Intelligent Resource Scheduling for Co-located Latency-critical Services: A Multi-Model Collaborative Learning Approach

Lei Liu, *Beihang University;* Xinglei Dou and Yuetao Chen, *ICT, CAS; Sys-Inventor Lab*

https://www.usenix.org/conference/fast23/presentation/liu

This paper is included in the Proceedings of the 21st USENIX Conference on File and Storage Technologies.

February 21–23, 2023 • Santa Clara, CA, USA

978-1-939133-32-8

Intelligent Resource Scheduling for Co-located Latency-critical Services: A Multi-Model Collaborative Learning Approach

Lei Liu[1]*, Xinglei Dou[2], Yuetao Chen[2]
[1]Beihang University; [2]ICT, CAS; Sys-Inventor Lab

Abstract

Latency-critical services have been widely deployed in cloud environments. For cost-efficiency, multiple services are usually co-located on a server. Thus, run-time resource scheduling becomes the pivot for QoS control in these complicated co-location cases. However, the scheduling exploration space enlarges rapidly with the increasing server resources, making schedulers unable to provide ideal solutions quickly and efficiently. More importantly, we observe that there are "resource cliffs" in the scheduling exploration space. They affect the exploration efficiency and always lead to severe QoS fluctuations in previous schedulers. To address these problems, we propose a novel ML-based intelligent scheduler – OSML. It learns the correlation between architectural hints (e.g., IPC, cache misses, memory footprint, etc.), scheduling solutions and the QoS demands. OSML employs multiple ML models to work collaboratively to predict QoS variations, shepherd the scheduling, and recover from QoS violations in complicated co-location cases. OSML can intelligently avoid resource cliffs during scheduling and reach an optimal solution much faster than previous approaches for co-located applications. Experimental results show that OSML supports higher loads and meets QoS targets with lower scheduling overheads and shorter convergence time than previous studies.

1 Introduction

Cloud applications are shifting from monolithic architectures to loosely-coupled designs, consisting of many latency-critical (LC) services (e.g., some microservices, interactive services) with strict QoS requirements [18,19,52,53]. Many cloud providers, including Amazon, Alibaba, Facebook, Google, and LinkedIn, employ this loosely-coupled design to improve productivity, scalability, functionality, and reliability of their cloud systems [2,3,18,52].

QoS-driven resource scheduling faces more challenges in this era. The cost-efficiency policy drives providers to co-locate as many applications as possible on a server. However, these co-located services exhibit diverse behaviors across the storage hierarchy, including multiple interactive resources such as CPU cores, cache, memory/IO bandwidth, and main memory banks. These behaviors also can be drastically different from demand to demand and change within seconds. Moreover, with the increasing number of cores, more threads contend for the shared LLC (last-level cache) and memory bandwidth. Notably, these shared resources interact with each other [32,33,45,80]. All these issues make resource scheduling for co-located LC services more complicated and time-consuming. Moreover, in reality, end-users keep increasing demands for quick responses from the cloud [15,47,53]. According to Amazon's estimation, even if the end-users experience a 1-second delay, they tend to abort the transactions, translating to $1.6 billion loss annually [1]. Briefly, unprecedented challenges are posed for resource scheduling mechanisms [8,10,31,47,52].

Some previous studies [17,31,33,45,61] design clustering approaches to allocate LLC or LLC together with main memory bandwidth to cores for scheduling single-threaded applications. Yet, they are not suitable in cloud environments, as the cloud services often have many concurrent threads and strict QoS constraints (i.e., latency-critical). Alternatively, the existing representative studies either use heuristic algorithms – increasing/decreasing one resource at a time and observing the performance variations [10] or use learning-based algorithms (e.g., Bayesian optimization [46]) in a relatively straightforward manner. The studies in [10,46] show that scheduling five co-located interactive services to meet certain QoS constraints can take more than 20 seconds on average. Existing schedulers still have room for improvement in scheduling convergence time, intelligence, and how to schedule complicated interactive resources (e.g., parallel computing units and complex memory/storage hierarchies) in a timely fashion. Moreover, the existing schedulers cannot easily avoid "resource cliffs", i.e., decreasing a resource only slightly during scheduling leads to a significant QoS slowdown. To the best of our knowledge, our community has been expecting new directions on developing resource-scheduling mechanisms to handle co-located LC services [16,30,31,45].

To this end, we design OSML, a novel machine learning (ML) based resource scheduler for LC services on large-scale servers. Using ML models significantly improves scheduling exploration efficiency for multiple co-located cloud services and can handle the complicated resource sharing, under/over-provision cases timely. ML has achieved tremendous success in improving speech recognition [54], benefiting image recognition [25], and helping the machine to beat the human champion at Go [13,24,51]. In OSML, we make progress in leveraging ML for resource scheduling, and we make the following contributions.

(1) Investigation in RCliff for Multiple Resources during Scheduling. In the context of cloud environment, we study resource cliff (RCliff, i.e., reducing a resource only slightly leads to a significant QoS slowdown) for computing and cache resources. More importantly, we show that RCliffs commonly

*Corresponding author (PI): lei.liu@zoho.com; liulei2010@buaa.edu.cn.

Table 1: Latency-critical (LC) services, including micro-/interactive services [18,19,52,68,46]. The max load - max RPS - is with the 99th percentile tail latency QoS target [10,18,46,52].

LC service	Domain	RPS (Requests Per Second)
Img-dnn [62]	Image recognition	2000,3000,4000,5000,6000 (Max)
Masstree [62]	Key-value store	3000,3400,3800,4200,4600
Memcached [65]	Key-value store	256k,512k,768k,1024k,1280k
MongoDB [64]	Persistent database	1000,3000,5000,7000,9000
Moses [62]	RT translation	2200,2400,2600,2800,3000
Nginx [66]	Web server	60k,120k,180k,240k,300k
Specjbb [62]	Java middleware	7000,9000,11000,13000,15000
Sphinx [62]	Speech recognition	1,4,8,12,16
Xapian [62]	Online search	3600,4400,5200,6000,6800
Login [68]	Login	300,600,900,1200,1500
Ads [68,52]	Online renting ads	10,100,1000

exist in many widely used LC services and challenge existing schedulers (Sec.3.3). Furthermore, we show ML can be an ideal approach that benefits scheduling (Sec.4.4).

(2) Collaborative ML Models for Intelligent Scheduling. OSML is an ML-based scheduler that intelligently schedules multiple interactive resources to meet co-located services' QoS targets. OSML learns the correlation between architectural hints (e.g., IPC, cache misses, memory footprint, etc.), optimal scheduling solutions, and the QoS demands. It employs MLP models to avoid RCliffs intelligently, thus avoiding the sudden QoS slowdown often incurred by the RCliffs in prior schedulers; it predicts the QoS variations and resource margins, and then delivers appropriate resource allocations. It leverages an enhanced DQN to shepherd the allocations and recover from the QoS violation and resource over-provision cases. Moreover, as OSML's models are lightweight and their functions are clearly defined, it is easy to locate the problems and debug them.

(3) An Open-sourced Data Set for Low-overhead ML. We have collected the performance traces for widely deployed LC services (in Table 1), covering 62,720,264 resource allocation cases that contain around 2-billion samples (Sec.4). These data have a rich set of information, e.g., the RCliffs for multiple resources; the interactions between workload features and the mainstream architectures. Our models can be trained and generalized with these data and then used on new platforms with low-overhead transfer learning. People can study the data set and train their models without a long period for data collection.

(4) Real Implementation and Detailed Comparisons. We implement OSML based on latest Linux. OSML is designed as a co-worker of the OS scheduler located between the OS kernel and the user layer. We compare OSML with the most related open-source studies [10,46] and show the advantages.

OSML captures the applications' online behaviors, forwards them to the ML models running on CPU or GPU, and schedules resources accordingly. Compared with [10,46], OSML takes 36~55% less time to meet the QoS targets; and supports 10~50% higher loads in 58% cases where Moses, Img-dnn, and Xapian can be co-scheduled in our experiments. Its ML models have low run-time overheads. OSML project and its ecological construction receive support from industry and academia; a version for research will be open-sourced via https://github.com/Sys-Inventor-Lab/AI4System-OSML.

Table 2: Our platform specification vs. a server in 2010~14 [80].

Conf. / Servers	Our Platform	Server (2010s)
CPU Model	Intel Xeon E5-2697 v4	Intel i7-860
Logical Processor Cores	36 Cores (18 phy. cores)	8 Cores (4 phy. cores)
Processor Speed	2.3GHz	2.8GHz
Main Memory / Channel / BW	256GB, 2400MHz DDR4 / 4 Channels / 76.8GB/s	8GB, 1600MHz DDR3 / 2 Channels / 25.6GB/s
Private L1 & L2 Cache Size	32KB and 256KB	32KB and 256KB
Shared L3 Cache Size	45MB - 20 ways	8MB - 16 ways
Disk	1TB,7200 RPM,HD	500GB,5400 RPM,HD
GPU	NVIDIA GP104 [GTX 1080], 8GB Memory	N/A

2 Background and Motivation

The cloud environment has a trend towards a new model [3,18,52], in which cloud applications comprise numerous distributed LC services (i.e., micro/interactive services), such as key-value storing, database serving, and business applications serving [18,19]. Table 1 includes some widely used ones, and they form a significant fraction of cloud applications [18]. These services have different features and resource demands.

In terms of the datacenter servers, at present, new servers can have an increased number of cores, larger LLC capacity, larger main memory capacity, higher bandwidth, and the resource scheduling exploration space becomes much larger than ever before as a result. Table 2 compares the two typical servers used at different times. On the one hand, although modern servers can have more cores and memory resources than ever before, they are not fully exploited in today's cloud environments. For instance, in Google's datacenter, the CPU utilization is about 45~53% and memory utilization ranges from 25~77% during 25 days, while Alibaba's cluster exhibits a lower and unstable trend, i.e., 18~40% for CPU and 42~60% for memory in 12 hours [32,49], indicating that a lot of resources are wasted. On the other hand, the larger resource scheduling exploration space, which consists of more diverse resources, prohibits the schedulers from achieving the optimal solution quickly. Additionally, cloud applications can have dozens of concurrent threads [10,46]. When several cloud applications run on a server, they share and contend resources across multiple resource layers – cores, LLC, memory bandwidth/banks. Previous studies show they may incur severe performance degradation and unpredictable QoS violations, and propose the scheduling approaches at architecture [9,23,44], OS [31,45,50,80], and user-level [10,37,38]. *Yet, do they perform ideally for scheduling co-located LC services on modern datacenter servers?*

3 Resource Scheduling for LC Services

To answer the above question, we study the LC services (Table 1) that are widely deployed in cloud environments.

3.1 Understanding the LC Services - Resource Cliff

We study how sensitive these LC services are to the critical resources, e.g., the number of cores and LLC capacity,

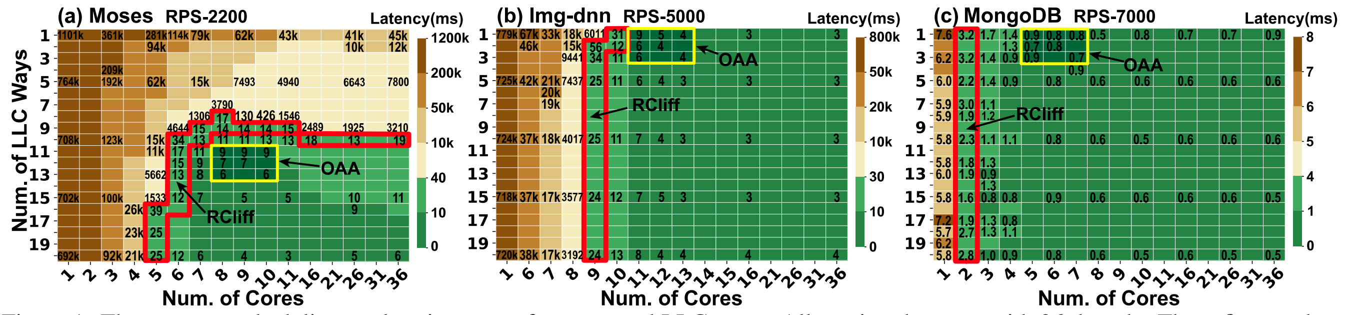


Figure 1: The resource scheduling exploration space for cores and LLC ways. All services here are with 36 threads. These figures show the sensitivity to resource allocation under different policies. Each col./row represents a specific number of LLC ways/cores allocated to an application. Each cell denotes the LC service's response latency under the given number of cores and LLC ways. The Redline highlights the RCliff (can be obtained by selecting the knee solution [69]). The green color cells show allocation policies that bring better performance (low response latency). OAA (Optimal Allocation Area) is also illustrated for each LC service. We test all of the LC services in Table 1, and we find the RCliff and OAA are always existing, though the RPS varies [79]. We only show several of them for the sake of saving space.

on a commercial platform ("our platform" in Table 2). For Moses, as illustrated in Figure 1-a, with the increasing number of cores, more threads are mapped on them simultaneously. Meanwhile, for a specific amount of cores, more LLC ways can benefit performance. Thus, we observe the response latency is low when computing and LLC resources are ample (i.e., below 10ms in the area within green color). The overall trends are also observed from other LC services.

However, we observe the Cliff phenomenon for these services. In Figure 1-a, in the cases where 6 cores are allocated to Moses, the response latency is increased significantly from 34ms to 4644ms if merely one LLC way is reduced (i.e., from 10 ways to 9 ways). Similar phenomena also happen in cases where computing resources are reduced. As slight resource re-allocations bring a significant performance slowdown, we denote this phenomenon as Resource Cliff (**RCliff**). It is defined as the resource allocation cases that could incur the most significant performance slowdown if resources (e.g., core, cache) are deprived via a fine-grain way in the scheduling exploration space. Take Moses as an example, on the RCliff (denoted by the red box in Figure 1-a), there would be a sharp performance slowdown if only one core or one LLC way (or both) is deprived. Figure 1-b and c show RCliffs for Img-dnn and MongoDB, respectively. They exhibit computing-sensitive features and have RCliff only for cores. From another angle, RCliff means that a little bit more resources will bring significant performance improvement. Figure 1-a shows that Moses exhibits RCliff for both core and LLC. Moreover, we test the services in Table 1 across various RPS and find the RCliffs always exist, though the RCliffs vary (8.8% on average) according to different RPS.

The underlying reason for the cache cliff is locality; for the core cliff, the fundamental reason is on queuing theory - the latency will increase drastically when the request arrival rate exceeds the available cores. RCliff alerts the scheduler not to allocate resources close to it because it is "dangerous to fall off the cliff" and incurs a significant performance slowdown, i.e., even a slight resource reduction can incur a severe slowdown. Notably, in Figure 1, we highlight each LC service's **Optimal Allocation Area (OAA)** in the scheduling exploration space,

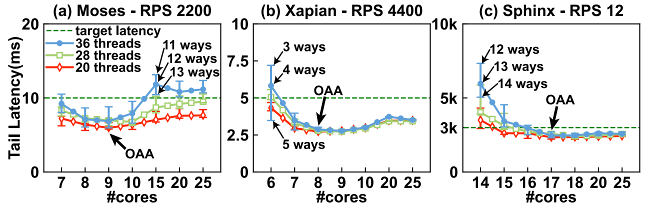


Figure 2: OAA exists regardless of the num. of concurrent threads.

defined as the ideal number of allocated cores and LLC ways to bring an acceptable QoS. More resources than OAA cannot deliver more significant performance, but fewer resources lead to the danger of falling off the RCliff. OAA is the goal that schedulers should achieve.

3.2 Is OAA Sensitive to the Number of Threads?

In practice, an LC service may have many threads for higher performance. Therefore, we come up with the question: *is the OAA sensitive to the number of threads, i.e., if someone starts more threads, will the OAA change?*

To answer this question, for a specific LC service, we start a different number of threads and map them across a different number of cores (the num. of threads can be larger than the num. of cores). Through the experiments, we observe - (i) More threads do not necessarily bring more benefits. Take Moses as an example, when more threads are started (e.g., 20/28/36) and mapped across a different number of cores, the overall response latency can be higher (as illustrated in Figure 2). The underlying reason lies in more memory contentions at memory hierarchy and more context switch overheads, leading to a higher response latency [20,36]. (ii) The OAA is not sensitive to the number of concurrent threads. For Moses in Figure 2, even if 20/28/36 threads are mapped to 10~25 cores, around 8/9-core cases always perform ideally. Other LC services in Table 1 also show a similar phenomenon, though their OAAs are different for different applications.

In practice, if the QoS for a specific LC service is satisfied, LLC ways should be allocated as less as possible, saving LLC space for other applications. Similarly, we also try to allocate fewer cores for saving computing resources. Here, we conclude that the OAA is not sensitive to the number

Table 3: The input parameters for ML models.

Feature	Description	Models
IPC	Instructions per clock	A/A'/B/B'/C
Cache Misses	LLC misses per second	A/A'/B/B'/C
MBL	Local memory bandwidth	A/A'/B/B'/C
CPU Usage	The sum of each core's utilization	A/A'/B/B'/C
Virt. Memory	Virtual memory in use by an app	A/A'/B/B'
Res. Memory	Resident memory in use by an app	A/A'/B/B'
Allocated Cores	The number of allocated cores	A/A'/B/B'/C
Allocated Cache	The capacity of allocated cache	A/A'/B/B'/C
Core Frequency	Core Frequency during run time	A/A'/B/B'/C
QoS Slowdown	Percentage of QoS slowdown	B
Expected Cores	Expected cores after deprivation	B'
Expected Cache	Expected cache after deprivation	B'
Cores used by N.	Cores used by Neighbors	A'/B/B'
Cache used by N.	Cache capacity used by Neighbors	A'/B/B'
MBL used by N.	Memory BW used by Neighbors	A'/B/B'
Resp. Latency	Average latency of a LC service	C

of threads. We should further reveal: *how do the existing schedulers perform in front of OAAs and RCliffs?*

3.3 Issues the Existing Schedulers May Meet

We find three main shortcomings in the existing schedulers when dealing with OAAs and RCliffs. **(1) Entangling with RCliffs.** Many schedulers often employ heuristic scheduling algorithms, i.e., they increase/reduce resources until the monitor alerts that the system performance is suffering a significant change (e.g., a severe slowdown). Yet, these approaches could incur unpredictable latency spiking. For example, if the current resource allocation for an LC service is in the base of RCliff (i.e., the yellow color area in Figure 1-a/b/c), the scheduler has to try to achieve OAA. However, as the scheduler doesn't know the "location" of OAA in the exploration space, it has to increase resources step by step in a fine-grain way, thus the entire scheduling process from the base of the RCliff will incur very high response latency. For another example, if the current resource allocation is on the RCliff or close to RCliff, a slight resource reduction for any purpose could incur a sudden and sharp performance drop for LC services. The previous efforts [10,32,50,53] find there would be about hundreds/thousands of times latency jitter, indicating the QoS cannot be guaranteed during these periods. Thus, RCliffs should not be neglected when designing a scheduler. **(2) Unable to accurately and simultaneously schedule a combination of multiple interactive resources (e.g., cores, LLC ways) to achieve OAAs in low overheads.** Prior studies [10,31,32,45] show that the core computing ability, cache hierarchy, and memory bandwidth are interactive factors for resource scheduling. Solely considering a single dimension in scheduling often leads to sub-optimal QoS. However, the existing schedulers using heuristic or model-based algorithms usually schedule one dimension resource at a time and bring high overheads on scheduling multiple interactive resources. For example, PARTIES [10] takes around 20~30 seconds on average (up to 60 seconds in the worst cases) to find ideal allocations when 3~6 LC services are co-running. The efforts in [16,41,42] also show the heuristics inefficiency due to the high overheads on scheduling various resources with complex configurations. **(3) Unable to provide accurate QoS predictions.** Therefore, the scheduler can hardly balance the global QoS and resource allocations across all co-located applications, leading to QoS violations or resource over-provision.

An ideal scheduler should avoid the RCliff and quickly achieve the OAA from any positions in the scheduling space. We claim it is time to design a new scheduler, and using ML can be a good approach to handle such complicated cases with low overheads.

4 Leveraging ML for Scheduling

We design a new resource scheduler - OSML. It differs from the previous studies in the following ways. We divide the resource scheduling for co-located services into several routines and design ML models to handle them, respectively. These models work collaboratively in OSML to perform scheduling. OSML uses data-driven static ML models (Model-A/B) to predict the OAA/RCliff, and balances the QoS and resource allocations among co-located LC services, and uses the reinforcement learning model (Model-C) to shepherd the allocations dynamically. Moreover, we collect extensive real traces for widely deployed LC services.

4.1 Model-A: Aiming OAA

Model-A Description. The neural network used in Model-A is a 3-layer multi-layer perceptron (MLP); each layer is a set of nonlinear functions of a weighted sum of all outputs that are fully connected from the prior one [21,24]. There are 40 neurons in each hidden layer. There is a dropout layer with a loss rate of 30% behind each fully connected layer to prevent overfitting. For each LC service, the **inputs** of the MLP include 9 items in Table 3 and the **outputs** include the OAA for multiple interactive resources, OAA bandwidth (bandwidth requirement for OAA), and the RCliff. Model-A has a shadow – A', which has the same MLP structure and 12 input parameters (Table 3), providing solutions when multiple LC services are running together.

Model-A Training. Collecting training data is an offline job. We have collected the performance traces that involve the parameters in Table 3 for the LC services in Table 1 on "our platform" in Table 2. The parameters are normalized into [0,1] according to the function: $\text{Normalized\_Feature} = (\text{Feature} - \text{Min})/(\text{Max} - \text{Min})$. Feature is the original value; Max and Min are predefined according to different metrics.

For each LC service with every common RPS demand, we sweep 36 threads to 1 thread across LLC allocation policies ranging from 1 to 20 ways and map the threads on a certain number of cores and collect the performance trace data accordingly. In each case, we label the corresponding OAA, RCliff and OAA bandwidth. For example, Figure 3 shows a data collection case where 8 threads are mapped onto 7 cores with 4 LLC ways. We feed the LC services with diverse RPS (Table 1), covering most of the common cases. Moreover, to train Model-A's shadow (A'), we map LC services on the remaining resources in the above process and get the traces for co-location cases. Note that the resources

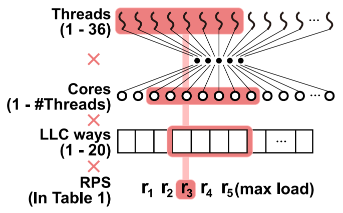


Figure 3: Model-A data collection.

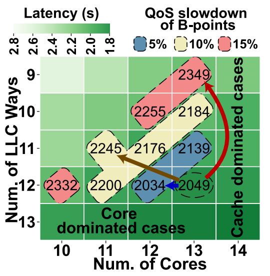


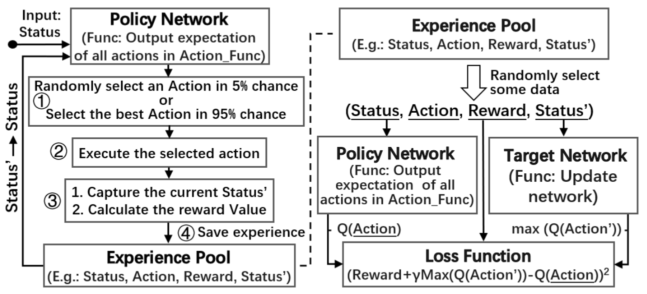
Figure 4: Model-B training.

are partitioned among applications. We test comprehensive co-location cases for LC services in Table 1, and find the LC services' RCliffs vary from 2.8% to 38.9%, and OAAs vary from 5.6% to 36.1%, respectively, when multiple LC services are co-running. Finally, we collect 43,299,135 samples (data tuples), covering 1,308,296 allocation cases with different numbers of cores, LLC ways, and bandwidth allocations. A large amount of traces may lead to higher accuracy for ML models. The workload characteristics are converted to traces consisting of hardware parameters used for fitting and training MLP to provide predictions.

4.2 Model-B: Balancing QoS and Resources

Model-B Description. Model-B employs an MLP with the same structure in Model-A' plus one more input item, i.e., QoS slowdown (Table 3). Model-B **outputs** the resources that a service can be deprived of under allowable QoS slowdown. As the computing units and memory resource can be fungible [10], Model-B's outputs include three policies, i.e., <cores, LLC ways>, <cores dominated, LLC ways> and <cores, LLC ways dominated>, respectively. The tuple items are the number of cores, LLC ways deprived and reallocated to others with the allowable QoS slowdown. The term "cores dominated" indicates the policy using more cores to trade the LLC ways, and vice versa. The allowable QoS slowdown is determined according to the user requirement or the LC services' priority and controlled by the OSML's central logic. We denote Model-B's outputs as B-Points.

Model-B trades QoS for resources. For example, when an LC service (E) comes to a server that already has 4 co-located services, OSML enables Model-A' to obtain <n+, m+>, which denotes at least n more cores and m more LLC ways should be provided to meet E's QoS. Then, OSML enables Model-B and uses the allowable QoS slowdown as an input to infer B-Points for obtaining resources from other co-located services. B-Points include the "can be deprived" resources from E's neighbors with the allowable QoS slowdown. Finally, OSML finds the best solution to match <n+, m+> with B-Points, which has a minimal impact on the co-located applications' current allocation state. Detailed logic is in Algo._1. Besides, we design Model-B' (a shadow of Model-B) to predict how much QoS slowdown will suffer if a certain amount of resources is deprived of a specific service. Model-B' has an MLP with the same structure in Model-A' plus the items that indicate the remaining cache ways and cores after deprivation.


Input: Status
Policy Network (Func: Output expectation of all actions in Action_Func)
① Randomly select an Action in 5% chance or Select the best Action in 95% chance
② Execute the selected action
③ 1. Capture the current Status' 2. Calculate the reward Value
④ Save experience
Status' → Status
Experience Pool (E.g.: Status, Action, Reward, Status')
Experience Pool (E.g.: Status, Action, Reward, Status')
Randomly select some data
(Status, Action, Reward, Status')
Policy Network (Func: Output expectation of all actions in Action_Func)
Target Network (Func: Update network)
Q(Action)
max (Q(Action'))
Loss Function (Reward+γMax(Q(Action'))-Q(Action))²
Figure 5: Model-C in a nutshell.

Model-B Training. For training Model-B and B', we reduce the allocated resources for a specific LC service from its OAA by fine-grain approaches, as illustrated in Figure 4. The reduction has three angles, i.e., horizontal, oblique, and vertical, i.e., B-Points include <cores dominated, LLC ways>, <cores, LLC ways>, <cores, LLC ways dominated>, respectively. For each fine-grain resource reduction step, we collect the corresponding QoS slowdowns and then label them as less equal to ($\leq$) 5%, 10%, 15%, and so on, respectively. Examples are illustrated in Figure 4, which shows the B-Points with the corresponding QoS slowdown. We collect the training data set for every LC service in Table 1. The data set contains 65,998,227 data tuples, covering 549,987 cases.

Model-B Function. We design a new loss function:

$$L = \frac{1}{n}\sum_{t=1}^{n}\left(\frac{y_t}{y_t + c} \times (s_t - y_t)\right)^2,$$

in which s_t is the prediction output value of Model-B, y_t is the labeled value in practice, and 'c' is a constant that is infinitely close to zero. We multiply the difference between s_t and y_t by $\frac{y_t}{y_t+c}$ for avoiding adjusting the weights during backpropagation in the cases where $y_t = 0$ and $\frac{y_t}{y_t+c} = 0$ caused by some non-existent cases (we label the non-existent cases as 0, i.e., $y_t = 0$, indicating we don't find a resource-QoS trading policy in the data collection process).

4.3 Model-C: Handling the Changes On the Fly

Model-C Description. Model-C corrects the resource under-/over-provision and conducts online training. Figure 5 shows the Model-C in a nutshell. Model-C's core component is an enhanced Deep Q-Network (DQN) [43], consisting of two neural networks, i.e., Policy Network and Target Network. The Policy and Target Network employ the 3-layer MLP, and each hidden layer has 30 neurons. Policy Network's **inputs** consist of the parameters in Table 3, and the **outputs** are resource scheduling actions (e.g., reducing/increasing a specific number of cores or LLC ways) and the corresponding expectations (defined as Q(action)). These actions numbered 0~48 are defined as Action_Function:$\{<m,n> | m \in [-3,3], n \in [-3,3]\}$, in which a positive m denotes allocating m more cores (i.e., add operation) for an application and a negative m means depriving it of m cores (i.e., sub operation); n indicates the actions on LLC ways. Figure 5 illustrates Model-C's logic. The scheduling action with the maximum expectation value (i.e., the action towards the best solution) will be selected in ① and executed in ②. In ③, Model-C will get the Reward value according to the Reward Function. Then, the tuple <Status, Action, Reward, Status'> will be saved in the Experience Pool in ④, which will be used during online training. The terms Status and Status' denote system's status described by the parameters in Table 3 before and after the Action is taken. Model-C can quickly have the ideal solutions in practice (about 2 or 3 actions). Please note that in ①, Model-C might randomly select an Action instead of the best Action with a 5% chance. By doing so, OSML avoids falling into a local optimum [43]. These random actions have a 44% chance of causing QoS violations when Model-C reduces resources. But OSML can handle the QoS violations by withdrawing the action (line 9 in Algo._3).

Model-C's Reward Function. The reward function of Model-C is defined as follow:

If $\mathrm{Latency}_{t-1} > \mathrm{Latency}_t$:
$$R_t = \log(1 + \mathrm{Latency}_{t-1} - \mathrm{Latency}_t) - (\Delta\mathrm{CoreNum} + \Delta\mathrm{CacheWay})$$
If $\mathrm{Latency}_{t-1} < \mathrm{Latency}_t$:
$$R_t = -\log(1 + \mathrm{Latency}_t - \mathrm{Latency}_{t-1}) - (\Delta\mathrm{CoreNum} + \Delta\mathrm{CacheWay})$$
If $\mathrm{Latency}_{t-1} = \mathrm{Latency}_t$:
$$R_t = -(\Delta\mathrm{CoreNum} + \Delta\mathrm{CacheWay}),$$
where $\mathrm{Latency}_{t-1}$ and $\mathrm{Latency}_t$ denote the latency in previous and current status, respectively; ΔCoreNum and ΔCacheWay represent the changes in the number of cores and LLC ways, respectively. This function gives higher reward and expectation to Action that brings less resource usage and lower latency. Thus, Model-C can allocate appropriate resources. Algo._2 and 3 show the logic on using Model-C in detail.

Offline Training. A training data tuple includes Status, Status', Action and Reward, which denote the current status of a LC service, the status after these actions are conducted (e.g., reduce several cores or allocate more LLC ways) and the reward calculated using the above functions, respectively.

To create the training data set for Model-C, we resort to the data set used in Model-A training. The process is as follows. Two tuples in Model-A training data set are selected to denote Status and Status', and we further get the differences of the resource allocations between the two status (i.e., the actions that cause the status shifting). Then, we use the reward function to have the reward accordingly. These 4 values form a specific tuple in Model-C training data set. In practice, as there are a large number of data tuples in Model-A training data set, it is impossible to try every pair of tuples in the data set, we only select two tuples from resource allocation policies that have less than or equal to 3 cores, or 3 LLC ways differences. Moreover, we also collect the training data in the cases where cache way sharing happens and preserve them in the experience pool. Therefore, Model-C can work in resource-sharing cases. To sum up, we have 1,521,549,190 tuples in Model-C training data set.

Online Training. Model-C collects online traces. The training flow is in the right part of Figure 5. Model-C randomly selects some data tuples (200 by default) from the Experience Pool. For each tuple, Model-C uses the Policy Network to get the Action's expectation value (i.e., Q(Action) [43]) with the Status. In Model-C, the target of the Action's expectation value is the Reward observed plus the weighted best expectation value of the next status (i.e., Status'). As illustrated in Figure 5, Model-C uses the Target Network to have the expectation values of Status' for the actions in Action_Function and then finds the best one, i.e., $\mathrm{Max}(\mathrm{Q}(\mathrm{Action}'))$. We design a new Loss Function based on MSE: $(\mathrm{Reward} + \gamma\mathrm{Max}(\mathrm{Q}(\mathrm{Action}')) - \mathrm{Q}(\mathrm{Action}))^2$. It helps the Policy Network predict closer to the target. The Policy Network is updated during online training. The Target Network's weights are synchronized periodically with the Policy Network's weights. Doing so enables the Target Network to provide stable predictions for the best expectation value of Status' within a predefined number of time steps, thus improving the stability of the training and prediction.

Table 4: The Summary of ML models in OSML.

ML	Model	Features	Model Size	Loss Function	Gradient Descent	Activation Function
A	MLP	9	144 KB	Mean Square Error (MSE)	Adam Optimizer	ReLU
A'	MLP	12	155 KB			
B	MLP	13	110 KB	Modified MSE		
B'	MLP	14	106 KB	MSE		
C	DQN	8	141 KB	Modified MSE	RMSProp	

4.4 Discussions on the design of ML Models

(1) Why using MLPs. Table 4 characterizes the ML models used in OSML. We employ three-layered MLPs in Model-A and B, because they can fit continuous functions with an arbitrary precision given a sufficient number of neurons in each layer [67], and we can use extensive training data to improve the accuracy of MLPs for predicting OAAs and RCliffs. Moreover, after offline training, using MLPs brings negligible run-time overheads to OSML. **(2) Why do we need the Three models?** We divide the OSML's scheduling logic into three parts, which the three models cover, respectively. Models work in different scheduling phases, and no single model can handle all cases. For example, model-A predicts the RCliffs and OAAs; Model-B predicts the QoS variations and resource margins in co-location cases. DQN in Model-C learns online to shepherd the scheduling results from Model-A/B. They are necessary and work cooperatively to cover the main scheduling cases. Moreover, they are easier to generalize than other approaches, e.g., a table lookup approach (Sec.6.4). *Why Not Only use the online learning Model-C?* Model-C uses DQN that depends on the start points. It starts with Model-A/B's outputs to avoid exploring the whole (large) scheduling space. Without the approximate OAA provided by Model-A for many unseen cases, only using Model-C will incur more scheduling actions (overheads). **(3) Insights on Generalization for Unseen apps and New servers.** (i) We use "hold-out cross validation", i.e., the training data (70% of the whole data set) excludes the testing data (30%) for each LC service. (ii) We train models with extensive representative traces from many

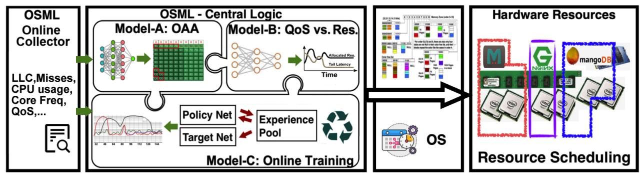


Figure 6: The overview of the system design of OSML.

typical services (e.g., memory/CPU intensive, random memory access, etc. [10,46]), fitting the correlation between architectural hints for mainstream features (e.g., IPC, cache miss, memory footprint, CPU/LLC utilization and MBL), OAA, and the QoS demands. For instance, the spearman correlation coefficient is 0.571, 0.499, and -0.457 between OAA and cache miss, MBL, and IPC, respectively. *On different platforms or for new/unseen applications*, these numbers might be varied; however, this correlation trend is not changed, enabling OSML to generalize to other situations. Even for errors caused by deviations from the training dataset for unseen applications/new platforms, model-C can learn online and correct them. (iii) Using transfer learning, collecting new traces on a new server for several hours will make OSML work well for it (refer to Sec.6.4). (iv) OSML is a long-term project open to the community; we continue adding new traces collected from new applications and new servers to the data set for enhancing models' performance for new cases.

5 OSML: System Design

5.1 The Central Control Logic

The overview of OSML is in Figure 6. OSML is a per-node scheduler. The central controller of OSML coordinates the ML models, manages the data/control flow, and reports the scheduling results. Figure 7 shows its overall control logic.

Allocating Resources for LC services. Algo._1 shows how OSML uses Model-A and B in practice. Figure 7 highlights its operations. For a newly coming LC service, the central controller calls Model-A via the interface *modelA_oaa_rcliff()* to get the OAA and RCliff. Suppose the current idle resources are not sufficient to satisfy the new LC service. In that case, OSML will enable Model-B through the interface *modelB_trade_qos_res()* to deprive some resources of other LC services with the allowable QoS slowdown (controlled by the upper-level scheduler) and then allocate them to the new one. The upper-level scheduler manages the cluster consisting of nodes using OSML. In the depriving process for a specific LC service, OSML reduces its allocated resources and gets close to the RCliff, but it will not easily fall off the RCliff unless expressly permitted (refer to Algo._4).

Dynamic Adjusting. Figure 7 shows the dynamic adjusting of Algo._2 and 3, in which Model-C works as a dominant role. During the run time, OSML monitors each LC service's QoS status for every second. If the QoS violation is detected, the central controller will enable Algo._2 and call Model-C to allocate more resources to achieve the ideal QoS. The interface is *modelC_upsize()*. If OSML finds an LC service is over-provisioned (i.e., wasting resources), Algo._3 will be used to reclaim them,

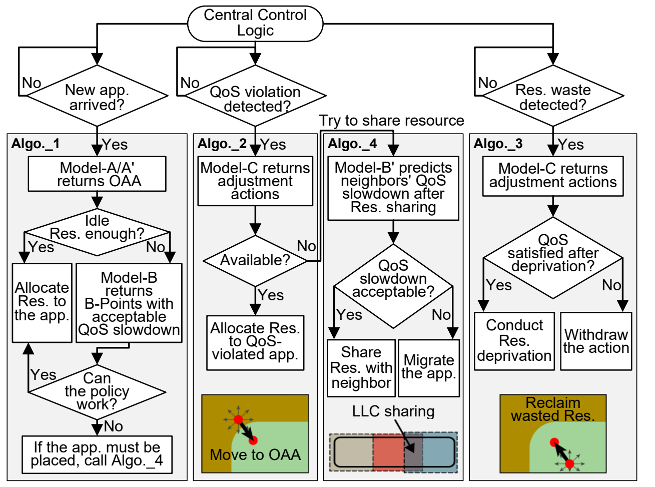


Figure 7: OSML's central logic.

and Model-C will be called via *modelC_downsize()*.

Moreover, Algo._4 will enable resource sharing (the default scheduling is to do hard partitioning of cores/LLC ways), if all of the co-located LC services are close to their RCliff and the upper scheduler still wants to increase loads on this server. Model-A and B work cooperatively to accomplish this goal in Algo._4. In practice, to minimize the adverse effects, resource sharing usually happens between only two applications. Note that Algo._4 might incur the resource sharing over the RCliff, and thus may bring higher response latency for one or more LC services. OSML will report the potential QoS slowdown to the upper scheduler and ask for the decisions. If the slowdown is not allowed, the corresponding actions will not be conducted.

Bandwidth Scheduling. OSML partitions the overall bandwidth for each co-located LC service according to the ratio $\mathrm{BW_j}/\sum \mathrm{BW_i}$. $\mathrm{BW_j}$ is a LC service's OAA bandwidth requirement, which is obtained from the Model-A. Note that such scheduling needs Memory Bandwidth Allocation (MBA) mechanism [4,5] in CPU.

5.2 Implementation

OSML learns whether the LC services have met their QoS targets. OSML monitors the run-time parameters for each co-located LC service using performance counters for every second (default). If the observation period is too short, other factors (e.g., cache data evicted from the cache hierarchy, context switch) may interfere with the sampling results. Moreover, OSML performs well with other interval settings and allows configuration flexibility (e.g., 1.5 or 2 seconds).

We design OSML that works cooperatively with OS (Figure 6). As the kernel space lacks the support of ML libraries, OSML lies in the user space and exchanges information with the OS kernel. OSML is implemented using python and C. It employs Intel CAT technology [4] to control the cache way allocations, and it supports dynamically adjusting. OSML uses Linux's taskset and MBA [5] to allocate specific cores and bandwidth to an LC service. OSML captures the online performance parameters by using the pqos tool [4] and PMU

Algorithm 1: using ML to have OPT resources allocations. In practice, only one policy in OAA will be selected.

```
1  For a coming LC service, map it on the idle resources and capture its
   run-time parameters in Table 3 for 1 second (default);
2  Forward these parameters as inputs to Model-A (A' when several LC
   services are co-located);
3  Model-A outputs: (1) OAA to meet the target QoS; (2) OAA bw (3)
   RCliff in current environment;
4  if idle resources are sufficient to meet OAA then
5      Allocate resources with a specific solution in OAA;
6  end
7  if idle resources are not enough then  // Enabling Model-B
8      Calculate the difference between the idle resources and its'
       OAA, i.e., <+cores, +LLC ways>  // required resource to meet
       its QoS;
9      Calculate the difference between the idle resources and RCliff,
       i.e., <+cores', +LLC ways'>  // should be used carefully;
10     for each previously running LC service do
11         if the one can tolerate a certain QoS slowdown then
12             Use Model-B to infer the B-Points with the acceptable
               QoS slowdown;
13             Model-B outputs the B-Points, i.e., <cores, LLC
               ways>, <cores dominated, LLC ways>, and etc.;
14         end
15     end
16     Record each LC service's B-Points with the QoS slowdown;
17     Find the best-fit solution to meet OAA/RCliff according to
       B-Points with at most 3 apps involved  // The less the better;
18     if the solution could meet OAA or RCliff then
19         Adjust allocations according to OAA (RCliff is alternative);
20     else
21         The LC service cannot be located on this server without
           sharing resources;
22     end
23 end  // Enabling Model-B
24 Report results to upper scheduler, call Algo._4 for sharing if needed.
```

[5]. The ML models are based on TensorFlow [6] with the version 2.0.4, and can be run on either CPU or GPU.

6 Evaluations

6.1 Methodology

We evaluate the per-node OSML performance on our platform in Table 2. Details on LC services are in Table 1. The metrics involve the QoS (similar to [10], the QoS target of each application is the 99th percentile latency of the knee of the latency-RPS curve. Latency higher than the QoS target is a violation.); Effective Machine Utilization (EMU) [10] (the max aggregated load of all co-located LC services) – higher is better. We first evaluate the scenarios where LC services run at constant loads, and the loads are from 10% - 100%. Then, we explore workload churn. We inject applications with loads from 20% - 100% of their respective max load. Furthermore, to evaluate the generalization of OSML, we employ some new/unseen applications that are not in Table 1 and the new platform in our experiments. If an allocation in which all applications meet their QoS cannot be found after 3 mins, we signal that the scheduler cannot deliver QoS for that configuration.

6.2 OSML Effectiveness

We compare OSML with the most related approaches in [10,46] based on the latest open-source version.

PARTIES [10]. It makes incremental adjustments in one-dimension resource at a time until QoS is satisfied – "trial and error" – for all of the applications. The core mechanism in [10] is like an FSM [60].

Algorithm 2: handling resource under-provision cases.

```
1  for each allocated LC service do
2      if QoS is not satisfied ∧ latency is increasing then
3          Forward the current running status parameters to Model-C;
4          Model-C selects a specific action in the Action_Fun;
5          Return Model-C's output (<cores+, LLC ways+>) to
           OSML's central controller;
6          if <cores+, LLC ways+> can be satisfied within current
             idle resources then
7              OSML allocates, and GOTO Line 2;
8          else
9              Call Algorithm_4.  // Share resources w/ others?
10         end
11     end
12 end
```

Algorithm 3: handling resource over-provision cases.

```
1  for each allocated LC service do
2      // Over-provision;
3      if its QoS is satisfied ∧ no idle resources left in system ∧ at least
         one of its neighbor's QoS is not satisfied then
4          Forward current run-time status parameters to Model-C;
5          Model-C selects a specific action accordingly;
6          Return Model-C's output (<cores-, LLC ways->) to
           OSML's central controller;
7          OSML reduces the resources accordingly;
8          if its QoS is not satisfied now then
9              OSML withdraws the actions.  // Rollback
10         end
11     end
12 end
```

Algorithm 4: handling resources sharing among apps.

```
1  // OSML tries to allocate resources cross over RCiff;
2  Obtain how many resources a LC service needs, i.e., <+cores, +LLC
   ways>, from the neighbors to meet its QoS using Model-A;
3  for each potential neighbor App do
4      Create sharing policies, i.e.,
       $\{< \mathrm{u}, \mathrm{v} > |\forall \mathrm{u} \leq (+\text{cores}) \wedge \forall \mathrm{v} \leq (+\text{LLC ways}); \mathrm{u}, \mathrm{v} \geq 0\}$;
5      Use Model-B' to predict the neighbor's QoS slowdown
       according to $\{< \mathrm{u}, \mathrm{v} >\}$;
6  end
7  if the neighbors' QoS slowdown can be accepted by OSML then
8      OSML conducts the allocation;
9  else
10     OSML migrate the LC service to another node.
11 end
```

CLITE [46]. It conducts various allocation policies and samples each of them; it then feeds the sampling results – the QoS and run-time parameters for resources – to a Bayesian optimizer to predict the next scheduling policy.

Unmanaged Allocation (baseline). This policy doesn't control the allocation policies on cores, LLC, and other shared resources for co-located LC services. This policy relies on the original OS schedulers.

ORACLE. We obtain these results by exhaustive offline sampling and find the best allocation policy. It indicates the ceiling that the schedulers try to achieve.

We show the effectiveness of OSML as follow.

(1) OSML exhibits a shorter scheduling convergence time. Using ML models, OSML achieves OAA quickly and efficiently handles cases with diverse loads. Figure 8-a shows the distributions of the scheduling results of 104 loads for OSML,

PARTIES and CLITE, respectively. Every dot represents a specific workload that contains 3 co-located LC services with different RPS. We launch the applications in turn and use a scheduler to handle QoS violations until all applications meet their QoS targets. The x-axis shows the convergence time; the y-axis denotes the achieved EMU. Generally, OSML can achieve the same EMU with a shorter convergence time for a specific load. Figure 8-b shows the violin plots of convergence time for these loads. On average, OSML takes 20.9 seconds to converge, while PARTIES and CLITE take 32.7 and 46.3 seconds, respectively. OSML converges 1.56X and 2.22X faster than PARTIES and CLITE. OSML performs stably – the convergence time ranges from 5.3s (best case) to 80.0s (worst case). By contrast, the convergence time in PARTIES ranges from 5.5s to 111.1s, and CLITE is from 14.0s to 140.6s. OSML converges faster mainly because the start point in the scheduling space provided by Model-A is close to OAA. PARTIES and CLITE take a longer convergence time, indicating that they require high scheduling overheads in cloud environments. In Cloud, jobs come and go frequently; thus, scheduling occurs frequently, and longer scheduling convergence time often leads to unstable/low QoS.

We further analyze how these schedulers work in detail. Figure 9-a/b/c show the actions used in OSML, PARTIES, and CLITE's scheduling process for case A in Figure 8. This case includes Moses, Img-dnn, and Xapian with 40%, 60%, and 50% of their maximum loads. For this load, PARTIES, CLITE and OSML take 14.5 seconds, 72.6 seconds and 8.2 seconds to converge, respectively. Figure 9 highlights scheduling actions using solid red lines to represent increasing resources and blue dotted lines to denote reducing resources. Figure 9-a shows PARTIES takes 7 actions for scheduling cores and 1 action for cache ways. It schedules in a fine-grained way by increasing/decreasing one resource at a time. CLITE relies on the sampling points in the scheduling exploration space. Figure 9-b shows CLITE repeats sampling until the "expected improvement" in CLITE drops below a certain threshold. CLITE only performs five scheduling actions according to its latest open-source version; but it takes the longest convergence time (72.6 seconds). The underlying reason is that CLITE's sampling/scheduling doesn't have clear targets. In practice, the improper resource partitions/allocations during sampling lead to the accumulation of requests, and the requests cannot be handled due to resource under-provision. Therefore, it brings a significant increase in response latency. Moreover, due to the early termination of CLITE's scheduling process, CLITE cannot schedule resources to handle QoS violations in a timely manner, leading to a long convergence time. Figure 9-c shows OSML achieves OAA for each LC service with 5 actions. Compared with prior schedulers, OSML has clear aims and schedules multiple resources simultaneously to achieve them. It has the shortest convergence time – 8.2 seconds.

Moreover, as the scheduling is fast, OSML often supports more loads. Figure 10 shows the OSML's results on schedul-

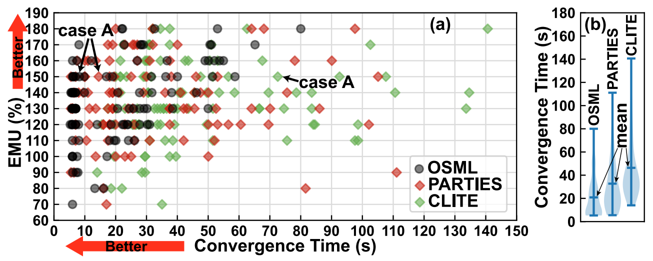


Figure 8: (a) The performance distributions for 104 loads that OSML, PARTIES and CLITE can all converge. (b) Violin plots of convergence time for loads in (a).

ing the three LC services – Moses, Img-dnn, and Xapian. For a specific scheduling phase, by using ML to achieve OAA, OSML supports 10~50% higher loads than PARTIES and CLITE in highlighted cells in Figure 10-d, accounting for 58% of the cases that can be scheduled. All schedulers perform better than the Unmanaged (Figure 10-a), as they reduce the resource contentions.

(2) Compared with PARTIES and CLITE, OSML consumes fewer resources to support the same loads to meet the QoS targets. On average, for the 104 loads in Figure 8-a, OSML uses 34 cores and 16 LLC ways; by contrast, PARTIES and CLITE exhaust all the platform's 36 cores and 20 LLC ways. As illustrated in Figure 9-a, PARTIES partitions the LLC ways and cores equally for each LC service at the beginning; once it meets the QoS target (using 8 actions), it stops. Thus, PARTIES drops the opportunities to explore alternative better solutions (i.e., using fewer cores or cache ways to meet identical QoS targets). PARTIES allocates all cores and LLC ways finally. CLITE also uses all cores and cache ways shown in Figure 9-b. By contrast, OSML schedules according to applications' resource requirements instead of using all resources. Figure 9-c shows that using Model-A, OSML achieves each LC service's OAA (the optimal solution) after 5 actions. OSML detects and reclaims over-provided resources using Model-C. For example, the last action in Figure 9-c reclaims 3 cores and 2 LLC ways from Xapian. Finally, OSML saves 3 cores and 9 LLC ways. As OSML is designed for LC services that are executed for a long period, saving resources means saving budgets for cloud providers.

(3) Using ML models, OSML provides solutions for sharing some cores and LLC ways among LC services, therefore supporting higher loads. PARTIES and CLITE don't show resource sharing in the original design. Using Algo._4, OSML lists some potential resource sharing solutions, and then enables Model-B' to predict the QoS slowdown for each case. The sharing solution with a relatively lower QoS slowdown is selected. More details refer to Figure 7. Figure 9-d shows how OSML shares resources for the highlighted case B in Figure 10-d. OSML enables Model-C to add resources for Moses in Algo._2 and uses Algo._4 to share 2 CPU cores with Xapian. Finally, the QoS is met. By enabling resource sharing, OSML can support higher loads than PARTIES and CLITE, and can even be close to ORACLE in Figure 10-e. If not

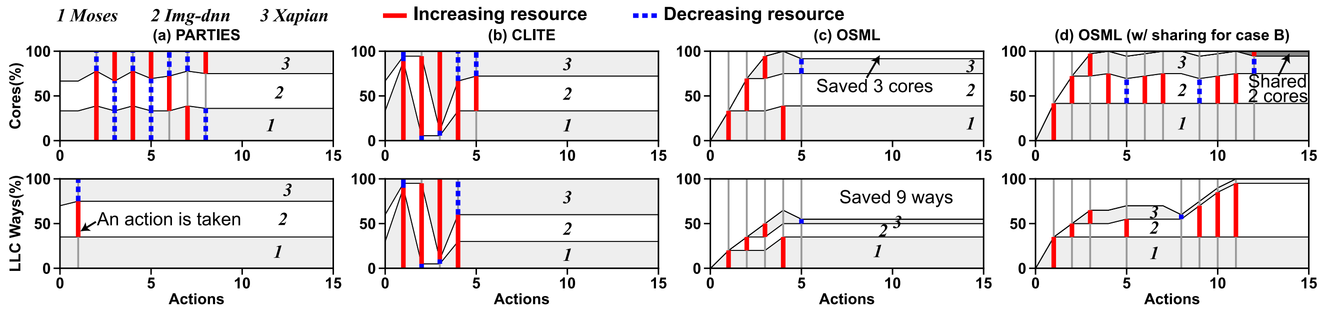


Figure 9: Resource usage comparisons for OSML, PARTIES, and CLITE.

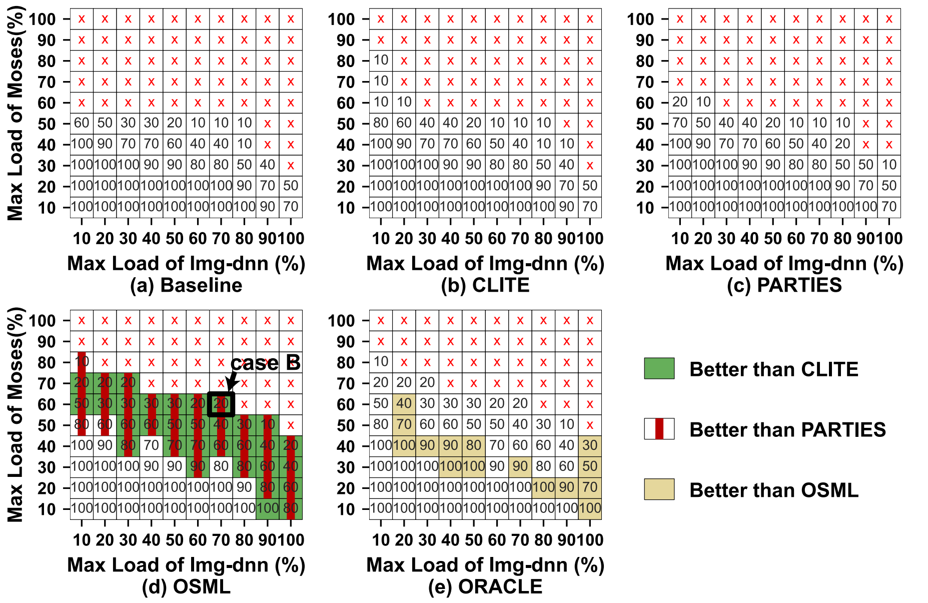


Figure 10: Co-location of Moses, Img-dnn and Xapian. The heatmap values are the percentage of the third LC service's (i.e., Xapian) achieved max load without QoS violations in these cases. The x- and y-axis denote the first and second app's fraction of their max loads (with QoS target), respectively. Cross means QoS target cannot be met. The related studies [10,46] use heat maps to show their effectiveness, so we also use heat maps for comparisons in this work.

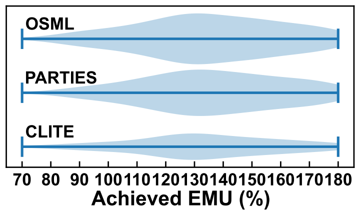


Figure 11: Throughput distribution.

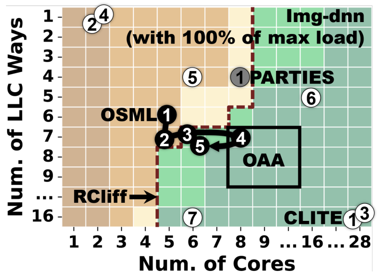


Figure 13: Highlighted the scheduling traces in scheduling space for all schedulers from time point 180 to 228 in Figure 12. Each circle denotes a specific scheduling policy conducted by a specific scheduler. The number in a circle denotes the sequence of these scheduling actions during the scheduling phase.

OSML, however, the "trial and error" approach has to try to share core/cache way in a fine-grain way among applications, and then observes the real-time latency for making a further decision, inevitably incurring higher scheduling overheads and bringing sharp QoS slowdown if falling off the RCliff.

(4) OSML promptly handles the resource under/over-provision and QoS violations using Model-C. Based on Model-A/B's results, Model-C shepherds and adjusts the allocations with several actions for each application in our experiments and converges more quickly than previous approaches. More experiments on dynamic, complicated cases can be found in Sec.6.3. *Can we only use Model-C or only use Model-A/B?* Enabling the three models is necessary for OSML. For the case in Figure 9-c, when OSML uses the three models collaboratively for scheduling, it takes 8.2s and 5 actions to achieve OAA for all applications. By contrast, if only enabling Model-C, it takes 18.5s and 13 actions. Because Model-A/B can at least provide an approximate OAA for scheduling, thus reducing the convergence time. Just using Model-A/B may lead to 4-core errors for an unseen LC service (Table 5). So, Model-C is needed to correct the errors. We cannot disable any models in OSML in practice.

(5) OSML performs well in various cases. Figure 11 shows the EMU distribution for converged loads among 302 loads (each has 3 apps with diverse RPS). EMU reflects the system throughput [10,32]. OSML can have scheduling results for 285 loads; PARTIES and CLITE can work for 260 and 148 loads, respectively. OSML's distribution in Figure 11 is wider than PARTIES and CLITE. This indicates that OSML works for more loads, including those with a high EMU (e.g., 130-180%). Even for the loads that OSML, PARTIES, and CLITE can all converge, OSML can converge faster (Figure 8).

6.3 Performance for Workload Churn

We evaluate how OSML behaves with dynamically changing loads. Each LC service's QoS is normalized to the solely running case. As illustrated in Figure 12-a, in the beginning, Moses with 60% of max load arrives; then Sphinx with 20% of max load and Img-dnn with 60% of max load arrive. We observe their response latency increases caused by the re-

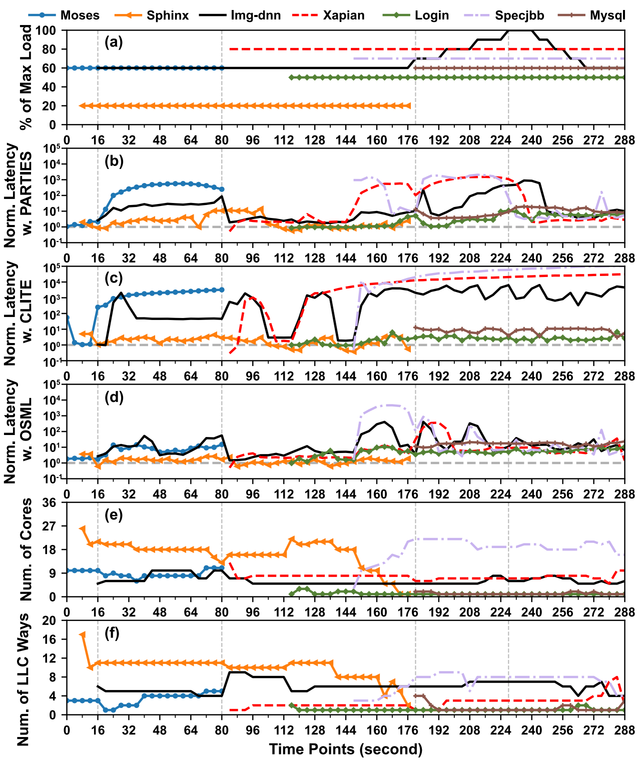


Figure 12: OSML's performance in reality with varying loads.

source contentions among them. In Figure 12-b, PARTIES aids the QoS violations step by step. During the scheduling, Moses always has high latency until it ends at 80 seconds. CLITE's scheduling relies on sampling and Bayesian optimizer. CLITE starts scheduling at time point 16, where all the three services arrive. At 32s, CLITE obtains the scheduling solution for these LC services after five sampling steps. But it doesn't satisfy these services' QoS targets. In Figure 12-c, Moses and Img-dnn still have high latency. By contrast, with Model-A's OAA predictions and Model-C's online scheduling, OSML quickly provides better scheduling solutions at time point 48 for all three services. During the identical scheduling phase (e.g., the time point 16 to 80), we can observe the lowest overall normalized latency in Figure 12-d. Moreover, Figure 12-e and f illustrate OSML's scheduling actions for achieving ideal solutions. In short, within a few scheduling actions (scheduling overheads) that schedule multiple resources, OSML quickly meets the QoS targets.

From 180 to 228, we increase the load for Img-dnn as illustrated in Figure 12-a. OSML meets Img-dnn's changing demands by using Model-C. PARTIES does not reflect quickly for this change, and it works for other services. Thus, as illustrated in Figure 12-b, the QoS violation is not aided until 244s, when Img-dnn's load decreases. For CLITE, it has to sample each time when the load changes. But during the sampling, a specific service might not have sufficient resources to handle the requests; thus, the requests are accumulated, leading to QoS fluctuations/violations during the scheduling (Figure 12-c). Figure 13 highlights the scheduling actions

Table 5: ML Models' performance on average. The units of errors are the number of cores/LLC ways. For Model B', the unit is the percentage of QoS slowdown.

ML	Outputs	Error		Errors for unseen LC services		Err on new platforms (TL)		MSE	Over-heads
		Core	LLC	Core	LLC	Core	LLC		
A	RCliff	0.589	0.288	1.266	0.198	2.142	0.542	0.0025	0.20s
	OAA	0.580	0.360	1.276	0.191	2.004	0.865		
A'	RCliff	1.072	0.815	3.403	1.835	0.772	0.411	0.0135	0.20s
	OAA	1.072	0.814	3.404	1.835	0.790	0.413		
B	B-Points	0.612	0.053	4.012	0.167	2.320	0.969	0.0012	0.18s
	B-Points,Core dominated	0.314	0.048	3.434	0.937	2.250	0.815		
	B-Points,Cache dominated	0.093	0.462	0.789	0.783	1.868	1.519		
B'	QoS reduction	7.87%		8.33%		11.28%		0.0035	0.19s
C	Scheduling actions	0.908	0.782	0.844	0.841	1.390	1.801	0.7051	0.20s

for Img-dnn from 180 to 228. During this phase, PARTIES does not add resources for Img-dnn; but it add more resources for Specjbb and Xapian as they are with higher latency. Img-dnn's response latency keeps increasing. CLITE samples several scheduling policies in the scheduling space, but does not converge and thus incurs QoS fluctuations. By contrast, OSML's Model-C can handle QoS violations of Img-dnn even the load increases. Moreover, as mentioned before, OSML saves resources and thus it can serve more workloads. For example, as shown in Figure 12, Mysql (an **unseen** workload in training) comes at time point 180; OSML allocates the saved cores to it without sharing or depriving other LC services of resources.

6.4 New/Unseen Apps and New Platforms

Generalization. Based on our data set, the ML models can be well trained. It will be at most 4-core error for unseen applications (Table 5). We evaluate OSML using unseen applications that are not used in training, i.e., Silo [62], Shore[62], Mysql [70], Redis [77], and Node.js [78]. They exhibit diverse computing/memory patterns. We evaluate the convergence time of OSML with 3 groups of workloads. Each group has 15 workloads, and each workload has 3 LC services. The workloads in Group 1 have 1 unseen application. The workloads in Group 2 and 3 have 2 and all unseen applications, respectively. OSML takes 24.6s, 29.3s, and 31.0s to converge for the 3 groups of workloads, on average, respectively. OSML performs well, even for unseen cases, showing good generalization performance. PARTIES uses 34.9s, 43.6s, and 42.8s; CLITE uses 58.5s, 60.6s, and 46.5s for these applications. Note that PARTIES and CLITE's scheduling don't need previous knowledge for applications, so their performance doesn't depend on whether the application is seen/unseen.

For new platforms, we use fine-tuning in transfer learning (TL). We freeze the first hidden layer of the MLPs; we retrain the last two-hidden layers and the output layer using the traces collected on two new platforms (w/ CPU Xeon Gold 6240M and E5-2630 v4, respectively). For each LC service, based on our data set, collecting new traces on a new platform for several hours will be sufficient (covering the more allocation cases, the better). The model updating can be accomplished in hours. The time consumption will be shorter if using multiple machines in parallel. Table 5 shows the average values of ML models' quality. The new models' prediction errors are slightly higher than the previous models on the original platforms, but OSML still handles them well. By contrast, if we use a table lookup approach instead, we have to use additional memory to store the data tuples, e.g., 60GB will be wasted for the current data set to replace Model-A. More importantly, this approach is difficult to generalize for new/unseen applications or platforms, as their traces and the corresponding OAAs don't exist in the current data set.

Overheads. OSML takes 0.2s of a single core for each time during the interval setting (e.g., 1 second by default in Sec.5.2). 0.01s for ML model and 0.19s for online monitoring. As our models are light-weighted (OSML uses only one core), running them on CPU and GPU has a similar overhead. If models are on GPU, it takes an extra 0.03s for receiving results from GPU. OSML doesn't monopolize GPU. Generally, the overhead doesn't hinder the overall performance. In the cloud, applications' behaviors may change every second due to the diversity of user demands. Thus, OSML plays a critical role during the entire run time. **For building models**, using our current data set on platform in Table 2, it takes 3.3 mins, 5 mins, and 8.3 hours to train Model-A, B, and C for one epoch (all training samples are used to train once), respectively. We train models for ten epochs. Training can be accelerated using the popular Multi-GPU training technology. Doing so is practical in datacenters, and training time will not impede practice.

7 Related Work and Our Novelty

ML for System Optimizations. The work in [55] employs DNN to optimize the buffer size for the database systems. The efforts in [22,56,34] leverage ML to optimize computer architecture or resource management in the network to meet various workloads. The studies in [9,39] use ML to manage on-chip hardware resources. CALOREE [41] can learn key control parameters to meet latency requirements with minimal energy in complex environments. The studies in [26,31,58,59] optimize the OS components with learned rules or propose insights on designing new learned OS. *In OSML, we design an intelligent multi-model collaborative learning approach, providing better co-location solutions to meet QoS targets for LC services faster than the latest work stably.*

ML for Scheduling. Decima [35] designs cluster-level data processing job scheduling using RL. Resource Central [12] builds a system that contains the historical resource utilization information of the workloads used in Azure and employs ML to predict resource management for VMs. The study in [40] uses RL to predict which subsets of operations in a TensorFlow graph should run on the available devices. Paragon [14] classifies and learns workload interference. Quasar [15] determines jobs' resource preferences on clusters. Sinan [74] uses ML models to determine the performance dependencies between microservices in clusters. They are cluster schedulers [14,15,74]. By contrast, OSML deeply studies scheduling in co-location cases. Selecta [72] predicts near-optimal configurations of computing and storage resources for analytics workloads based on profiling data. CLITE [46] uses Bayesian optimization for scheduling on-node resources. The work in [48] applies ML to predict the end-to-end tail latency of LC service workflows. Twig [63] uses RL to characterize tail latency for energy-efficient task management. CuttleSys [76] leverages data mining to identify suitable core and cache configurations for co-scheduled applications. *For complicated co-location cases, OSML can avoid RCliffs and quickly achieve the ideal allocations (OAA) for multiple interactive resources simultaneously for LC services. Moreover, OSML performs well in generalization.*

Resource Partitioning. PARTIES [10] partitions cache, main memory, I/O, network, disk bandwidth, etc., to provide QoS for co-located services. The studies in [17,28,57,71] design some new LLC partitioning/sharing policies. The efforts in [23,27,44,45,73] show that considering cooperative partitioning on LLC, memory banks and channels outperforms one-level memory partitioning. However, the cooperative partitioning policies need to be carefully designed [29,30,37], and [16,32] show the heuristic resource scheduling approach could be ineffective in many QoS-constrained cases. [7,11] study the "performance cliff" on cache for SPECCPU 2006 applications and Memcached. Caladan [75] doesn't involve cache optimizations, and core/cache cliffs cannot be avoided, causing QoS fluctuations in some cases. *By contrast, OSML is the first work that profoundly explores cache cliff and core cliff simultaneously (i.e., RCliff) for many widely used LC services in co-location cases. OSML is a representative work using ML to guide the multiple resources partitioning in co-location cases; OSML is cost-effective in new cloud environments.*

8 Conclusion

We present OSML, an ML-based resource scheduler for co-located LC services. We learn that straightforwardly using a simple ML model might not handle all of the processes during the scheduling. Therefore, using multiple ML models cooperatively in a pipe-lined way can be an ideal approach. More importantly, we advocate the new solution, i.e., leveraging ML to enhance resource scheduling, could have an immense potential for OS design. In a world where co-location and sharing are a fundamental reality, our solution should grow in importance and benefits our community.

Acknowledgement

We thank the reviewers and our shepherd Young-ri Choi for the invaluable comments. This work is supported by the Key-Area R&D Program of Guangdong (No.2021B0101310002), NSFC (No.62072432). L. Liu thanks Hecheng's arriving. X. Dou and Y. Chen are students in Sys-Inventor Lab led by L. Liu.

References

[1] "How 1s could cost amazon $1.6 billion in sales." https://www.fastcompany.com/1825005/how-one-second-could-cost-amazon-16-billion-sales

[2] "Microservices workshop: Why, what, and how to get there," http://www.slideshare.net/adriancockcroft/microservices-workshop-craft-conference

[3] "State of the Cloud Report," http://www.righscale.com/lp/state-of-the-cloud. Accessed: 2019-01-28

[4] "Improving real-time performance by utilizing cache allocation technology," https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/cache-allocation-technology-white-paper.pdf, Intel Corporation, April, 2015

[5] "Intel 64 and IA-32 Architectures Software Developer's Manual," https://software.intel.com/en-us/articles/intel-sdm, Intel Corporation, October, 2016

[6] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng, "TensorFlow: A System for Large-Scale Machine Learning," in OSDI, 2016

[7] Nathan Beckmann, Daniel Sanchez, "Talus: A Simple Way to Remove Cliffs in Cache Performance," in HPCA, 2015

[8] Daniel S. Berger, Benjamin Berg, Timothy Zhu, Siddhartha Sen, Mor Harchol-Balter, "RobinHood: Tail Latency Aware Caching – Dynamic Reallocation from Cache-Rich to Cache-Poor," in OSDI, 2018

[9] Ramazan Bitirgen, Engin Ipek, Jose F. Martinez, "Coordinated Management of Multiple Interacting Resources in Chip Multiprocessors: A Machine Learning Approach," in Micro, 2008

[10] Shuang Chen, Christina Delimitrou, José F. Martínez, "PARTIES: QoS-Aware Resource Partitioning for Multiple Interactive Services," in ASPLOS, 2019

[11] Asaf Cidon, Assaf Eisenman, Mohammad Alizadeh, Sachin Katti, "Cliffhanger: Scaling Performance Cliffs in Web Memory Caches," in NSDI, 2016

[12] Eli Cortez, Anand Bonde, Alexandre Muzio, Mark Russinovich, Marcus Fontoura, Ricardo Bianchini, "Resource Central: Understanding and Predicting Worloads for Improved Resource Management in Large Cloud Platforms," in SOSP, 2017

[13] Jeff Dean, David A. Patterson, Cliff Young, "A New Golden Age in Computer Architecture: Empowering the Machine-Learning Revolution," in IEEE Micro, 2018

[14] Christina Delimitrou, Christos Kozyrakis, "QoS-Aware Scheduling in Heterogenous Datacenters with Paragon," in ACM TOCS, 2013

[15] Christina Delimitrou, Christos Kozyrakis, "Quasar: Resource-Efficient and QoS-Aware Cluster Management," in ASPLOS, 2014

[16] Yi Ding, Nikita Mishra, Henry Hoffmann, "Generative and Multi-phase Learning for Computer Systems Optimization," in ISCA, 2019

[17] Nosayba El-Sayed, Anurag Mukkara, Po-An Tsai, Harshad Kasture, Xiaosong Ma, Daniel Sanchez, "KPart: A hybrid Cache Partitioning-Sharing Technique for Commodity Multicores," in HPCA, 2018

[18] Yu Gan and Christina Delimitrou, "The Architectural Implications of Cloud Microservices," in IEEE Computer Architecture Letters, 2018

[19] Yu Gan, Yanqi Zhang, Kelvin Hu, Dailun Cheng, Yuan He, Meghna Pancholi, Christina Delimitrou, "Leveraging Deep Learning to Improve Performance Predictability in Cloud Microservices with Seer," in ACM SIGOPS Operating Systems Review, 2019

[20] Mark D. Hill, Michael R. Marty, "Amdahl's Law in the Multicore Era," in IEEE Computers, 2008

[21] Kurt Hornik, "Approximation Capabilities of Multilayer Feedforward Networks," in Neural Networks, 1991

[22] Engin Ipek, Onur Mutlu, José F. Martínez, Rich Caruana, "Self-Optimizing Memory Controllers: A Reinforcement Learning Approach," in ISCA, 2008

[23] Min Kyu Jeong, Doe Hyun Yoon, Dam Sunwoo, Michael Sullivan, Ikhwan Lee, Mattan Erez,"Balancing DRAM Locality and Parallelism in Shared Memory CMP Systems,"in HPCA, 2012

[24] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, Doe Hyun Yoon, "In-Datacenter Performance Analysis of a Tensor Processing Unit," in ISCA, 2017

[25] Alex Krizhevsky, Ilya Sutskever, Geoffrey Hinton, "ImageNet Classification with Deep Convolutional Neural Networks," in Advances in neural information processing systems, 2012

[26] Yanjing Li, Onur Mutlu, Subhasish Mitra, "Operating System Scheduling for Efficient Online Self-Test in Robust Systems," in ICCAD, 2009

[27] Lei Liu, et al, "A Software Memory Partition Approach for Eliminating Bank-level Interference in Multicore Systems," in PACT, 2012

[28] Jiang Lin, Qingda Lu, Xiaoning Ding, Zhao Zhang, Xiaodong Zhang, P. Sadayappan, "Gaining insights into mlticore cache partitioning: bridging the gap between simulation and real systems," in HPCA, 2008

[29] Fang Liu, Yan Solihin, "Studying the Impact of Hardware Prefetching and Bandwidth Partitioning in Chip-Multiprocessors," in Sigmetrics, 2011

[30] Seung-Hwan Lim, Jae-Seok Huh, Yougjae Kim, Galen M. Shipman, Chita R. Das, "D-Factor: A Quantitative Model of Application Slow-Down in Multi-Resource Shared Systems," in Sigmetrics, 2012

[31] Lei Liu, et al, "Rethinking Memory Management in Modern Operating System: Horizontal, Vertical or Random?" in IEEE Trans. on Computers, 2016

[32] David Lo, Liqun Cheng, Rama Govindaraju, Parthasarathy Ranganathan, Christos Kozyrakis, "Heracles: Improving Resource Efficiency at Scale," in ISCA, 2015

[33] Lei Liu, et al, "Hierarchical Hybrid Memory Management in OS for Tiered Memory Systems," in IEEE Trans. on Parallel and Distributed Systems, 2019

[34] Hongzi Mao, Mohammad Alizadeh, Ishai Menache, Srikanth Kandula, "Resource Management with Deep Reinforcement Learning," in HotNet-XV, 2016

[35] Hongzi Mao, Malte Schwarzkopf, Shaileshh B. Venkatakrishnan, Zili Memg, Mohammad Alizadeh, "Learning Scheduling Algorithms for Data Processing Clusters," in SIGCOMM, 2019

[36] Yashwant Marathe, Nagendra Gulur, Jee Ho Ryoo, Shuang Song, and Lizy K. John, "CSALT: Context Switch Aware Large TLB," in Micro, 2017

[37] Jason Mars, Lingjia Tang, Robert Hundt, Kevin Skadron, Mary Lou Soffa, "Bubble-Up: Increasing Utilization in Modern Warehouse Scale Computers via Sensible Co-locations," in Micro, 2011

[38] Jason Mars, Lingjia Tang, Mary Lou Soffa, "Directly Characterizing Cross Core Interference Through Contention Synthesis," in HiPEAC, 2011

[39] Jose F. Martinez, Egin Ipek, "Dynamic multicore resource management: A machine learning approach," in IEEE Micro 29 (5):8-17 (2009)

[40] Azalia Mirhoseini, Hieu Pham, Quoc V. Le, Benoit Steiner, Rasmus Larsen, Yuefeng Zhou, Naveen Kumar, Mohammad Norouzi, Samy Bengio, Jeff Dean, "Learning Device Placement in Tensorflow Computations," in Arxiv 1706.04972

[41] Nikita Mishra, Connor Imes, John D. Lafferty, Henry Hoffmann,

"CALOREE: Learning Control for Predictable Latency and Low Energy," in ASPLOS, 2018

[42] Nikita Mishra, Harper Zhang, John Lafferty, Henry Hoffmann, "A probabilistic Graphical Model-based Approach for Minimizing Energy Under Performance Constraints," in ASPLOS, 2015

[43] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, Demis Hassabis, "Human-level control through deep reinforcement learning," in Nature 518 (7540): 529-533, 2015

[44] Sai Prashanth Muralidhara, Lavanya Subramanian, Onur Mutlu, Mahmut Kandemir, Thomas Moscibroda, "Reducing Memory Interference in Multicore Systems via Application-Aware Memory Channel Partitioning," in Micro, 2011

[45] Jinsu Park, Seongbeom Park, Woongki Baek, "CoPart: Coordinated Partitioning of Last-Level Cache and Memory Bandwidth for Fairness-Aware Workload Consolidation on Commodity Servers," in EuroSys, 2019

[46] Tirthak Patel, Devesh Tiwari, "CLITE: Efficient and QoS-Aware Co-Location of Multiple Latency-Critical Jobs for Warehouse Scale Computers," in HPCA, 2020

[47] Henry Qin, Qian Li, Jacqueline Speiser, Peter Kraft, and John Ousterhout, "Arachne: Core-Aware Thread Management," in OSDI, 2018

[48] Joy Rahman, Palden Lama, "Predicting the End-to-End Tail Latency of Containerized Microservices in the Cloud," in IC2E, 2019

[49] Yizhou Shan, Yutong Huang, Yilun Chen, Yiying Zhang, "LegoOS: A Disseminated, Distributed OS for Hardware Resource Disaggregation," in OSDI, 2018

[50] Prateek Sharma, Ahmed Ali-Eldin, Prashant Shenoy, "Resource Deflation: A New Approach For Transient Resource Reclamation," in EuroSys, 2019

[51] David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, Demis Hassabis, "Mastering the game of Go with deep neural networks and tree search," in Nature, 529 (7587), 2016

[52] Akshitha Sriraman, Abhishek Dhanotia, Thomas F. Wenisch, "SoftSKU: Optimizing Server Architectures for Microservice Diversity @Scale," in ISCA, 2019

[53] Akshitha Sriraman, Thomas F. Wenisch, "µTune: Auto-Tuned Threading for OLDI Microservices," in OSDI, 2018

[54] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scoott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, Andrew Rabinovich, "Going deeper with convolutions," in CVPR, 2015

[55] Jian Tan, Tieying Zhang, Feifei Li, Jie Chen, Qixing Zheng, Ping Zhang, Honglin Qiao, Yue Shi, Wei Cao, Rui Zhang, "iBTune: Individualized Buffer Tuning for Large-scale Cloud Databases," in VLDB, 2019

[56] Stephen J. Tarsa, Rangeen Basu Roy Chowdhury, Julien Sebot, Gautham Chinya, Jayesh Gaur, Karthik Sankaranarayanan, Chit-Kwan Lin, Robert Chappell, Ronak Singhal, Hong Wang, "Post-Silicon CPU Adaptations Made Practical Using Machine Learning," in ISCA, 2019

[57] Xiaodong Wang, Shuang Chen, Jeff Setter, Jose F. Martínez, "SWAP: Effective Fine-Grain Management of Shared Last-Level Caches with Minimum Hardware Support," in HPCA, 2017

[58] Zi Yan, Daniel Lustig, David Nellans, and Abhishek Bhattacharjee, "Nimble Page Management for Tiered Memory Systems," in ASPLOS, 2019

[59] Yiying Zhang, Yutong Huang, "Learned Operating Systems," in ACM SIGOPS Operating Systems Review, 2019

[60] Zhijia Zhao, Bo Wu, Xipeng Shen, "Challenging the "Embarrassingly Sequential": Parallelizing Finite State Machine-based Computations through Principled Speculation," in ASPLOS, 2014

[61] Xiaoya Xiang, Chen Ding, Hao Luo, Bin Bao, "HOTL: A higher order theory of locality," in ASPLOS, 2013

[62] Harshad Kasture, Daniel Sanchez, "Tailbench: a benchmark suite and evaluation methodology for latency-critical applications," in IISWC, 2016

[63] Rajiv Nishtala, Vinicius Petrucci, Paul Carpenter, Magnus Sjalander, "Twig: Multi-Agent Task Management for Colocated Latency-Critical Cloud Services," in HPCA, 2020

[64] MongoDB official website. http://www.mongodb.com

[65] Memcached official website. https://memcached.org

[66] NGINX official website. http://nginx.org

[67] https://en.wikipedia.org/wiki/Universal_approximation_theorem

[68] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, Kelvin Hu, Meghna Pancholi, Yuan He, Brett Clancy, Chris Colen, Fukang Wen, Catherine Leung, Siyuan Wang, Leon Zaruvinsky, Mateo Espinosa, Rick Lin, Zhongling Liu, Jake Padilla, Christina Delimitrou, "An Open-Source Benchmark Suite for Microservices and Their Hardware-Software Implications for Cloud & Edge Systems," in ASPLOS, 2019

[69] Kalyanmoy Deb, Shivam Gupta, "Understanding Knee Points in Bicriteria Problems and Their Implications as Preferred Solution Principles," in Engineering Optimization, 43 (11), 2011

[70] www.mysql.com

[71] Harshad Kasture, Daniel Sanchez, "Ubik: Efficient Cache Sharing with Strict QoS for Latency-Critical Workloads," in ASPLOS, 2014

[72] Ana Klimovic, Heiner Litz, Christos Kozyrakis, "Selecta: Learning Heterogeneous Cloud Storage Configuration for Data Analytics," in USENIX ATC, 2018

[73] Harshad Kasture, Xu Ji, Nosayba El-Sayed, Xiaosong Ma, Daniel Sanchez, "Improving Datacenter Efficiency Through Partitioning-Aware Scheduling," in PACT, 2017

[74] Yanqi Zhang, Weizhe Hua, Zhuangzhuang Zhou, G. Edward Suh, Christina Delimitrou, "Sinan: ML-Based and QoS-Aware Resource Management for Cloud Microservices," in ASPLOS, 2021

[75] Joshua Fried, Zhenyuan Ruan, Amy Ousterhout, Adam Belay, "Caladan: Mitigating Interference at Microsecond Timescales," in OSDI, 2020

[76] Neeraj Kulkarni, Gonzalo Gonzalez-Pumariega, Amulya Khurana, Christine A Shoemaker, Christina Delimitrou, David H Albonesi, "Cuttlesys: Data-driven Resource Management for Interactive Services on Re configurable Multicores," in Micro, 2020

[77] Redis official website. https://redis.io/

[78] Node.js official website. https://nodejs.org/en/

[79] Lei Liu, "QoS-Aware Resources Scheduling for Microservices: A Multi-Model Collaborative Learning-based Approach," in arXiv:1911.13208v2, 2019

[80] Lei Liu, et al, "Going Vertical in Memory Management: Handling Multiplicity by Multi-policy," in ISCA, 2014