QoS-Aware Resource Scheduling for Microservices: A Multi-Model Collaborative Learning-based Approach

Abstract

Microservices have been dominating in the modern cloud environment. To improve cost efficiency, multiple microservices are normally co-located on a server. Thus, the run-time resource scheduling becomes the pivot for QoS control. However, the scheduling exploration space enlarges rapidly with the increasing server resources (cores, cache, bandwidth, etc.) and the diversity of microservices. Consequently, the existing schedulers might not meet the rapid changes in service demands. Besides, we observe that there exist "resource cliffs" in the scheduling space; they not only impact the exploration efficiency, making it difficult to converge to the optimal scheduling solution, but also result in severe QoS fluctuation.

To overcome these problems, we propose a novel machine learning-based scheduling mechanism called OSML. It uses resources and run-time states as the inputs and employs two MLP models and a reinforcement learning model (DQN) to perform scheduling space exploration. Thus, OSML can reach an optimal solution much faster than traditional approaches. More importantly, it can automatically detect resource cliffs and avoid them during exploration. To verify the effectiveness of OSML and obtain a well-generalized model, we collect a data set containing over 2-billion samples from 11 typical microservices running on off-the-shelf servers over 11 months. Experimental results show OSML supports higher loads (up to 60%) and meets QoS targets with 3.2~5.5× scheduling speed than state-of-the-art studies.

1 Introduction

As cloud computing enters a new era, cloud services are shifting from monolithic designs to microservices, which exist as numbers of loosely coupled functions and work together to serve the end-users [18,19,52,53]. Microservices have been rapidly growing since 2018. Most cloud providers, including Amazon, Alibaba, Facebook, Google, and LinkedIn, have deployed microservices for improving the scalability, functionality, and reliability of their cloud systems [2,3,18,52]. QoS is a critical metric for microservices. 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 give up the transactions, translating to $1.6 billion loss annually [1].

Microservices make QoS-driven resource scheduling even more challenging. The cost efficiency policy drives providers to co-locate as many applications as possible on a server. These co-located microservices, however, exhibit different behaviors across multiple interactive resources, such as CPU cores, cache, bandwidth, main memory banks. These behaviors also can be drastically different from demand to demand and change within the microsecond-level. Moreover, with the increasing number of cores, more threads share and contend for the shared LLC (last-level cache) and memory bandwidth. Notably, these shared resources interact with each other. All these issues enlarge the scheduling exploration space, making scheduling more complicated and time-consuming. Figure 1 shows examples of such complex exploration space. Briefly, unprecedented challenges are posed for resource scheduling mechanisms [8,10,31,47,52].

Some prior studies [17,31,33,45] design clustering approaches for allocating LLC or LLC together with main memory bandwidth among single-threaded applications. Yet, they are not suitable for microservices, as microservices often have many concurrent threads and strict QoS constraints. Moreover, these studies rely on accurate performance models, which might bring high scheduling overheads during run-time and incur non-negligible porting efforts; and, designing a precise model is still a challenging job [61]. Alternatively, the state-of-the-art 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. These approaches cannot handle users' diverse demands in a timely fashion on modern devices with increasingly parallel computing units and complex memory hierarchies. For instance, scheduling 5 co-located microservices to meet certain QoS constraints, [10,46] can take more than 20 seconds on average, which delays the specific critical tasks and causing significant financial losses as a result [1]. To our knowledge, no prior studies can work ideally for microservices. Thus, our community has been expecting new directions on developing resource-scheduling mechanisms [16,30,31,45].

To this end, we design OSML, a novel machine learning (ML) based resource scheduler for microservices on large-scale servers. OSML abstracts resources and microservices' run-time states as the inputs and employs multiple collaborative ML models to explore the scheduling exploration space. Over the past decade, 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]. To make better use of ML to enhance the scheduler, we find 4 problems needed to be solved. (1) Lack of a thorough understanding of the behaviors that microservices typically exhibit, preventing from exactly customizing the ML model functions. (2) Lack of accurate abstractions of ML models for low-level fine-grain resource scheduling, making the design of the overall scheduling mechanism difficult. (3) Scarce training data, leading to inaccurate inference results from ML models. (4) Lack of clarity in the design of software stack hierarchy when ML is involved for scheduling, therefore it is hard to design the interfaces and interactive control/data flow with existing OS and hardware systems. In OSML, we conquer these problems and make the following contributions.

(1) Investigation in RCliff for Multiple Resources. For microservices, we are the first conducting a study on resource cliff (RCliff – reducing only a slight resource leads to a significant QoS slowdown) for both Core and Cache. We further show that RCliff commonly exists in many widely used microservices. More importantly, we are the first to show the

Table 1. Microservice details. The max load - max RPS - is with the 95th percentile tail latency QoS target [10,18,52].

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

prior schedulers can hardly avoid RCliff (described in Sec. 3.3), and use ML technologies to avoid RCliff.

(2) Using Collaborative ML Models for Optimal Scheduling. OSML is the first ML-based scheduler that simultaneously schedules multiple complicated interactive resources to achieve the optimal allocations for co-located microservices' QoS targets. OSML employs MLP models to avoid RCliff and quickly deliver optimal solutions; to provide accurate scheduling solutions, OSML leverages an enhanced DQN to shepherd the allocation and perform reallocation in the QoS violation and resource over-provision cases. OSML avoids the QoS slowdown often incurred by the RCliff in prior schedulers.

(3) Collecting 11 Months Traces for Popular Microservices. To train ML models, we collect the performance traces for widely deployed microservices (in Table 1), covering 77,509,740 cases that include more than 2-billion samples mainly in a productive environment for over 11 months. More importantly, we make our training data set publicly available (Link), and we believe our efforts can benefit our community.

(4) Real Implementation and Detailed Comparisons. We implement OSML based on Linux kernel 4.19. OSML is designed as a co-worker of OS kernel located between the OS and the user layer. Further, we conduct a comprehensive comparison between OSML and the state-of-the-art work [10,46]. We believe our explorations will inspire future designers.

In practice, OSML captures the microservices' online behaviors and forwards them to the ML models running on CPU or GPU, and schedules resources according to the feedbacks. Compared with [10,46], OSML meets the QoS targets with merely 1/3~1/5 scheduling actions and supports 10~60% of higher loads. OSML and its ML models are with low run-time overhead. We make OSML open source as planned.

2 Background

The cloud environment has a growing trend towards the microservice implementation model [3,18,52]. Modern cloud applications comprise numerous distributed microservices such as key-value storing, database serving, access-control management and business applications serving [18,19]. Table 1 includes several typical microservices, which are widely used and form a significant fraction of cloud applications [18]. These microservices are with different features and resource demands. We study them in this article.

In terms of the datacenter servers, nearly a decade before, a datacenter server could have an Intel i7-series CPU with 4/8 cores/threads, 8 MB LLC, and supports 12.8 GB/s memory bandwidth per channel. 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.

Table 2. Our platform specification vs. a server used 10 yrs. before.

Conf. / Servers	Our Platform	Server (10 Years Ago)
CPU Model	Intel ® Xeon ® CPU 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

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 large number of resources are wasted every day and night.

On the other hand, the larger resource scheduling exploration space, which consists of more combinations of diverse resources, making the existing schedulers cannot quickly achieve the optimal solution. Additionally, as microservices can have dozens of concurrent threads [10,46], when several of them are running on a specific server, they have to share and contend resources across multiple resources layers – cores, LLC, memory bandwidth, and banks (e.g., DRAM banks). Previous studies show it may incur severe performance degradation and unpredictable QoS violation, and they propose the scheduling mechanisms at hardware architecture, OS, and user-level [9,10,23,31,37,38,44,45,50]. *Yet, do the existing schedulers serve microservices well?*

3 Investigation into Resource Scheduling for Microservices – Leveraging ML

To answer the above question, we study microservices that are widely deployed as the key components in cloud environments. The details of them are illustrated in Table 1.

3.1 Understanding the Microservices – Resource Cliff

We study how sensitive these microservices behave to the critical resources, e.g., the number of cores and LLC capacity, on a modern commercial platform ("our platform" in Table 2). We show the results of 6 typical microservices.

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 relatively low in the cases where computing and LLC resources are ample (i.e., below 10ms for Moses in the area with green color). The overall trends are observed from other microservices.

However, we observe the Cliff phenomenon for these microservices. In Figure 1-a, Moses exhibits this phenomenon. For instance, in the cases where 6 cores are allocated to Moses,

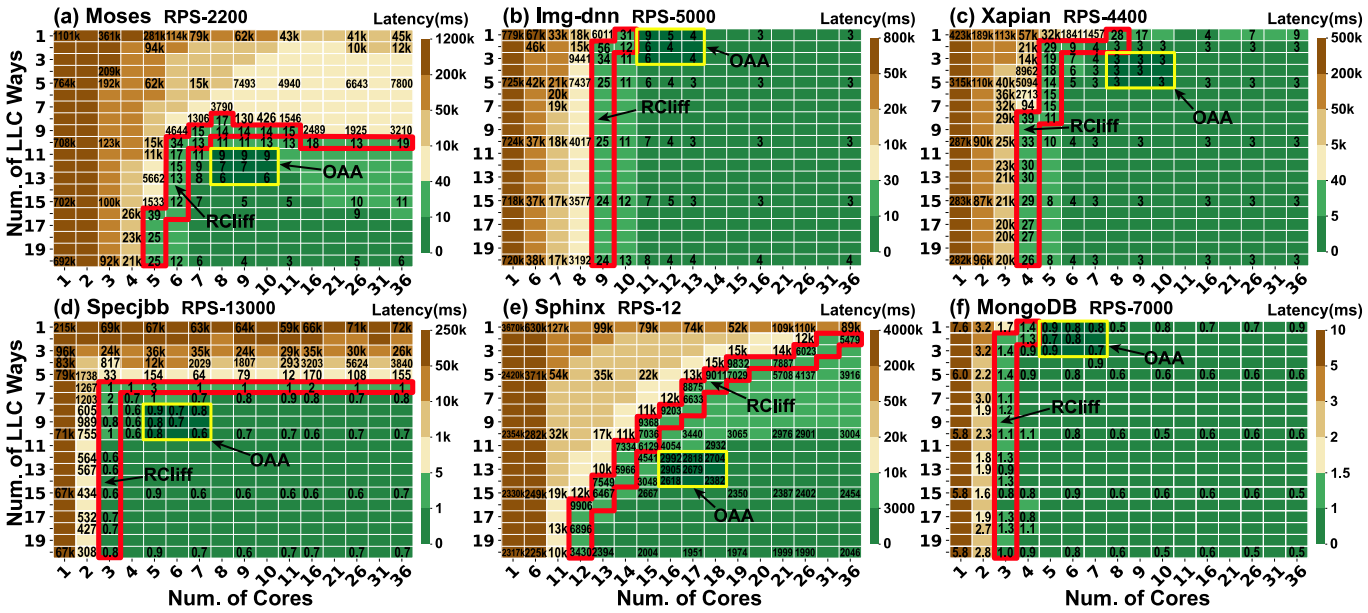

Figure 1. The resource scheduling exploration space for core and LLC ways. These figures show the sensitivity to resource allocation under different policies. Each col./row represents a specific number of LLC ways/cores allocated to an application. Each cell denotes the microservice's response latency under the given number of cores and LLC ways. The Redline highlights the RCliff. The green color cells show allocation policies that bring better performance (low response latency). OAA (Opt Allocation Area) is also illustrated for each microservice. We test all of the microservices in Table 1. Due to the space problem, we only list several of them. As we don't want the figures to look too dense, we only have some typical data on them.

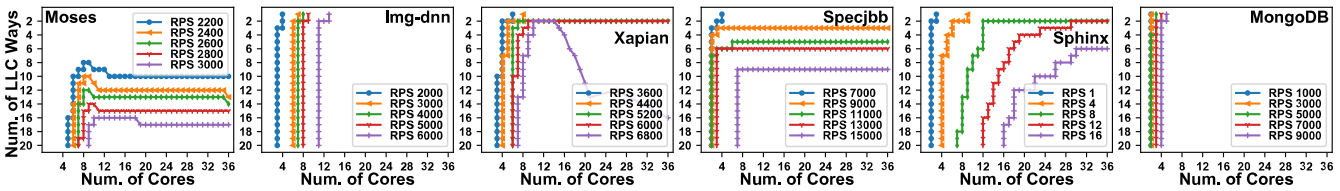

Figure 2. Sensitivity to RCliff under different RPS. We can see the RCliff is always existing, though the RPS varies. On average, RCliff exhibits 8.80% variation (Moses is with maximum variation 15.0% and MongoDB is with minimum 2.77%).

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. For example, in the cases where 13 ways are allocated, the response latency is sharply increased from 13ms to 5662ms when we allocate 5 cores instead of 6 cores. We denote this phenomenon as Resource Cliff (**RCliff**). **On the RCliff, as illustrated in Figure 1-a, there would be a sharp performance slowdown if only one core or one LLC way is deprived of**. From another point of view, 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.

Compared with Moses, Img-dnn mainly exhibits the RCliff phenomenon for cores. In Figure 1-b, the response latency can be reduced from 15,000ms to 56ms if 9 cores are allocated instead of 8 cores. Meanwhile, for a specific number of cores, allocating more LLC ways has much less impact than cores. Additionally, though some of the microservices' RCliffs do not exhibit significant performance change, as Moses, Xapian and Sphinx do (above 100×), we can also observe several times variation around RCliff, e.g., MongoDB in Figure 1-f.

Is the RCliff always existing? We test these microservices across different RPS in Table 1, and find the RCliff always exists, though the RCliff may change according to different RPS. Figure 2 illustrates the details. For Moses, with the increasing of user demands, i.e., RPS ranges from 2.2K to 3K, Moses' RCliff shifts accordingly; and, Img-dnn's RCliff line shifts from 3-core to 11-core cases, when the RPS ranges from 2K to 6K. Xapian, Specjbb and Sphinx also show the trend.

RCliff should be considered when designing a scheduler. 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 as a result, i.e., even a little bit resource reduction may incur severe slowdown. Notably, in Figure 1, we highlight each microservice's **Optimal Allocation Area (OAA) in the scheduling exploration space, which indicates the ideal number of allocated cores and LLC ways that can bring optimal performance.** OAA is the goal that schedulers should achieve; but RCliff is something that needs to avoid.

3.2 Is OAA Sensitive to the Number of Threads?

In practice, an arbitrary number of threads might be started for a microservice, as people may intuitively assume that more threads can bring a higher performance. For instance, people may start 20 threads when Moses is launched and regardless of only 8 cores are available. Here, we come up to the question: *is the OAA sensitive to the number of threads, i.e., if someone starts more threads, will the OAA change?*

To further study this problem, for a specific microservice, we start a different number of threads and map them across a different number of cores (the num. of threads can be larger

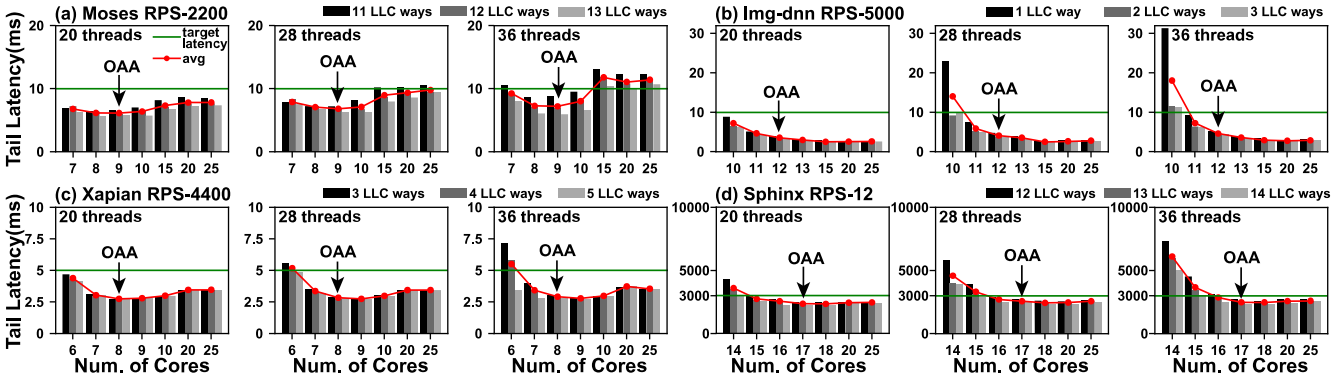

Figure 3. OAA exists regardless of the num. of concurrent threads. Due to the space problem, we only show some of the cases.

than the num. of cores). Through the experiments, we observe two things. (1) More threads do not necessarily bring more benefits. Take Moses as an example, 8 threads mapped to 8 cores can be the ideal solution with low response latency (in the OAA); however, when more threads are started (e.g., 20/28/36), the overall response latency can be higher (as illustrated in Figure 3). The underlying reason lies in more memory contentions at memory hierarchy and more context switch overheads, thus leading to a higher response latency [20,36]. (2) The OAA is not sensitive to the number of concurrent threads. Illustrated in Figure 3, although the overall latency becomes higher with the increasing number of threads, the OAA doesn't change obviously. For Moses in Figure 3, even if 20/28/36 threads are mapped to 10~25 cores, around 8/9-core cases always perform ideally. Other applications also show a similar phenomenon, though the OAA differs from each other.

As LLC is always a scare resource, the scheduler should allocate it carefully. Figure 3 shows the LLC allocations in microservices' OAAs. In practice, if the QoS for a specific microservice is satisfied, e.g., below 10ms latency for Moses, LLC ways should be allocated as less as possible (e.g., assigning 11 ways is more cost-effective than allocating 12 or 13 ways), saving LLC space for other applications. We also try to allocate fewer cores to meet the QoS target for saving computing resources. Here, we conclude that the OAA always exists, and it is not sensitive to the number of threads. And, we meet another question: *how do the existing schedulers perform in front of OAA and RCliff?*

3.3 Is It Necessary to Design a New ML-based Scheduler?

Through our study, we find the existing schedulers often have three shortcomings when meet OAA and RCliff. **(1) Entangling with RCliff.** As many schedulers often employ heuristic 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), these approaches could incur an unpredictable latency spiking. For example, if the current resource allocation for a microservice is in the base of RCliff (i.e., the base area is with yellow color in Figure 1-a), the scheduler has to try to achieve OAA. However, as the scheduler doesn't know the "location" of OAA, 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 cause a severe performance slowdown, incurring a sudden and sharp performance drop for microservices. 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. **(2) Failing to simultaneously schedule a combination of multiple interactive resources – core counts, LLC ways and bandwidth usage – to achieve OAA in low overhead.** 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 often leads to sub-optimal QoS. However, existing schedulers using heuristic or model-based algorithms are usually with high overheads on handling the complexity. For example, the state-of-the-art work [10] brings around 20~30 seconds on average (up to 60 seconds in the worst cases) to find ideal allocations when 3~6 microservices are co-running together. [16,41,42] also show the heuristics inefficiency due to the high overheads on scheduling various resources with complex configurations. **(3) QoS prediction is often inaccurate.** Therefore, the scheduler can hardly balance the global QoS and resource allocations across all applications, leading to QoS violations or resource over-provision. For example, we find the previous scheduler [10] (without an accurate QoS predictor) misses 29.6% opportunities to support higher RPS in the case where Img-dnn, Moses and Xapian are co-located.

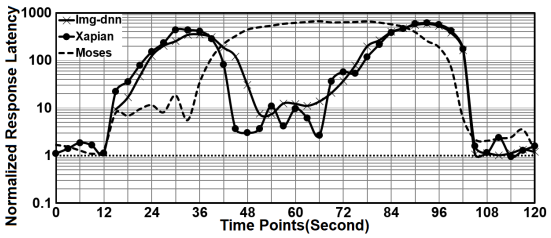

Figure 4. A case for heuristic scheduling approach.

We conduct experiments to show the issues. We use a similar idea with the same configuration in [10], which increases/decreases one-dimension resource at a time by a fine-grain "trial-and-error" way. The baseline is the optimal case in which microservice solely runs on our platform with all available resources. Figure 4 shows the QoS for 3 microservices (normalized to baseline). As illustrated, the scheduling process incurs a high and unpredictable response latency (e.g., about 500~800× latency at time point 30 for Img-dnn and Xapian; at time point 60 for Moses), and taking an extended period (~100 seconds) to achieve a better solution at time point 108

Table 3. The input parameters for ML models (MDL).

Feature	Description	Input for MDL
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
Memory Util	The memory footprint of an app	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'
LLC Occupied	LLC footprint of an app	A/A'/B/B'/C
Allocated Core	The number of allocated cores	A/A'/B/B'/C
Allocated Cache	The number of allocated LLC ways	A/A'/B/B'/C
Core Frequency	Core Frequency in run time	A/A'/B/B'/C
QoS Slowdown	Percentage of QoS slowdown	B
Target Core	Expected cores after deprivation	B'
Target Cache	Expected cache ways after deprivation	B'
Cores used by N.	Cores used by Neighbors	A'/B/B'
Cache used by N.	Cache ways used by Neighbors	A'/B/B'
MBL used by N.	Memory BW used by Neighbors	A'/B/B'
Resp. Latency	Average latency of a microservice	C

finally. We observe that the scheduler keeps trying to identify the "optimal" allocation by striving to reduce/increase core/cache ways for each application during the scheduling because it is not aware of RCliff and OAA in the scheduling space. This design quickly "falls off the cliff," (e.g., from time point 42 for Moses), incurring a high response latency that is hard to be recovered in a subsequent short period.

Toward this end, we claim it is time to design a new resource scheduling approach for microservices. Though the OS is arguably responsible for scheduling, *we have the insight that ML is the potential to offer an optimized resource scheduling solution and with the nature of handling such complicated cases in considerable low overhead.*

4 The Art of Using ML for Scheduling

The overview. We use machine learning (ML) to build a new resource scheduler - OSML. Our work differs from previous studies that use ML in the following ways. (1) We design three small ML models – Model-A/B/C – work collaboratively to perform scheduling. Model-A predicts the Optimal Allocation Area (OAA) and the RCliff for a specific microservice. Model-B shepherds the OAA by trading the QoS and allocated resources. Model-C is an online learning model that dynamically handles the cases where misprediction occurs, environment changes, and other unpredictable situations happen. (2) To train these models, we collaborate with a B2C company (i.e., Alibaba) and obtain the traces from multiple real-world applications (after warm-up). Additionally, we also simulated the behaviors of these real-world applications by deploying the microservices with diverse, commonly used RPS in Table 1 on our lab machines to generate more test traces. More details refer to the following contents.

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 microservice, the inputs of the MLP include 11 items in Table 3 and the **outputs** include the OAA, OAA bandwidth (bandwidth requirement in OAA), and

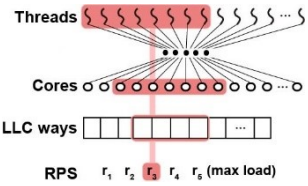

Figure 5. Model-A data collection.

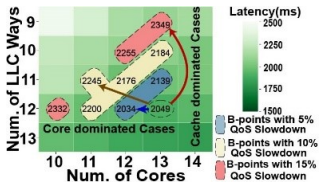

Figure 6. Model-B training.

the RCliff. Model-A has a shadow – A', which has the same MLP structure and 14 input parameters (Table 3), providing solutions when multiple microservices are running together.

In practice, OAA is slightly away from the RCliff (Figure 1) to avoid a significant QoS slowdown when some of a microservice's resources are shared or deprived of. For example, if a microservice needs at least 3 cores and 6 MB LLC capacity to meet its QoS (i.e., on the edge of RCliff), the OAA has 5 cores and 8 MB LLC capacity. OAA alerts the OS scheduler not to blindly allocate cores, LLC ways, and local bandwidth for any microservices, thus reducing the memory interferences among them in co-location cases. Model-A provides outputs within 1~2 seconds by default.

Model-A Training. Collecting training data is an expensive job. To cover the common cases, we have collected the performance traces including the parameters in Table 3 for the microservices in Table 1 for over 11 months. The parameters are normalized into [0,1] according to the function: $\text{Normalized\_Feature} = (\text{Feature} - \text{Min})/(\text{Max} - \text{Min})$, in which Feature is the original value, and the Max and Min are predefined according to different metrics. The details are as below.

For each microservice with a specific RPS demand (e.g., RPS-2200 for Moses), we first launch 36 threads and map them across 36 cores, 35 cores, 34 cores and so on until 1 core, respectively; for each threads-cores mapping case, we allocate LLC with different ways ranging from 1 to 20 (maximum) and we collect the performance traces accordingly. Next, we launch 35 threads for the microservice and map them to 36~1 core with LLC allocations from 1~20 ways, and collect the performance traces. Similarly, we conduct the mapping and trace collecting for a number of threads from 34 to 1, respectively. In summary, for each microservice 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 5 shows a data collection case where 8 threads are mapped onto 7 cores with 4 LLC ways. We feed the microservices with diverse RPS (Table 2), covering most of the common cases. Moreover, for A', we randomly map microservices on the rest resources in the above process; then, we get the traces for co-locating cases.

Finally, we collect 173,491,200 data tuples, covering 1,430,160 allocation cases with different numbers of cores, LLC ways, and bandwidth allocations. A large amount of traces leads to higher accuracy for ML models. In fact, the workload characteristics are converted to comprehensive traces consisted of diverse hardware parameters; they are used for fitting and training MLP to provide accurate predictions.

Model-A Function. Model-A uses the function ReLU (Rectified Linear Unit), i.e., $f(x) = \max(0, x)$, as the activation

function. It is efficient and effective, especially for backpropagation. The loss function employs Mean Square Error, and the gradient update is Adam Optimizer.

4.2 Model-B: Balancing QoS and Resources Allocations

Model-B Description. Model-B employs an MLP with the same structure in Model-A'; its inputs contain the same parameters. Besides, Model-B has one more input item, i.e., QoS slowdown. Model-B **outputs** the policies that, with the acceptable QoS slowdown (controlled by OSML), how many resources can be deprived of from a specific microservice. As the computing units and memory resource can be fungible [10], Model-B's output includes 3 policies, i.e., <cores, LLC ways>, <cores dominated, LLC ways> and <cores, LLC ways dominated>, respectively. The items in the tuple are the number of cores and LLC ways that can be deprived and reallocated to others with the acceptable QoS slowdown. The term "cores dominated" indicates the policy using more cores to trade the LLC ways, and vice versa. The acceptable QoS slowdown is determined according to the user requirement or the microservices' priority and controlled by the OSML's central logic. We denote Model-B's outputs as B-Points.

Model-B works as a complementary of Model-A, trading QoS for resources. It infers the least amount of resources that would be deprived of from a microservice with a specific QoS slowdown. For example, when a microservice (called E) comes to a server that already has 4 co-located microservices, 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 with predefined QoS slowdown as an input to infer B-Points for each running microservice, which includes the "can be deprived" resources with the allowable QoS slowdown. Finally, OSML tries to match <n+, m+> with B-Points and find the best solution, which should have a minimal impact on current allocation status for the existing applications. Moreover, OSML will return failure to the upper-level scheduler if it fails to find an acceptable solution. Detailed logic is in Algo._1. Besides Model-B, we also design Model-B' (a shadow of Model-B) for predicting how much QoS slowdown will suffer if a certain amount of resources is deprived of from a specific microservice. The NN structure of Model-B' is similar to Model-B's.

Model-B Training. For training Model-B and B', we reduce the allocated resources for a specific microservice from its OAA by fine-grain approaches, as illustrated in Figure 6. 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 6, which shows the B-Points with the corresponding QoS slowdown. We collect the training data set for every microservice in Table 1. The data set contain 463,917,600 data, covering 7,650,780 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). This function can be seen as a modified MSE.

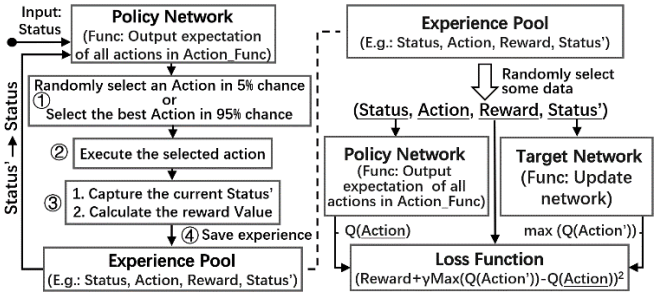

Figure 7. Model-C in a nutshell.

4.3 Model-C: Handling the Changes On the Fly

Model-C Description. Model-C corrects the inappropriate resource allocations (e.g., resource under/over-provision) and collects data for online training. Figure 7 shows the Model-C in a nutshell. The critical component in Model-C is an enhanced Deep Q-Network (DQN) [43], which is devised according to our new requirements. Model-C consists of two neural networks, i.e., Policy Network and Target Network. The Policy Network is a 3-layer MLP that includes 3 hidden layers (each layer has 30 neurons). The structure of Target Network is identical to the Policy Network. 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 7 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 can avoid falling into a local optimum [43].

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

If $Latency_{t-1} > Latency_t$:
$R_t = \log(Latency_{t-1} - Latency_t) - (\Delta CoreNum + \Delta CacheWay)$
If $Latency_{t-1} < Latency_t$:
$R_t = -\log(Latency_t - Latency_{t-1}) - (\Delta CoreNum + \Delta CacheWay)$
If $Latency_{t-1} = Latency_t$:
$R_t = -(\Delta CoreNum + \Delta CacheWay)$,

where $Latency_{t-1}$ and $Latency_t$ denotes the latency in previous and current status, respectively; ΔCoreNum and ΔCacheWay

Table 4. The Summary of ML models in OSML.

ML	Model	Features	Model Size	Loss Function	Gradient Descent	Activation Function
A	MLP	11	144 KB	Mean Square Error (MSE)	Adam Optimizer	ReLU
A'	MLP	14	155 KB			
B	MLP	15	110 KB	Modified MSE		
B'	MLP	16	106 KB	MSE		
C	DQN	10	141 KB	Modified MSE	RMSProp	

represent the change of the number of cores and LLC ways, respectively. This function gives the Action leading to less resource usage and lower latency the higher rewards and expectations. Thus, Model-C can guide to allocate appropriate resources in any case. Algo._2 and 3 show the OSML's 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 microservice, 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. Generally, 2 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 are responsible for 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. For example, we use 2 data tuples that one is from <3 cores, 4 LLC ways> allocation while another is from <5 cores, 4 LLC ways> allocation, implying the actions that 2 more cores are allocated or reduced. Moreover, we also collect the training data in the cases where LLC sharing occurs among different microservices and save them in the Experience Pool. Using them, Model-C can have the first step knowledge on selecting actions in resource sharing cases, and avoid the stuck instances in practice. To sum up, we have 1,710,726,000 tuples in Model-C training data set.

Online Training. Model-C supports online training. The overall workflow is shown in the right part of Figure 7. Model-C randomly selects some data tuples (200 by default) from the Experience Pool. Then, for each tuple, Model-C uses the Policy Network to get the Action's expectation value (i.e., Q(Action) [43]) with the Status. Model-C uses the Target Network to have the expectation values of Status' across the actions in Action_Function and then finds the max one, i.e., $\mathrm{Max}(\mathrm{Q}(\mathrm{Action}'))$. Illustrated in Figure 7, we design a new Loss Function based on MSE: $(\mathrm{Reward}+\gamma\mathrm{Max}(\mathrm{Q}(\mathrm{Action}'))-\mathrm{Q}(\mathrm{Action}))^2$. It indicates whether OSML can have an optimal scheduling solution by taking this Action. The Policy and Target Network will be updated according to the online training results. With updating, Model-C provides more accurate predictions for future actions, even for unpredictable cases.

4.4 Discussions

Table 4 summarizes ML models. We show several of our considerations here. **(1) ML model selection.** We want to leverage our large-scale training traces captured in reality to achieve an accurate prediction with low overhead. As a supervised ML algorithm, MLP can satisfy both of our requirements. We also want to predict future actions based on history and online information, so we use DQN because of its high accuracy, high efficiency, and low complexity. MLP and DQN are the ideal choices for resource scheduling with OS. More evaluations are in Sec. 6.4. **(2) Why do our models work?** These models are trained using extensive performance traces that reflect the correlations between the computing units and memory hierarchy across diverse typical workloads. Model-A and B are carefully tuned, and the training data sets continue to grow for more platforms, configurations, and workloads. Model-C is a dynamic model, which collects the run-time information for online training, correcting the misprediction caused by Model-A/B while enhancing itself through online learning. **(3) Why do we need the three models? Why don't use Model-C directly?** Model-C's actions are based on Model-A/B's outputs. With Model-A and B, Model-C can try to have the solutions from the predicted OAA, saving time on exploring the scheduling space. In practice, Model-C only needs some small calibrations to achieve the ideal results, outperforming the heuristic-based approaches. **(4) Generalization.** For better generalization performance, (i) we follow the rule "leave one subject out", i.e., the training data sets don't include the testing one for each microservice; (ii) we try to collect traces for the popular workloads as many as possible on off-the-shelf servers. Table 5 summarizes these models' performance.

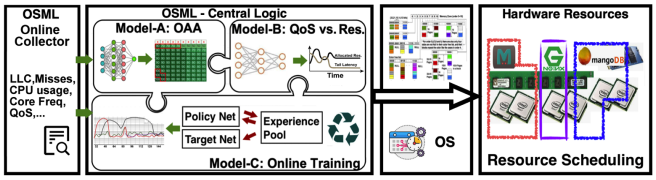

Figure 8. The overview of OSML.

5 OSML: System Design

In this section, we show the design details for OSML. As illustrated in Figure 8, OSML's central logic controls the workflow of ML models. The online profiling module collects the run-time parameters for microservices. OSML leverages OS interfaces to schedule hardware resources.

5.1 The Central Control Logic

The central controller has the overall responsibility to coordinate the ML models, manage the data/control flow and report the scheduling results to upper scheduler. Figure 9 shows its whole control logic. More details are as follow.

Allocating Resources for Microservices. Algo._1 shows how OSML uses Model-A and B in practice. Figure 9 also highlights its critical operations. For a new coming microservice, the central controller calls Model-A via the interface *modelA_oaa_rcliff()* to get the OAA and RCliff. If the current idle resources are not sufficient to meet the new microservice, OSML will enable Model-B through the interface *modelB_trade_qos_res()* to deprive some resources of other microservices with the acceptable QoS slowdown (controlled by upper-level scheduler), and then allocate them to the new one. In the depriving process for a specific microservice, OSML reduces its allocated resources and gets close to the RCliff, but it will not easily fall off RCliff unless expressly permitted (refer to Algo._4).

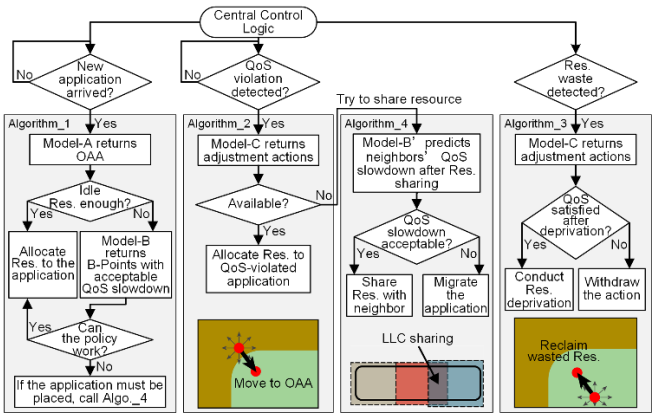

Figure 9. OSML's central logic.

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

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

Dynamic Adjusting. OSML has the capability of handling the cases where (i) environments/user demands change, leading to Model-A/B inaccurate; (ii) misprediction happens; (iii) resource sharing is allowed for co-locating more loads.

Figure 9 demonstrates the dynamic adjusting process in Algo._2 and 3, in which Model-C works as a dominant role. In the run time, OSML monitors each microservice'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 a microservice is over-provision (i.e., wasting resources), Algo._3 will be used to reclaim them, and Model-C will be called via *modelC_downsize()*. The outputs are <core+, LLC way+> and <core-, LLC way->, respectively.

Algorithm 2: handling the cases in which resources are insufficient (under-provision).

```
for each allocated microservice do
    if its QoS is not satisfied then   // Higher latency
        Forward the current running status parameters to Model-C;
        Model-C selects a specific action in the Action_Fun;
        Return Model-C's output (<cores+, LLC ways+>) to
          OSML's central controller;
        if <cores+, LLC ways+> can be satisfied within current
          idle resources then
            OSML allocates, and GOTO Line 2;
        else
            Call Algorithm_4.   // Share resources w/ others?
        end
    end
end
```

Algorithm 3: handling allocation cases where resources are over-provision.

```
for each allocated microservice do
    // Over-provision;
    if QoS increases ∧ Instructions/second decreases then
        Forward current run-time status parameters to Model-C;
        Model-C selects a specific action accordingly;
        Return Model-C's output (<cores-, LLC ways->) to
          OSML's central controller;
        OSML reduces the resources accordingly;
        if its QoS is not satisfied now then
            OSML withdraws the actions.   // Rollback
        end
    end
end
```

Algorithm 4: handling resources sharing among applications.

```
// OSML tries to allocate resources cross over RCiff;
Obtain how many resources a microservice needs, i.e., <+cores,
  +LLC ways>, from the neighbors to meet its QoS using Model-A;
for each potential neighbor App do
    Create sharing policies, i.e.,
    {< u, v > |∀u ≤ (+cores) ∧ ∀v ≤ (+LLC ways); u, v ≥ 0};
    Use Model-B' to predict the neighbor's QoS slowdown
    according to {< u, v >};
end
if the neighbors' QoS slowdown can be accepted by OSML then
    OSML conducts the allocation;
else
    OSML migrate the microservice to another node.
end
```

Moreover, Algo._4 will enable resource sharing, if all of the co-located microservices 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. (i) Model-A infers how many resources are required by the new microservice in addition to its currently allocated resources. (ii) Then, Model-B is enabled to predict the QoS slowdown for the microservices that will be involved, if the required resources are shared with them. In practice, to minimize the adverse effects, at most 3 microservices will be included to share their resources. Using Model-A and B, OSML can neatly handle the complicated resource sharing cases without significant scheduling overheads. Note that Algo._4 might incur the resource sharing over the RCliff, and thus may bring higher response latency for one or more microservices. 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 carried on.

5.2 Monitor Parameters and the Design Considerations

OSML interacts with upper scheduler or clients to obtain the latency of all requests and checks whether they have met their QoS targets. OSML monitors the run-time parameters for each co-located microservice 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, we find the OSML indeed performs well with other interval settings and allows the flexibility to be configured as needed (e.g., 1.5 or 2 seconds).

Bandwidth Scheduling. OSML partitions the overall bandwidth for each co-located microservice according to the ratio $BW_j / \sum BW_i$. BW_j is a microservice's OAA bandwidth requirement, which is obtained from the Model-A. Note that such scheduling requires CPU to have MBA support [4,5].

5.3 Implementation

We design OSML that works cooperatively with OS (Figure 8). As the kernel space lacks the support of ML libraries, OSML lies in the user space that exchanges information with 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 Intel MBA [5] to allocate specific cores and bandwidth to a microservice. OSML captures the online performance parameters by using the pqos tool [4] and PMU [5]. The ML models are based on TensorFlow [6] with the version 1.13.0-rc0, and can be run on either CPU or GPU.

6 Evaluations

6.1 Methodology

We evaluate OSML on our platform in Table 2. The metrics include the QoS, which is measured by the response latency, and EMU denoted as the max aggregated load of all co-located microservices [10] – higher is better.

6.2 OSML Effectiveness

We compare OSML with the following competing approaches:
PARTIES [10]. It is the state-of-the-art work, which 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].
CLITE [46]. It conducts various allocation policies (20 in [46]) 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. We implement [10,46] in our work.
Unmanaged Allocation (baseline). This policy doesn't control the allocation policies on cores, LLC, and other shared resources for co-located microservices. 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) Using Model-A/B/C, OSML achieves OAA quickly, and responses timely when the workloads' demands change. Figure 10 shows the distributions of the scheduling results for 591 (197*3) cases. Every dot represents a scheduling

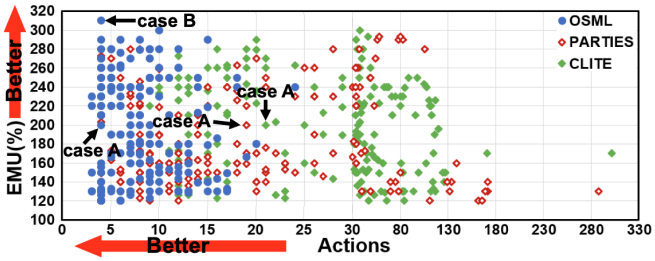

Figure 10. The performance distributions for OSML, PARTIES, and CLITE; 197 different loads are tested for every scheduler.

result for a specific workload that contains several co-located microservices with diverse RPS. The x-axis shows the number of scheduling actions; the y-axis denotes the achieved EMU using these actions. On average, OSML uses 8.1 actions to deliver similar or higher EMU, compared with PARTIES and CLITE, who use 26.2 and 44.6 actions, respectively. OSML performs stably – the number of actions ranges from 3 (best case) to 24 (worst case), indicating it handles diverse cases efficiently. In contrast, the actions in PARTIES range from 4 to 288, and CLITE is from 4 to 302, indicating they can incur higher scheduling overheads because more scheduling actions are conducted in many cases.

Figure 11-a, b, and c show how OSML and [10,46] schedule for case A in Figure 10, which includes Masstree, Specjbb, Xapian, and MongoDB with 30%, 30%, 90% and 50% of their maximum loads, respectively. Figure 11 highlights actions using solid red lines to represent increasing resources and blue dotted lines to denote reducing resources. The schedulers (OSML and [10,46]) enable scheduling actions for every second. Figure 11-a shows [10] takes 11 actions for scheduling cores and 8 actions for cache ways. [46] relies on the sampling points in scheduling exploration space; more points indicate a better scheduling solution. Figure 11-b shows [46] repeats scheduling and sampling until it meets the QoS targets. CLITE performs 21 scheduling actions; all of them involve both cache and cores for all of the applications. It will take 0.021s to enable CAT configuration each time, plus context switch and cache line eviction overheads, therefore the scheduling overheads are high. For case A, OSML achieves OAA for each microservice with only 4 actions (Figure 11-c). Apparently, [10] schedules multiple resources one by one and step by step; [46] blindly enables many scheduling actions without clear targets; OSML is the best, which has clear aims and schedules multiple resources simultaneously to achieve them.

Moreover, as the scheduling is fast, OSML often supports more loads than [10,46] during a specific period. Figure 12 shows the OSML's results on scheduling 4 microservices – Masstree, Specjbb, Xapian, and MongoDB (with 50% of maximum load – RPS-5000 – in the background). For a specific scheduling phase, by using ML to achieve OAA, OSML supports 10~40% higher percentage of loads in 17~36% cases than [10,46] (e.g., highlighted cells in Figure 12-d). Figure 12-c shows that PARTIES outperforms CLITE in 19% cases; CLITE outperforms PARTIES in 7% cases. All schedulers perform better than the Unmanaged (the baseline in Figure 12-a), as they reduce the resource contentions.

(2) Compared with [10,46], OSML uses fewer resources to support identical loads to meet the QoS targets. As illustrated in Figure 11-a, [10] partitions the LLC ways and cores equally for each microservice at the beginning; once it meets the

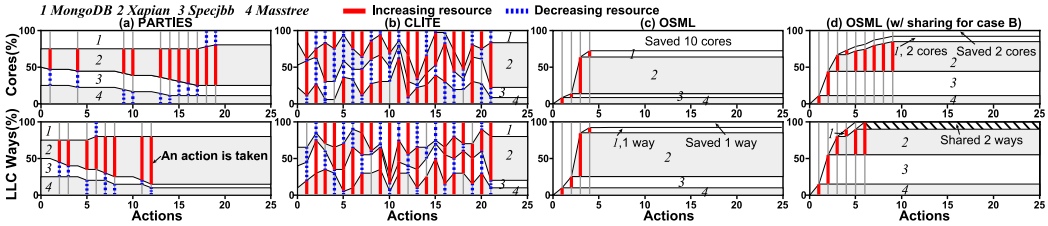

Figure 11. Resource usage comparisons for OSML, PARTIES, and CLITE.

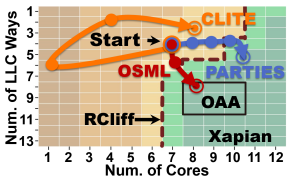

Figure 13. Highlighted actions in scheduling space.

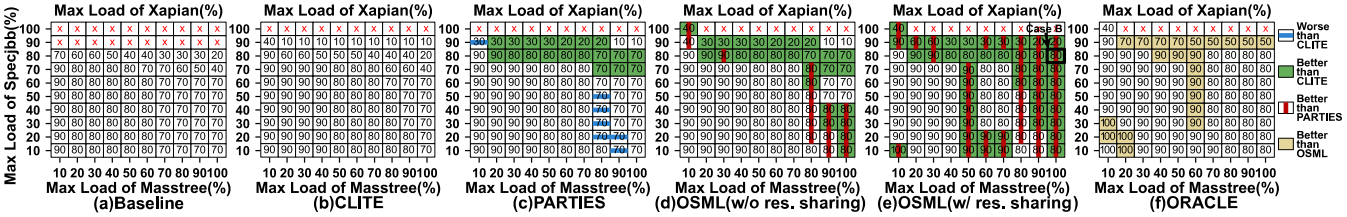

Figure 12. Co-location of Masstree, Specjbb, Xapian and MongoDB. The heatmap values are the percentage of third microservice's (i.e., Xapian) achieved max load without QoS violations in these cases. The x- and y-axis denote the first and second app's fraction of their max loads (with QoS target), respectively. Cross means QoS target cannot be met.

QoS target (using 19 actions), it stops. Thus, [10] drops the opportunities to explore alternative better solutions (i.e., using fewer cores or cache ways to meet identical QoS targets). [10] allocates all cores and LLC ways finally. [46] also uses all cores and cache ways shown in Figure 11-b. In contrast, Figure 11-c shows that using Model-A, OSML achieves each microservice's OAA (the optimal solution) only after 4 actions, saving 10 cores and 1 LLC way finally. As OSML is designed for microservices that are executed for a long period, saving resources means saving more banquets for cloud providers.

(3) Using ML models, OSML quickly provides solutions for sharing some cores and LLC ways among microservices, therefore supporting higher loads. But, [10,46] don't support 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 9. Figure 11-d shows how OSML shares resources for the highlighted case B in Figure 10 and 12-e. OSML enables Model-C to add resources for Xapian in Algo._2 and uses Algo._4 to share 2 LLC ways with MongoDB (the best solution). Finally, the QoS is met, and OSML supports 10% higher loads for Xapian in case B. Figure 12-e further highlights the cases supporting higher loads by using resource sharing in OSML, outperforming [10,46] by 10~60%, and even OSML without resource sharing by 10~50%, respectively. OSML is close to ORACLE in Figure 12-f; exhibits identical performance in 79% case. If not OSML, however, the "trial and error" approach has to try to share core/cache way in a fine-grain 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 enables Model-C when resource under/over-provision, QoS violations and other cases happen. In our experiments, after Model-A/B, Model-C adjusts multiple resources simultaneously, and usually accomplishes its job with 2 or 3 actions for each microservice, rather than exploring in the scheduling space for coordinating multiple resources step by step. In contrast, to handle QoS violations caused by resource under-provision, [10] and [46] use 8.7 and 6 actions, on average, respectively; to tackle over-provision, [10] uses 6 actions and [46] uses 12.3 actions, on average. OSML performs better than them.

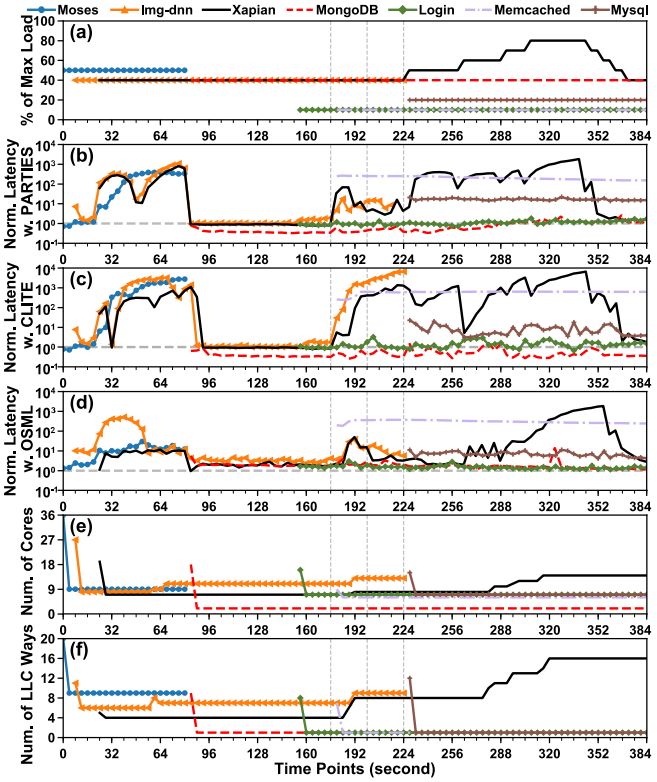

Figure 14. OSML's performance in reality with varying loads.

6.3 Performance for Fluctuating load

We evaluate OSML employing dynamical changing loads. Each microservice's QoS is normalized to the solely running case. As illustrated in Figure 14, in the beginning, Moses with 50% of max load arrives; then Img-dnn and Xapian with 40% of max load arrive. We observe their response latency increases caused by the resource contentions among them. PARTIES' "trial and error" approach takes 61 seconds to aid the QoS violations. CLITE performs similarly, as it has to sample in as many scheduling solutions as possible for accuracy, and feed them into Bayesian optimizer. In contrast, OSML quickly provides the ideal scheduling solutions to meet QoS

targets with less than 10 seconds for all of the microservices. Figure 14-e and f illustrate OSML's scheduling actions for achieving ideal solutions. In short, within only a few scheduling actions (about 1/5 of PARTIES) that schedule multiple resources, OSML meets the QoS targets.

For RCliff and OAA, Figure 13 highlights the scheduling actions for Xapian from the time period 176~200s, during which Memcached comes. [10] uses 4 actions to have an acceptable solution (not the best solution), but OSML's Model-C merely uses 2 actions to achieve OAA – the best solution. [46] performs worst; it is always at the bottom of the RCliff. Note that OSML provides the best QoS supports; it quickly jumps to OAA by using ML models.

From the time point 224, we increase the load for Xapian, and find its latency increases for [10,46]. OSML meets Xapian's demands quickly. Moreover, as OSML saves resources, it can serve more workloads. For example, as shown in Figure 14, the Login comes at 153; OSML allocates the saved cores to it without sharing or depriving other microservices of resources. At 228, OSML handles Mysql (unseen case) well by scheduling saved cores to it; CLITE, however, deprives Memcached of cores to satisfy Mysql. Thus, Memcached is with a better performance using OSML.

6.4 Evaluations for ML Models

We summarize the average performance for ML models in Table 5. ML models' prediction errors are low. Even for the unseen cases, at most 4 cores are wrongly allocated, which can be corrected by Model-C within two actions. For overheads, enabling ML models takes below 0.2s for each time. As our models are light weighted, running them on CPU and GPU are with similar overhead. If models are on GPU, it takes 0.23s for receiving results from GPU. Generally, the overhead doesn't hinder the overall performance.

In cloud, microservice's behaviors could change every second due to the diversity of user demands. Thus, OSML can play an essential role during the entire run time. If GPU is used, OSML will not monopolize it. It can be used for other purposes in idle time. If many servers need scheduling, their data can be gathered via the network; OSML enables GPU to process them. We will show results on this in our future work.

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. [34] uses deep reinforcement learning for resource management in a networking environment. The efforts in [22,56] leverage ML to optimize computer architecture, making CPU or memory controller adaptive to workloads. [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 3 light-weight ML models and train them using microservices' real performance traces collected for over 11 months, making OSML provide better scheduling solutions to meet QoS targets much faster than the state-of-the-art work stably.*

ML for Scheduling. Decima [35] designs a cluster-level data processing job scheduling, i.e., TPC-C/Spark, using RL. Resource Central [12] builds a system – RC – that contains the historical resource utilization information of the workloads used in Azure. When a new VM starts, the scheduler contacts RC and employs ML to predict resources management. The work in [40] leverages RL to predict which subsets of operations in a TensorFlow graph should run on which of the available devices. Paragon [14] and Quasar [15] use collaborative filtering techniques to classify workloads for heterogeneity and interference in shared resources. [48] applies ML to predict the end-to-end tail latency of microservice workflows. CLITE [46] leverages Bayesian optimization for scheduling; its online sampling overheads on scheduling space exploration, however, prevent it from efficiently providing an optimal scheduling result. *In contrast, for today's microservices, none of them like OSML uses ML to predict the fine-grain allocations for the complicated multiple interactive on-node resources (cores, cache ways, and memory bandwidth) simultaneously to meet the QoS in a timely fashion.*

Resource Partitioning. PARTIES [10] partitions cache, main memory, I/O, network, disk bandwidth, etc. to provide QoS for co-located services. [17,28,57] design some LLC partitioning mechanisms. [23,27,44,45] show that cooperatively partition LLC, memory banks, and channels outperform that merely partition one level memory resource. 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. Moreover, [7,11] only study the "cache cliff" phenomenon for SPECCPU 2006 applications and Memcached. *In contrast, OSML is the first work that profoundly explores cache cliff and core cliff simultaneously that significantly affects the resource scheduling for many popular microservices. And, OSML is the first work using ML to guide the multiple resources partitioning, which is a cost-effective way in new cloud environments.*

Table 5. ML Models' performance.

ML	Outputs	Error		Errors for unseen microservices		MSE	Overheads
		Core	LLC	Core	LLC		
A	RCliff	0.589	0.288	2.216	1.076	0.0025	0.20s
	OAA	0.580	0.3595	1.954	1.582		
A'	RCliff	1.072	0.815	3.403	1.835	0.0135	0.20s
	OAA	1.072	0.814	3.404	1.835		
B	B-Points	0.612	0.053	4.012	0.167	0.0012	0.18s
	B-Points,Core dominated	0.314	0.048	3.434	0.937		
	B-Points,Cache dominated	0.093	0.462	0.789	0.783		
B'	QoS reduction	7.87%		8.33%		0.0035	0.19s
C	Scheduling actions	0.908	0.782	0.844	0.841	0.7051	0.20s

8 Conclusion

We have presented OSML, an online resource scheduler for microservices. OSML leverages ML in its key components to preserve QoS for the co-scheduled services. We evaluate OSML against the state-of-the-art work and show that it performs better. 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. We hope our efforts could benefit researchers in our community.

REFERENCES

[1] "How 1s could cost amazon $1.6 billionin sales." https://www.fastcompany.com/1825005/how-one-second-could-costamazon-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, "Resouce 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 38 (2): 21-29 (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, Zehan Cui, Mingjie Xing, Chengyong Wu, "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, Yong Li, Chen Ding, Hao Yang, Chengyong Wu, "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, Shengjie Yang, Lu Peng, Xinyu Li, "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: Systematic and Efficient Co-location of Multiple Latency-Critical and Throughput-Oriented Workloads,” 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