QoS-Aware Machine Learning-based Multiple Resources Scheduling for Microservices in Cloud Environment

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. It not only impacts the exploration efficiency, making it difficult to converge to the optimal scheduling solution, but also results in severe QoS fluctuation.

To overcome these problems, we propose a novel machine learning-based scheduling mechanism called OSML. It uses resources and runtime states as the input and employs two MLP models and a reinforcement learning model to perform scheduling space exploration. Thus, OSML can reach an optimal solution much faster than traditional approaches. More importantly, it can automatically detect the resource cliff and avoid them during exploration. To verify the effectiveness of OSML and obtain a well-generalized model, we collect a dataset containing over 2-billion samples from 11 typical microservices running on real servers over 9 months. Under the same QoS constraint, experimental results show that OSML outperforms the state-of-the-art work, and achieves around 5× scheduling speed.

I. 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 can work together to serve the end-users [14,15,45,46]. 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 [3,5,14,46]. QoS (i.e., Quality of Service; response time) is a critical metric for microservices. In reality, end-users keep increasing demands for quick response from the cloud [12,20,45]. 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 lost annually [4].

In fact, the resource scheduling for QoS has become an even more challenging problem in this era. On the one hand, the cost efficiency policy drives providers to co-locate as many applications as possible on a server. These co-located microservices, however, exhibit diverse behaviors across multiple resources, including CPU cores, cache, bandwidth, main memory banks, I/O, etc. In addition, their behaviors change from time to time, and from demand to demand. On the other hand, with the increasing number of cores, more threads share and contend for the LLC (last-level cache) and memory bandwidth interactive with each other and pose more challenges for resource scheduling mechanisms [7,9,20,29,46]. All these issues enlarge the exploration space, making scheduling more complicated and time-consuming.

Some prior approaches based on heuristic algorithms – increasing/decreasing one resource at a time and observing the performance variations – might not handle users' diverse requirements in a timely fashion on platforms with increasingly parallel computing units and complex memory hierarchies. Some alternative mechanisms employ on-line clustering approaches for allocating LLC or LLC together with main memory bandwidth among single-thread applications. However, they are not suitable for microservices that contain concurrent threads. Additionally, they always rely on accurate performance models, which might bring high scheduling overheads during runtime and incur non-negligible porting efforts. In addition, designing an accurate performance model is still a challenging work. Thus, the community is expecting new directions on designing resource scheduling mechanisms [9,10,19,25,27,29].

In this paper, we design OSML, a novel machine learning (ML) based resource scheduling mechanism for microservices. OSML abstracts resources and microservice run-time states as the input and employs ML models to perform scheduling space exploration. Over the past decade, ML has achieved tremendous success in improving speech recognition [42], benefitting image recognition [23], and helping the machine to beat the human champion at Go [11,21,43]. Yet, it is still an open question on how to use ML to enhance the scheduling mechanism, which works as a system's key component.

In our study, we find that there are three underlying reasons why ML has not been widely used for resource scheduling: (1) scarce training data, leading to inaccurate inference results from ML models; (2) lack of clear abstractions of ML models that are suitable for low-level resource scheduling, making the design of overall scheduling mechanism difficult; (3) lack of a clarity in 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.

OSML includes the following contributions. (1) We analyze the performance bottlenecks, and we collect the performance traces for widely deployed microservices, e.g., Memcached, MongoDB, Moses, Sphinx, etc., with diverse configurations (in Table 1), covering 72,776,880 cases including more than 2-billion samples in a productive environment for over 9 months. More importantly, we make all of the training data sets along with OSML publicly available at (Link), and we believe our efforts can benefit our community. (2) We reveal the resource cliff (RCliff) phenomenon in scheduling exploration, i.e., QoS suffers a sharp slowdown even only a slight resource is deprived. RCliff significantly affects existing schedulers' performance.

Table 1. Microservice details. The max load (RPS) is with the 95th percentile tail latency QoS target [9,14,46].

Microservice	Domain	RPS (Requests Per Second)
Img-dnn	Image recognition	2000,3000,4000,5000,6000
Masstree	Key-value store	2800,3400,3800,4200,4600
Memcached	Key-value store	256k,284k,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
Ads	Online renting ads	10,100,1000

(3) Based on our studies, we employ two MLP models and a reinforcement learning model (DQN) to guide scheduling. To the best of our knowledge, OSML is the first work that addresses RCliff in its scheduling, providing ideal solutions in a short time and avoiding the QoS spiking often incurred by the existing schedulers. (4) We implement OSML in reality based on Linux kernel with the version 4.19. And we don't add more components to the existing OS kernel. OSML is designed as a co-worker of OS kernel that is located between the OS layer and user layer.

In practice, OSML captures the microservices' online behaviors and forwards them to the ML models run on GPU. OSML makes the scheduling decision according to the results from the GPU. On average, compared to the state-of-the-art, OSML achieves the better solutions and meets the QoS targets within with merely 1/5 overhead.

II. Background and Motivation

New Trend in Cloud Environments. The cloud environment has a growing trend towards the microservice implementation model [3,14,46]. Modern cloud applications comprise numerous distributed microservices such as key-value storing, database serving, access-control management, business applications serving, etc. [14,15]. Table 1 includes several typical microservices, which are widely used and form a significant fraction of cloud applications [14]. These microservices are with different features and resource requirements. We study these microservices in this article.

New challenges for resource scheduling. Nearly a decade before, a datacenter server equipped an Intel i7-series CPU with 4/8 cores/threads, 8 MB LLC, and supports 12.8 GB/Sec memory bandwidth per channel. Now, new servers have an increased number of cores, larger LLC capacity, larger main memory capacity, and higher bandwidth. Table 2 compares the two typical datacenter servers used at different times. *Our platform* is used as the testbed in this work.

However, 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 [24,44], indicating that a large number of resources are wasted every day and night.

Now, we need to perform a comprehensive study on how to timely meet the resource demands for co-located microservices. In practice, each of the microservices has its own QoS constraint [9,37,40]. However, they have to share and contend resources across multiple resources layers, e.g., cores, LLC, memory bandwidth, and banks (e.g., DRAM banks), therefore bringing unpredictable QoS fluctuations [9,29,22,39]. Previous studies show the contentions involve multiple resources incur serious performance degradation and QoS violation and propose the scheduling mechanisms at hardware architecture, OS and user-level [8,9,19,29,41]. Nevertheless, we still face two key open questions: *do the existing approaches serve microservices well? If not, how to design a cost-effective scheduler that avoids the common problems in existing solutions?*

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 physical cores)	8 Cores (4 physical 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	8MB
Disk	1TB,7200 RPM, HD	500GB, 5400 RPM, HD
GPU	NVIDIA GP104 [GTX 1080], 8GB Memory	N/A

III. Investigation into Resource Scheduling for Microservices

In this paper, we study microservices that are widely deployed as the key components in cloud environments. The details of them are illustrated in Table 1.

A. 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 showcase the results across 6 typical microservices.

For Moses, as illustrated in Figure 1-a, with the increasing number of cores, more threads can be mapped on them simultaneously. Meanwhile, for a specific number 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 trend can be observed from other microservices.

However, we observe the *Cliff* phenomenon for these microservices. In Figure 1-a, Moses exhibits this phenomenon clearly. For instance, in the cases where 6 cores are allocated to Moses, the response latency is increased significantly from 34ms to 4644ms if merely one more LLC way is deprived (i.e., from 10 ways to 9 ways). Similar phenomena also happen in cases where computing resource is deprived. 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 (i.e., on the edge of it),

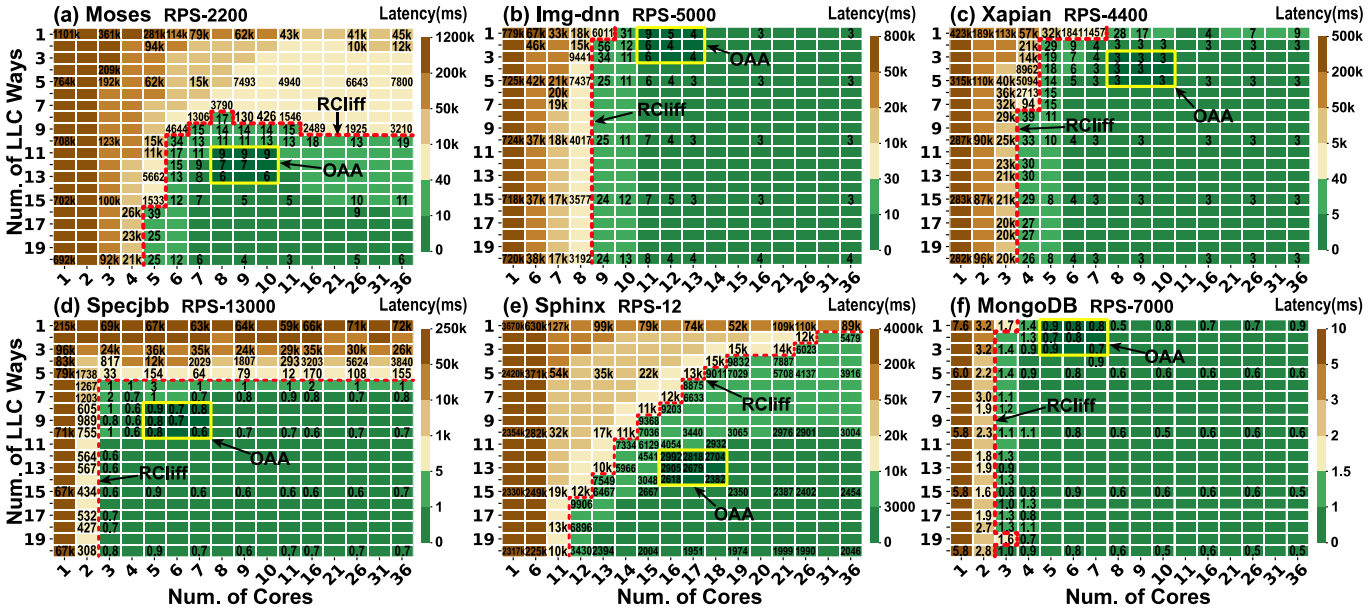

Figure 1. Sensitivity to resource allocation under different policies. Each col./row represents a specific number of LLC ways/cores allocated to an application. Each cell denotes the microservice's response latency under the given number of cores and LLC ways. The Redline shows the RCliff. The green color cells show allocation policies that bring better performance (low response latency). OAA 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.

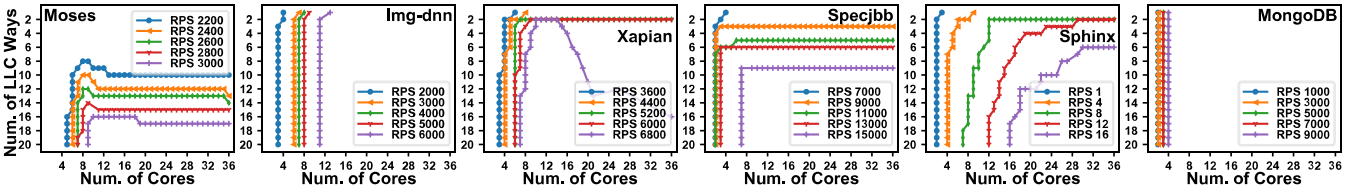

Figure 2. Sensitivity to RCliff under different RPS. We can see the RCliff is always existing, though the RPS varies. On average, RCliff exhibits 8.80% variation (Moses is with maximum variation 15.0% and MongoDB is with minimum 2.77%).

there would be significant performance slowdown if only one core or one LLC way is deprived of a microservice. From another point of view, RCliff means that a little bit more resources will bring significant performance improvement. Illustrated in Figure 1-a, Moses exhibits RCliff for both core and LLC dimensions.

Compared with Moses, Img-dnn only 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 changes, 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 still 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.

To provide ideal resource scheduling policies, RCliff should be considered seriously. RCliff alerts the scheduler not to allo--cate resources close to it, because it is "dangerous to fall into cliff" and incurs a significant performance slowdown, i.e., even a little bit resource reducing may incur severe slowdown. In Figure 1, we highlight each microservice's Optimal Allocation Area (OAA)[1], which indicates the ideal number of allocated cores and LLC ways that can bring optimal performance. Generally, OAA is not that close to RCliff. OAA is the goal that schedulers should achieve.

B. 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 one 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 than the num. of allocated cores). From 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., 20~36), the overall response latency can be higher (as illustr-

[1] To provide a better understanding of RCliff and OAA, we use the golf game as an example to explain the underlying principle. OAA is analogous to the "putting green'' in a golf course. The scheduling exploration process is analogous to hitting a ball to the putting green. And a RCliff can be considered as the boundary of a water hazard or a sand trap. If the exploration hits a RCliff, the performance is greatly degraded, just like hitting balls into a water hazard or a sand trap.

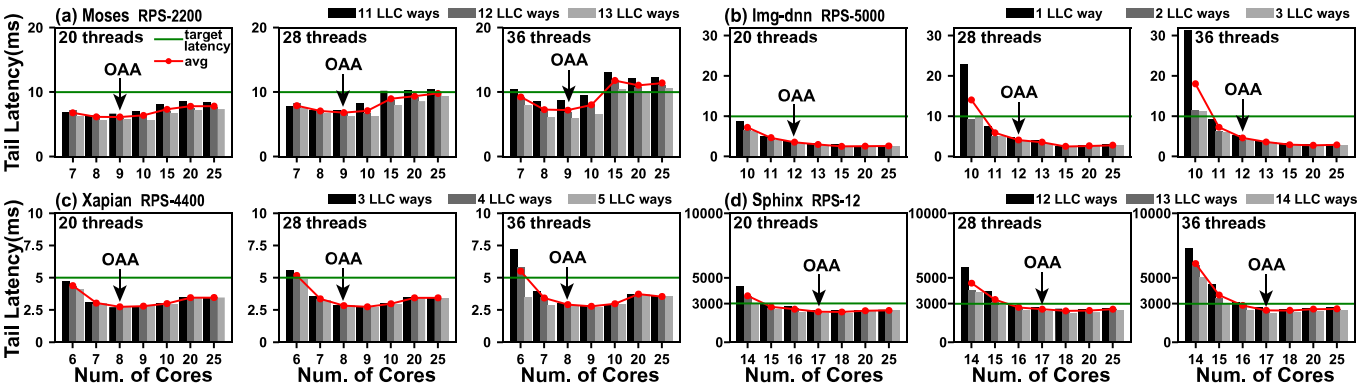

Figure 3. OAA exists regardless of the num. of concurrent threads. Due to the space problem, we only show some of the cases.

-ated in Figure 3). A similar trend can be observed in other microservices. The underlying reason lies in more memory contentions at memory hierarchy and more context switch overheads, thus leading to a higher response latency [17,36]. (2) The OAA is not quite 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 is always there. For Moses in Figure 3, when 20/28/36 threads are mapped to 7~25 cores, around 8/9-core cases always perform ideally. Other applications also show the similar phenomenon, though the OAA differs from each other.

For LLC ways in OAA, as LLC is always a scare resource, it should be allocated carefully. For each microservice, Figure 3 shows the LLC allocations in their OAA. 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 a better policy than allocating 12/13 ways), saving LLC space for other applications. We also try to allocate fewer cores to meet the QoS target for saving computing resources. To this end, we conclude that the OAA is always existing, and it is not too much sensitive to the number of threads in practice. Here, we meet a question: *how to find an Optimal Allocation Area at runtime efficiently?*

C. Existing Schedulers might not be Effective

Through our study, we find the existing schedulers often have three shortcomings to meet microservices. **(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 is in the base of RCliff (i.e., the base area is with yellow color in Figure 1-a), the scheduler will attempt to achieve OAA. However, as the schedulers do not 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 microservices. For another example, if the current resource allocation is on the (edge of) RCliff or close to RCliff, a fine-grain resource reduction for any purpose could cause a severe performance slowdown, incurring a sudden and sharp performance drop for microservices. The previous efforts [9,24,41,45] find there

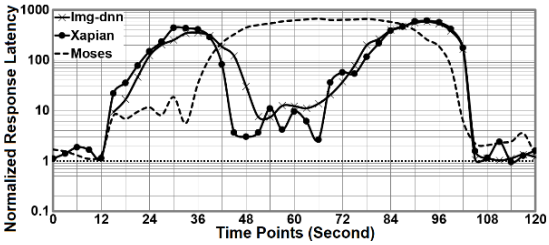

Figure 4. A case for heuristic scheduling approach.

would be about hundreds/thousands of times latency jitter, indicating the QoS cannot be assured during these periods. **(2) Failing to have an optimal schedule for microservices by simultaneously considering a combination of multiple resources – core counts, LLC ways and bandwidth usage.** Previous studies [9,19,24,29] show that the core computing ability, cache hierarchy, and memory bandwidth are interactive factors. Solely considering a single dimension for resource scheduling of co-located applications often leads to suboptimal QoS and performance. However, existing schedulers using heuristic or model-based algorithms are usually failed to consider multi-dimensions simultaneously, resulting sub-optimal solutions. **(3) Incurring high overheads.** The heuristic approaches' time consuming is not negligible. For example, the state-of-the-art [9] brings around 20~30 seconds on average (up to 60 seconds in the worst cases) to find an ideal co-locating scheme when 3~6 microservices are co-running together. [10,32,34] also show the heuristics inefficiency due to the high overheads on scheduling resources with varies and complex configurations.

We conduct experiments to show the issues. We try the similar idea in [9], 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 performance of 3 microservices (normalized to baseline). As illustrated, the whole 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 a long time (about 100 seconds) to finally achieves a better scheduling solution at time point 108 for all applications. We observe that the scheduler keeps trying to identify the "optimal" allocation by reducing/increasing core/cache resources for each application because it is not aware of RCliff and OAA. This design will quickly "jump into

the cliff," incurring a high response latency that is hard to be recovered in a subsequent short period, especially in the cases where multiple resources need to be involved in scheduling.

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 potential to offer an optimized resource scheduling solution and with the nature of handling such complicated case in a considerable low overhead.*

IV. THE ART OF USING ML FOR RESOURCE SCHEDULING

In this paper, we use machine learning (ML) to build a new resource scheduler, providing robust support for OS. We denote our design as OSML. We build fine-grained models in OSML to achieve accurate prediction results. To effectively handle the diverse cases in reality, we design 3 ML models, denoted as Model-A/B/C, work cooperatively to provide solutions. Model-A is used for finding the Optimal Allocation Area (OAA) and the RCliff for a specific microservice; Model-B is used for trading the QoS and allocated resources; Model-C is an online learning model that dynamically handles the cases where misprediction occurs, environment and user demand changes, resources sharing and other unseen cases happen. To train these models, we collect the parameters in Table 3. More details refer to the following contents.

A. Model-A: Finding OAA

Model-A's Target. For a specific microservice, Model-A is used for inferring the resource allocation policies. At the runtime, after a sampling period (within 2 seconds by default), OSML enables Model-A to obtain the OAA (Optimal Allocation Area) to meet its QoS constraint. Besides, Model-A also outputs the RCliff in the current environment. OAA is slightly away from the RCliff, because OSML is designed to incurring 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., RCliff), an OAA might have 5 cores and 8 MB LLC capacity. OAA will guide the OS allocator not to blindly allocate core, LLC ways and local bandwidth, potentially reducing the memory interferences among microservices in co-location cases. In the cases where the idle resource is ample, Model-A can have the solution after a short sampling period (less than 2 seconds).

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 [16,21]. There are 40 neurons in each hidden layer. For each running microservice, the input of the MLP includes 11 items in Table 3. The output of this MLP includes the OAA, OAA bandwidth (bw requirement in OAA), and the RCliff for a specific microservice.

Model-A Training. Collecting training data is an expensive task. To cover the common cases, we have collected the performance traces according to the parameters in Table 3 for the microservices in Table 1, primarily on our platform, for over 9 months. The details are as below.

Table 3. The Involved Parameters

Feature	Description	Used in Model
IPC	Instructions per clock	A/B/C
Cache Misses	LLC misses per second	A/B/C
MBL	Local memory bandwidth	A/B/C
CPU Usage	The sum of each core's utilization	A/B/C
Memory Util	The memory footprint of an app	A/B/C
Virt. Memory	Virtual memory in use by an app	A/B
Res. Memory	Resident memory in use by an app	A/B
LLC Occupied	LLC footprint of an app	A/B/C
Allocated Core	The number of allocated cores	A/B/C
Allocates Cache	The number of allocated LLC ways	A/B/C
Core Frequency	Core Frequency at runtime	A/B/C
QoS Slowdown	Percentage of QoS slowdown	B
Resp. Latency	Average latency of a microservice	C

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 untill 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.

Finally, we collect 171,072,000 data tuples, covering 1,425,600 allocation cases with different numbers of cores, LLC ways, and bandwidth. We believe that more traces lead to better model accuracy. Moreover, as the workload features are converted to comprehensive traces consisted of hardware parameters, we think that they can be used for fitting and training MLP to provide predictions for the unseen cases.

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 is defined as follows.

$$L_{MSE} = \frac{1}{n}\sum_{t=1}^{n}(s_t - y_t)^2$$

Gradient descent is Adam Optimizer, in which m_t and v_t are defined as:

$$m_t = \beta_1 m_{t-1} + (1-\beta_1)g_t;\ v_t = \beta_2 v_{t-1} + (1-\beta_2)\cdot g_t^2$$

And, the deviation correction includes:

$$\widehat{m}_t = \frac{m_t}{1-\beta_1^t};\ \widehat{v}_t = \frac{v_t}{1-\beta_2^t}$$

Gradient update is defined as:

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\widehat{v}_t}+\varepsilon}\widehat{m}_t.$$

B. Model-B: Balancing QoS and Resources

Model-B's Target. In the limited resource condition, Model-B works as a complementary of Model-A to trade QoS for res-

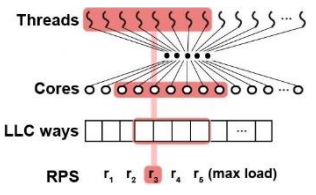

Figure 5. Model-A data collection.

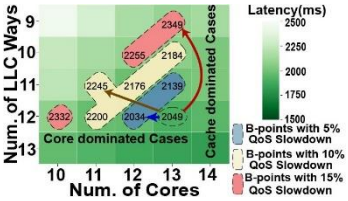

Figure 6. Model-B training.

-ources. It works in the cases where several microservices are already located on a server, and the idle cores and unallocated LLC capacity cannot meet the new application's requirements. Then, Model-B will try to deprive the already existing co-located microservices of some resources with their allowable/minimum QoS slowdown and allocate these resources to the new microservice. For short, Model-B is designed for inferring the least amount of resources that would be deprived of from a microservice with a specific QoS slowdown.

Compared with Model-A, the input of Model-B has one more item, i.e., QoS slowdown. Model-B's input also includes the parameters that are similar to those used in Model-A. Model-B's output contains 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 [9], 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 corresponding QoS slowdown. The term "cores dominated" indicates the policy that 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. We denote the outputs from Model-B as B-Points.

By using Model-B, OSML can have an ideal resource allocation solution when resources are limited. For example, when a microservice (called E) is scheduled to a server that already has 4 co-located microservices, OSML enables Model-A and then finds out that to meet E's QoS, OSML should provide at least n more cores and m more LLC ways (denoted as <n+, m+>). Then, OSML enables Model-B with predefined QoS slowdown on each running microservice to output B-Points. Finally, OSML tries to match <n+, m+> with B-Points and find the best solution, which should have the minimal impact on current allocation status for the existing applications. Moreover, OSML will return failure to upper-level scheduler if it fails to find an acceptable solution. More details can be found in Algorithm_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.

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, corresponding to different outputs of Model-B, 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 (<=) 5%, 10%, 15% and so on, respectively. Examples are illustrated in Figure 6, which shows the cases with the corresponding QoS slowdown, i.e., the B-Points. We collect the training data sets for every microservice in Table 1. The training data sets are with 350,697,600 data, covering 2,922,480 cases.

Model-B Function. We design a new loss function for Model-B,

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

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). Model-B' also uses this loss function.

C. Model-C: Handling the Changes On the Fly

Model-C's Target. Model-C handles the cases where QoS is violated due to environment changes, user demand/application behavior changes and other unseen problems happen. And, Model-C can correct the impropriate resource allocations (e.g., resource wasting) on the fly and can collect data for on-line training. Figure 7 shows the Model-C in a nutshell. In our design, the critical component in Model-C is an enhanced Deep Q-Network (DQN) [38], which is redesigned according to the new scheduling requirement. Model-C contains 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 input consists 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 are defined as Action_Function: {<m, n> | m ∈ [-3,3], n∈[-3,3]}, in which a positive m denotes allocating m more cores for an application and a negative m means depriving it of m cores. The n indicates the actions on LLC ways. 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

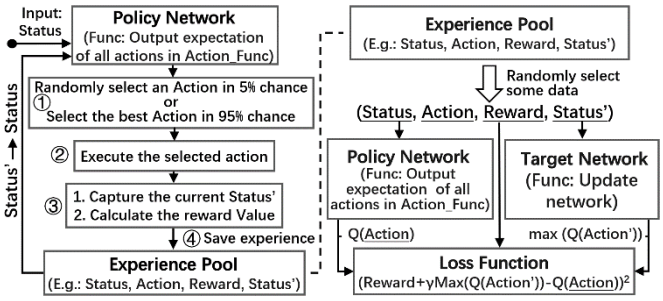

Figure 7. Model-C in a nutshell.

in practice (around 2 steps). 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 [38].

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

If $Latency_{t-1} > Latency_t$:

$$R_t = \log(Latency_{t-1} - Latency_t) - (\triangle CoreNum + \triangle CacheWay)$$

If $Latency_{t-1} < Latency_t$:

$$R_t = -\log(Latency_t - Latency_{t-1}) - (\triangle CoreNum + \triangle CacheWay)$$

If $Latency_{t-1} = Latency_t$:

$$R_t = -(\triangle CoreNum + \triangle CacheWay),$$

where $Latency_{t-1}$ and $Latency_t$ denotes the latency of previous and current status, respectively; and $\triangle CoreNum$/$\triangle CacheWay$ is the change of the number of core and LLC ways, respectively. This function gives higher rewards and expectations to the Action that can lead to less resource usage and lower latency. Thus, no matter how many resources the previous allocation policies provides, Model-C can guide to allocate appropriate resources. Details on using Model-C are in Algorithm_2 and 3.

Offline Training. The format of the 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.

In terms of the training dataset for Model-C, we rely on the data set used in Model-A training. The process is as follows. In general, 2 tuples in Model-A training dataset 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 dataset. 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 dataset, 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 data tuples in Model-C training data set.

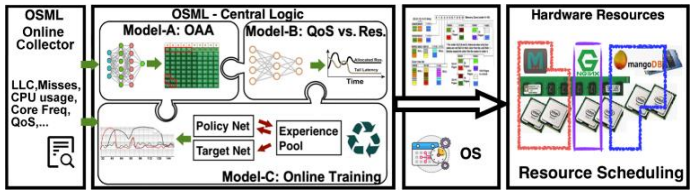

Figure 8. The overview of OSML.

Online Training. Model-C also supports online training. The overall workflow is shown in 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 have the Action's expectation value (i.e., Q(Action) [38]) with the Status; 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., Max(Q(Action')). Illustrated in Figure 7, the Loss Function is calculated as $(\text{Reward} + \gamma\text{Max}(Q(\text{Action}') - Q(\text{Action})))^2$, indicating 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. After updating, they perform better, providing more accurate action predictions for the unseen cases.

D. Discussions

Why do these models work? These models are trained using extensive data sets 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 runtime information for online training, correcting the misprediction caused by Model-A/B while enhancing itself through online learning.

Why don't we use Model-C directly? Model-C is with an online dynamic adjusting approach. Model-C's action is based on Model-A/B's output. With Model-A and B, Model-C can try to have the solutions from the predicted OAA, saving time on exploring the scheduling space and providing more accurate results. In practice, Model-C only needs some small calibrations to achieve the ideal results, performing better than heuristic-based approaches.

V. OSML: System Design

This section details the overall system design of OSML. The key components include the central controller, profiling module, and ML models. The ML models work on GPU. The profiling module captures the applications' online information, and then forward them to the ML models. Central controller receives the ML models' results and makes scheduling decisions accordingly. Figure 8 illustrates OSML in a nutshell.

A. The Central Controlling Logic

The central controller has the overall responsibility to coordinate the ML models, manage the data/control flow and

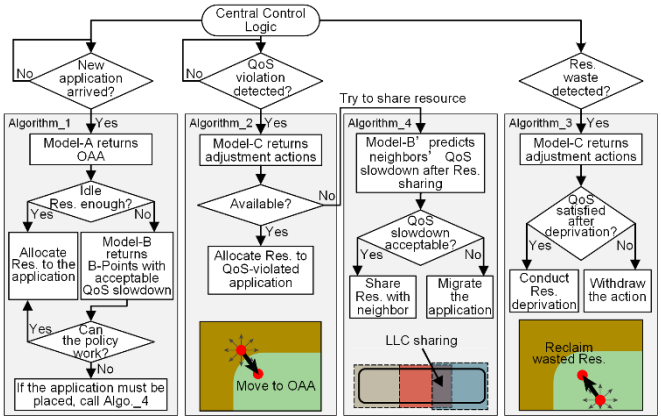

Figure 9. OSML's central logic.

report the scheduling results. Figure 9 shows its whole control logic. The scheduling principle is attempting to reach OAA without resource sharing for microservices first, and only enabling resource sharing in exceptional cases. More details are as follow.

A.1. Allocating Resources for Microservices

During the runtime, OSML enables Model-A to have the OAA and RCliff for each new coming application. Model-A has the duty of resource allocation in the first step. Model-B could help to co-locate a new one in the cases where the idle/unallocated resources cannot meet its QoS requirement. If the current idle resources are not sufficient to meet the QoS requirement for this microservice, OSML will enable Model-B to deprive some resources of other co-locating microservices with the acceptable QoS slowdown (controlled by OSML or upper-level scheduler), and allocate them to the new coming one. After, OSML will conduct the resource scheduling accordingly or reports the exceptional.

Algorithm_1 shows how OSML uses Model-A and B in practice. Model-A's output includes the OAA and RCliff, alerting the central scheduler to notice the allocation policies that might incur QoS slowdown sharply. And, in the resource depriving process, OSML moves away from the OAA to somewhere close to RCliff (saving resources), but will not easily step into it unless expressly permitted (see Algo. _4).

----------**Algorithm_1**----------

Function: Using ML to have OPT resources allocations. In practice, only one policy in OAA will be selected.

---

1. For a new coming microservice, map it on the idle resources and capture its runtime parameters for n seconds (n is 2 by default)
2. Forward these parameters to Model-A
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 policy in OAA
6. **END IF**
7. **IF** idle resources are not enough **THEN** //Enabling Model-B
8. Calculate the difference between the idle resources and 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 microservice **DO**
11. **IF** the microservice 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 IF**
15. **END FOR**
16. Record each microservice's B-Points with the QoS slowdown
17. Find the best-fit solution to meet OAA/RCliff according to B-Point 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 microservice cannot be located on this server without sharing resources with others
22. **END IF**
23. **END IF** //Enabling Model-B
24. Report to upper scheduler about the scheduling policies #

A.2. Dynamic Adjusting

In our design, OSML has the capability of handling the cases where (i) environments or user demands change, leading to Model-A/B performs inaccurately; (ii) misprediction happens; (iii) resource sharing is allowed for co-locating more applications and (iv) unseen cases occur.

Figure 9 also demonstrates the dynamic adjusting process, in which Model-C works as a dominated role. In the runtime, OSML monitors each microservice's QoS status. If the QoS violation is detected, the central controller will enable Algorithm_2, which helps to allocate more resources and achieve the ideal QoS. It usually achieves the goal within two steps. If OSML finds a microservice is allocated with more resources than its OAA (i.e., wasting resources), Algorithm_3 will be used to reclaim them.

Moreover, if all of the co-located microservices' resources are close to their RCliff and the upper scheduler must place a new application onto this server, Algorithm_4 will be enabled to find a solution that allows the applications sharing some resources with others. Note that Algorithm_4 might cause resource sharing over the RCliff, and thus may incur higher latency. OSML will report these situations to the upper scheduler and ask for the decision. If the slowdown is not allowed, the corresponding actions will be withdrawn. In Algorithm_4, Model-A is used in the first step to infer how many resources are needed by the program in addition to the currently allocated resources. Then, Model-B is enabled to predict the QoS slowdown if the required resources are partially/entirely shared with a specific microservice.

----------**Algorithm_2**----------

Function: handling the cases in which resources are insufficient

---

1. **FOR** each allocated microservice **DO**
2. **IF** its QoS is not satisfied **THEN** //Higher latency
3. Obtain and 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 IF
11.     END IF; END FOR  #
```

----------Algorithm_3----------

Function: handling allocation cases where resources are surplus

```
1. FOR each allocated microservice DO
    //More resources are allocated, wasting resources.
2.      IF its allocated number of cores/LLC Ways > its RCliff's+2
        THEN
3.          Forward current status parameters to Model-C
4.          Model-C selects a specific action accordingly
5.          Return Model-C's output (<cores-, LLC ways->) to
            OSML's controller
6.          OSML reduces the resources accordingly
7.          IF its QoS is not satisfied now THEN
8.              OSML withdraw the actions //Rollback
9.          END IF
9.      END IF; END FOR  #
```

----------Algorithm_4----------

Function: handling resources sharing among applications

```
//OSML try to allocate resources cross over RCiff
1. Obtain how many resources a microservice needs, i.e., <+cores,
   +LLC ways>, from the neighbors to meet its QoS using Model-A
2. FOR each potential neighbor App DO
3.      Create sharing policies, i.e., {<u,v>|∀u<=(+cores) ∧
        ∀v<=(+LLC ways); u, v>=0}
4.      Use Model-B' to predict the neighbor's QoS slowdown
        according to {<u, v>}
5. END FOR
6. IF the neighbors' QoS slowdown can be accepted by OSML
   THEN
7.     OSML conducts the allocation
8. ELSE
9.     OSML migrate the microservice to another node
10. END IF  #
```

B. Parameters and the Design Considerations

OSML monitors the performance parameters of each co-located jobs using performance counters, and checks whether they have met their QoS targets. OSML has configured the default scheduling period to be 2 seconds, during which the sampling model can observe enough information for making decisions. 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.

ML model selection. We want to leverage our large-scale training traces and also achieve an accurate prediction for complex unseen cases. As a supervised ML algorithm, MLP can satisfy both of our requirements. We also want to predict future actions based on historical information, so a proper reinforcement learning model is also required. We use DQN because of its high accuracy, high efficiency, and low complexity. According to our evaluation, these models achieve both high prediction accuracy and low latency. Thus, they are the ideal choice for resource scheduling with OS.

Bandwidth Scheduling. OSML partitions the overall bandwidth for each co-located microservice according to the ratio $BW_j/\Sigma BW_i$. BW_j is a microservice's OAA bandwidth requirement, which is obtained from the Model-A. Note that such scheduling may require specific hardware support. For example, a CPU having MBA support [1,2] can achieve this goal with OSML.

C. 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. We do not modify the OS kernel significantly. OSML is implemented using python and C. It employs Intel CAT technology [1] to control the cache way allocation, and it supports dynamically adjusting the cache allocation. OSML uses Linux's taskset and Intel MBA [2] to allocate specific cores and bandwidth to a microservice. OSML captures the online performance parameters by using the pqos tool [1] and PMU [2]. The sampling interval is 1 second. The ML models are based on TensorFlow [6] with the version 1.13.0-rc0.

VI. EVALUATION

A. Methodology

We evaluate OSML on our testbed in Table 2. The metrics include the QoS and EMU, which are measured by the response latency of microservices and the max aggregated load of all collocated microservices [9].

B. OSML Effectiveness

We compare OSML with the following competing policies:

(1) **PARTIES** [9]. It is among the state-of-the-art studies, which makes incremental adjustments in one-dimension resource at a time until QoS is satisfied for all of the applications. The core mechanism in [9] is like an FSM [52]. We implement it in our work, as it is not opensource.

(2) **Unmanaged Allocation (baseline)**. This policy randomly maps the microservice's threads to cores and doesn't control the allocation polices on LLC and other shared resources. This policy relies on the OS to schedule multiple resources.

(3) **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.

Figure 10~12 shows the highest allowable load of co-located microservices without QoS violation for different policies. Figure 10 compares the performance by using Xapian, Img-dnn, and Moses. Generally, PARTIES outperforms the unmanaged cases, and OSML exhibits better performance than PARTIES. As illustrated in Figure 10-c, under the same QoS constraint, OSML can help to support higher loads for Moses in highlighted cells with red boxes. Even for these cells with identical load, OSML can achieve it in low overhead (at most 2~3 actions for each application), on average. Figure 11 shows the cases where 4 services (Moses, Specjbb, Xapian, Sphinx) are co-located. Sphinx is in the background and with 10% of the maximum load. Figure 11-c highlights the cells in

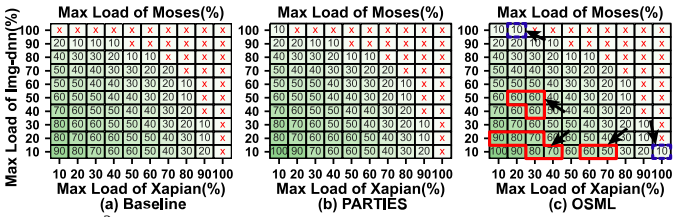

Figure 10[2]. Co-location of Xapian, Img-dnn and Moses.

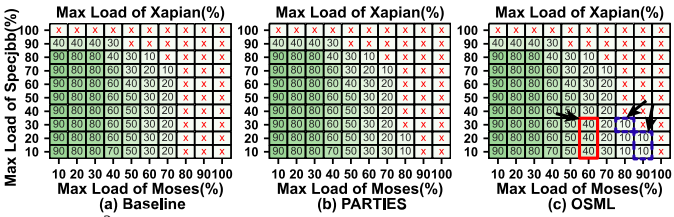

Figure 11[2]. Co-location of Moses, Specjbb, Xapian and Sphinx.

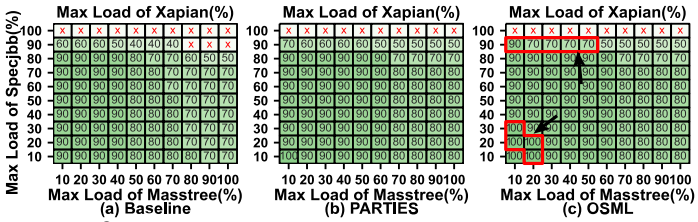

Figure 12[2]. Co-location of Masstree, Specjbb, Xapian and MongoDB.

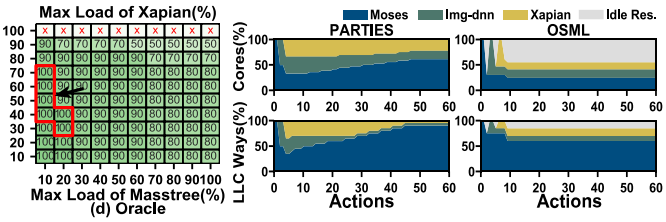

Fig.13. Resource usage comparisons.

which OSML achieves better solutions, indicating that OSML can satisfy a higher RPS for Xapian. Moreover, to our surprise, we find OSML can explore allocation policies the previous approach cannot achieve. For example, the highlighted cells with blue boxes in 11-c, OSML is able to support 10% of the maximum load of Xapian when Moses is with 90% and Specjbb is with 10~20%, respectively.

Figure 12 further shows OSML effectiveness for scheduling 4 microservices that include Masstree, Specjbb, Xapian, and MongoDB (with 50% of maximum load – RPS-5000 – in the background). We also observe that OSML can support a higher percentage of load (shown in highlighted cells in Figure 12-c). Figure 12-d shows Oracle cases. We can see that OSML behaves similarly to Oracle. For these exceptional cases, it can also support 90% of Oracle, e.g., the highlighted cells in Figure 12-d, in which Xapian is with 100% of its max loads. Generally, Figure 15 shows that OSML can bring higher EMU [9] than PARTIES.

The underly reasons are multi-fold. (1) OSML can achieve OAA in a short time and change the scheduling policies quickly according to the workloads' demands by using ML models. In other words, it can respond quickly to rapidly changing situations. Thus, we can see it meets higher loads in some cases. (2) OSML allows flexible sharing some of the LLC ways among microservices (more allocation policies), therefore bringing higher resource utilization. Always enabling strict partitioning among microservices can hurt performance. (3) OSML doesn't use the expensive heuristic "trial and error" approach and can explore different resource combinations using algorithms with ML technologies. Moreover, we find the previous work cannot quickly achieve the ideal solution when more than 4 challenging workloads are running together. And it needs carefully tuned; otherwise, it will incur high response latency due to RCliff. Moreover, once it meets the QoS constraint, it stops. Therefore, it cannot find more allocation policies to meet more workloads.

Figure 13 explores resource usage during the scheduling period for workloads in Figure 10. We observe that OSML performs differently with PARTIES, spending less time to achieve OAA (trying fewer scheduling actions), and saving more cores and LLC ways – idle core/LLC ways. The main reason is OSML quickly achieves microservices' OAA, but the "trial and error" way has to search in the large scheduling exploration space step by step. Apparently, if OSML is used widely, it will help to save banquet for cloud providers.

C. Performance for Fluctuating load

We evaluate OSML employing dynamically changing load. The results are normalized to the baseline (solely running cases, similar to the cases in Figure 4). 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 increase caused by the resource contentions among them. PARTIES uses "trial and error" algorithm to allocate resources for each application one by one, incurring relatively high latency for others, though Xapian gets more resources and behaves better for a short time. On contrast, OSML performs better, making resource scheduling decision quickly, and thus brings lower latency for all of them. Note that their response latency increases when a new service (MongoDB) comes at time point 80. OSML quickly detects it and adjusts the resource allocation policies; thus, their response latency decreases accordingly. However, previous work cannot handle this case in a timely fashion; thus, Moses is always with high latency until it is migrated to another server, and Xapian experiences a latency fluctuation before meeting its QoS constraint. Note the sub-figures for the num. of cores/LLC ways, OSML only uses a few of scheduling actions, indicating it can achieve better solutions with low overhead. However, the previous approach has to try many allocation actions. Figure 15 summarizes the average scheduling overhead is merely 1/5 of the prior approach.

From the time point 224, we increase the load for Xapian, and find its latency increases as a result for PARTIES. Yet, OSML helps to meet Img-dnn's demands in a short time using ML models. Moreover, OSML saves resources and thus can serve more applications. Figure 13 shows the resource usage for cases in Figure 10, it saves cores and thus can allocate them for memory non-intensive microservices. Shown in Figure 14, Login comes at about the time point 160, OSML allocates idle cores to meet it without sharing or depriving others of resources. Moreover, OSML handles Txt index (an unseen one) well by scheduling cores to it, but PARTIES has

[2] Figure 10~12 show the results when we collocate 3~4 microservices together. The heatmap values are the percentage of third microservice's (e.g., Moses in Fig.10, Xapian in both 11 and 12) 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.

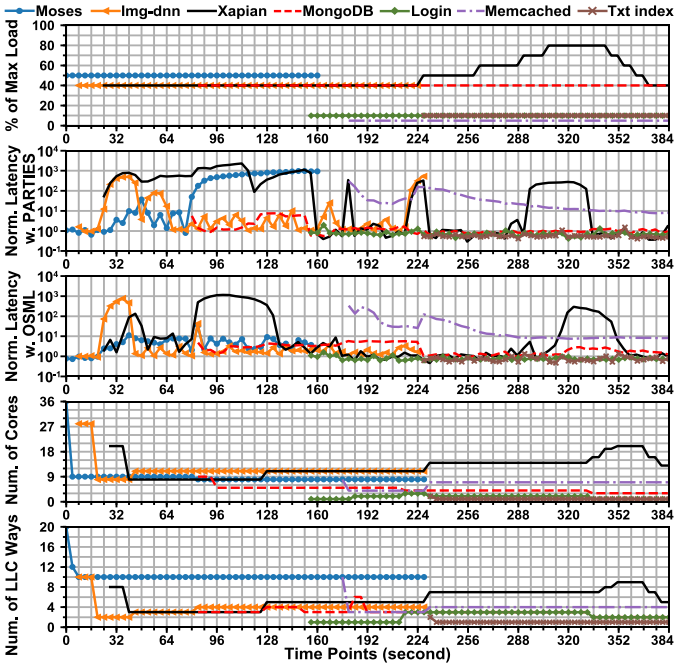

Figure 14. How OSML performs in reality.

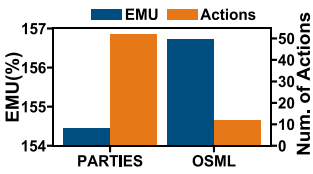

Fig.15. OSML's performance.

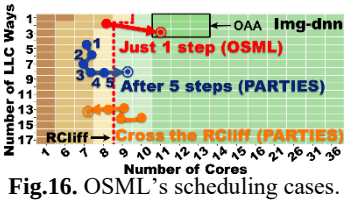

Fig.16. OSML's scheduling cases.

to let it share cores with Memcached. Therefore, Memcached is with a better performance with OSML.

For RCliff and OAA, Figure 16 shows a concrete example. At the time point 44, PARTIES uses 5 actions to have a better solution, but OSML only uses 1 action/step to achieve OAA. At the time point 56, PARTIES deprives Img-dnn of cores and LLC ways, and then allocates them to Xapian, leading to the RCliff phenomenon, incurring high latency for Img-dnn. We see clearly that the previous scheduler incurs high scheduling overheads from Figure 16. Again, Figure 16 shows the advantages and necessity of using ML in resource scheduling.

D. Disscussions

(1) ML models. In our study, we find 3 parameters – the num. of cores, LLC ways, and local bandwidth – in Table 3 play more important role than others in ML models. It is reasonable, and OSML performs well on scheduling them. **(2) RCliff**. OSML can effectively avoid RCliff, and we find that, in the scenarios with heavily resource contentions, RCliff brings a relatively lower impact for some applications, but it still obvious. Moreover, it is an easy job for OSML handle the microservices that do not have significant RCliff. **(3) Overheads**. OSML detects the QoS of microservices for every second, and once the QoS violation is detected, it will enable ML models. It takes 0.23 second for receiving results from models on GPU. Moreover, Figure 15 shows that OSML's scheduling actions are only 1/5 of state-of-the-art scheduler, on average, bringing low scheduling overhead.

VII. Related Work

(1) ML for Systems. Employing ML technologies for system design and optimizations can be a good idea. The work in [47] employs DNN to optimize the buffer size Database system. [35] uses deep reinforcement learning for resource management in a networking environment. Some efforts in [18,48] use ML to optimize computer architecture, making C-PU or memory controller adaptive to workloads. [8,33] employs ML for managing interactive on-chip resources. CALOREE in [32] can learn key control parameters to meet latency requirements with minimal energy in complex environments. Our work can be orthogonal with these studies. In OSML design, we abstract the resource scheduling problem's structure and then design ML models to handle them. **(2) ML for OS.** It is time to rethink the OS design by incorporating ML technologies. The efforts in [26,29,50,51] try to optimize the OS components with learned rules or propose insight on how to design a new learned OS or OS components. We think these studies could be worth exploring by future practitioners. In our work, OSML is designed to work closely and interact with OS. OSML is an attempt to marry OS and ML. **(3) Resource Partitioning.** Partitioning is a widely used resource scheduling scheme. [9] designs PARTIES that partitions cache, main memory, I/O, network, disk bandwidth, etc. to provide QoS for co-located services in cloud environments. [28,49] propose LLC partitioning for the multi/manycore platforms. [13] partitions LLC for diverse clusters of applications. The efforts in [19,22,31,39] show that cooperatively partition LLC, main memory banks, channel/bandwidth outperforms the approaches that merely partition one level memory resource, e.g., sole bank partitioning. However, the cooperative partitioning policies need to be carefully designed [27,30,40], and [10,24] shows the heuristic resource scheduling approach could be ineffective in many QoS constraint cases. OSML is the first work that uses Neural Network to handle the cross-layers resource partitioning problem, providing ideal QoS for co-located interactive applications in cloud environments. **(4) Microservice.** The work in [14,46] studies the implications, characteristics of microservices for designing/optimizing cloud servers; [15,45] enhance the performance of systems comprised of microservices using ML or auto-tuned approaches. Our design is partially inspired by these studies. OSML schedules resources using ML technologies, which could be a cost-effective way in new cloud environments.

VIII. Conclusion

We have presented OSML, an online resource scheduling mechanism for microservices. OSML employs ML in its key components to preserve QoS for the co-scheduled services. We evaluate OSML against state-of-the-art mechanism and show that it performs better in many cases. 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 colocation and sharing are a fundamental reality, our solution should grow in importance. We hope our efforts could be helpful to future researchers in our community.

REFERENCES

[1] "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

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

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

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

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

[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] 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

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

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

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

[11] 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)

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

[13] 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

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

[15] 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

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

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

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

[19] 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

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

[21] 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

[22] 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

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

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

[25] 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

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

[27] 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

[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] 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

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

[31] Lei Liu, Zehan Cui, Mingjie Xing, Chengyong Wu, "A Software Memory Partition Approach for Eliminating Bank-level Interference in Multicore Systems," in PACT, 2012

[32] Nikita Mishra, Connor Imes, John D. Lafferty, Henry Hoffmann, "CALOREE: Learning Control for Predictable Latency and Low Energy," in ASPLOS, 2018

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

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

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

[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, Mary Lou Soffa, “Directly Characterizing Cross Core Interference Through Contention Synthesis,” in HiPEAC, 2011

[38] 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

[39] 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

[40] 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

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

[42] 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

[43] 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

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

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

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

[47] 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

[48] 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

[49] 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

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

[51] Yiying Zhang, Yutong Huang, “Learned Operating Systems”, in ACM SIGOPS Operating Systems Review, 2019

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