Sturgeon: Preference-aware Co-location for Improving Utilization of Power Constrained Computers

*Pu Pang, *†Quan Chen, ‡Deze Zeng,
*Chao Li, *Jingwen Leng, *Wenli Zheng, *†Minyi Guo
†*Shanghai Institute for Advanced Communication and Data Science, Shanghai Jiao Tong University, China*
**Department of Computer Science and Engineering, Shanghai Jiao Tong University, China*
‡*School of Computer Science, China University of Geosciences, Wuhan, China*
avengerispp@sjtu.edu.cn, {chen-quan, lichao, leng-jw, zheng-wl, guo-my}@cs.sjtu.edu.cn, deze@cug.edu.cn

***Abstract*—Large-scale datacenters often host latency-sensitive services that have stringent Quality-of-Service requirement and experience diurnal load pattern. Co-locating best-effort applications that have no QoS requirement with latency-sensitive services has been widely used to improve the resource utilization with careful shared resource management. However, existing co-location techniques tend to result in the power overload problem on power constrained computers due to the ignorance of the power consumption. To this end, we propose *Sturgeon*, a runtime system proactively manages resources between co-located applications in a power constrained environment, to ensure the QoS of latency-sensitive services while maximizing the resource utilization. Our investigation shows that, at a given load, there are multiple feasible resource configurations to meet both QoS requirement and power budget, while one of them yields the maximum throughput of best-effort applications. To find such a configuration, we establish models to accurately predict the performance and power consumption of the co-located applications. *Sturgeon* monitors the QoS periodically in order to eliminate the potential QoS violation caused by the unpredictable interference. The experimental results show that *Sturgeon* improves the throughput of best-effort applications by 24.96% compared to the state-of-the-art technique, while guaranteeing the 95%-ile latency within the QoS target.**

***Index Terms*—QoS, Improved Utilization, Power Constrained Computers**

I. Introduction

Contemporary datacenters consume tens of megawatts of power and cost $12 to $15 per watt to build them [6]. The capacity of power infrastructure and the cooling system of a datacenter constraints the available power of the datacenter. Due to the huge energy consumption and the resulting unsustainable problem, datacenters adopt commodity servers with power constraint, in order to control costs. *Latency-Sensitive* (LS) services (e.g., web search [2], and memcached [3]) are critical business workloads in datacenters. It is crucial to consistently maintain the low tail latency of LS services. While LS services often experience diurnal load pattern [5], the resource utilization and power utilization of the datacenter are low. It is cost-effective to co-locate LS services with *Best-Effort* (BE) applications that have no QoS requirement.

A large amount of prior work has been proposed to improve resource utilization while guaranteeing the QoS of LS services [9], [11], [23], [31]. However, these researches mainly focus on managing the directly controllable shared

Quan Chen is the corresponding author.

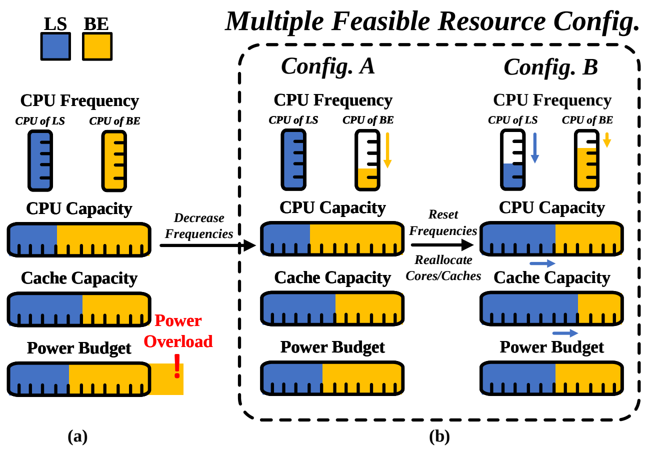

Fig. 1. (a) The co-location leads to the power overload (b) There are multiple feasible resource configurations (*Config. A* and *Config. B*). They can meet both the QoS requirement and power budget.

resources (core, cache, memory, disk and so on) but failing to consider the power constraint. Generally, when an LS service is co-located with BE applications, prior researches harvest all idle resources and allocate them to the BE applications after reserving enough resources for the LS service. Relying on some resource isolation techniques to eliminate shared resource contention, they work well in improving resource utilization, but can lead to the power overload.

To better explain this problem, Fig. 1(a) shows an example where an LS service is co-located with a BE application at low load. Observed from Fig. 1(a), when the power budget is enough to handle the peak load of the LS service, the co-location results in the power overload problem. This is mainly because BE applications may consume higher power than LS services with the same amount of resources. A straightforward solution to handle this problem is decreasing the frequencies of the cores allocated to BE applications ($Config.A$ in Fig. 1(b)). This solution is also adopted in some prior work [23]. However, this solution may result in the low throughput of the BE application due to the lower frequency.

Our investigation shows that there are multiple feasible resource configurations with different combinations of shared resources (e.g., $Config.A$ and $Config.B$) for a valid co-location. They can meet both QoS requirement and power budget but resulting in different throughput of the co-located BE applications. It is nontrivial to find the configuration yielding the maximum throughput, as it depends on the characteristic and resource preference of the BE application. Specifically, in $Config.A$, the BE application gets more cores and cache ways but lower core frequency, while in $Config.B$ it gets fewer cores and cache ways but higher core frequency. If the BE application scales well in multi-threading, $Config.A$ yields higher throughput; otherwise, $Config.B$ outperforms.

Therefore, we are motivated to design a resource management system that ensures the QoS of LS services, maximizes the throughput of BE applications, and limits the total power consumption within a given budget. Designing such a system faces three critical challenges: 1) there is no prior knowledge of the latency and power consumption of LS services and BE applications under different resource configurations. 2) there are multiple feasible configurations, but they result in different throughput of BE applications. 3) the contention on the unmanaged shared resources between the co-located applications may result in QoS violation.

To this end, we propose *Sturgeon*, a runtime system consisting of an **online performance/power predictor** and a **preference-aware resource balancer**. Sturgeon runs on each node of the datacenter and manages shared resources proactively. When the load of the LS service changes, the predictor predicts the performance and power consumption under different resource configurations using offline-trained models. For LS services, the predictor identifies whether a configuration can meet the QoS requirement and predicts the consequent power consumption. For BE applications, the predictor estimates the throughput and power consumption of a configuration. Based on the predictions, the configuration that ensures the QoS and maximizes the throughput under the given power budget can be found. Due to the unpredictable interference (e.g., contention on unmanaged resources, interrupt handling of OS), an LS service may still suffer from QoS violation at co-location. The resource balancer monitors the QoS of the LS service, harvests some resources from the BE application and reallocates them to the LS service based on the resource preference of the BE application.

Specifically, this paper makes three main contributions:

- **Comprehensive analysis of application co-location under power constraint.** The analysis shows that multiple feasible resource allocations exist for application co-location. The finding reveals the possibility to identify the allocation that maximizes the throughput of BE applications under QoS and power constraints.
- **The design of low-overhead performance and power consumption models.** The models enable precise prediction of the performance and power consumption under different loads and resource configurations.
- **The design of a compensation mechanism to eliminate the QoS violation caused by uncontrollable interference.** The mechanism guarantees the QoS of LS services at co-location.

Our real-system evaluation shows that *Sturgeon* improves the throughput of BE applications by 24.96% compared to PARTIES [11], the state-of-the-art technique, while guaranteeing 95%-ile latency of LS services within their QoS targets.

II. Background and Related Work

A. Hosting LS Services on Dedicated Datacenters

The cost of building and operating a large datacenter is expensive, on both capital and operational expenditure (e.g., power provisioning and energy costs) [6]. Accurately assessing the actual resource demands of services is significant for datacenter operators to right-size the infrastructure to lower the expenditure. While, the QoS of LS services is highly related to the revenue of many large online service providers such as Facebook and Microsoft, who usually deploy LS services on their private dedicated clusters [19], [27]. In this way, they can not only effectively guarantee the QoS, but also accurately assess the cost-effectiveness and resource requirements. Accordingly, it is also possible to plan the long-term capacity and budget accordingly. Meanwhile, the extremely high energy consumption incurred by large datacenters has increasingly raised concerns of datacenter operators on energy efficiency. For example, many datacenter operators have limitations on power capacity of IT equipment. In addition to lowering operation costs, it also eliminates the risk that unexpected changes in workload or sudden jitter in power consumption that trip circuit breaker, leading to unplanned downtime. Generally, the power capacity is set according to the server power usage at its peak, as opposed to the average usage [8]. As a result, much recent effort has been devoted to improving the resource utility and the energy efficiency of datacenters. We next elaborate some representative state-of-the-art work.

B. Improving the Resource Utilization in Datacenters

Most LS services are provided directly to the end-users characterized by diurnal pattern, as the load reaches the maximum near midday and the lowest during night. The resources become idle at the low load, leading to low resource utilization. Remarkably, the idle time constitutes a non-negligible portion in many clusters. For instance, Google web search servers have an average idleness of 30% over a 24 hour period [23]. The mainstream approach to improve the utilization is to co-locate other BE applications without QoS requirement at idle time. Co-location inevitably imposes performance impact to the LS services. Previous works mainly focus on how to lower such performance impact and generally fall into two categories: profile-based and feedback-based.

Profile-based approaches first identify whether a co-located pair is “safe” (i.e., without QoS violation) or not. Bubble series [24], [31] only allow the “safe” pair to co-locate and count on OS to manage shared resources. Profile-based approaches can always strictly guarantee the QoS of the LS service but wastes potential co-location opportunity at a fluctuating load.

Feedback-based approaches explore the online information to dynamically adjust the resource allocation between the co-located applications. Heracles [23] decreases the frequencies of cores allocated to the BE application, so as to ensure sufficient power slack to the LS service. PARTIES [11] is unaware of the power constraint and the resource configuration generated by its controller may lead to power overload. Dirigent [33] develops an LS service execution time prediction technique and adjusts resource allocation based on the prediction.

TABLE I
COMPARING STURGEON WITH PRIOR RELATED WORK.

QoS-aware Systems	Online Res. Management	Co-locate LS+BE	Power Constraint	Consider Res. Preference
Bubble		✓		
PARTIES	✓	✓		LS
Dirigent	✓	✓		LS
PowerChief	✓		✓	
Rubik	✓	✓		
Sturgeon	✓	✓	✓	LS+BE

C. *Improving the Power Efficiency in Datacenters*

When only LS services run on datacenters with power-limitations, several recent studies have discussed on the power efficiency issue. For example, EEWA [10] improves the power efficiency by using dynamic voltage and frequency scaling (DVFS) to properly tune the frequencies of the cores according to the online workload information. PowerChief [32] identifies the bottleneck of multi-stage LS service and adaptively applies boosting techniques to mitigate the latency under power constraint. When LS services and BE applications co-exist, co-location is inevitably considered for improving resource utilization. Rubik [20] uses DVFS to quickly adapt to the variability of LS services for power consumption minimization. BE applications run on a fixed frequency with the maximum throughput per watt only when LS services are idle.

Summarizing existing studies discussed above, we notice that they more or less exhibit certain limitations when performing co-location in power constrained datacenters. For example, Bubble, Heracles and Rubik may fail to harvest the opportunity of maximizing the throughput of BE applications. EEWA and PowerChief are inapplicable when co-location is considered. This motivates us to propose *Sturgeon* that allows more fine-grained co-location for higher resource utilization and energy efficiency without violating both QoS requirement and power limitation. Table I compares *Sturgeon* with previous works.

III. MOTIVATION

In this section, we investigate the challenges of performing application co-location on power constraint datacenters.

A. *Experimental Setup*

In the investigation, we use three widely-used services: *Memcached*, *Xapian*, and *Img-dnn* selected from CloudSuite [14] and Tailbench [21] as LS services and six applications: *blackshcoles (bs), facesim (fa), ferret (fe), raytrace (rt), swaptions (sp) and fluidanimate (fd)* from PARSEC [7] as the BE applications. The BE benchmarks show different characters thus cover a large spectrum of real-system applications. The details of the LS services are listed below. Table II lists the hardware specification of the experimental platform.

1) **Memcached** [3], a high-performance in-memory key-value caching system, has became the key component in cloud services to increase the concurrency. We take it from CloudSuite [14] and follows its setting. We use the scaled Twitter dataset [14] as the input of memcached.
2) **Xapian** [4], an open-source web search engine, which is widely used in popular websites and software frameworks. We take it from Tailbench [21] and follow Tailbench's setup to configure it as a leaf node. In the experiments, the search index is built from a English Wikipedia from July 2013.
3) **Img-dnn** [1], a handwriting recognition application based on OpenCV, which is widely used for image-based search, automatic image tagging and various online services. We take it from Tailbench [21] as well. The input data is randomly chosen from MNIST database.

Same to prior work [14], [21], we use 10ms as the QoS target of *memcached* and *img-dnn*, and 15ms as the QoS target of *xapian*. Note that setting different QoS targets does not affect the investigation, because *Sturgeon* works for either longer or shorter QoS targets.

TABLE II
HARDWARE SPECIFICATION

CPU	Intel Xeon E5-2630 v4
OS	Ubuntun 16.04 x86 64 (kernel 4.14)
Cores/Sockets	2 sockets, 10 cores per socket
Core frequency	1.2GHz - 2.2GHz
Hyperthreading	Enabled, 2 threads per core
L3 (Last-level) Cache	25MB, 20 ways

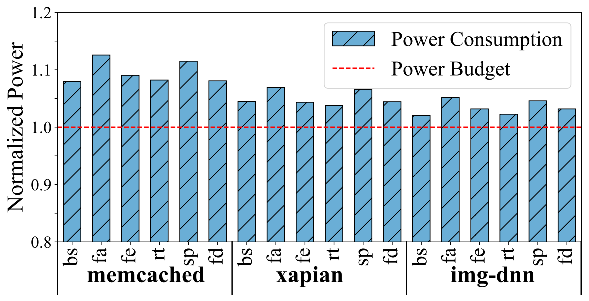

Fig. 2. The power consumption of the computer at co-location normalized to the power budget.

B. *Power Overload Problem at Co-location*

In this subsection, we seek to answer the question that whether the co-location results in the power overload problem. Specifically, we co-locate LS services and BE applications on the same machine and report the power consumption.

For all the $3 \times 6 = 18$ co-location pairs, Fig. 2 shows the power consumption normalized to the power budget, while the resource allocation ensures the QoS of LS services. We set the load of each LS service to be 20% of the peak load in this experiment, while experimenting with other loads conveys similar results. Besides, because datacenters achieve high cost-effectiveness by right-sizing the power budget based on the needs of the primary applications (i.e., the LS services) [15], [30], the power budget for a server is set to be the power consumption when the server runs the LS service at the peak load. In Fig. 2, the x-axis shows the co-location pairs (e.g., *bs* under *memecached* represents the LS service *memcached* is co-located with the BE application *blackscholes*). Observed from this figure, the actual power consumption of all the co-location pairs exceeds the power budget by 2.04% to 12.57%.

According to our measurement, at 20% of the peak load, 4 cores at 1.6GHz and 6 LLC ways are enough for *memcached*, while 4 cores at 1.8GHz and 5 LLC ways are enough for

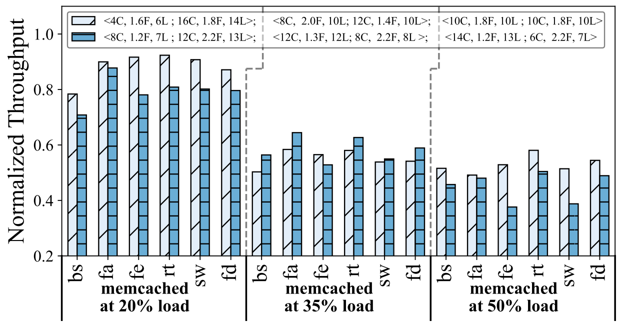

Fig. 3. The throughput of BE applications co-located with memcached under different resource configurations at different loads. <8C, 1.2F, 7L; 12C, 2.2F, 13L> indicates a resource configuration that allocates 8 cores at 1.2GHz and 7 LLC ways to the LS service (*memcached*) while allocating 12 cores at 2.2GHz and 13 LLC ways to the BE application.

xapian and *img-dnn*. Without considering power consumption, the rest cores and LLC space are allocated to the BE application and the cores allocated to the BE application run at the highest frequency 2.2GHz. The high power consumption of the cores allocated to the BE application results in the power overload problem at co-location.

C. Multiple Feasible Resource Configurations

To keep the total power consumption within the power budget, the frequencies of the cores allocated to BE applications are often scaled down. However, this strategy may result in the low throughput of the co-located BE applications.

There are multiple configurations that meet both QoS requirement and power constraint by adjusting the allocation of cores and their frequencies (we refer these configurations as "feasible"), but they bring different throughput of the BE applications. Take *memcached* as a representative LS service, Fig. 3 shows the throughput of the BE application (normalized to its solo-run throughput) at co-location with two feasible configurations. More feasible configurations for each co-location exist but are not shown in Fig. 3 due to the limited space. Since cache allocation has little influence on the overall power consumption, we allocate "just-enough" cache ways to the LS service to maintain QoS in each configuration. Other LS services and other loads show similar results.

In the figure, $<C_1, F_1, L_1; C_2, F_2, L_2>$ represents the resource configuration that allocates C_1 cores operating at frequency F_1 and L_1 LLC ways to the LS service while allocating C_2 cores operating at frequency F_2 and L_2 LLC ways to the BE application.

Observed from Fig. 3, in 13 out of the 18 co-locations, more cores result in higher throughput of BE applications, while higher core frequencies result in higher throughput in 5 co-locations. The performance gap between the two feasible configurations is large. For example, <12C, 1.3F, 12L; 8C, 2.2F, 8L> results in higher throughput of *bs* compared with <8C, 2.0F, 10L; 12C, 1.4F, 10L> when *bs* is co-located with *memcached* at 35% load (higher frequency is better for the throughput of *bs* in this case). Note that <8C, 2.0F, 10L; 12C, 1.4F, 10L> is the configuration by simply reducing the frequency of the cores allocated to BE applications. Meanwhile, <4C, 1.6F, 6L; 16C, 1.8F, 14L> results in higher throughput of *bs* compared with <8C, 1.2F, 7L; 12C, 2.2F, 13L> when *bs* is co-located with *memcached* at 20% load (more cores are better for *bs* in this case).

It is nontrivial that more cores or higher frequencies are better for the BE applications to achieve higher throughput at co-location. The feasible configuration that maximizes the throughput of BE applications while ensuring the QoS of LS services under power constraint depends on (1) *the load of the LS service*, (2) *the resource preferences of both the LS services and BE applications*, and (3) *the given power budget*.

We use Fig. 3 as an example to explain the above finding. Observed from Fig. 3, more cores are better for BE applications when the load of memcached is 20% of its peak load, but higher frequency are better when the load is 35% of the peak load (The load of LS service matters). Meanwhile, more cores result in higher throughput of *fe* but higher frequency results in higher throughput of other BE applications when the load of *memcached* is 35% of the peak load (The resource preferences of LS service and BE application matters). Generally, if the LS service is scalable but the BE application is not, allocating fewer cores operating at high frequency to the BE application may be a better choice, and vice versa. Since the power budget affects the remaining power for the BE application and there is no linear relationship between the frequency of a core and its power consumption, the power budget affects the selecting of the best resource configuration (The power budget matters).

D. Challenges

Based on the above analysis, we propose *Sturgeon*, a runtime system that efficiently finds and applies the configuration that maximizes the throughput of BE applications while ensuring the QoS of LS services under a given power budget. Specifically, *Sturgeon* faces three key problems.

- **There is no prior knowledge of the latency and power consumption of LS services and BE applications under different resource configurations.** An method is required to predict the QoS of LS services and the total power consumption at co-location with various resource configurations, and identify all feasible configurations.
- **The search space of the feasible configurations is large.** While there are a large number of feasible configurations, *Sturgeon* needs to identify the appropriate configuration that maximizes the throughput of BE applications quickly at runtime because the load of LS services changes fast.
- **Co-located applications contend for shared resources.** Some occasional contentions may result in the QoS violation of LS services. An mechanism is required to monitor the QoS of LS services and mitigates the potential QoS violation due to unpredictable interference.

IV. Design of Sturgeon

Fig. 4 presents an overview of *Sturgeon*. In a datacenter, the queries sent by users are first dispatched to each server by the cluster-level scheduler. *Sturgeon* runs on each node and manages shared resources between co-located applications.

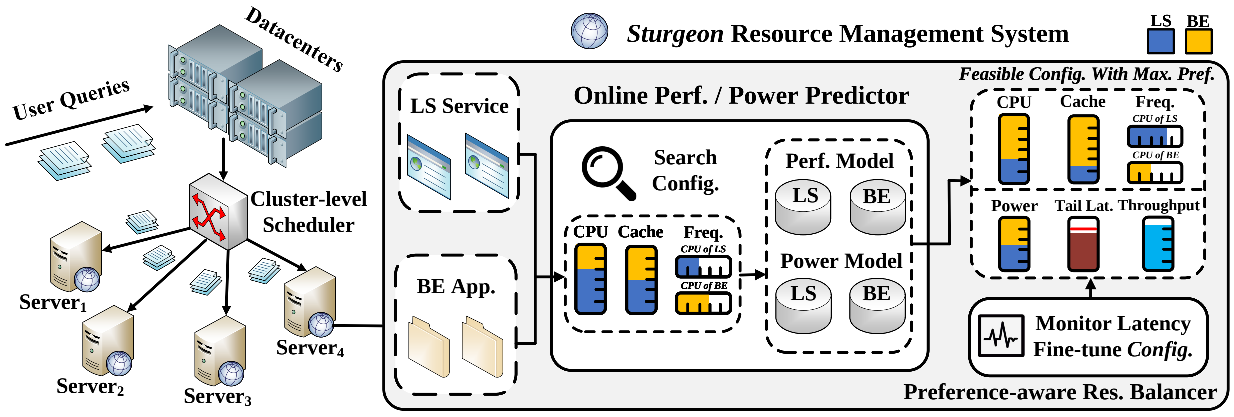

Fig. 4. The overview of *Sturgeon*. The cluster-level scheduler dispatches user queries to different servers. *Sturgeon* is deployed on each server to manage the shared resources.

Algorithm 1: Sturgeon Top-level Controller

```
1 initialize resource allocation;
2 while TRUE do
3     monitor the load and tail latency at an interval of 1s;
4     $slack \leftarrow (target - latency)\ /\ target$;
5     if $slack < \alpha$ or $slack > \beta$ then
6         find and apply a new configuration (predictor);
7         fine-tune the configuration if necessary (balancer);
```

As shown in Fig. 4, *Sturgeon* is comprised of an **online performance/power predictor** and a **preference-aware resource balancer**. The predictor adopts offline-trained per-application performance and power models to predict the QoS and throughput of LS services and BE applications, respectively. Meanwhile, the power consumption of different applications is also predicted. With the help of the predictor, *Sturgeon* is able to effectively find a feasible resource configuration that maximizes the throughput of BE applications while meeting both QoS requirement and power constraint. However, due to the contention on unmanaged resources or uncontrollable system interference, the LS service may still suffer from QoS violation when running under the resource configuration generated by the predictor. In this case, the balancer is invoked to fine-tune the resource allocation to eliminate the risk of QoS violation.

Based on the architecture, Algorithm 1 shows the top-level control flow of *Sturgeon*. The control flow starts from resource allocation initialization. Because the initial load of the LS service is unknown, a generic approach is adopted to initially allocate all resources to the LS service for QoS guaranteed. Thereafter, the number of queries and tail latency of the service are sampled every second. *Sturgeon* measures *slack* between QoS target and tail latency to decide whether to tune the resource configuration at runtime [11], [23].

Sturgeon reconfigures the resource allocation whenever the slack is too large or too small. That is, if the *slack* is smaller than the lower bound α, more resources will be allocated to maintain the QoS and if it is larger than the upper bound β, some resources will be released to BE applications. Note that the loads of LS services change quickly, the predictor must also react quickly to identify all the feasible resource configurations and find the one that maximize the BE throughput. We will detail our design of predictor in Section V. The two bounds (i.e., α and β) are set according to the sensitivity of specific applications. A larger α leads to strict protection of QoS but it is prone to raise false alarm and lower the resource utilization; a smaller β makes the system free the resources of LS services more proactively, but it imposes higher QoS violation risk at the same time. In this paper α and β are respectively set to 10% and 20% by default.

TABLE III
RESOURCE PARTITIONING AND POWER MEASUREMENT TOOLS

Item	Tool
Core	Linux's *cpuset cgroups*
LLC	Intel Cache Allocation Technology [18]
Frequency	The ACPI frequency driver
Power	Intel Running Average Power Limit (RAPL) interface [18]

After resource reconfiguration, the balancer monitors the tail latency of LS services. The balancer shall harvest and allocate more resources from BE application to the LS service if it tends to suffer from (or is suffering from) QoS violation. The challenge is on how to harvest resources in a proper way with the minimum impact on the BE application throughput. We will detail our design in Section VI.

Sturgeon is not involved in the call stack of any LS/BE application. It runs as a daemon thread on each node of a datacenter and only adjusts the resource allocation in the background using the lightweight tools in Table III. Since the adjustment takes effect in a few milliseconds [23], the runtime overhead caused by *Sturgeon* is negligible.

V. Online Performance/Power Predictor

As discussed above, it is non-trivial to identify whether a resource configuration is "feasible" and which configuration can yield the maximum throughput. Targeting at this problem, we build performance and power models of LS services and BE applications, respectively.

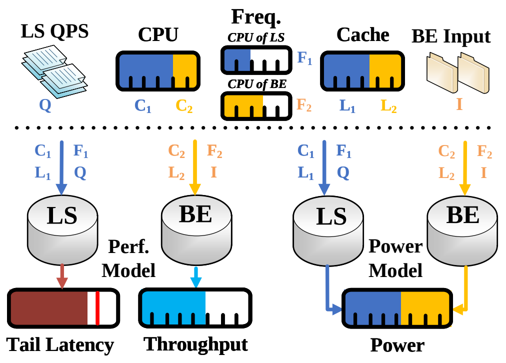

Fig. 5. The prediction of a configuration $< C_1, F_1, L_1; C_2, F_2, L_2 >$.

A. Design of the Predictor

Sturgeon builds a performance model and a power model for each application. In a dedicated cluster, it is feasible to collect the training samples of application performance and power under different resource configurations. We use the 95%-ile latency for LS services and Instruction Per Clock (IPC) for BE applications as their performance metrics, which can be obtained through the application instrumentation in a dedicated cluster. The power consumption can be obtained through some power meters [18]. It should be noted that the power consumption can spike for an elongated duration during the whole lifetime of some applications, which leads to a potential risk of the power overload. To resolve this problem, *Sturgeon* builds power models for applications based on their peak powers conservatively. In practice, some telemetry systems have been deployed in modern datacenters to collect these metrics periodically [22], [29] and hence these models can be trained offline with no runtime overhead.

When building the models, four features with high correlations are selected by Lasso regression [28] and treated as the input of the models: input size, number of cores, core frequency and LLC ways. For LS services, the query per second (QPS) is treated as the input size. For BE applications, PARSEC has defined six-level input sets with different sizes for each application and they are treated as the input size of BE applications.

Fig. 5 illustrates the prediction process. For a resource configuration $<C_1, F_1, L_1; C_2, F_2, L_2>$, the performance model of the LS service takes the number of cores C_1, core frequency F_1, LLC ways L_1 and QPS Q as input to predict the tail latency. The performance model of the BE application takes C_2, F_2, L_2 and input size I as input to predict the throughput. Similarly, the power models take these features to predict the power consumption. By checking whether the tail latency is within QoS target and overall power consumption within budget, *Sturgeon* identifies whether the resource configuration is "feasible". Besides, the throughput yielded by this configuration is also exposed.

B. Searching the Configuration with Max. Throughput

Since which resource configuration can yield the maximum throughput is uncertain, we have to search for it. However, the exhaustive search is not practical in the real system due to the large search space. In our current experimental platform with 20 cores, 10-level frequencies and 20 LLC ways, there are 20 $\times$ 10 $\times$ 20 $\times$ 10 = 40000 resource configurations to search. Our experiment shows that it averagely takes 6.4s to find the best configuration via exhaustive search, failing to satisfy the runtime scheduling interval of 1s.

An important insight in reducing search space is that the BE application gains more resources and hence higher throughput only when the LS service takes up fewer resources. Hence we just need to search those configurations with "just-enough" resources to maintain QoS. Note that the performance of applications has a positive correlation with the number of shared resources [9], [23], binary search can be applied to effectively find such resource configurations. The algorithm first fixes F_1 and L_1 at their maximum value and finds the minimum C_1 that ensures QoS (assessed by performance model) through binary search. After $C_{1,min}$ is determined, the minimum L_1 and F_1 can be determined using binary search similarly. Then C_2, L_2 can be determined by a simple subtraction according to the CPU/cache capacity. At this time, we can not directly set F_2 to be its maximum value because of the power constraint. Hence the algorithm searches the maximum possible value of F_2 using binary search without power overload (assessed by power model).

So far, we have determined one resource configuration with "just-enough" cores for the LS service such that BE application gains the most cores. However, this resource configuration is only a candidate; it is not necessarily the one with the maximum throughput as discussed in Section III. Therefore, the algorithm then gradually increases $C_{1,min}$ to search more resource configurations until F_2 reaches its maximum value such that BE application gains the highest core frequency. During the above search process, all resource configurations are regarded as candidates, among which *Sturgeon* selects the one with maximum throughput to apply.

In summary, the above algorithm based on binary search reduces the time complexity from $O(N_C \times N_F \times N_L \times N_F)$ = $O(N^4)$ to $O(N_C + (N_C + 1) \cdot (logN_F + logN_L + logN_F))$ = $O(NlogN)$, where N_C, N_F and N_L represent the capacity of the core, frequency and cache respectively. The algorithm can be extended to support multiple LS/BE applications by independently searching the configuration for each application.

C. Selecting Modeling Techniques

Obviously, the efficiency of predictors is critical to the performance of *Sturgeon*. Many different modeling techniques with different characteristics are available to build the models, e.g., Decision Tree (DT) [25], K-Nearest Neighbor (KNN) [12], Support Vector (SV) [16], Multi-layer Perceptron Neural Network (MLP) [17], Logistic Regression [13] and Linear Regression (LR) [26]. Note that the LS service performance model only needs to tell whether the QoS is violated or not and therefore it can be built via classification models. While, regression models that output a precise value can be applied to power models of both LS service and BE application as well as performance model of the BE application.

The results on the accuracy of performance and power models are reported in Fig. 6 and 7, respectively. The coefficient-

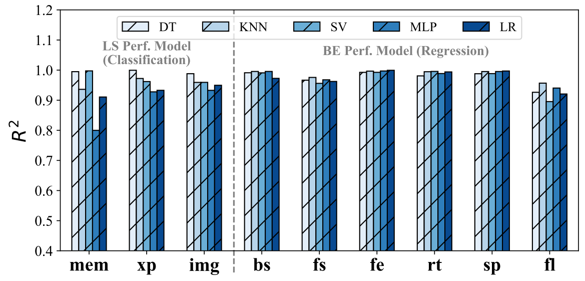

Fig. 6. The coefficient of determination R^2 of performance models, where LR refers to logistic regression and linear regression for classification model and regression model respectively.

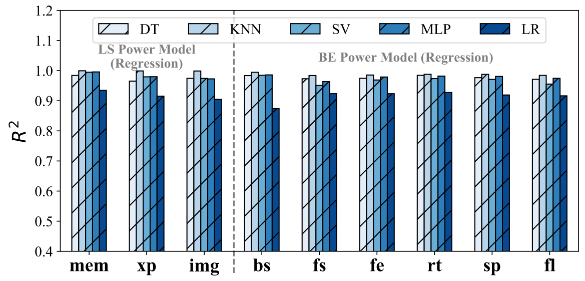

Fig. 7. The coefficient of determination R^2 of power models.

of-determination R^2 is used to evaluate the accuracy, where the value of R^2 closer to 1.0 represents higher accuracy.

From the consideration of accuracy, we notice that DT Classification is the most suitable for the performance model of LS services; KNN and MLP Regression are suitable for the performance model of BE applications; KNN Regression is the most suitable for the power model of both LS/BE applications. In fact, all offline-trained models are stored on the server and the most suitable one can be deployed according to the specific application running on the server. Besides, our experiment shows that all models make a prediction within 0.04ms.

VI. Preference-aware Resource Balancer

Running with the resource configuration generated by the predictor, the LS service may still suffer from QoS violation due to the contention on unmanaged resources or the uncontrollable system interference. In order to mitigate such problem to eliminate the potential QoS violation, we also design a preference-aware resource balancer in *Sturgeon*.

After applying a resource configuration, the balancer is invoked to monitor the tail latency of the LS service periodically. If the slack of tail latency is rather small (or even negative), the resource balancer will harvest some resources from BE application and assign them to the LS service. Each time the resource balancer tries to harvest just enough resources from the BE application so as to guarantee the QoS of LS service while minimizing the loss of throughput of the BE application.

Algorithm 2 describes how the resource balancer works. We adopt "binary-harvest" concept and set the harvesting granularity to the half of the resource quantity that the BE application owns, for each type of resources (line 2). Since there are multiple types of resources, the resource balancer will identify which type of resources to harvest can lead to

Algorithm 2: Resource Balance Algorithm

```
1  // initialize the granularity to harvest $G$;
2  $G \leftarrow 0.5 \times$ Resource that BE application owns now;
3  while $slack < \alpha$ or $slack > \beta$ do
4      // Determine the target to harvest;
5      for each type of resources $R_i$ do
6          // Use the performance/power predictors;
7          Predict the throughput if harvest it by $G$;
8          Predict if it leads to the power overload;
9      harvest $R_i$ with minimum throughput loss by $G$;
10     monitor tail latency in the next interval;
11     if excessively harvest then
12         revert $0.5 \times G$ $R_i$ to BE application;
13         Predict if it leads to the power overload;
14     $G \leftarrow 0.5 \times G$;
```

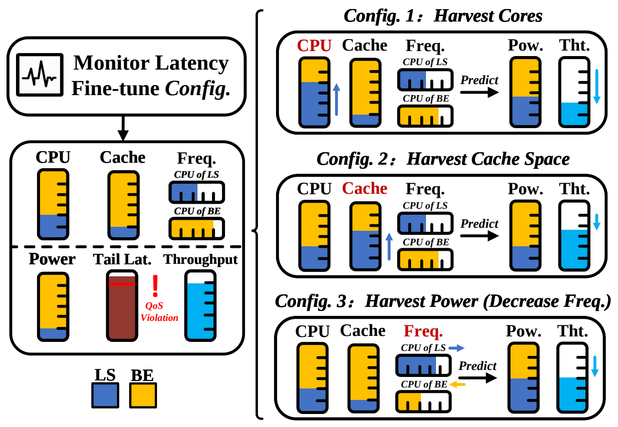

Fig. 8. When harvesting resources from the BE application, there will be there different targets to harvest: cores, cache space and power (through decreasing the frequencies of BE cores). Harvesting different type of resources leads to different throughput loss.

the minimum throughput loss (line 5-8). As shown in Fig. 8, there are three choices: 1) Harvesting cores. 2) Harvesting cache space. 3) Harvesting power, namely, decreasing the core frequency of the BE application and increasing the core frequency of the LS service. The resource balancer uses the predictors as in Section V to estimate the consequent power consumption and throughput of the three choices. Then, the one with the minimum throughput loss while meeting the power budget will be applied. Note that harvesting resources may also result in power overload. After resource reconfiguration, the resource balancer takes the tail latency in the next interval as feedback. If the latency suddenly becomes very low, it means that harvesting such an amount of resource is excessive, and the resource balancer reverts half of it back to the BE application (line 11-13); otherwise, the resource balancer keeps reducing the granularity by half until the tail latency goes into a suitable range.

VII. Evaluation

In this section, we first evaluate the performance of *Sturgeon* in improving the throughput at co-location while ensuring the QoS of LS services on power-constrained environment. Then,

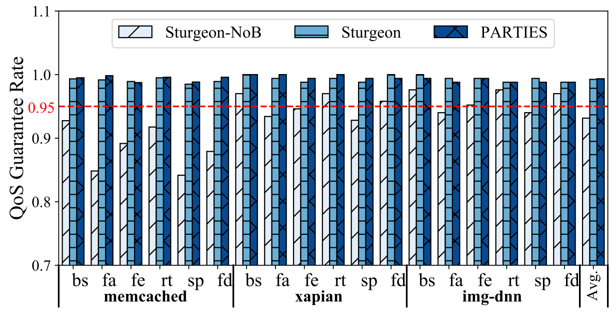

Fig. 9. The QoS guarantee rate of 18 co-location pairs, which means the number of queries completed within the QoS target in the evaluation.

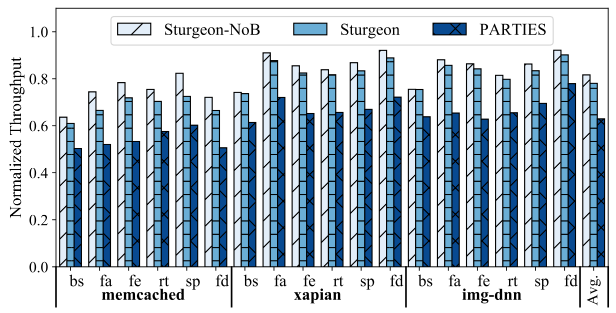

Fig. 10. The normalized throughput of BE applications of 18 co-locations.

we show the effectiveness of the resource balancer in Sturgeon followed by the discussion on the overhead of Sturgeon.

A. Experimental Setup

Since *Sturgeon* is a server-level resource management system, we evaluate it on a node of the datacenter. The specification of our experimental platform, selected LS services and BE applications can be found in Section III. In our experiment, we enable Hyper-Threading so that there are 20 logical cores on the socket and each LS service or BE application will start with 20 threads.

In previous work, PARTIES [11] and Heracles [23] are most relevant to *Sturgeon*. They have been proved to work well in improving resource utilization while guaranteeing the QoS of LS services. Although PARTIES is designed to support co-locating multiple LS services, it still works well in co-locating 2 applications and outperforms Heracles. Besides, it proactively adjusts the core frequencies of both co-located applications in its controller. We thus choose PARTIES as the comparison. In the original implementation, PARTIES adjusts one type of resources and monitors the following latency as feedback. PARTIES allocates more resources to the LS service with small slack. If the latency does not become shorter, PARTIES will choose another type of resources to allocate. It harvests resources from the LS service with large slack. If the consequent slack of this LS service becomes quite small, PARTIES will revert the resource back. Because there is no consideration of power budget in the original implementation, we enhance PARTIES: if allocating one type of resources leads to the power overload problem, PARTIES will revert it back and try another. Our experiment shows that the original implementation of PARTIES results in the power overload problem as we explained in Section III.

Because co-locating LS services with BE applications is motivated by the fluctuating load, we evaluate *Sturgeon* and PARTIES in a fluctuating input. The load of an LS service first increases from 20% to 80% and then decrease to 20% of its peak load. In our experimental platform, 60K, 3500, 3000 QPS are the peak load of *memcached*, *xapian* and *img-dnn*.

B. QoS, Throughput and Power

Fig. 9 shows the *QoS guarantee rate* of the LS services for all co-location pairs with *Sturgeon* and PARTIES. *QoS guarantee rate* shows the number of queries completed within the QoS target in the evaluation. For example, the QoS guarantee rate of co-locating *bs* with *memcached* under the management of *Sturgeon* is 99.33%, which means the latencies of 99.33% queries are within QoS target, namely, the 99%-ile latency of *memcached* in this co-location pair can be guaranteed within the QoS target. Observed from the figure, the 95%-ile latency of all co-location pairs can be guaranteed within the QoS target with *Sturgeon* and the enhanced PARTIES.

Fig. 10 shows the normalized throughput of BE applications at co-location with Sturgeon and PARTIES. The throughput of a BE application at co-location is normalized to its solo-run performance on the same hardware. Observed from the figure, *Sturgeon* outperforms PARTIES in all co-location pairs. On average, with *Sturgeon*, BE applications achieves 24.96% higher throughput than with PARTIES. Generally, PARTIES results in the low throughput of BE applications at co-location because it does not consider the resource preference of the co-located applications. The selected resource configuration is not optimal. The detailed reason will be analyzed in Section VII-D.

As mentioned in Section V, *Sturgeon* conservatively uses the peak power of an application to train the power model in order to eliminate the potential power overload problem. In the experiment, *Sturgeon* limits the power consumption within the budget for all 18 co-location pairs. Meanwhile, even if we enhance PARTIES, 7 out of the 18 co-locations still suffer from power overload. This is because the feedback-based controller in PARTIES still requires several iterations to converge when a power overload is detected. The power overload happens before the search converges.

In summary, *Sturgeon is able to effectively improve the throughput of BE applications while ensuring the QoS of LS services in the power constrained environment.*

C. Effectiveness of the Resource Balancer

To evaluate the effectiveness of the preference-aware resource balancer in eliminating QoS violation due to the contention on unmanaged resources or uncontrollable system interference, we implement a variation of *Sturgeon* that disables the resource balancer (denoted as "*Sturgeon-NoB*"). Fig. 9 and Fig. 10 also shows the performance of *Sturgeon-NoB* in guaranteeing the QoS of LS services and improving the throughput of BE applications at co-location.

Observed from Fig. 9 and Fig. 10, LS services in 12 out of the 18 co-locations suffer from QoS violation with *Sturgeon-*

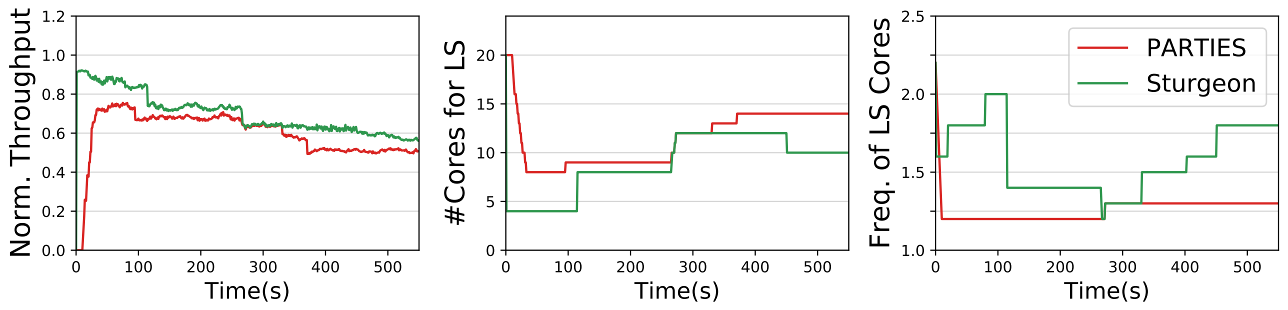

Fig. 11. The normalized throughput and resource allocations of a co-location pair (*memcached* and *raytrace*) with *Sturgeon* and PARTIES. The load of *memcached* increases from 20% to 50% of its peak load.

NoB, while BE applications achieve higher throughput in most co-locations compared to *Sturgeon*. This is because when the resource balancer is enabled, it may harvest some shared resources from the BE application and allocate them to the LS service, in order to indirectly adjust the allocation of unmanaged resources (e.g., harvesting cache space to indirectly regulate memory bandwidth because memory traffic is closely related to the cache hit rate.) or improve the performance of the LS service for handling the sudden interference. Harvesting resources from the BE application will lead to throughput loss but it is still necessary.

By enabling the preference-aware resource balancer in *Sturgeon*, the QoS of LS services can be guaranteed with the cost of only minor throughput degradation of BE applications (4.38% on average) at co-location.

The preference-aware resource balancer can eliminate the potential QoS violation due to the contention on unmanaged resources or uncontrollable system interference effectively.

D. Fluctuating Load Example

To further investigate the difference in resource allocation strategy between *Sturgeon* and PARTIES, we choose a pair (*memcached* and *raytrace*) to co-locate under the management of *Sturgeon* and PARTIES with the load increasing from 20% to 50% of the peak load. We report the throughput of the BE application, allocation of cores and the frequency settings as in Fig. 11. Other co-location pairs show similar results.

In order to guarantee the QoS, the LS service *memcached* gets all the resources in the beginning. This is an over-provisioning at 20% load and the resource management system starts to free the resources allocated to *memcached*. PARTIES continues harvesting cores as well as LLC ways from *memcached* and decreasing the frequencies and the configuration of *memcached* finally stabilizes at <8C, 1.2F, 10L> (the BE application *raytrace* is allocated 12 cores operating at 2.2GHz and 10 LLC ways at this time). Instead, *Sturgeon* decides to allocate 4 cores operating at 1.6GHz and 6 LLC ways to the *memcached* based on the online performance/power predictors, while *raytrace* is allocated 16 cores operating at 1.8GHz and 14 LLC ways at this time. Note that, *Sturgeon* tries to allocates "just-enough" LLC ways to the LS service as in Section V-B. As shown in Fig. 11, the resource allocation generated by *Sturgeon* leads to higher throughput of *raytrace* compared with the one generated by PARTIES.

When the load of *memcached* increases, it needs more resources to ensure the QoS. At about 100s, PARTIES harvests a core from the *raytrace* (<9C, 1.2F, 10L> for *memcached* at this time). However, as shown in Section III, at around 20% of the peak load, *raytrace* achieves higher throughput when it is allocated more cores. *Sturgeon* identifies this preference and decides to increase to frequencies of the cores allocated to *memcached* to maintain the QoS (from 1.6GHz to 1.8GHz and then to 2.0GHz), and the frequencies of the cores allocated to *raytrace* will therefore decrease. As we can see, the configuration generated by *Sturgeon* outperforms.

At around 300s, PARTIES continues harvesting core from *raytrace*. The number of cores allocated to *memcached* increases from 9 to 13, while there is some interference that causes PARTIES to increase the number of cores and frequencies in turn. Instead of increasing the core frequency, *Sturgeon* decides to allocate more cores to *memcached* while decreasing their frequencies (from <8C, 1.4F, 8L> to <12C, 1.3F, 12L>). At around 35% of the peak load, *raytrace* achieves higher throughput when the frequency of its cores is higher. Both *Sturgeon* and PARTIES adopt the strategy that allocating more cores to the LS service (the BE application hence gains fewer cores but higher core frequency) at this time, so the throughput is close.

From 300s to the end, PARTIES still harvests core from *raytrace* because it indeed reduces the tail latency of *memcached*, while *Sturgeon* prefers to increase the frequencies of the cores allocated to *memcached* and leaves more cores to *raytrace*. The configuration generated by *Sturgeon* outperforms because *raytrace* prefers more cores at this load.

To conclude, *Sturgeon is able to fast converge to a fairly good resource configuration by considering the system load, resource preferences of both co-runners and the given power budget comprehensively.*

E. Overhead and Discussion

The overhead of *Sturgeon* exists in the predictor that searches through a large number of configurations for the most suitable one, and the balancer that fine-tunes resource allocation for ensuring the QoS.

For the predictor, when *Sturgeon* needs to adjust the resource configuration, there are 40000 configurations to search. The exhaustive search takes 40000 × 4 × 0.04ms = 6.4s (each time 4 models will be used and predicting with each model takes 0.04ms according to our real system measurement).

While the sampling and resource adjustment interval is 1s, such a long search time is unacceptable.

Adopting the algorithm in Section V-B, *Sturgeon* needs $\log_2(20) + \log_2(10) + \log_2(20) + \log_2(10)$ = 16 times to find the configuration with minimum number of cores ($C_{1,min}$) allocated to the LS service. Based on the above configuration, *Sturgeon* increases $C_{1,min}$ gradually to find more candidates until the frequencies of cores allocated to the BE application reach the maximum in a configuration. Since there are at most 19 candidates and it takes $\log_2(10) + \log_2(20) + \log_2(10)$ = 11 times to search the specific allocation in each candidate, *Sturgeon* takes at most (16 + 11 $\times$ 19) $\times$ 4 $\times$ 0.04ms = 36ms to find the feasible configuration with the maximum throughput of BE applications.

According to our real-system measurement, *Sturgeon* uses at most 120ms to find the best configuration, including the overhead of model computation and other inherent cost (maintaining data structures and so on). While the search runs in background and the resource adjustment interval is 1s, the best configuration can be returned in time. If longer adjustment interval is adopted (e.g., 15s in Heracles [23]), the overhead of search can be ignored. Moreover, the search can also be further accelerated using multithreading.

For the resource balancer, since it only needs to estimate the performance and power consumption of three configurations (as in Fig. 8) when it is invoked, the overhead of the resource balancer is 3 $\times$ 4 $\times$ 0.04 = 0.48ms. It is negligible compared with the 1s adjustment interval.

VIII. Conclusions

This paper presents *Sturgeon*, a runtime system that manages resources between co-located applications in a power constrained environment, to ensure the QoS of LS services while maximizing the resource utilization. With the accurate online performance/power predictor, *Sturgeon* effectively finds a feasible resource configuration with the maximum throughput of BE applications. With the preference-aware resource balancer, *Sturgeon* eliminates the potential QoS violation. The experimental results show that *Sturgeon* improves the throughput of BE applications at co-location by 24.96% compared to the state-of-the-art technique, while guaranteeing 95%-ile latency within the QoS target.

Acknowledgment

This work is partially sponsored by the National R&D Program of China (No. 2018YFB1004800), the National Natural Science Foundation of China (NSFC) (61632017, 61772480, 61872240, 61832006, 61702328).

References

[1] A deep network handwriting classifier. 2019. https://github.com/xingdi-ericyuan/multi-layer-convnet.

[2] Apache Solr. 2019. http://lucene.apache.org/solr.

[3] Memcached. 2019. http://memcached.org.

[4] Xapian project. 2019. https://xapian.org.

[5] L. A. Barroso and U. Holzle. The case for energy-proportional computing. *Computer*, 40(12):33–37, 2007.

[6] L. A. Barroso and U. Hölzle. The datacenter as a computer: An introduction to the design of warehouse-scale machines. *Synthesis lectures on computer architecture*, 4(1):1–108, 2009.

[7] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The parsec benchmark suite: Characterization and architectural implications. In *PACT*, pages 72–81. ACM, 2008.

[8] H. P. Capping. Hp dynamic power capping for proliant servers. *Technology Brief,*, 2011.

[9] Q. Chen, Z. Wang, J. Leng, C. Li, W. Zheng, and M. Guo. Avalon: towards qos awareness and improved utilization through multi-resource management in datacenters. In *ICS*, pages 272–283. ACM, 2019.

[10] Q. Chen, L. Zheng, M. Guo, and Z. Huang. Eewa: Energy-efficient workload-aware task scheduling in multi-core architectures. In *IPDPSW*, pages 642–651. IEEE, 2014.

[11] S. Chen, C. Delimitrou, and J. F. Martínez. Parties: Qos-aware resource partitioning for multiple interactive services. In *ASPLOS*, pages 107–120. ACM, 2019.

[12] T. Cover and P. Hart. Nearest neighbor pattern classification. *IEEE Transactions on Information Theory*, 13(1):21–27, 1953.

[13] B. Efron. Logistic regression, survival analysis, and the kaplan-meier curve. *Publications of the American Statistical Association*, 83(402):414–425, 1988.

[14] M. Ferdman, A. Adileh, O. Kocberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. D. Popescu, A. Ailamaki, and B. Falsafi. Clearing the clouds: a study of emerging scale-out workloads on modern hardware. In *ASPLOS*, pages 37–48. ACM, 2012.

[15] S. Govindan, J. Choi, B. Urgaonkar, A. Sivasubramaniam, and A. Baldini. Statistical profiling-based techniques for effective power provisioning in data centers. In *Eurosys*, pages 317–330. ACM, 2009.

[16] S. R. Gunn et al. Support vector machines for classification and regression. *ISIS technical report*, 14(1):5–16, 1998.

[17] K. Hornik, M. Stinchcombe, and H. White. Multilayer feedforward networks are universal approximators. *Neural networks*, 2(5):359–366, 1989.

[18] Intel. Intel® 64 and ia-32 architectures software developer's manual. 2016.

[19] M. Jeon, S. Kim, S.-w. Hwang, Y. He, S. Elnikety, A. L. Cox, and S. Rixner. Predictive parallelization: Taming tail latencies in web search. In *SIGIR*, pages 253–262. ACM, 2014.

[20] H. Kasture, D. B. Bartolini, N. Beckmann, and D. Sanchez. Rubik: Fast analytical power management for latency-critical systems. In *Micro*, pages 598–610. IEEE, 2015.

[21] H. Kasture and D. Sanchez. Tailbench: a benchmark suite and evaluation methodology for latency-critical applications. In *IISWC*, pages 1–10. IEEE, 2016.

[22] J. Lee, C. Kim, K. Lin, L. Cheng, R. Govindaraju, and J. Kim. Wsmeter: A performance evaluation methodology for google's production warehouse-scale computers. In *ASPLOS*, pages 549–563. ACM, 2018.

[23] D. Lo, L. Cheng, R. Govindaraju, P. Ranganathan, and C. Kozyrakis. Heracles: Improving resource efficiency at scale. In *ISCA*, volume 43, pages 450–462. ACM, 2015.

[24] J. Mars, L. Tang, R. Hundt, K. Skadron, and M. L. Soffa. Bubble-up: Increasing utilization in modern warehouse scale computers via sensible co-locations. In *Micro*, pages 248–259. ACM, 2011.

[25] J. R. Quinlan. Induction on decision tree. *Machine Learning*, 1(1):81–106, 1986.

[26] G. A. Seber and A. J. Lee. *Linear regression analysis*, volume 329. John Wiley & Sons, 2012.

[27] A. Sriraman, A. Dhanotia, and T. F. Wenisch. Softsku: optimizing server architectures for microservice diversity@ scale. In *ISCA*, pages 513–526. ACM, 2019.

[28] R. Tibshirani. Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society: Series B (Methodological)*, 58(1):267–288, 1996.

[29] J. L. Vincent and J. Kuzma. Using platform level telemetry to reduce power consumption in a datacenter. In *Thermal Measurement, Modeling & Management Symposium*, 2015.

[30] D. Wang, C. Ren, and A. Sivasubramaniam. Virtualizing power distribution in datacenters. In *ISCA*, pages 595–606. ACM, 2013.

[31] H. Yang, A. Breslow, J. Mars, and L. Tang. Bubble-flux: Precise online qos management for increased utilization in warehouse scale computers. *ISCA*, 41(3):607–618, 2013.

[32] H. Yang, Q. Chen, M. Riaz, Z. Luan, L. Tang, and J. Mars. Powerchief: Intelligent power allocation for multi-stage applications to improve responsiveness on power constrained cmp. In *ISCA*, pages 133–146. ACM, 2017.

[33] H. Zhu and M. Erez. Dirigent: Enforcing qos for latency-critical tasks on shared multicore systems. In *ASPLOS*, pages 33–47. ACM, 2016.