Industry Practice of Configuration Auto-tuning for Cloud Applications and Services

Runzhe Wang*
Alibaba Group
China

Qinglong Wang*
Alibaba Group
China

Yuxi Hu
Alibaba Group
China

Heyuan Shi†
Central South University
China

Yuheng Shen
Tsinghua University
China

Yu Zhan
Central South University
China

Ying Fu
Ant Group
China

Zheng Liu
Alibaba Group&Zhejiang University
China

Xiaohai Shi
Alibaba Group
China

Yu Jiang
Tsinghua University
Chinal

ABSTRACT

Auto-tuning attracts increasing attention in industry practice to optimize the performance of a system with many configurable parameters. It is particularly useful for cloud applications and services since they have complex system hierarchies and intricate knob correlations. However, existing tools and algorithms rarely consider practical problems such as workload pressure control, the support for distributed deployment, and expensive time costs, etc., which are utterly important for enterprise cloud applications and services. In this work, we significantly extend an open source tuning tool – KeenTune to optimize several typical enterprise cloud applications and services. Our practice is in collaboration with enterprise users and tuning tool developers to address the aforementioned problems. Specifically, we highlight five key challenges from our experiences and provide a set of solutions accordingly. Through applying the improved tuning tool to different application scenarios, we achieve 2%-14% improvements for the performance of MySQL, OceanBase, nginx, ingress-nginx, and 5%-70% improvements for the performance of ACK cloud container service.

CCS CONCEPTS

• **Software and its engineering** → **Software performance**; *Software maintenance tools*;

KEYWORDS

Automatic tuning, configuration optimization, software performance, machine learning

*The first and second authors contribute equally.
†Heyuan Shi is the corresponding author.

ESEC/FSE '22, November 14–18, 2022, Singapore, Singapore

ACM ISBN 978-1-4503-9413-0/22/11…$15.00
https://doi.org/10.1145/3540250.3558962

ACM Reference Format:
Runzhe Wang, Qinglong Wang, Yuxi Hu, Heyuan Shi, Yuheng Shen, Yu Zhan, Ying Fu, Zheng Liu, Xiaohai Shi, and Yu Jiang. 2022. Industry Practice of Configuration Auto-tuning for Cloud Applications and Services. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '22), November 14–18, 2022, Singapore, Singapore.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3540250.3558962

1 INTRODUCTION

Auto-tuning is increasingly adopted to improve the performance or quality of service (QoS) of software by automatically tuning the software parameters with machine learning algorithms. Since auto-tuning spares engineers with the requirements for expert knowledge and manual work, it is particularly useful for optimizing complex software such as the Linux kernel, database, and compiler, etc. There have been many research efforts on developing tuning tools from both the industry and academia. For example, A-Tune [15], ResTune [41], and NNI [23] are developed by enterprise developers for optimizing operating systems, database applications, and machine learning models, respectively. In academic research, existing studies mainly focus on developing or applying different tuning algorithms, including the typical Gaussian Process (GP) based Bayesian optimization (BO) [33, 36], Tree-structured Parzen Estimator (TPE) [7], HORD [16], and HyperBand [20], etc.

Despite these endeavors, it remains rather challenging to apply auto-tuning in real production environments. The challenges mainly come from the gaps between design considerations and practical concerns in the following perspectives. First, tuning cloud application and service usually involves high-dimensional parameters across different software layers. For example, it is common to confront an application configuration file that contains dozens of parameters. Furthermore, a Linux kernel can be configured by thousands of parameters with continuous, discrete, or categorical forms. This often results in extremely large search spaces that take existing algorithms for unacceptable tuning time. Second, it is critical to configure the values of software parameters carefully in production. Improper settings may lead to unexpected or abnormal behaviors such as crashes, reboot, hung, or memory leaks, etc. However, existing work rarely discuss the safety of parameter values. Third, while

many enterprise applications (e.g.,distributed database) are served in clusters, existing tuning tools or algorithms pay little attention to the support of distributed tuning. These gaps are the sources of many obstacles for developers to apply auto-tuning in practice.

In this work, we introduce several extensions to KeenTune [17] – an open source auto-tuning tool to optimize the performance of various cloud applications and services. We demonstrate the generality of our development by using KeenTune to adjust kernel and application parameters of MySQL [28], OceanBase [25], and nginx [24]. These applications are deployed in real cloud environments supported by the Elastic Compute Service (ECS) [4] and Alibaba Cloud Container Service for Kubernetes (ACK) [5]. We summarize from the experiences in applying KeenTune to these scenarios to highlight the problems that are associated to the aforementioned gaps. Specifically, these problems include improper parameter ranges, insufficient benchmark workload, lack of distributed tuning, expensive tuning time, and redundant repetitive tuning. To address these problems, we develop different supporting features in KeenTune. In summary, we implement distributed and continuous tuning in KeenTune to significantly improve its scale of deployment. We also introduce a parameter selection step to improve overall tuning efficiency. With these extensions, we observe performance improvements from 2% to 14% for the MySQL, OceanBase, and nginx, and improvements from 5% to 70% for ACK container services that support four typical workloads, i.e., high CPU load, high network throughput, low network latency, and high IO throughput. The main contributions are summarized as follows:

- We summarize and categorize several main challenges in applying auto-tuning to cloud applications and services.
- We provide and implement different solutions to address these challenges with an open source tuning tool.
- We demonstrate the effectiveness of the enhanced tuning tool with several typical cloud applications and services.

The rest of this paper is organized as follows. In Section 2, we introduce typical tuning tools and algorithms, and particularly highlight KeenTune. Then we provide an overview of the tuning procedure as well as the target applications and services evaluated in this work in Section 3. We summarize the main challenges in applying auto-tuning in practice and introduce our solutions in Section 4. The experimental results are presented and discussed in Section 5. At last, we discuss several interesting observations in Section 6 and conclude the paper in Section 7.

2 BACKGROUND

In this section, we briefly introduce auto-tuning followed by several typical tuning tools and algorithms. Then we introduce an open-source tuning tool named as KeenTune with more details and its application in industry practice.

2.1 Auto-tuning

It is common for a cloud application to have different components in its complex software hierarchy that individually or jointly affect the performance. For example, for a web server or any network intensive application, besides its configuration parameters that directly affect the performance, the parameters with the prefix *net* among the thousands of parameters in the Linux kernel [21] can also be particularly influential. For a database application such as MySQL, there are dozens of parameters that control the CPU, memory usage, and I/O resource, which jointly determine its QoS and resource utilization performance [41]. Furthermore, a typical auto-tuning practice usually optimizes the performance of an application on a standard benchmark that simulates real-world workloads. In such cases, the domains of tunable parameters also include parameters of the selected benchmark. In fact, an improper configuration of benchmark parameters may severely limit the overall tuning performance. As such, benchmark parameters also need to be carefully specified. The diversity of these different software parameters and their intricate interaction make it rather challenging to estimate the effect of tuning any parameter. This makes auto-tuning more appealing than manual tuning, which is often tedious and fruitless .

There are different tuning tools [15, 17, 23] provided by enterprise developers and many tuning algorithms [1, 13, 31, 41] designed by academia and industry efforts. In particular, Bayesian optimization (BO) is widely adopted by these studies as the backbone for developing improved algorithms or serving different application scenarios. A generic BO problem usually requires a surrogate to be built for predicting the tuning results of different parameter configurations. This surrogate is trained with certain initially collected data samples and then used for guiding the search to find better parameter configurations more efficiently. There have been various studies that adapt this framework to optimize database applications [1, 13, 41], operating systems [15, 17, 31], application placement [6], scheduling policy [29], and machine learning models [23]. There are also several open source tools, such as NNI [23], A-Tune [15], and KeenTune [17], that support different popular BO algorithms, including Gaussian Process (GP) based BO, TPE, and HyperBand, etc. Reinforce learning (RL) is also studied for optimizing database applications [19, 40]. However, its application scope is more limited due to the expensive cost of running RL models.

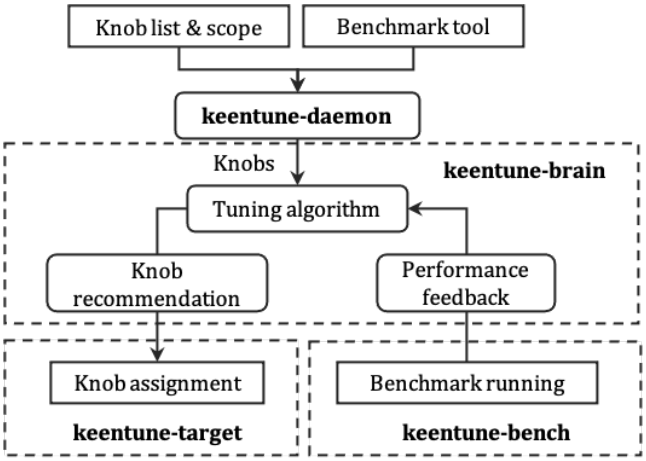

Figure 1: The framework of KeenTune.

2.2 KeenTune

Despite the availability and effectiveness of many previously mentioned tools, their practicality is limited since they are either devoted to specific tuning tasks or not suitable for distributed environments. Among aforementioned tools, KeenTune [17] has a lightweight and flexible framework that is relatively easier to be extended to support general system tuning tasks and distributed tuning. As shown in Figure 1, KeenTune consists of four components, i.e., keentune-daemon, keentune-brain, keentune-bench, and keentune-target. Since tunable parameters are described as knobs in KeenTune, in the following, unless otherwise specified, we follow its taxonomy and use knob and parameter interchangeably.

Keentune-daemon is a daemon process that acts as the central control component for scheduling and monitoring other components to complete a tuning task. Its core functionality is to schedule other components by communicating to them with the intermediate tuning data. Keentune-brain, as the name suggests, is the component that runs a selected tuning algorithms. KeenTune-brain supports several typical tuning algorithms, e.g., random search, grid search, TPE, and customization of new BO algorithms. It communicates with other components to specify the tunable knobs and their ranges of values, then apply the selected tuning algorithm to recommend configurations. The recommended configurations are further sent to keentune-target, which is responsible for applying the recommended configurations in the target environment. It also reads and keeps default knob values to ensure the recovery of the target environment. Keentune-bench is the last component in the tuning pipeline and calls different benchmark tools with build-in scripts to evaluate the recommendation. After the evaluation, it sends the results to keentune-brain as feedback to the tuning algorithm. KeenTune bench can be customized to support different benchmark tools by defining configuration files that specifying the set of tunable knobs and their ranges of values. In the following, we highlight several important properties of KeenTune that make it suitable to be extended for production use.

Target independence: The generic framework of KeenTune allows it to be flexibly extended for various tuning needs. For example, KeenTune can be configured to support application knobs, kernel knobs, and even their combinations. KeenTune also allows flexible customization of different benchmarks and tuning tasks. In comparison, it is more difficult to adapt a tuning tool like ResTune [41], which is developed for tuning database applications with many careful designs, to other tasks.

Portability: The components of KeenTune have good compatibility. For example, keentune-daemon and keentune-brain only need few dependencies to support popular tuning and machine learning libraries, e.g., Hyperopt [8], Scikit-learn [30], and SHAP [22], etc. These libraries support most existing tuning algorithms and explainable machine learning algorithms. This enables KeenTune to easily expand its arsenal with state-of-the-art tuning algorithms to improve the tuning effectiveness and efficiency. Moreover, KeenTune is compatible with multiple OS distributions, including CentOS [10], Debian [12], Anolis OS [26], Alibaba cloud Linux [3], and Ubuntu [9]. As for keentune-target and keentune-bench, since they need to evaluate the target application, they are often deployed in the same environment as the target application. This makes it possible to deploy keentune-target and keentune-bench independently from keentune-daemon and keentune-brain. In KeenTune, these components communicate only through network connections. This further allow keentune-target and keentune-bench to have negligible impact to the deployed environment.

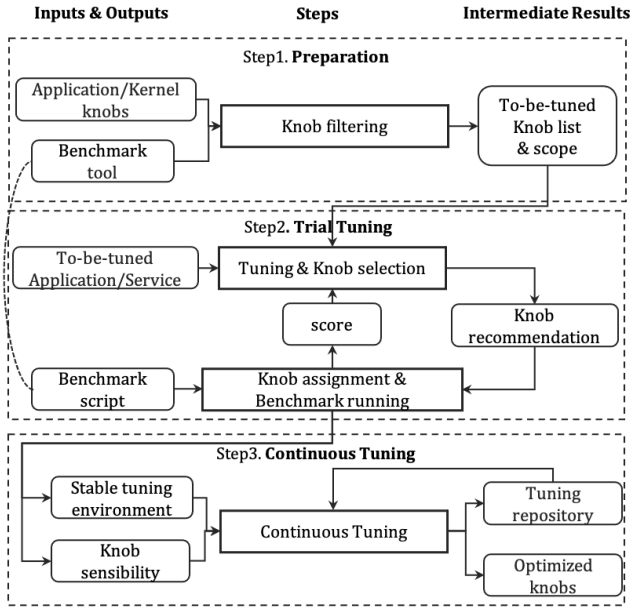

Figure 2: An overview of the auto-tuning procedure with KeenTune.

Community supports: KeenTune is an open source software that is part of the openAnolis [27] community. This community is organized by the Alibaba Group and their business partners. It provides constant support and maintenance to KeenTune. The supports include extension of new tuning algorithms from academia, expert knowledge from industry practice, and various different application scenarios. In particular, the broad adoption of KeenTune for optimizing internal enterprise applications and services in the Alibaba Group provides KeenTune a long-term ingestion of industry experiences.

3 TUNING PROCEDURE AND TARGETS

In this section, we first describe the typical procedure of applying KeenTune for auto-tuning. An overview of the procedure is provided in Figure 2. Then we introduce several applications and services that are mainly investigated in this work for evaluating the effectiveness of KeenTune.

3.1 Tuning Procedure

As shown in Figure 2, there are three main steps in a typical tuning pipeline. Specifically, in the first step of preparation, there are several requirements that are necessary to initiate a tuning task:

- Environment consistency. It is important to first validate that the tuning environment is consistent with the actual production environment. For example, both environments use the same type of cloud server instances.

Table 1: A summary of the target cloud applications and services.

Type	App./Serv.	Env.	OS	Tuning Metric	Bench.	App. Knobs	Kernel Knobs
Database	Mysql	ECS	Anolis OS 8.2	tpm-c	TPC-C	my.cnf	-
	OceanBase	ECS	Anolis OS 8.2	query response time	TPC-H	-	/proc/sys/*
Web server	nginx	ECS	Alibaba Cloud Linux 3	requests/sec (https long)	wrk	nginx.conf	-
	nginx	ECS	Alibaba Cloud Linux 3	requests/sec (https short)			-
	ingress-nginx	ACK	Alibaba Cloud Linux 3	requests/sec (https long)			/proc/sys/*
	ingress-nginx	ACK	Alibaba Cloud Linux 3	requests/sec (https short)			/proc/sys/*
Container	ACK	ACK	Alibaba Cloud Linux 3	total QPS	stressapptest	-	/proc/sys/*
				avgIO	stressapptest		
				requests/sec	netperf		
				latency	netperf		

- Target knobs. We need to predetermine a set of knobs and their tunable ranges of values. Specifically, these knobs can be kernel knobs, application knobs, or even their combinations. This avoids the occurrence of invalid configurations that may lead to unexpected behaviors.
- Evaluation Benchmark. It is necessary to specify a benchmark with a default configuration of its own parameters that are able to generate sufficient test workload. The evaluation results generated by the benchmark are expected to be sensitive to different configurations of the target knobs. This allows a tuning algorithm to obtain useful information for updating its surrogate and search strategy.

After the preparation step, the pipeline enters the trial tuning step. In this step, the target software runs with randomly configured knob values for multiple trials. The multiple trials help verify the consistency of the tuning environment before initiating a long-term tuning. Moreover, it examines if the specified ranges of target knobs may trigger unintended software abnormal states.

In the final step of continuous tuning, KeenTune allows tuning with multiple instances to accelerate the overall tuning process. After the tuning stops, the recommended knob configurations of all tuning instances are collected. The optimal configuration is then selected from these configurations and taken as the final output. The configurations generated by different instances are saved and further used to accelerate the tuning of similar tasks in the future.

3.2 Tuning Targets

In Table 1, we introduce several popular cloud applications and services that have adopted KeenTune for performance optimization. We select these example applications and services, including databases (MySQL and OceanBase), web servers (nginx and ingress-nginx), and container service (ACK), since they are crucial for providing various cloud services and supporting the business values of cloud vendors. The massive deployment of these applications and services also makes it much easier to accumulate and refine tuning profiles. The profiles that are widely verified are particularly useful for optimizing similar applications and services. Also, these applications and services provide KeenTune a comprehensive set of configurable knobs across different domains. Furthermore, the diversity of the deployment environments, including different compute services (ECS and ACK) and OS distributions (the cloud-specific distribution of Alibaba Cloud Linux and the general distribution of Anolis OS), drives KeenTune to better acknowledge the characteristics of application and services deployed in different environments. As such, the representative knobs and deployment environments improve the generality of KeenTune to cover a wide spectrum of cloud applications and services.

In the following, we introduce the characteristics of aforementioned applications and services, along with their corresponding tuning objectives and requirements.

3.2.1 Database. The throughput of a database application is a key metric for evaluating the tuning results. This metric is commonly measured by standard benchmark tools such as TPC-C [37] and TPC-H [38]. These tools measure the transactions processed per-minute and the queries processed per-hour, respectively. As shown in the "Database" row in Table 1, we use TPC-C and TPC-H to evaluate the throughput of a database application or service.

As previously introduced in Section 3.1, we need to specify the domains of knobs to be tuned for the target application. Specifically, we consider two scenarios where a MySQL [28] application is deployed with the Anolis OS 8.2[1] in an Elastic Compute service (ECS) instance [4], and a OceanBase[2] application is deployed with Alibaba Cloud Linux 3[3] in a ECS cluster. For MySQL, its application knobs can be obtained from `/etc/my.cnf`. For OceanBase, in the production environment, its application knobs are already manually tuned by experts. As such, we mainly focus on tuning the knobs of the underlying operating system. Specifically, we use `sysctl` [21] – a software utility of Unix-like OS that reads and modifies the attributes of the system – to dynamically adjust the kernel knobs listed in `/proc/sys/`.

[1] Anolis OS [26] is an open source Linux distribution maintained by the OpenAnolis community [27]. We use Anolis OS 8.2, which is based on the long-term support (LTS) Linux kernel version of 5.10 and has enhanced features and better optimization.
[2] OceanBase [25] is a native distributed relational database developed by Alibaba group and Ant group.
[3] Alibaba Cloud Linux 3 [3] is an open source Linux distribution and one of the downstream Linux distributions of Anolis OS, it is specifically used for the ECS environment.

Table 2: An overview of challenges

Step	Challenge	MySQL	OceanBase	nginx	ingress-nginx	ACK
Environment preparation	Unreasonable knobs range	-	✓	-	✓	✓
Trial Tuning	Invalid benchmark workload	-	-	✓	✓	✓
Trial Tuning	No support for distributed software tuning	-	✓	-	-	-
Trial Tuning	Unacceptable tuning time cost	✓	✓	-	-	-
Continuous Tuning	Inefficient repetitive tuning	-	-	-	-	✓

Table 3: Problems caused by improper knob values.

Knobs	Problem
fs.file-max, fs.nr_open	The risk of unable to open file if the value is too small.
net.ipv6.*	Irrelevant knobs in ipv4.
net.ipv4.forward.*	The risk of network interruption if the value is modified.
vm.extra_free_kbytes	More direct reclaim and severely jammed if the value is too small.
vm.min_free_kbytes	The risk of deadlock if the value is too small.
vm.max_map_count	The risk of out of memory if the value is not zero.
vm.overcommit_memory, vm.overcommit_ratio	The risk of "OSError: Cannot allocate memory."
vm.dirty_background_ratio, vm.dirty_ratio	Ineffective setting if vm.dirty_background_ratio is larger than vm.dirty_ratio.

3.2.2 Web Server. We select nginx [24] and ingress-nginx[4] as two representative web server applications. The tuning task is designed as maximizing the throughput of a web server for processing the workloads generated by a standard benchmark wrk [39]. Similar to the scenarios in tuning database applications, we consider two cases where a nginx instance is deployed in an ECS instance and an ingress-nginx instance is deployed in an ACK instance. Both the ECS and ACK instances use Alibaba Cloud Linux 3. For nginx, its application knobs are defined in `/etc/nginx.conf`. For ingress-nginx [18], we specify the domains of knobs to include both application and kernel knobs. We expect this larger domain of knobs to bring more room for better tuning results.

3.2.3 Container Service. Container service is one of the most important products for cloud vendors. As such, it is crucial to maintain the performance of applications that are deployed in containers. For this scenario, we consider four typical cases where applications are required to have high CPU loading, high input/output (IO) throughput, high network throughput, or low network latency while they are deployed in ACK instances. We use netperf [14] to simulate workloads and evaluate an ACK instance for its performance of unidirectional throughput and end-to-end latency. To evaluate the CPU load and IO throughput of ACK instances, we further use stressapptest [35] to simulate a high CPU load test case that maximizes randomized traffics from processor and I/O to memory. For all aforementioned cases, we specify the same set of tunable knobs from `/proc/sys/`.

4 CHALLENGES AND SOLUTIONS

In Table 2, we summarize several major challenges that are crucial for applying tuning algorithms in practice. While some challenges can be tackled from the algorithmic perspective, many of these challenges need to addressed through introducing new mechanisms. In the following, we describe each challenge and our proposed solution in sequence.

4.1 Tuning Preparation

Recall that during the preparation step, we need to first specify the domain of target knobs. The domain consists of a list of selected knobs and their ranges of configurable values. However, given a large number of candidate knobs and their heterogeneous configurable values, it is rather challenging to define a reasonable domain of configurable knobs.

Challenge 1: Improper knob ranges. A common practice for selecting knobs and their value ranges is to refer to the user manual or source code. However, this may leave too many knob candidates. For example, there can be hundreds of different tunable knobs for different OS distributions [3, 26, 27]. It is just too expensive to include all possible knobs since the resulting vast search space will significantly lower the tuning efficiency. Furthermore, we observe that if some knobs have their values assigned out of their safe ranges, they may cause severe problems such as network interruption and system crash. In Table 3, we list several

Table 4: The prefix of selected kernel knobs for knowledge-based knob selection.

Domain	Count	Effect
fs.*	48	IO, File System
kernel.*	208	CPU scheduling
net.*	634	network
vm.*	48	memory

[4]Ingress-nginx uses an ingress controller for Kubernetes using nginx as a reverse proxy and load balancer [18]

example problems that can be caused by improper settings of knob values. In practice, it is very challenging to unify the safe ranges for knobs of virtual machines with different specifications. For example, the values of `worker_rlimit_nofile` and `worker_connections` of nginx cannot exceed the `ulimit` setting of the underlying system. Moreover, more context-switch costs may occur if the value of `worker_processes` of nginx is larger than the number of CPUs of its deployed environment.

Solution 1: Knowledge-based knob selection.

From our experiences, we often observe that there are only a small number of knobs that dominate the tuning results. For example, dozens of application knobs are commonly more influential than hundreds of kernel knobs for affecting the performance. This is because the former knobs affect the application more directly. In the meantime, the generality of kernel knobs make it relatively easier to specify the candidates and their ranges of values. In comparison, the values of application knobs need to be carefully configured for different cases. As such, if all knobs are treated equally, more attention will be spent for tuning kernel knobs while it is the application knobs that need more subtle tuning efforts.

In Table 4, we provide several categories of knobs that are characterized by their prefix. These knobs are selected based on Anolis OS 8.2 and have good generality to be reused for various tuning tasks. Specifically, we first identify a set of knobs and their ranges of values according to the actual tuning requirements. Then we refer to Table 3 to eliminate invalid knobs or range configurations. At last, we use a trial tuning step (introduced in the following) to validate safety of the remained knob specifications.

4.2 Trial Tuning

Besides the validating the domain of configurable knobs specified in the previous step, trial tuning also performs a quick estimation of the tuning effectiveness and costs. Specifically, in practice, there are cases where different knob values cause negligible difference to the tuning results. This indicates the existence of some bottleneck that invalidates the tuning efforts. Also, a few iterations of trial tuning provide an estimate of the overall tuning time. If the time cost of each iteration is prohibitively high due to some complicated or expensive operations (e.g., the data loading and compiling processes performed in TPC-C), the total number of iterations may need to be adjusted according to the time budget. It is important to note that, for tuning tasks with high-dimensional search space, the BO algorithms integrated in existing tuning tools cannot efficiently control the tuning process. As such, certain external mechanism is needed to ensure the converge of tuning.

Challenge 2: Invalid benchmark workload. As previously mentioned in Section 3, the tuning results are usually represented by the performance measured by a selected benchmark. Standard benchmark tools such as sysbench [2] and wrk [39] often have their own configurable parameters. It is often assumed that the selected benchmark has already been configured to generate sufficient workloads to approach the full capacity of the underlying environment. If the benchmark is not configured properly, the target system or application may not reach its top performance.

We take sysbench as an example for demonstrating the aforementioned effect in Figure 3. Specifically, sysbench is used to evaluate the results of optimizing the QPS performance of a MySQL application and its configurable parameters are specified as `threads`, `thread-stack-size`, `tables`, and `table-size`. As shown in Figure 3 that, by limiting the maximal `threads` to different numbers, the optimal QPS values that can be reached are rather different. It is clear that a lower maximum significantly limits the tuning performance. Furthermore, the results shown in Figure 3 demonstrate that it is not always necessary to configure the `threads` parameter of sysbench with the maximal value. The tuning results obtained with `threads=40` and `threads=80` are very close at the end of tuning.

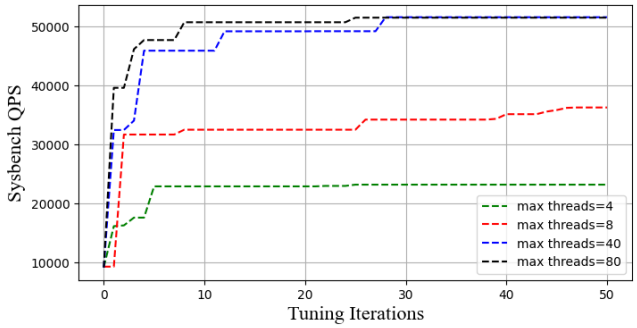

Figure 3: The QPS performance obtained by setting the `threads` parameter of sysbench with different maximums.

Solution 2: Automatic pressure control. In the previous example, we use sysbench to demonstrate the necessity of performing pressure control. While it appears that the threads parameter simply needs to be set to the maximum without careful configuration, this does not hold for all benchmarks. To generate sufficient workloads, we add to KeenTune a module that automatically adjusts the configurations of benchmark parameters before auto-tuning. In this module, we again need to first specify the ranges for benchmark parameters. Since there are often a limited number of benchmark parameters, it is much easier to specify their ranges of configurable values. In case when there are benchmark parameters with unknown tuning ranges, we need to automatically find the allowed extreme values by detecting the underlying environment. Specifically, before tuning, keentune-brain first sends to keentune-bench commands for detecting different resources. The commands include finding the maximum number of threads or the maximum memory size of the deployed environment. Keentune-brain then receives the detected values returned by keentune-bench and sets the ranges of configurable values accordingly.

Challenge 3: Lack of support for distributed tuning. In Figure 4, we present an example architecture of an OceanBase cluster which consists of three zones while each zone holds three OB servers. For distributed database, data is usually duplicated and stored in servers in different zones for safety purpose. As such, the OB Server 1 in Figure 4 across three zones usually have the same specification and configuration. Accordingly, when applying the recommended knob configurations in this cluster, one needs to send the same recommendation (e.g., Configuration A) to the servers with the same ID (e.g., ID=1) across different zones. KeenTune does not naturally support these cases where the support for multi-target

knob configuration are needed. To support these cases, we need to deploy a keentune-target component in each server and have a keentune-bench component in the resource pool of tenant to operate benchmark tools.

Solution 3: Extension for distributed tuning. In Figure 4, we present an example case where OB servers with the same ID are deployed in different zones. To introduce to KeenTune a distributed tuning feature, we need to define a tuning group for the target application or service with different IP addresses and the same knob setting. The knobs associated with this group are renamed by keentune-daemon to add to their knob names a suffix, e.g., knob-name@group1 and knob-name@group2. Then keentune-daemon is able to assign the same knob setting to different IP addresses with the same group ID. We introduce this renaming scheme since it extends KeenTune to support distributed tuning while causing minimal changes to other components.

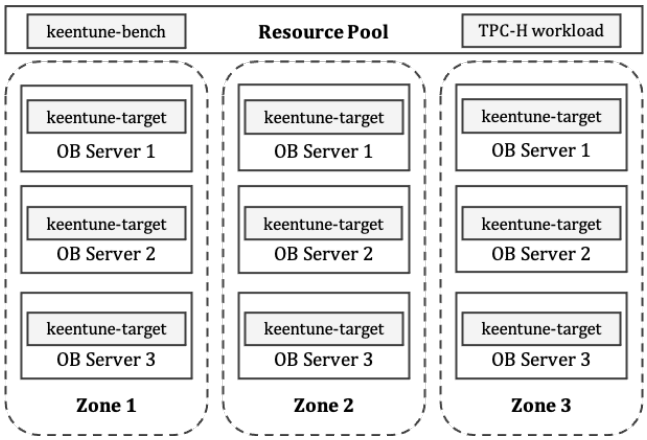

Figure 4: The distributed tuning example of applying KeenTune for tuning OceanBase clusters.

Challenge 4: Unacceptable tuning time cost. For a tuning task, the time efficiency is an important metric for evaluating the adopted tuning tool. The overall tuning time consists of the time for running tuning algorithms and the time for running benchmark tools. There are certain benchmarks (e.g., wrk, benchmarksql) that have long execution time due to expensive operations such as data loading and compiling. It is usually difficult or irrational to adjust the execution time of benchmark tools. If the total budge of tuning time is restricted, it directly limits the number of tuning iterations for identifying better knob configurations. Therefore, to improve the time efficiency for arbitrary tuning algorithms supported by KeenTune, we aim at reducing the tuning iterations by only focusing on knobs that are more influential to tuning results.

Solution 4: Sensitive knob identification. Besides manually screening knobs as introduced in Section 4.1, we propose to quantify knob sensitivities through data-driven approaches. Several tuning tools [1, 13] have applied different methods to estimate knob sensitivity. Specifically, in iTuned [13], the sensitivity of each knob is represented by its weight coefficient of a linear regression model, which is different to the surrogate model and independently trained with some historical tuning data samples. This approach assumes linear dependency between knobs and the tuning results. In OtterTune [1], the sensitivity of a knob is extracted from the GP surrogate used in a BO algorithm. The sensitivity values of different knobs are estimated by decomposing the total variance of the tuning results to all knobs. This approach does not rely on the linear assumption, but can be too expensive when the number of knobs is large. Recent work [22, 32] on explainable machine learning has proposed various methods to assign attribution scores to each input features and are efficient for high dimensional feature space. We integrate KeenTune with SHAP [22] to use the initially collected trial samples to estimate knob importance. Specifically, we first train a machine learning model, e.g., XGBoost [11], with the same dataset used to initiate the surrogate model. Then we use SHAP to obtain the estimated knob sensitivity values. By specifying a threshold for knob sensitivity, the knob space can be significantly reduced since many knobs have trivial contributions to the tuning results.

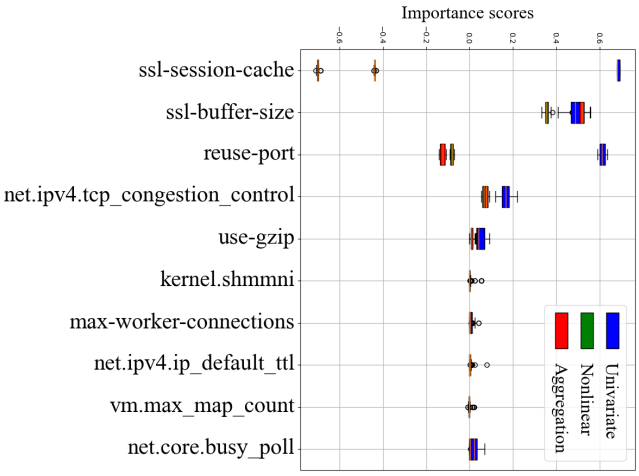

Figure 5: Knob sensitivity scores for tuning ingress-nginx with HTTPS short connection.

In the following, we conduct a case study to demonstrate the effect of performing auto-tuning with or without knob selection. To make sure that the search space has a sufficiently high dimensionality, we apply different tuning algorithms to jointly tune the kernel and application knobs. The target application used in this case study is ingress-nginx (https short) and the tuning objective is to maximize the throughput. For each tuning algorithm, we mark the cases of with or without knob sensitivity identification with * as shown in Figure 6. In the vertical axis of Figure 6, we present the relative loss metric values of all tuning algorithms, This metric is calculated by comparing the throughput obtained an algorithm at a certain step to the maximum throughput among the tuning results obtained by all tuning algorithms.

Figure 6 show that knob selection indeed improves tuning since all algorithms with knob selection obtain better results in comparison with themselves runs in solo. Among three algorithms, TPE achieves the best performance (lowest relative loss) with or without knob selection. However, knob selection contribute to different algorithms differently. Specifically, the improvements obtained by

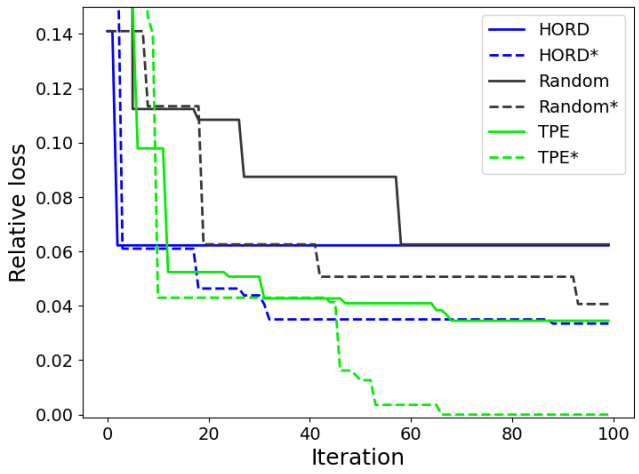

Figure 6: The tuning process for ingress-nginx with HTTPS short connection.

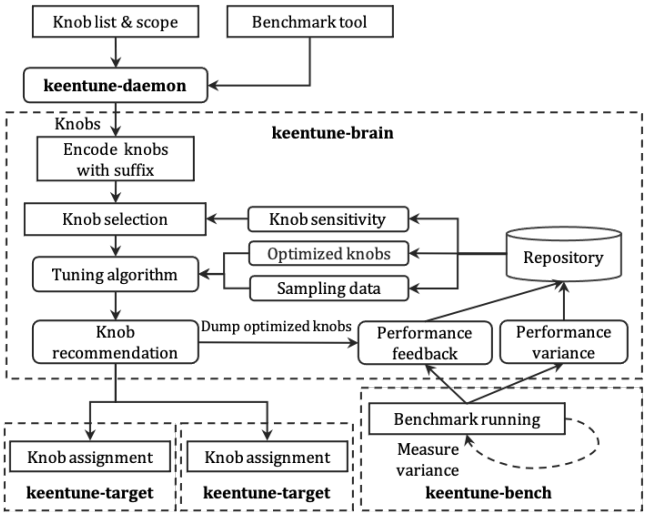

Figure 7: The continuous tuning framework of KeenTune.

TPE* and HORD* are much more significant that the improvements obtained by Random*. The larger improvements obtained by TPE and HORD, which are two representative BO algorithms, may be attributed to the reduced search space. The reduced search space amplifies the benefit of avoiding unnecessary tuning efforts spent for less sensitive knobs.

To compare different knob selection methods, in Figure 5, we demonstrate the sensitivity values assigned by three different methods. Due to space limit, we only show the top 10 knobs selected by different methods. Specifically, we calculate univariate sensitivity and nonlinear knob sensitivity. The univariate sensitivity is obtained by estimating the mutual information between each knob and the tuning results. This method does not rely on the linear assumption, hence is more general than the aforementioned linear estimation method. The nonlinear knob sensitivity is obtained with XGBoost [11] and SHAP [22], as introduced previously. The aggregation sensitivity is obtained by linearly combining the univariate and nonlinear sensitivity. In Figure 5, the box plot is obtained from the results of 30 trials of running each knob selection method. We observe that the univariate and nonlinear methods nearly identify the same set of knobs that dominate the tuning results. In such cases, different methods cross-validate each other and their consistency makes the result more trustworthy. Also, the sensitivities in Figure 5 are more stable and less ambiguous (thin boxes). This implies that the knob space in this setting is relatively easy to be learned. As such, the searching radius of TPE can be more concentrated, hence leading to its better performance as discussed above.

4.3 Continuous Tuning

In previous sections, we mainly introduce different challenges that can be mitigated by different solutions individually. To further improve the tuning efficiency, we summarize from experiences to further highlight a higher level issue that affect the overall and long-term tuning efficiency.

Challenge 5: Repetitive tuning. During a typical auto-tuning process, there may be many repetitive behaviors that severely lower the overall tuning efficiency and limit the final tuning performance. More specifically, the repetitive behaviors can be classified into the following three categories:

(1) Run benchmarks repetitively. In production environments, it is common to confront environmental fluctuations. These fluctuations cause a benchmark tool to report inaccurate results and mislead the tuning algorithm. As a result, t, we often find it necessary to run a benchmark tool for multiple times to neutralize the fluctuations and use the averaged result as the feedback to the tuning algorithm.
(2) Generate the same evaluation result repetitively. This repetitive effect is usually caused by a tuning algorithm that either over-explores less sensitive knobs or over-exploits a narrow vicinity in the knob space. In the former case, the algorithm loses informative signals to guide its search. As a result, the algorithm acts similarly as random search. In the latter case, the algorithm only recommends knob configurations are too close to each other. Therefore, the recommendations bring little difference to the evaluation results.
(3) Rerun similar tasks repetitively. In practice, it is common to run a tuning task from zero even though a similar task has just finished. These operations are inefficient in terms of re-sampling data samples and re-exploring unfavorable regions in the knob value space.

Solution 5: Extension for continuous tuning. To alleviate the impact from the aforementioned repetitive behaviors, we extend KeenTune with continuous tuning support by introducing a repository that keeps historical intermediate and final tuning data. The continuous tuning framework that we implement in KeenTune is shown in Figure 7.

Specifically, in our extension, we store the following three types of data in a separate repository:

Table 5: Improvement of auto-tuning in Alibaba cloud application and service.

App./Serv.	Tuning Metric	Unit	Pre-tuned Knobs	Tuning Knobs	Baseline	Auto-tuning	Improvement
MySQL	tpm-c	count	-	app.	8417	9606	14.1%
OceanBase	query response time	second	app.	kernel	128.1	125.53	2.1%
nginx	requests/sec (https long)	count	-	app.	426263	452117	6.0%
nginx	requests/sec (https short)	count	-	app.	14037	15482	10.3%
ingress-nginx	requests/sec (https long)	count	app.	app.+kernel	91692	96638	5.4%
ingress-nginx	requests/sec (https short)	count	app.	app.+kernel	24829	27474	10.65%
ACK	totalQPS	count	-	kernel	33220	35022	5.42%
	avgIO	MB/s	-	kernel	199	340	70.85%
	requests/sec	count	-	kernel	2124	2800	31.83%
	latency	ms	-	kernel	144	135	6.67%

(1) Lists of sensitive knobs. We effectively prevent a tuning algorithm from tuning less relevant knobs by maintaining and refining lists of sensitive knobs for different cloud applications or services. This directly reduces the repetitions caused by the above mentioned over-exploration problem.
(2) Profiles of recommended configurations. Before starting a tuning task, we first search in the repository for profiles obtained from similar tasks. Once an existing profile is matched, it is used to warm start the tuning algorithm. Through this approach, we can accelerate the overall tuning process.
(3) Historical data samples. The historical data can be used in an offline manner to accumulate or refine knob sensitivity values that are collected previously. These data samples are also used for training or refining the surrogate model used in a selected tuning algorithm.

We further modify keentune-bench to temporarily keep intermediate benchmark results. Then the mean of these intermediate results is used to provide a more stable feedback to the tuning algorithm. The maintenance of the intermediate tuning results also opens the opportunity of finer control of the tuning process. For example, the variance of the intermediate tuning results can be used to trigger an early stopping mechanism that aborts the tuning task before wasting too many iterations.

5 RESULTS

In this section, we demonstrate the results of applying our enhanced KeenTune to optimize different applications and services in Table 5. Specifically, for MySQL, auto-tuning improved the average number of transactions per minute (tpm-c) by 14.1%. For OceanBase applications, recall that the application configurations are specified by expert knowledge. From the third row of Table 5, we observe an improvement of 2.1% although the configurable knobs are limited to kernel knobs. This encourages the use of auto-tuning as an auxiliary step to further boost the performance that has already been optimized by human experts. As shown in the fourth and fifth rows in Table 5, for nginx servers, we were able to improve the the number of requests processed per minute increases by 5.4% to 10.6% by applying auto-tuning. In the case of tuning ingress-nginx, we jointly tuned both the kernel and nginx knobs. We improved the number of requests per second by 5.4% and 10.65% with https-long and https-short service workloads, respectively. For ACK services, we applied auto-tuning to four types of tasks as shown in the bottom four rows of Table 5. We specify different typical metrics for these tasks and observed improvements of 5.42%, 70.85%, 31.83% and 6.67%, respectively. These evaluation results clearly demonstrates the effectiveness of applying auto-tuning for improving the performance of various cloud applications and services.

Note that for enterprise services such as ACK, it is possible to reproduce this performance improvement (to some extent) by transferring the profile obtained from similar cases. For example, given a task of tuning the kernel knobs of an ACK instance that works in a high CPU load scenario and holds a JVM which runs a java application (SPECjbb [34]), by applying the same profile obtained from this task, we achieved an increase of 3.8% for the throughput (max jops) and an increase of 2% for the response time. This may be explained by the fact that both JVM and high CPU load cases use the throughput and response time as key metrics for evaluating the operating performance. The similarity between the tuning metrics somehow shapes knob values in a similar way that benefits both tuning tasks. These results highlight the possibility of significantly expanding some local tuning tasks to a larger scale since the tuning profiles can be transferred to similar tasks. We will provide more discussion of this effect in Section 5.

In Table 5, we mark in the fourth column if a tuning task is initiated with pre-tuned knobs. We notice from the results that, auto-tuning generally brings more improvement if an application is configured with the default knob setting than the case when an application is configured with some pre-tuned knob setting. Specifically, as shown in the first two rows in Table 5, the performance of MySQL (with the default configuration) is improved by 14.1% while the performance of OceanBase (with the pre-tuned configurations) is improved by only 2%. This is as expected since the pre-tuned configurations are often provided by experts. These configurations can be considered as some sufficiently good initial points for the search of even better configurations. Therefore, the room for improvement is usually limited. On one hand, these results indicate that it is necessary to take auto-tuning as the "last mile" of tuning to pursuit the optimal performance. On the other hand, the use of pre-tuned configurations in turn benefits auto-tuning for significantly improving the overall tuning efficiency and effectiveness.

Another interesting observation is that, in general, tuning application knobs introduces higher improvement than tuning kernel knobs. One explanation is that application knobs directly affect the performance of the target application or service. Also, there are commonly much less application knobs than kernel knobs. As a result, the tuning efforts can be more concentrated for exploring a much smaller search space. In the meantime, the kernel knobs have better generality since their configurations are easier to be transferred to different tasks. Therefore, a tuning practice should not ignore kernel knobs due to their inferior contributions to the tuning results. Moreover, for certain applications, their application knobs are often dependant on kernel knobs. For example, the value of *worker-connections*, which is a knob of nginx, is influenced by both *ulimit* and *file-max*. The latter is a kernel knob that limits the maximum number of file handles that an application can use.

6 LESSONS LEARNED

From our intensive experiments, we gain many insights of applying auto-tuning in the production environments. We summarize these insights into the following three aspects and discuss the inspiration for guiding other research and our future work.

Explainabilily is crucial for auto-tuning users. It is well known that the black-box nature of many machine learning models is the one of the key obstacles for large-scale application. This black-box nature of machine learning models imposes threats to the safety of applying auto-tuning in industry environments. Therefore, it is crucial to provide certain explanations for the recommended knob configuration. The explanations can be examined for safety purpose. They may also provide knowledge that is not known by experts. Besides, it is necessary to have a rule-based validation step before applying the recommended knob configuration. In this work, we introduce to KeenTune a knob sensitivity identification mechanism that adopts explainable machine learning algorithms to provide certain explainability.

No ideal environments for simplified benchmarks. In most research work and enterprise tools developed for auto-tuning, there are often two assumptions about the tuning environments. Specifically, it is usually assumed that a selected benchmark tool can simulate sufficiently representative workloads. Also, it is commonly assumed to have a stable environment for running the selected benchmark tool. However, both assumptions are difficult to satisfy in practice. For the first assumption, one can rely on domain experts to advise the selection of benchmark tools and examine their parameter configurations. For the second assumption, as previously introduced in Section 4.3, it can be inevitable to confront fluctuations that severely affect the reliability of the benchmark results in a real production environment. The unreliable benchmark results then mislead the tuning algorithm and compromise the entire tuning task. The second problem is more general yet is rarely considered by existing methods. In this work, we propose to trade efficiency for accuracy and alleviate this problem by averaging the benchmark results from multiple runs. Our future work will focus on developing more comprehensive solutions to this problem.

No free lunch for tuning algorithms The characteristics of system and application configurations require careful design or selection of tuning algorithms. Specifically, the domain of configurable values usually contains continuous, discrete, and categorical values. For a certain knob, its configurable values may not be distributed evenly. Moreover, a rigorous requirement of the overall tuning time may rule out certain tuning algorithms or prefer a greedy tuning strategy. Therefore, the property of an adopted tuning algorithm needs to match the characteristics of a specific auto-tuning task.

An auto-tuning tool needs re-usability and extensibility. As previously introduced in Section 4.3, both the lists of knobs and their recommended configurations can be reused for similar tuning tasks or applications. This is particularly important for industry applications since reusable tuning profiles can reduce the costs of auto-tuning significantly. However, since industry applications often have rapid updates and changes in their deployed environments, it is also very challenging to generating tuning profiles that scale to different tuning tasks. Moreover, the intensive research efforts in developing new tuning algorithms make it necessary for an auto-tuning tool to have a framework that supports standardized interface and easy extension of newly developed algorithms.

7 CONCLUSION

In this paper, we conduct an industry practice of auto-tuning on four typical cloud applications and a public cloud service. We identify the main challenges in deploying an example auto-tuning tool, i.e., KeenTune, with enterprise Linux kernels and provide different solutions to address these challenges. Specifically, we first reduce the number of candidate knobs and validate their ranges of values with domain knowledge. Then we highlight the problem of using insufficient benchmark workloads and propose a pressure control mechanism that automatically adjusts benchmark parameters to provide sufficient testing pressure. We also propose a sensitive knob identification mechanism to reduce the tuning time by focusing on more influential knobs. Furthermore, we add the support for distributed application tuning and continuous tuning. The experimental results show the effectiveness of proposed solutions, which are integrated into a continuous integration system to improve the performance of cloud applications and services.

This paper tries to reduce the gap between the functionality provided by current tuning tools and the practical requirements of cloud applications and services. We aim to draw the attention of cloud users and developers to practical auto-tuning and will pursuit the following goals in our future work: 1) Extend the repository of expert knowledge to provide effective management and reuse of tuning profiles. 2) Explore and combine dynamic and static tuning methods to improve the scalability of historical tuning experiences. 3) Investigate auto-tuning for cloud applications and services deployed with heterogeneous devices to build a cross-platform full-stack auto-tuning tool. 4) Develop energy-efficient benchmark selection methods by utilizing historical records of tuning process and resource utilization.

REFERENCES

[1] Dana Van Aken, Andrew Pavlo, Geoffrey J. Gordon, and Bohan Zhang. 2017. Automatic Database Management System Tuning Through Large-scale Machine Learning. In *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*, Semih

Salihoglu, Wenchao Zhou, Rada Chirkova, Jun Yang, and Dan Suciu (Eds.). ACM, 1009–1024. https://doi.org/10.1145/3035918.3064029
[2] Alexey Kopytov. 2017. *sysbench*. https://github.com/akopytov/sysbench
[3] Alibaba. 2019. Alibaba Cloud Linux. https://github.com/alibaba/cloud-kernel.
[4] Alibaba Group. 2015. *ECS*. https://www.alibabacloud.com/product/ecs
[5] Alibaba Group. 2018. *ACK*. https://www.alibabacloud.com/product/kubernetes
[6] Omid Alipourfard, Hongqiang Harry Liu, Jianshu Chen, Shivaram Venkataraman, Minlan Yu, and Ming Zhang. 2017. CherryPick: Adaptively Unearthing the Best Cloud Configurations for Big Data Analytics. In *NSDI*. USENIX Association, 469–482.
[7] James Bergstra, Rémi Bardenet, Yoshua Bengio, and Balázs Kégl. 2011. Algorithms for Hyper-Parameter Optimization. In *NIPS*. 2546–2554.
[8] James Bergstra, Daniel Yamins, and David D. Cox. 2013. Making a Science of Model Search: Hyperparameter Optimization in Hundreds of Dimensions for Vision Architectures. In *ICML (1) (JMLR Workshop and Conference Proceedings, Vol. 28)*. JMLR.org, 115–123.
[9] Canonical. 2004. The ubuntu Project. https://ubuntu.com.
[10] Centos. 2019. The CentOS Project. https://www.centos.org.
[11] Tianqi Chen and Carlos Guestrin. 2016. XGBoost: A Scalable Tree Boosting System. In *KDD*. ACM, 785–794.
[12] Debian. 1997. The Debian Project. https://www.debian.org.
[13] Songyun Duan, Vamsidhar Thummala, and Shivnath Babu. 2009. Tuning Database Configuration Parameters with iTuned. *Proc. VLDB Endow.* 2, 1 (2009), 1246–1257. https://doi.org/10.14778/1687627.1687767
[14] Hewlett Packard. 2005. *netperf*. https://github.com/HewlettPackard/netperf
[15] Huawei. 2021. *A-Tune*. https://gitee.com/openeuler/A-Tune
[16] Ilija Ilievski, Taimoor Akhtar, Jiashi Feng, and Christine Annette Shoemaker. 2017. Efficient Hyperparameter Optimization for Deep Learning Algorithms Using Deterministic RBF Surrogates. In *AAAI*. AAAI Press, 822–829.
[17] KeenTune. 2021. *KeenTune*. https://openanolis.cn/sig/keentune
[18] Kubernetes. 2016. *ingress-nginx*. https://github.com/kubernetes/ingress-nginx
[19] Guoliang Li, Xuanhe Zhou, Shifu Li, and Bo Gao. 2019. QTune: A Query-Aware Database Tuning System with Deep Reinforcement Learning. *Proc. VLDB Endow.* 12, 12 (2019), 2118–2130. https://doi.org/10.14778/3352063.3352129
[20] Lisha Li, Kevin G. Jamieson, Giulia DeSalvo, Afshin Rostamizadeh, and Ameet Talwalkar. 2017. Hyperband: Bandit-Based Configuration Evaluation for Hyperparameter Optimization. In *ICLR (Poster)*. OpenReview.net.
[21] Linux. 2020. *sysctl*. https://man7.org/linux/man-pages/man8/sysctl.8.html
[22] Scott M. Lundberg and Su-In Lee. 2017. A Unified Approach to Interpreting Model Predictions. In *NIPS*. 4765–4774.
[23] Microsoft. 2021. *Neural Network Intelligence*. https://github.com/microsoft/nni
[24] NGINX, Inc. 2020. *nginx*. https://nginx.org/en/
[25] OceanBase. 2010. *oceanbase*. https://github.com/oceanbase/oceanbase
[26] OpenAnolis. 2020. Anolis OS. https://openanolis.cn/anolisos.
[27] OpenAnolis. 2020. OpenAnolis community. https://openanolis.cn.
[28] Oracle Corporation. 2010. *MySQL 8.0 Reference Manual*. https://www.mysql.com
[29] Tirthak Patel and Devesh Tiwari. 2020. CLITE: Efficient and QoS-Aware Co-Location of Multiple Latency-Critical Jobs for Warehouse Scale Computers. In *HPCA*. IEEE, 193–206.
[30] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. 2011. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research* 12 (2011), 2825–2830.
[31] Alex Renda, Yishen Chen, Charith Mendis, and Michael Carbin. 2020. DiffTune: Optimizing CPU Simulator Parameters with Learned Differentiable Surrogates. In *MICRO*. IEEE, 442–455.
[32] Marco Túlio Ribeiro, Sameer Singh, and Carlos Guestrin. 2016. "Why Should I Trust You?": Explaining the Predictions of Any Classifier. (2016), 1135–1144.
[33] Jasper Snoek, Hugo Larochelle, and Ryan P. Adams. 2012. Practical Bayesian Optimization of Machine Learning Algorithms. In *NIPS*. 2960–2968.
[34] Standard Performance Evaluation Corporation. 2015. *SPECjbb2015*. https://www.spec.org/jbb2015
[35] Stressapptest. 2017. *stressapp*. https://github.com/stressapptest/stressapptest
[36] Kevin Swersky, Jasper Snoek, and Ryan Prescott Adams. 2013. Multi-Task Bayesian Optimization. In *NIPS*. 2004–2012.
[37] TPC. 1992. *TPC-C*. https://www.tpc.org/tpcc
[38] TPC. 2007. *TPC-H*. https://www.tpc.org/tpch
[39] Will Glozer. 2013. *wrk*. https://github.com/wg/wrk
[40] Ji Zhang, Yu Liu, Ke Zhou, Guoliang Li, Zhili Xiao, Bin Cheng, Jiashu Xing, Yangtao Wang, Tianheng Cheng, Li Liu, Minwei Ran, and Zekang Li. 2019. An End-to-End Automatic Cloud Database Tuning System Using Deep Reinforcement Learning. In *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*, Peter A. Boncz, Stefan Manegold, Anastasia Ailamaki, Amol Deshpande, and Tim Kraska (Eds.). ACM, 415–432. https://doi.org/10.1145/3299869.3300085
[41] Xinyi Zhang, Hong Wu, Zhuo Chang, Shuowei Jin, Jian Tan, Feifei Li, Tieying Zhang, and Bin Cui. 2021. ResTune: Resource Oriented Tuning Boosted by Meta-Learning for Cloud Databases. In *SIGMOD Conference*. ACM, 2102–2114.