Reducing the Tail Latency of Microservices Applications via Optimal Configuration Tuning

G. Somashekar, A. Suresh, S. Tyagi, V. Dhyani, K, Donkada, A. Pradhan, A. Gandhi
PACE Lab, Stony Brook University, Stony Brook, NY, USA
{gsomashekar, amsuresh, satyagi, vdhyani, kdonkada, anpradhan, anshul}@cs.stonybrook.edu

***Abstract*—The microservice architecture is an architectural style for designing applications that supports a collection of fine-grained and loosely-coupled services, called microservices, enabling independent development and deployment. An undesirable complexity that results from this style is the large state space of possibly inter-dependent configuration parameters (of the constituent microservices) which have to be tuned to improve application performance.**

This paper investigates optimization algorithms to address the problem of configuration tuning of microservices applications. To address the critical issue of large state space, practical dimensionality reduction strategies are developed based on available system characteristics. The evaluation of the optimization algorithms and dimensionality reduction techniques across three popular benchmarking applications highlights the importance of configuration tuning to reduce tail latency (by as much as 46%). A detailed analysis of the efficacy of different dimensionality reduction techniques in capturing the most important parameters is performed using ANOVA techniques. Results show that the right combination of optimization algorithms and dimensionality reduction can provide substantial latency improvements by identifying the right subset of parameters to tune, reducing the search space by as much as 83%.

***Index Terms*—ML for systems, microservices, configuration tuning, optimization, dimensionality reduction, tail latency**

I. Introduction

The emerging microservice architecture allows applications to be decomposed into different, interacting modules, each of which can then be independently managed for agility, scalability, and fault isolation [1], [2]. Each module or microservice typically implements a single business capability with inter-microservice communication enabled via Application Programming Interfaces (APIs). Applications deployed using the microservice architecture thus enable flexible software development.

The microservice architecture is especially well suited for designing online, customer-facing applications where performance and availability are paramount [3], [4]. For example, an online application can be deployed as front-end microservices (e.g., Nginx), a set of microservices that implement the logic of the application each of which can have their own database (e.g., MongoDB) and caching (e.g., Memcached) microservices. Consequently, an application can have *numerous* microservices. Given the benefits of the modular architecture, microservices architecture is widely replacing existing deployments implemented using monolithic or multi-tier architectures at Amazon, Netflix, and Twitter [1].

Despite the benefits of the microservice architecture, a specific challenge that this distributed deployment poses is that of ***tuning the configuration parameters of the constituent microservices***. A change in configuration parameters can substantially impact application performance, motivating our investigation of configuration tuning. For example, sweeping over the valid range of values for the *worker_process* parameter of the nginx [5] microservice in the social networking application [1] (while keeping the rest of the parameters at default) can provide up to 13% improvement in latency over the default configuration. However, *joint optimization* of all the application parameters can provide 46% latency improvement over the default configuration (see Section IV). On the other hand, setting a sub-optimal (but still valid) value for the *worker_process* parameter of the nginx while setting the rest of the parameters to the optimal values can *deteriorate* the performance by up to 100× compared to the default configuration. Tuning the parameters of monolithic or N-tier applications for maximizing performance is already a difficult task [2], [6]–[10] (see Section II). With microservice applications, configuration tuning is especially complicated owing to the following challenges:

- ***Very large configuration space.*** Microservices applications have hundreds to thousands of interacting microservices that each have several parameters that can be configured [11]. Frameworks that aid microservices development, such as Apache Thrift [12] and gRPC [13], introduce additional parameters that impact application performance. These parameters can take values that are discrete, continuous, or categorical, complicating attempts to optimize their values.
- ***Inter-dependent parameters.*** The parameter setting of a microservice can influence the optimal value of a different parameter of the *same* microservice. As a result, the numerous parameters of a given microservice cannot be independently optimized (see Section IV). For example, for MongoDB, a low value of the cache size parameter can amplify the number of concurrent read transactions, making it difficult to independently tune the latter parameter [14].
- ***Dependency between parameters of different microservices.*** The dependency between parameter values extends beyond a single microservice; parameters of upstream services are often dependent on the parameter settings of downstream services [8]. For example, the thread pool size of a microservice may dictate how many concurrent requests are sent to the downstream microservice.
- ***Interference among colocated microservices.*** Microservices, typically deployed as containers, can be colocated on the same physical host. Due to potential resource contention, the resource configuration of a microservice can impact the performance of other colocated microservices. For example, the cache size of two colocated caching microservices should not be set independently as they share the host's memory resources.
- ***Non-linear relationship between microservices parameters and performance.*** Application performance need not be monotonically or linearly dependent on parameter values, making it difficult to determine optimal configuration parameter settings.

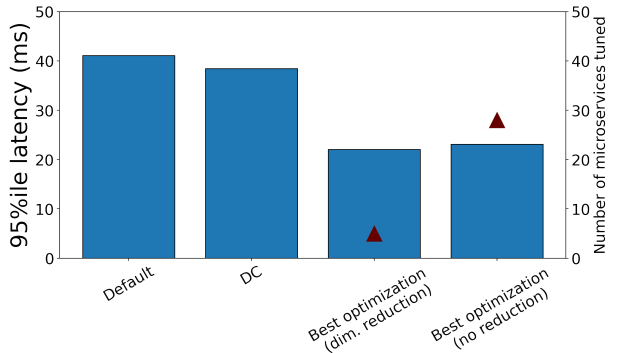


Fig. 1: Comparison of 95^{th} percentile of latency for the social networking application [1] under (i) default configuration values (Default), (ii) the configuration used by DeathStarBench benchmark developers [15] (DC), and the configuration found by the best optimization technique among those we explored (iii) with dimensionality reduction (considering only a subset of microservices for tuning) and (iv) without any reduction (tuning all microservices). The right y-axis shows the number of microservices tuned for (iii) and (iv).

The thread pool size parameter is a classic example whereby a low value results in under-utilization of the CPU and a very high value results in contention for network sockets or CPU resources [2].

There is little prior work on the specific problem of configuration tuning of microservices, and that work relies on empirically exploring the configuration setting of only specific parameters of just stateless microservices [2] . There are, however, prior works that focus on optimizing the configuration of individual services [9], [16], but as explained above, the dependencies between the parameters of microservices makes it infeasible to optimize them in isolation.

This paper explores the problem of configuration tuning of microservices applications. To address the problem of inter-dependent parameters, we consider *joint optimization* of the parameter space. We conduct an *extensive experimental investigation* of six black-box optimization algorithms with the goal of minimizing the tail latency of a given microservice application deployment. As shown in Figure 1, the best optimization algorithm can significantly improve application tail latency (95^{th} percentile), by as much 46% and 43%, compared to the default configuration setting and the suggested configuration in prior work [1], respectively. We also find that combining different algorithms can result in efficient solutions that quickly (with few exploration points) explore the state space and provide significant latency improvements.

To address the key challenge of a large configuration space when jointly tuning microservices applications, we investigate various *dimensionality reduction* approaches to identify a subset of microservices that are most likely to impact end-to-end application latency. As illustrated by the two rightmost bars in Figure 1, by employing dimensionality reduction, we can achieve significant improvement in tail latency while only having to tune about 18% of all microservices. In fact, within a given budget on the number of iterations of the optimization algorithm, optimizing with dimensionality reduction can further improve tail latency (by about 6.5%) compared to when optimizing without any dimensionality reduction since the search space is reduced, thereby aiding the optimization.

Our investigation of different algorithms reveals that the optimal choice is application-dependent. While the hybrid algorithm we devise performs best for the social networking and the train ticket applications, Bayesian optimization performs best for media microservices application, in terms of tail latency reduction. In terms of time taken to run the algorithm, dynamically dimensioned search (DDS) performs the best.

This paper makes the following contributions:

- We formulate configuration tuning of microservices application as a *joint optimization problem*, making it amenable to optimization algorithms. Contrary to serial tuning, this provides an opportunity for the optimization algorithms to learn the dependencies among parameters of the same microservices and across microservices.
- We implement a *framework* [17] to experimentally explore and evaluate the configuration space of parameters for microservices. The framework is fully automated and can be integrated with any optimization technique.
- We implement six different representative optimization algorithms using open-sourced libraries and compare their efficacy in choosing the best configuration with respect to minimizing the application tail latency. To assess the optimization algorithms' applicability in practice, we also analyze their convergence and overhead.
- Based on our analysis of different algorithms, we design and evaluate an efficient hybrid algorithm that combines the strengths of different algorithms. In particular, the algorithm quickly explores the state space using heuristic-based search and then uses the results of this search to initialize a model-based search algorithm.
- For scalability, we investigate different approaches, including *critical path* and *variability tracking*, to reduce the overhead of optimization by limiting the set of microservices whose parameters will be configured. We analyze the ability of these different techniques to capture the most important parameters that impact application tail latency.
- We use functional analysis of variance (fANOVA) [18] to find the most important parameters and analyze the values assigned to them by different optimization algorithms. We also examine the change in service time of individual microservices to assess the impact of optimization on different request types in the workload.

II. Background and Prior Work

Microservice architecture is a style of architecture where the application is implemented as a set of loosely coupled services, called microservices. This shift in design of distributed applications requires revisiting some of the problems that have been addressed for monolithic and N-tier architectures. Resource management [19] and bottleneck mitigation [20] for microservices applications are some of the problems that have garnered significant attention from the research and development community. We take a different approach to improving the performance of distributed applications implemented using a microservices architecture. In particular, we tackle the problem of tuning the parameters of microservices to improve the performance metric of interest (e.g., tail latency or throughput).

The general problem of tuning parameters of computer systems has gained significant attention [9], [10], [16]. However, these works do not focus on the specific problem of microservices configuration where several, inter-dependent parameter configurations have to be tuned. The one extensive prior work on configuration tuning of microservices that we are aware of is by Sriraman et al. [2]. In this work, the authors explore the tuning of a small subset of microservices parameters, limited to thread pool size and threading model. However, the state space of configuration parameters for microservices is very large, as discussed in Section I, and hence a more comprehensive investigation of parameters is required for performance optimization of microservices applications.

A naive approach to address the large state space of configurations for microservices applications is to tune one microservice at a time. While this approach significantly reduces the state space dimensionality for configuration tuning, it does proportionally increase the tuning effort. Further, this serial tuning approach cannot capture the complex relationship between different parameters and the cascading effects between different microservices [20], [21].

Based on the above discussion, we argue that there is a need to investigate *joint optimization* of the microservices' parameters. The joint optimization is needed in order to capture the impact of *multiple* parameters of one microservice on its performance as well as the impact of a microservice's parameters on the performance of other microservices. Further, mechanisms are needed to reduce the configuration state space, given the numerous parameters and microservices employed by modern applications.

We now briefly discuss prior works related to the general problem of configuration tuning in systems before we formalize our specific problem in Section III-A.

Application configuration tuning. There has been considerable research in parameter tuning for individual applications, such as Apache web server [8], Memcached [22], database [9] and storage systems [16], [23], etc. While the above works can be used to tune individual microservices in isolation, the dependencies between microservices necessitates global optimization across microservices.

SmartConf [7] is a control-theoretic framework that automatically sets and dynamically adjusts parameters of software systems to optimize performance metrics while meeting the operating constraints set by the user. However, SmartConf is only applicable to parameters that have a linear relationship with performance; this is not necessarily the case for parameters of microservices [2]. BestConfig [6] uses sampling and search-based methods to tune parameters of software systems. However, the sampling effort required increases exponentially with the number of parameters, suggesting that BestConfig is infeasible for microservices applications with a large configuration space. Fekry et al. [24] concentrate on dynamically tuning configurations of data analytics frameworks for varying workloads and environments. While online tuning is an interesting research direction, it significantly limits the number of parameters available for (online) tuning. Alabed et al. [25] tune 10 parameters of RocksDB by optimizing multiple objectives using Bayesian optimization. However, finding the low-level metrics and reducing the dimensionality of each optimization task requires expert knowledge of the system being tuned which is not feasible for microservices architecture due to the variety of microservices that are part of each application.

Resource allocation tuning. Bilal et al. [26] perform an exhaustive comparison of existing black-box techniques for the problem of finding the best cloud configuration that minimizes the execution time or cost. Vanir [21] optimizes the cloud configuration for analytics clusters using Mondrian forest-based performance model and transfer learning. OPTIMUSCLOUD [10] jointly optimizes VM configurations and database configurations for cloud-deployed database systems by training a performance prediction model. CherryPick [27] uses Bayesian Optimization (BO) to build a performance model for Big Data systems, which is then used to find the best cloud configuration for these systems.

While some of the optimization algorithms explored in our evaluation are similar to the ones employed by the above works, we note that our focus is on tuning the *parameters of the numerous microservices that make up an application*, as opposed to only focusing on a handful of resource allocation parameters, such as number of CPUs, memory capacity, etc.

Reducing the configuration space. Kanellis et al. [28] employ learning-based techniques to find the most important parameters of database systems that impact performance. Carver [29] employs Latin Hypercube Sampling to explore the effect of different parameters on storage system performance and uses the variance in performance caused by a parameter as an indicator of the parameter's importance. As discussed in Section IV, focusing on microservices on the critical path is a more effective approach.

III. Problem Formulation and System Design

In this section, we formulate the microservices configuration setting problem as an optimization problem. We then describe our system design for the automated framework (which we have made publicly available [17]) that aids our experimental evaluation (presented in Section IV).

A. Microservices configuration setting problem

Let $f(c)$ denote the objective function (or performance metric) for the microservices application under the configuration c; here, c is the (potentially large) vector of parameter settings for all tunable parameters of all microservices. Note that a parameter refers to a configurable option and a configuration is a combination of parameter values. Let C denote the set of all configurations, i.e., all feasible values that vector c can take. Finally, let $c_{opt} \in C$ denote the configuration that minimizes the performance metric, $f()$. Thus, $c_{opt} = \text{argmin}_{c \in C} f(c)$. We could consider metrics that need to be maximized by minimizing the negative of the objective function. Our problem statement is *to find c_{opt} or a near-optimal configuration*. We focus on the realistic case where no assumptions can be made on the structure of $f()$ or on the availability of offline training data. We further assert, for practical purposes, that the (near-)optimal configuration should be determined in a reasonable amount of time.

While $f()$ can represent any metric of interest, including combinations of metrics, we consider the 95^{th} percentile of end-to-end application latency to be our metric, $f()$. We note that customer-facing applications often employ such tail latency metrics to assess application performance [3], [4].

Given the dependencies between parameters and the possible non-linear relationship between performance and parameter values (as described in Section I), it is unlikely that $f()$ can be determined or inferred accurately. Thus, classic convex optimization techniques cannot be readily applied to determine c_{opt}. However, for a given

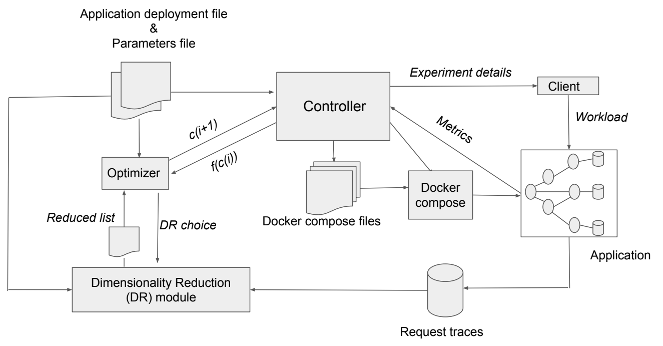


Fig. 2: Illustration of our solution framework. $f()$ is the objective function or performance metric of interest and $c(i)$ is the configuration setting for iteration i. The *optimizer* takes in the observed objective function value for a configuration, $f(c(i))$, and outputs the next configuration to employ, $c(i+1)$. The *dimensionality reduction module* trims the configuration vector size to speed up the optimization process. The *controller* interfaces with the application and invokes the execution based on the required configuration.

c, the value of $f(c)$ can be observed or measured by setting the parameter values in c for the microservices and running an experiment. This suggests that black-box optimization techniques, that iteratively observe the value of $f()$ at a given c and determine the next configuration value c' to explore, can be applied to find c_{opt} or near-optimal c values.

B. Automated framework to aid optimization

Unlike prior works [16], [26] that run optimization algorithms over readily available datasets, we evaluate the value of the objective function, $f()$, by running an experiment. To streamline the iterative exploration of configurations (for determining c_{opt}), we thus require a robust framework that can automatically: (i) configure the parameters of the microservices selected by the dimensionality reduction technique and run the application with these parameter settings, (ii) collect the required metrics, and (iii) run the optimization algorithm to obtain the next configuration to experiment with.

Figure 2 illustrates the design of our automated framework that we use to conduct our experiments. The *application deployment file* has information necessary to create the docker-compose files, viz., the list of microservices, their images, the host details, environment variables, etc. The *parameters file* contains the list of parameters being tuned and their range. The size of this list depends on the *dimensionality reduction* method being employed. The *controller* passes the value of the measured objective function, $f(c(i))$, of the current iteration, i, and queries the *optimizer* for the next configuration setting, $c(i+1)$.

The *optimizer*, in its first iteration, queries the *dimensionality reduction module* to obtain a subset of the microservices parameters that will be subject to optimization. The *dimensionality reduction module* uses the *application deployment file*, *parameters file*, and the *request traces* to pass a *reduced list* of parameters to the optimizer. The dimensionality reduction techniques are discussed in Section IV-B2. The *optimizer* then generates, via the optimization algorithm, the next configuration setting, $c(i+1)$, for the *reduced list* of parameters.

Using the details in the *application deployment file* and the $c(i+1)$ configuration passed by the *optimizer*, the *controller* generates *docker-compose files* on the fly with the necessary network settings and mounts. The application is then deployed on the servers using these *docker-compose files* and the *client* sends the workload to the application. The *request traces* are collected by a tracing framework and the latency metrics are calculated by the *client*. These metrics are passed to the *controller* which then calculates the objective function, $f(c(i+1))$, and repeats the process iteratively until a good enough configuration is found or until an exploration time limit is reached. Our framework supports any combination of average, median, or tail latency for the objective function.

The framework currently supports automatic configuration management for the most widely used microservices [30]: Memcached, Redis, MongoDB, MySQL, Nginx, and microservices implemented using the thrift framework. The parameters of some of these microservices can be modified by creating a configuration file (e.g., for Nginx) whereas others expect them as command line arguments with varying syntax. The user can be agnostic to these intricacies and treat all parameters similarly. The framework can be employed for any microservices application consisting of the supported microservices by including the *application deployment file* for that application. Optimization algorithms can be added by inheriting the *Optimizer* class and implementing its methods.

IV. Evaluation

In this section, we first discuss our experimental setup and methodology, and then present our experimental results. Our evaluation goal is to (i) investigate the efficacy of various optimization algorithms with respect to their running time and their ability to improve tail latency by configuration tuning; and (ii) investigate dimensionality reduction techniques that can speed up the optimization algorithms in practice.

A. Experimental setup

We use a cluster with four servers, each with 24 (hyper)cores, 40 GB of memory, and 250GB of disk space. We deploy the microservices of the application on these servers based on their functionality: one hosts back-end microservices as is the practice in industry [11], one server hosts front-end microservices, one hosts the microservices that implement the logic, and one server is dedicated for monitoring the microservices and the application performance. We restrict monitoring services, Jaeger [31] with Elasticsearch [32] back-end, to a different server to avoid interference with the application. *docker-compose* is used to deploy the application and *overlay* network connects the microservices across the servers.

Applications. We employ the *social networking* and *media microservices* applications from the DeathStarBench benchmark suite [1] and *train ticket* [33] application to evaluate the efficacy of different black-box optimization algorithms.

The social networking application has 28 microservices that together implement several features of real-world social networking applications. The constituent microservices are Nginx, Memcached, MongoDB, Redis, as well as microservices that implement the logic of the application. The application workload consists of 10%

requests that create a post, 30% requests that read the timeline of other users, and 60% requests that read the user's own timeline.

The media microservices application implements a movie review system and consists of 31 microservices. The constituent microservices are similar to the ones in the social networking application. The workload consists of 25% requests that add a movie review, 70% requests that read a movie review, and 5% requests that read the plot of the movie.

The train ticket application is a train ticket booking system implemented using 41 microservices. In addition to the microservices that are part of the social networking application, this application also uses MySQL microservice. The application workload consists of 50% of requests that search for a train between two stations, and 50% of requests that reserve a train ticket.

We change the type of server for social networking and media microservices applications to *TNonblockingServer*. The Apache Thrift C++ *TNonblockingServer* provides better performance and exposes numerous settings for the developer to customize the server [12]. We also make modifications to change the thread pool size dynamically based on the value suggested by the optimizer for each iteration.

B. Evaluation methodology

For evaluation, we consider the *95th percentile of latency* as the performance metric; other latency metrics can be readily used as well. For each microservice, we select at most five parameters to tune; we refer to product documentation [5], [12], [34]–[36] to identify the performance-impacting parameters. Our framework supports parameters that can take continuous (e.g., factor parameter of memcached), discrete (e.g., number of processes in Nginx), or categorical values (e.g., maxmemory-policy in Redis). The range of allowed values for each parameter is decided based on product documentation (e.g., internal cache size of mongoDB) or the limits of the hardware (e.g., number of threads in memcached).

We report results averaged across multiple experimental runs and provide error bars where appropriate. Each run lasts for 20 minutes, with the first few minutes (5 minutes for social networking and media microservices and 10 minutes for train ticket) considered as warm up until the cache hit rate stabilizes. Performance metrics are collected after the warm up period.

1) Black-box optimization algorithms: We consider six existing representative optimization algorithms in our evaluation, and then propose a seventh hybrid algorithm based on our analysis of the existing six algorithms. The first 2 are representative of *heuristic-based probabilistic algorithms*, the next 2 are evolutionary algorithms inspired by population-based biological evolution, and the next 2 are *sequential model-based optimization algorithms* that approximate the objective function with a cheaper, surrogate function [37] to aid optimization. We use skopt [38], Hyperopt [39], and Nevergrad [40] libraries to implement the algorithms. We also compare the results of these algorithms with the best configuration obtained by performing a random search of the configuration space. Note that we also tried tuning one microservice at a time (as opposed to a joint tuning), but the results are inferior and are so omitted.

Simulated Annealing (SA) [41] exploits the neighbourhood points based on the value of the objective function at these points, with the degree of exploration determined by a time-varying parameter that decreases with each iteration (annealing). Since SA is known to be better at global optimization than the hill climbing algorithm [41], we do not evaluate the latter.

Dynamically Dimensioned Search (DDS) starts with an initial configuration and perturbs the values of the parameters of the configuration based on a perturbation factor [42]. With each iteration, the probability of each parameter being included in the optimization reduces uniformly, thereby reducing the search space.

Particle Swarm Optimization (PSO) [41] works by moving a population (called swarm) of candidate solutions (called particles) around the search space depending on the particle's best-known position and the global best position.

Genetic Algorithms (GA) [41] mimic natural selection by first selecting a subset of candidate solutions based on the objective function value and then randomly changing the configurations of some parameters (mutation) and combining configurations of the candidates (crossover) to generate new candidates.

Bayesian Optimization (BO) starts with a prior distribution of the search space guided by the surrogate; we experiment with the popular Gaussian Process (GP) [37], Gradient Boosted Regression Trees (GBRT) [26], and Random Forests (RF) [26] surrogate models. The posterior distribution is updated at each step of exploration using Bayesian method.

Tree-structured Parzen Estimator (TPE) is similar to BO, but models the likelihood and prior instead of the posterior [37].

Hybrid algorithm is a new algorithm that we construct by combining the strengths of BO and DDS. BO models the relationship between performance and the parameters to efficiently search for the optimal configuration with a convergence rate that is dependent on the initial samples [25]. On the other hand, DDS is a computationally efficient heuristic-based search algorithm that performs well (See Section IV). Since DDS is not model-based, it makes no attempt to learn about the parameter space. With hybrid, we combine the *light-weight searching feature* of DDS with the *model-based searching feature* of BO. Specifically, the DDS algorithm is run for a fixed number of iterations and the resulting best configurations are used as initial samples for the Bayesian algorithm with the popular Gaussian Process as the surrogate model [21], [27]. By contrast, when not using hybrid, the initial samples for Bayesian are (by default) randomly generated.

2) Dimensionality reduction strategies: If an application has m microservices each with p_i parameters (for $i = 1,2,...,m$), then the number of dimensions in a configuration vector c is $n = \sum_{i=1}^{m} p_i$. For the purpose of illustration, if each parameter can take v different values, then the number of possible configurations is $|C| = v^n$. Clearly, the search space of configurations grows exponentially with the number of microservices. To reduce the search space, we thus consider strategies that allow us to focus our configuration tuning effort on only a subset of the microservices. Another advantage of dimensionality reduction is that several optimization algorithms, such as Bayesian Optimization (BO), do not work well in high dimensions (number of tunable parameters, in our case) [43]. We note that our dimensionality reduction strategies have a different goal than those used in the machine learning community since our focus is on using system characteristics to reduce dimensions in a practical manner. For example, Principal Component Analysis (PCA) [44] can reduce the configuration space dimensions but would make it difficult to reconstruct the configuration value after optimization.

1) ***Critical path.*** In the call graph of a request, the critical path is the path formed by microservices that determine the latency of the request. Tuning the parameters of the microservices that fall on the critical path of a request is important as any performance improvements in these microservices will reduce the end-to-end latency of the request. Algorithm 1 provides an overview of our critical path determination algorithm. The algorithm takes the request traces as input T and outputs a list of microservices that form the critical path of each trace. In summary, the algorithm traverses the call graph of a request to find all the microservices on the critical path that have non-negligible latency (at least 1ms). We rely on the service time (or span) measurements provided by Jaeger for each microservice to determine the critical path. Using our algorithm, we identify microservices present on the critical path of most of the request types for all applications.

Algorithm 1 Find microservices along the Critical Path.

```
1:  **Input:** Request traces, $T$.
2:  **Output:** List of microservices along the Critical Path.
3:  $criticalPathAll \leftarrow \emptyset$
4:  **for** $t \in T$ **do**
5:      currentCriticalPath $\leftarrow$ getCriticalPath($t.root$)
6:      append($criticalPathAll$,$currentCriticalPath$)
7:  **procedure** GETCRITICALPATH($node$)
8:      criticalPath $\leftarrow \emptyset$
9:      **if** $node.children\ is\ NULL$ **then**
10:         $lastChild$ = nextChild($node$)
11:         getCriticalPath($lastChild$)
12:         $node.duration$ = updateDuration($node$)
13:     **if** $node.duration > 1ms$ **then**
14:         append($criticalPath$, $node$)
```

2) ***Bottlenecks.*** FIRM [19] uses a Support Vector Machine (SVM) to detect microservices that could be potential bottlenecks for an application. We train an SVM model using the publicly available tracing data [45] for the social networking, media microservices, and the train ticket applications. We use this model to predict potential bottlenecks in all applications and tune only these.
3) ***Performance variance.*** Reducing the source of performance variance can improve the system performance [3], [46]. Accordingly, we consider configuration tuning only for microservices that have a high service time coefficient of variation (above 0.5 in our experiments).
4) ***Performance variance along the critical path.*** To combine the strengths of different dimensionality reduction techniques, we consider the approach of first determining the critical path (via Algorithm 1) and then selecting the top five microservices on the critical path that have the highest variance in service time.

C. *Experimental results*

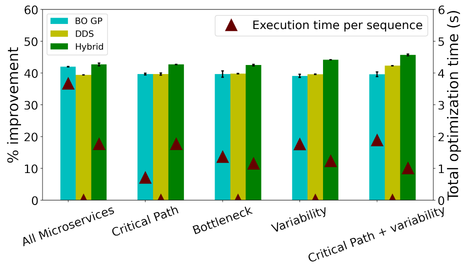
In practice, the optimization algorithms cannot be run indefinitely. Unless otherwise specified, we limit the number of configurations to be explored for each optimization algorithm to 15. We note that running each iteration of the algorithm involves bringing up the application, applying the configuration, and running the workload, which together takes about half an hour. By contrast, the time taken by an optimization algorithm to suggest a new configuration is typically in the order of seconds. Thus, a budget on the optimization time as a stopping criteria is not as practical as the number of iterations of the algorithm. For initialization, the optimization algorithms, except Hybrid, start with a random configuration. For the evaluation to be fair, we initialize all the algorithms with the same random samples. Note that (re)setting the configuration parameters between iterations does incur some overhead and may require restarting some microservices; during this time, the application may be momentarily offline. We acknowledge that this can be concerning for production deployments where application downtime is not tolerated. However, in a production deployment, the reconfiguration step can be carried out during planned maintenance or upgrade windows to avoid additional disruption to the application [47]. We defer online configuration tuning of microservices to future work.

1) Efficacy of dimensionality reduction strategies: Figure 3 shows the percentage improvement in tail (95^{th} percentile) latency of all applications under different dimensionality reduction techniques, compared to the tail latency when using the default configuration for all parameters. For ease of illustration, we show results for three specific optimization algorithms. Note that comparison across optimization algorithms will be discussed in the next subsection and is not the focus here. Error bars in the figures indicate the standard deviation around the reported mean results.

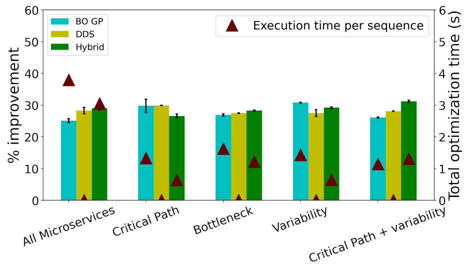
In Figure 3a, we see that tuning all 28 microservices of the social networking application provides about 39–43% improvement in tail latency. Tuning all the microservices on the critical path provides similar improvements. However, tuning only the microservices on the critical path that show high variability (5 microservices) provides 40–46% improvement. Note that this improvement is greater than that obtained by tuning all 28 microservices. This is because dimensionality reduction reduces the configuration search space, enabling a more efficient tuning within the budget of 15 configurations to explore. Tuning the known bottlenecks provides around 42% improvements, suggesting that the critical path approach correctly identifies the microservices that have the most impact. Finally, by focusing on the variability causing microservices, the latency improvement is about 39–44%.

In Figure 3b, we observe that tuning all the 31 microservices of the media microservices application produces about 25–29% improvements. We observe that tuning only the microservices on the critical path that show high variability (5 microservices) again provides superior performance with up to 31.2% improvement, highlighting the impact of dimensionality reduction. The performance improvements for the critical path, the bottleneck, and the variability techniques are 28–30%, 27–28%, and 28–30%, respectively

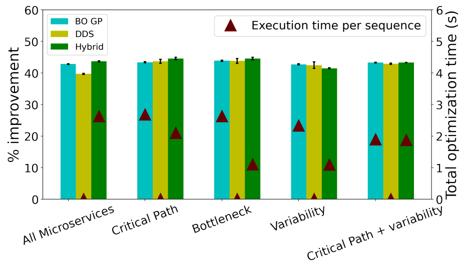
In Figure 3c, we see that tuning the 26 microservices of train ticket application results in 39–43% improvement. Tuning only the microservices on the critical path provides up to 46% improvement in tail latency. Since the train ticket application has the most parameters, the benefits of dimensionality reduction are more pronounced. The performance improvements are around 44%, 42%, and 43% for bottleneck, variability, and critical path+variability, respectively.



(a) Social networking application.



(b) Media microservices application.



(c) Train ticket application.

Fig. 3: Evaluation of different dimensionality reduction techniques with respect to improvement in latency over the default configuration under different optimization algorithms. Error bars indicate the standard deviation around the reported mean over 3 runs. The total optimization time is the time taken by the algorithm across the 15 iterations (excluding the time to run the application with the required configurations).

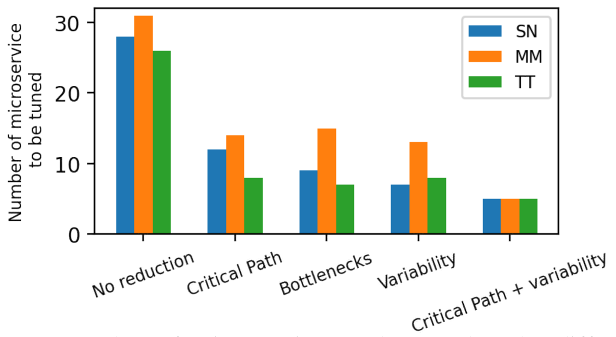


Fig. 4: Number of microservices to be tuned under different dimensionality reduction strategies for social networking (SN), media microservices (MM), and train ticket (TT).

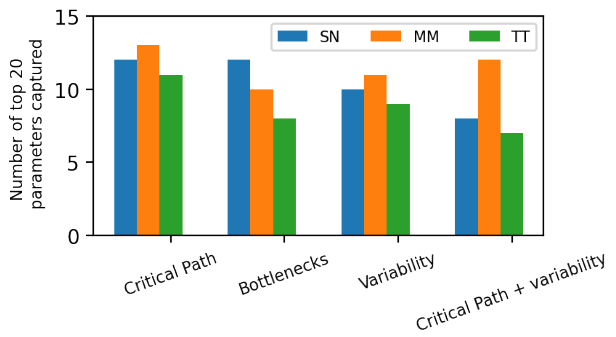


Fig. 5: Illustration of coverage of the top 20 parameters captured by different dimensionality reduction techniques.

Figure 4 shows the number of microservices tuned under different dimensionality reduction techniques, compared to no reduction, for all the three applications. While all techniques reduce the number of microservices to be tuned by at least 50%, the "Critical path + variability" approach (Performance variance along the critical path) allows us to customize and aggressively reduce the number to just 5. Despite this substantial reduction in the number of microservices to be tuned, the "Critical path + variability" approach provides significant tail latency reduction for the applications we consider, as highlighted in Figure 3.

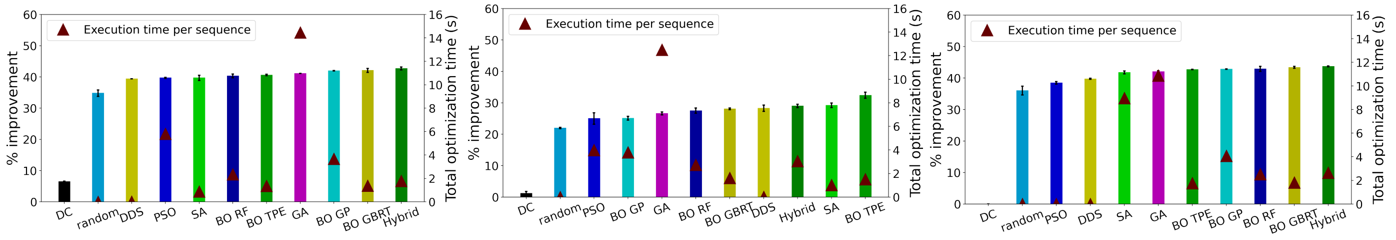
To further contrast the four different dimensionality reduction techniques, we consider the overlap in subsets of microservices chosen by the techniques. For the social networking application, we find that only two microservices are common among all the subsets: (i) *post-storage-memcached* is an important microservice as it caches posts that are read by requests that constitute 90% of the workload; and (ii) *compose-post-service* is critical in the call graph of the request that writes posts as it is called multiple times per request. This shows that, despite differences in the subsets, all techniques have the ability to identify some of the important, performance-impacting microservices.

A potential drawback of reducing the dimensions by omitting microservices for optimization is that a dimensionality reduction technique could miss out on important parameters. To evaluate this hypothesis, we find the 20 most important parameters using the offline and expensive fANOVA [18] approach and determine how many of these 20 parameters are captured by different dimensionality reduction techniques in Figure 5. We find that while the different dimensionality reduction techniques do not capture all 20 important parameters, they do capture 3–4 parameters out of the top 5. We note that fANOVA parameter importance analysis can be used to reduce the number of dimensions, but the amount of training data and effort required makes this approach impractical.

2) Comparing different optimization algorithms: Figures 6a, 6b, and 6c show the (sorted) percentage improvement (on left y-axis) in tail latency over the default configuration afforded by different optimization algorithms with no dimensionality reduction for the social networking, media microservices, and the train ticket applications, respectively. For comparison, we show (as DC) the improvement afforded by the configuration employed by the developers of the DeathStarBench [15] and the train ticket application [33].

For the social networking application in Figure 6a, we see that Hybrid algorithm provides the best improvement of around 43%, followed closely by BO GBRT (42%) and BO GP (41.8%). Using the configuration chosen by the developers provides a modest improvement of 6% over the default configuration. To evaluate the overhead of different optimization algorithms, we plot (as red triangles with right y-axis) the time taken by the optimization across all iterations in Figure 6. We find that DDS requires the least amount of time (10ms), followed by SA (0.8s) and BO TPE (1.4s). Hybrid is also relatively quick, requiring about 1.7s. GA and PSO incur a high overhead; this is expected as evolutionary algorithms are computationally intensive.

For the media microservices application, as seen in Figure 6b, the BO TPE algorithm provides the best configuration with an improvement of around 32%. DDS again takes the least amount



(a) Social networking application. (b) Media microservices application. (c) Train ticket application.

Fig. 6: Improvement in latency compared to default configuration (left y-axis) and the time incurred by the optimization (right y-axis) for all algorithms when tuning the microservices of the applications with no dimensionality reduction.

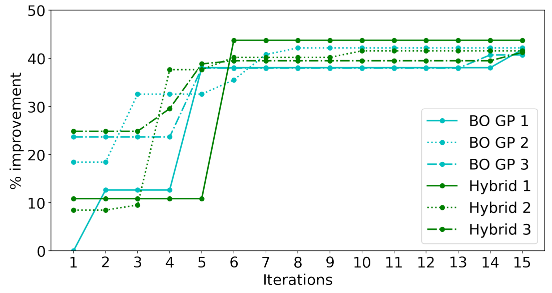


Fig. 7: Efficiency of various algorithms over 15 iterations when tuning on the critical path of social networking application.

of time, about 9ms. The Hybrid algorithm also performs well, with an improvement of around 29% and requiring about 3s of time. Using the configuration provided by the developers only provides a nominal 2% improvement over the default configuration.

For the train ticket application, the Hybrid algorithm again performs the best with 43% improvement over the default configuration, closely followed by BO GBRT (42.84%) and BO RF (42.72%). The developer's configuration performs worse than the default configuration because of which it is excluded from Figure 6c. It is interesting to note the impressive performance of random search (36% improvement) considering the negligible run time ($\sim$ 1ms). The existence of multiple optimal regions, as discussed in Section IV-C, is one likely reason for its good performance. Further, randomized configuration settings have been shown to perform well when tuning databases [16], [25].

Based on the above results, we conclude that, for our evaluation, Hybrid is the best performing algorithm for the social networking and train ticket applications whereas Tree-structured Parzen Estimator (TPE) provides a good tradeoff between latency improvement and optimization runtime for media microservices.

3) Convergence analysis of algorithms: The results shown thus far are based on the best configuration picked by the algorithms from among 15 iterations. To analyze the significance of number of iterations and variance across different sequences (runs), we plot the best improvement afforded until different iterations for BO GP and Hybrid, across 3 different sequences of these algorithms, in Figure 7 for the social networking application. Although the different sequences vary during the initial iterations, they eventually converge well within 15 iterations. This suggests that the variability *between* runs is low, explaining the narrow error bars in our results.

We also analyzed the results for 100 iterations and found that the additional performance benefit afforded over 15 iterations is only about 1–2% compared to the best solution in Figure 6, suggesting that the optimization algorithms converge quickly. This is useful in practice given that each additional iteration imposes certain overhead and application downtime.

4) Significance of initial configuration: The optimization algorithms typically start with a randomly sampled configuration. To assess the significance of this initial configuration on performance improvement and convergence, we specifically set the initial configuration of the social networking application to one that we know performs poorly to check how the optimization recovers; we use BO GP for this evaluation. For example, we limit the number of processes for the Nginx microservice to 1, set the Memcached cache size to 16MB, etc. We find that, despite the poor initial configuration, the algorithm does provide significant improvement over the default configuration, with only a 3.4% relative drop in performance compared to the randomly chosen initial configuration case.

5) Analysis of configurations set by algorithms: To better understand the optimal configurations, we now analyze the specific parameter configuration values determined by different algorithms. Without loss of generality, we consider the social networking application and analyze the values selected by each algorithm for the top 5 important parameters. To identify the important parameters, we employ fANOVA [18], which uses an empirical performance model based on random forests to analyze how much of the observed performance variation in the configuration space is explained by a single parameter or combinations of few parameters. To obtain the data for fANOVA, we sample the configuration space by running up to 1000 experiments for various configurations and collecting the corresponding 95^{th} percentile latency.

For the social networking application, the top 5 parameters (along with the associated microservice), in the order of importance, and the values assigned to them by each algorithm, are given in Table I. The top parameter is the *worker_processes* parameter of the *frontend* microservice (NGINX). While the default value of this parameter is 1, for a fair comparison, we override the default value to the number of cores in the server (24) as suggested in product documentation [5]. We set the allowable range for this parameter to be 1–48. As seen in Table I, the values set by different algorithms are close to 24. Since the worker processes for the social networking application do not perform any I/O, a high value for *worker_process* would lead to contention and a low value would lead to decreased processor utilization. This shows that all algorithms judiciously choose the *worker_process* value.

Parameter (associated microservice)	Range	Default	Optimization algorithms								
			BO GP	BO GBRT	TPE	DDS	PSO	SA	hybrid	GA	BO RF
worker_processes (frontend-nginx)	1-48	24	24	21	23	19	20	26	23	19	20
zset-max-ziplist-entries (social-graph-redis)	64-512	128	64	108	77	68	92	238	109	120	89
io_threadpool_size (social-graph-service)	1-48	13	33	36	43	34	33	46	29	33	30
memory_limit (MB) (post-storage-memc)	32-20k	64	4k	6.8k	5.8k	7k	8k	9.1k	8.8k	13k	6.4k
hz (user-timeline-redis)	1-100	10	43	64	60	57	41	61	52	71	39

TABLE I: Top-5 important parameters (as identified by fANOVA analysis) for the social networking application.

The second most important parameter is the *zset-max-ziplist-entries* of the social-graph-redis microservice. This parameter sets a limit on the number of entries allowed in a ZSET (sorted sets) for it to be encoded as a ziplist. The memory savings due to ziplist come at the cost of CPU usage—CPU cycles are spent on decoding every read, partially re-encoding every write, and may require moving data in memory. As seen from the table, the value of this parameter is always set below the default, except for the one generated by SA, signifying the benefits of prioritizing CPU over memory savings.

The next important parameter is the *io_threadpool_size* of the *social-graph-service*, which dictates the size of the I/O thread pool for TNonblockingServer [12]. We see that the *io_threadpool_size* value selected by different algorithms is consistently higher than the default value (13), suggesting that the default configuration was under-utilizing the resources.

The next important parameter is the *memory_limit* value for *post-storage-memcached* micorservice, which is set at 64MB by default. Table I shows that the various optimization algorithms have a *memory_limit* value of at least 4GB for this microservice. This is a critical microservice that is along the critical path of 90% of the requests, substantiating the extra memory allocation to improve performance.

Finally, the *hz* parameter sets the frequency of invocations of background tasks to remove expired keys in Redis [35]. For the *user-timeline-redis* microservice, the value selected for *hz* is much higher than the default of 10, indicating that the additional use of CPU by this microservice (at the expense of other microservices) is worth the improvement in performance.

Table I highlights the similarities (e.g., for *worker_processes*) and differences (e.g., for *zset-max-ziplist-entries*) in the parameter values chosen by the optimization algorithms. The differences suggest that the algorithms do converge to different locally optimal configurations (as opposed to a single globally optimal one); despite the different configurations, the resulting latency benefits are comparable (as seen in Figures 6a, 6b, and 6c). The similarities suggest that minor differences in values (within a range) of some parameters may not significantly impact performance; a valuable future direction is to discretize some of the parameter ranges to reduce the configuration space.

6) Microservice-level analysis of latency reduction afforded by the best configuration: The workloads used in our experiments consist of a mixture of different request types (see Section IV-A). The 95^{th} percentile latency depends heavily on the request type that takes the longest time. To analyze the ability of optimizations in prioritizing microservices that serve the long-tailed request types, we compare the service time (95^{th} percentile) of all microservices along the call graph of different request types for the best configuration across all experiments (i.e., across all algorithms and all dimensionality reduction strategies) with the default configuration.

For the social networking workload, based on the experiment logs, we find that read-user-timeline requests influence the 95^{th} percentile of the workload latency the most, followed by read-home-timeline. *post-storage-memcached* and *post-storage-mongo* microservices, which are along the critical path of both these request types, can thus have a significant impact on workload latency. In case of the read-user-timeline request type, the best configuration results in 65% and 28% reduction in service time of *post-storage-memcached* and *post-storage-mongo* microservices, respectively. The *user-timeline-redis*, which is on the critical path of read-user-timeline, sees a 56% reduction in its service time. On the other hand, the microservices along the call graph of light-tailed compose-post request type experience a nominal *increase* in service time, notably *user-timeline-mongo* (7% increase), where the user's post IDs are written as part of the compose-post request type. For the *user-timeline-mongo* microservice, the best configuration across all experiments chooses *zlib* as the compression algorithm which uses more CPU than the default (*snappy*) [34]. Likewise, the best configuration sets *wiredTigerConcurrentWriteTransactions* to 74 (lower than the default of 128), limiting the maximum concurrent writes, and increasing the service time of *user-timeline-mongo*.

We find similar patterns (of prioritizing parameters of microservices that serve heavy-tailed request types) for other applications as well. For example, for media microservices, compose-movie-review request type influences the workload's 95^{th} percentile latency the most. *compose-review-service* microservice, which is along the critical path of the compose-movie-review requests, sees a 35% reduction in 95^{th} percentile of the service time when the best configuration across all experiments is applied. The key takeaway here is that despite the optimization algorithms being oblivious to the workload mix, they sample the search space well enough to find configurations that are near-optimal for the workload.

V. Conclusion

Despite the recent shift in application design to microservices architecture, the fundamental problem of setting the configuration of individual microservices to improve performance has received very little attention, with practitioners instead settling for sub-optimal performance via default or ad-hoc configuration settings. This paper *makes the case for configuration tuning of microservices*.

Our investigation of different joint optimization techniques shows that significant improvements in tail latency, up to 46%, can be realized via configuration tuning. While most algorithms perform well, the optimal algorithm is application-dependent; further, combining different algorithms can provide superior performance for some applications. Our analysis reveals that the optimal configuration of a microservice (e.g., MongoDB) need not be the same across applications or even across instances within the same application.

We also investigate techniques to reduce the tuning effort across algorithms. We consider different approaches to dimensionality reduction and find that focusing on tuning the microservices on the critical path that have the highest service time variability is an effective dimensionality reduction technique. We conclude that dimensionality reduction based on system characteristics is an effective approach to the otherwise intractable problem of optimizing a large state space.

Acknowledgment: This work was supported by NSF CNS grants 1730128, 1750109, 1909356, and 2106434.

References

[1] Y. Gan, Y. Zhang, D. Cheng, A. Shetty, P. Rathi, N. Katarki, A. Bruno, J. Hu, B. Ritchken, B. Jackson, K. Hu, M. Pancholi, B. Clancy, C. Colen, F. Wen, C. Leung, S. Wang, L. Zaruvinsky, M. Espinosa, Y. He, and C. Delimitrou, "An Open-Source Benchmark Suite for Microservices and Their Hardware-Software Implications for Cloud and Edge Systems," in *Proceedings of the Twenty Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2019.

[2] A. Sriraman and T. F. Wenisch, "μtune: Auto-tuned threading for OLDI microservices," in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, 2018.

[3] J. Dean and L. A. Barroso, "The Tail at Scale," *Communications of ACM*, vol. 56, no. 2, pp. 74–80, 2013.

[4] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: Amazon's highly available key-value store," in *Proceedings of Twenty-First ACM SIGOPS Symposium on Operating Systems Principles*, ser. SOSP '07, 2007.

[5] "Beginner's guide," nginx.org/en/docs/beginners_guide.html.

[6] Y. Zhu, J. Liu, M. Guo, Y. Bao, W. Ma, Z. Liu, K. Song, and Y. Yang, "Bestconfig: Tapping the performance potential of systems via automatic configuration tuning," in *Proceedings of the 2017 Symposium on Cloud Computing*, ser. SoCC '17, 2017.

[7] S. Wang, C. Li, H. Hoffmann, S. Lu, W. Sentosa, and A. I. Kistijantoro, "Understanding and auto-adjusting performance-sensitive configurations," *SIGPLAN Not.*, 2018.

[8] Q. Wang, S. Zhang, Y. Kanemasa, C. Pu, B. Palanisamy, L. Harada, and M. Kawaba, "Optimizing n-tier application scalability in the cloud: A study of soft resource allocation," *ACM Trans. Model. Perform. Eval. Comput. Syst.*

[9] B. Zhang, D. Van Aken, J. Wang, T. Dai, S. Jiang, J. Lao, S. Sheng, A. Pavlo, and G. J. Gordon, "A demonstration of the ottertune automatic database management system tuning service," *Proc. VLDB Endow.*, 2018.

[10] A. Mahgoub, A. M. Medoff, R. Kumar, S. Mitra, A. Klimovic, S. Chaterji, and S. Bagchi, "OPTIMUSCLOUD: Heterogeneous configuration optimization for distributed databases in the cloud," in *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, 2020.

[11] S. Luo, H. Xu, C. Lu, K. Ye, G. Xu, L. Zhang, Y. Ding, J. He, and C. Xu, *Characterizing Microservice Dependency and Performance: Alibaba Trace Analysis*, 2021.

[12] R. Abernethy, *The Programmer's Guide to Apache Thrift*, 2018.

[13] K. Indrasiri and D. Kuruppu, *gRPC: Up and Running*, 2020.

[14] "Mongodb jira," jira.mongodb.org/browse/SERVER-19911.

[15] "Deathstarbench," github.com/delimitrou/DeathStarBench.

[16] Z. Cao, V. Tarasov, S. Tiwari, and E. Zadok, "Towards better understanding of black-box auto-tuning: A comparative analysis for storage systems," in *2018 USENIX Annual Technical Conference (USENIX ATC 18)*.

[17] "Framework for tuning microservices applications," https://github.com/PACELab/microservices-tuning.

[18] F. Hutter, H. Hoos, and K. Leyton-Brown, "An efficient approach for assessing hyperparameter importance," in *Proceedings of International Conference on Machine Learning 2014 (ICML 2014)*, 2014.

[19] H. Qiu, S. S. Banerjee, S. Jha, Z. T. Kalbarczyk, and R. K. Iyer, "FIRM: An intelligent fine-grained resource management framework for slo-oriented microservices," in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, 2020.

[20] Y. Gan, Y. Zhang, K. Hu, Y. He, M. Pancholi, D. Cheng, and C. Delimitrou, "Seer: Leveraging Big Data to Navigate the Complexity of Performance Debugging in Cloud Microservices," in *Proceedings of the Twenty Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2019.

[21] M. Bilal, M. Canini, and R. Rodrigues, "Finding the right cloud configuration for analytics clusters," in *Proceedings of the 11th ACM Symposium on Cloud Computing*, ser. SoCC '20.

[22] M. Wajahat, S. Masood, A. Sau, and A. Gandhi, "Lessons Learnt from Software Tuning of a Memcached-Backed, Multi-Tier, Web Cloud Application," in *Proceedings of the 8th International Green and Sustainable Computing Conference*, ser. IGSC '17, 2017.

[23] Y. Zhu, S. Krishnan, K. Karanasos, I. Tarte, C. Power, A. Modi, M. Kumar, D. Zhang, K. Muthyala, N. Jurgens, S. Sakalanaga, S. Darbha, M. Iyer, A. Agarwal, and C. Curino, "Kea: Tuning an exabyte-scale data infrastructure," in *Proceedings of the 2021 International Conference on Management of Data*, ser. SIGMOD/PODS '21, 2021.

[24] A. Fekry, L. Carata, T. Pasquier, A. Rice, and A. Hopper, "To tune or not to tune? in search of optimal configurations for data analytics," in *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2020.

[25] S. Alabed and E. Yoneki, "High-dimensional bayesian optimization with multi-task learning for rocksdb," ser. EuroMLSys '21, 2021.

[26] M. Bilal, M. Serafini, M. Canini, and R. Rodrigues, "Do the best cloud configurations grow on trees? an experimental evaluation of black box algorithms for optimizing cloud workloads," *Proc. VLDB Endow.*, 2020.

[27] O. Alipourfard, H. H. Liu, J. Chen, S. Venkataraman, M. Yu, and M. Zhang, "Cherrypick: Adaptively unearthing the best cloud configurations for big data analytics," in *Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'17, 2017.

[28] K. Kanellis, R. Alagappan, and S. Venkataraman, "Too many knobs to tune? towards faster database tuning by pre-selecting important knobs," in *12th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 20)*, 2020.

[29] Z. Cao, G. Kuenning, and E. Zadok, "Carver: Finding important parameters for storage system tuning," in *18th USENIX Conference on File and Storage Technologies (FAST 20)*.

[30] R. Laigner, Y. Zhou, M. A. V. Salles, Y. Liu, and M. Kalinowski, "Data management in microservices: State of the practice, challenges, and research directions," 2021.

[31] "Jaeger," https://www.jaegertracing.io/.

[32] "Elastic search," https://www.elastic.co/elasticsearch/.

[33] X. Zhou, X. Peng, T. Xie, J. Sun, C. Xu, C. Ji, and W. Zhao, "Poster: Benchmarking microservice systems for software engineering research," in *2018 IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion)*, 2018.

[34] "MongoDB," https://docs.mongodb.com/manual/reference/parameters/.

[35] "Redis configuration," https://redis.io/topics/config.

[36] "memcached(1)," https://linux.die.net/man/1/memcached.

[37] J. Bergstra, R. Bardenet, Y. Bengio, and B. Kégl, "Algorithms for hyper-parameter optimization," in *Advances in Neural Information Processing Systems*, 2011.

[38] "Skopt," https://scikit-optimize.github.io.

[39] J. Bergstra, D. Yamins, and D. D. Cox, "Making a science of model search: Hyperparameter optimization in hundreds of dimensions for vision architectures," in *Proceedings of the 30th International Conference on International Conference on Machine Learning*, ser. ICML'13, 2013.

[40] J. Rapin and O. Teytaud, "Nevergrad - A gradient-free optimization platform," https://GitHub.com/FacebookResearch/Nevergrad, 2018.

[41] S. Luke, *Essentials of Metaheuristics*. Lulu, 2013.

[42] B. A. Tolson and C. A. Shoemaker, "Dynamically dimensioned search algorithm for computationally efficient watershed model calibration," *Water Resources Research*, 2007.

[43] R. Moriconi, M. Deisenroth, and K. Sesh Kumar, "High-dimensional bayesian optimization using low-dimensional feature spaces." *Mach Learn 109, 1925–1943*, 2020.

[44] R. E. Millsap, *Journal of Educational and Behavioral Statistics*, 1995.

[45] H. Qiu, S. S. Banerjee, S. Jha, Z. T. Kalbarczyk, and R. Iyer, "Pre-processed tracing data for popular microservice benchmarks," 2020. [Online]. Available: https://doi.org/10.13012/B2IDB-6738796_V1

[46] A. Suresh and A. Gandhi, "Using variability as a guiding principle to reduce latency in web applications via os profiling," in *The World Wide Web Conference*, ser. WWW '19, 2019.

[47] "Planned maintenance window in aks," https://azure.microsoft.com/en-in/updates/public-preview-planned-maintenance-windows-in-aks/.