Towards Optimal Configuration of Microservices

Gagan Somashekar and Anshul Gandhi
PACE Lab, Stony Brook University
Stony Brook, New York, USA
{gsomashekar,anshul}@cs.stonybrook.edu

Abstract

The microservice architecture allows applications to be designed in a modular format, whereby each microservice can implement a single functionality and can be independently managed and deployed. However, an undesirable side-effect of this modular design 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 workshop paper investigates optimization techniques and dimensionality reduction strategies for tuning microservices applications, empirically demonstrating the significant tail latency improvements (as much as 23%) that can be achieved with configuration tuning.

Keywords: ML for systems, microservices, configuration tuning, optimization, tail latency

ACM Reference Format:
Gagan Somashekar and Anshul Gandhi. 2021. Towards Optimal Configuration of Microservices. In *The 1st Workshop on Machine Learning and Systems (EuroMLSys '21), April 26, 2021, Online, United Kingdom.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3437984.3458828

1 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 [24, 25, 27, 29, 40]. Each module or microservice typically implements a single business capability. The communication between the microservices, usually stateless, is via well-defined light weight APIs.

The microservice architecture is especially well suited for designing online, customer-facing applications where performance and availability are paramount [12, 20, 21, 25]. For example, an online application can be deployed as front-end microservices (e.g., Nginx), database microservices (e.g., MongoDB), caching microservices (e.g., Memcached), along with services that implement the logic of the application. The latter services that implement the logic may each have their own database and cache microservices. Consequently, an application can have numerous microservices. Distributed applications implemented using the microservices architecture are widely replacing existing deployments implemented using monolithic or multi-tier architectures at Amazon, Netflix, Uber, and Twitter [25].

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.* Tuning the parameters of monolithic or N-tier application deployments for maximizing performance is already a difficult task [33, 40, 44, 45, 45–47] (see Section 4). With microservice applications, configuration tuning is especially complicated owing to the following challenges:

- ***Very large configuration space.*** Microservices applications have numerous, interacting microservices that each have several parameters that can be configured. Further, frameworks that aid microservices development, such as Apache Thrift [10] and gRPC [28], introduce additional parameters that impact application performance.
- ***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 cannot be independently optimized (see Section 3). 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 [8].
- ***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 [44]. 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. Consequently, due to potential resource contention, the resource configuration of a microservice can impact the performance of all other colocated microservices.
- ***Non-linear relationship between microservices parameters and performance.*** Application performance need

EuroMLSys '21, April 26, 2021, Online, United Kingdom

ACM ISBN 978-1-4503-8298-4/21/04…$15.00
https://doi.org/10.1145/3437984.3458828

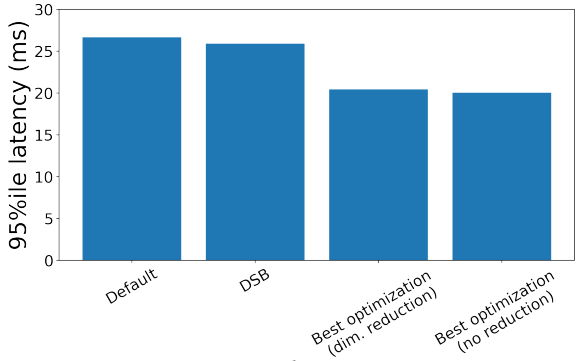

Figure 1. Comparison of 95^{th} percentile of latency for the social networking application [25] under (i) default configuration values (Default), (ii) the configuration used by DeathStarBench benchmark developers [2] (DSB), and the configuration found by the best optimization technique among those we explored (iii) with dimensionality reduction (considering only a subset of microservices for tuning) and (iv) without any reduction (tuning all microservices).

not be monotonically or linearly dependent on parameter values, making it difficult to determine optimal configuration parameter settings. The thread pool size parameter is a classic example whereby a low value results in underutilization of the CPU and a very high value results in contention for network sockets or CPU resources [40].

There is very little prior work on the specific problem of configuration tuning of microservices, and that work relies on empirical analysis for the specific parameters of thread pool size and threading model [40]. There are, however, prior works that focus on optimizing the configuration of individual services [19, 46], but as explained above, the dependencies between the parameters of microservices makes it infeasible to optimize them in isolation. To the best of our knowledge, there is no prior work that focuses on the problem of comprehensively tuning the configuration parameters of a microservice application.

This workshop paper explores the problem of configuration tuning of microservices applications. We conduct an extensive experimental investigation of various 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 reduce the tail latency of the social networking microservice application [25] by as much 23% and 21% compared to the default configuration setting and the suggested configuration in prior work [25], respectively.

To address the key challenge of a large configuration space when tuning microservices applications, we first identify potential performance-impacting parameters for popularly deployed microservices, such as Nginx, Memcached, MongoDB, etc. We then 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 roughly the same improvement in tail latency (0.2% difference) while only tuning about 43% of all microservices (roughly 57% reduction in number of parameters tuned).
This workshop paper makes the following contributions:

1. ***Problem formulation.*** We frame the configuration tuning problem as an optimization problem, making it amenable to optimization algorithms.
2. ***Automated framework to aid the configuration optimization.*** We implement a framework 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.
3. ***Experimental evaluation of different optimization algorithms.*** 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.
4. ***Dimensionality reduction.*** For scalability, we investigate techniques to reduce the overhead of optimization algorithms by limiting the set of microservices whose parameters will be configured. We consider various subsets of microservices: (i) those on the critical path, (ii) those that cause performance variability, and (iii) those identified by prior works to be performance bottlenecks.

2 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 that aids our experimental evaluation (presented in Section 3).

2.1 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. 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.

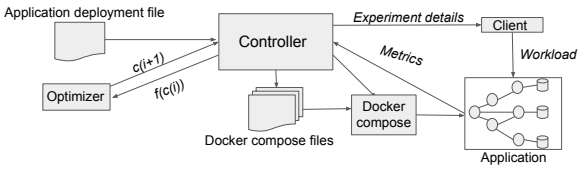

Figure 2. Illustration of our solution framework. $f()$ is objective function or performance metric of interest and $c(i)$ is the configuration setting for iteration i.

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 [21].

Given the dependencies between parameters and the possible non-linear relationship between performance and parameter values (as described in Section 1), 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 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.

2.2 Automated framework to aid optimization

Unlike prior works [16, 19] 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 as directed and run the application, (ii) collect the required metrics, and (iii) run the optimization algorithm to obtain the next configuration to experiment with. The framework should also be application-agnostic and should allow the optimization algorithm to be a pluggable module.

Figure 2 illustrates the design of our automated framework that we use to conduct our experiments. The *application deployment file* has the list of microservices, their images, the host details, etc. 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, $c(i+1)$. 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 *controller*. 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 currently supports any linear combination of average, median, or tail latency for the objective function. The framework can be employed for any microservices application by including the *application deployment file* for that application. Any optimization algorithm can be added by inheriting the *Optimizer* class and implementing its methods.

3 Evaluation

In this section, we first discuss our experimental setup and methodology, and then present our experimental results.

3.1 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 server hosts front-end microservices, one hosts back-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 [4] with Elasticsearch [3] 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.

We employ the *social networking* application from the DeathStarBench benchmark [25] 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 user's own timeline, and 60% requests that read the timeline of other users; this division is based on an empirical estimate of a user's typical behaviour.

We change the type of server in the social networking application of DeathStarBench to *TNonblockingServer*. The Apache Thrift C++ *TNonblockingServer* provides good performance and exposes numerous settings for the developer to customize the server [10]. We also make modifications to change the thread pool size dynamically based on the value suggested by the optimizer for each iteration.

3.2 Evaluation methodology

For evaluation, we consider the 95^{th} 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 [1, 5–7, 10] to identify the performance-impacting parameters. For

each configuration to be tested, we run the application under that configuration three times for a duration of 5 minutes each, and collect latency metrics across all runs. We next discuss the optimization algorithms and dimensionality reduction strategies we investigate in our evaluation.

3.2.1 Black-box optimization algorithms. We consider 6 optimization algorithms in our evaluation. The first 2 are representative of *heuristic-based probabilistic algorithms*, the next 2 are evolutionary algorithms inspired by population-based biological evolution, and the last 2 are *sequential model-based optimization algorithms* that approximate the objective function with a cheaper, surrogate function [13] to aid optimization. We use skopt [9], Hyperopt [14], and Nevergrad [38] libraries to implement the algorithms.

1. ***Simulated Annealing*** (SA) [36] starts with an initial configuration, c_0, and at each iteration considers a neighbouring configuration, c_n. It then picks the next configuration based on the value of the objective function at c_0 and c_n and a time varying parameter, T, whose value slowly decreases (annealing) with each iteration leading to more exploitation and less exploration.
2. ***Dynamically Dimensioned Search*** (DDS) starts with an initial configuration and then perturbs the values of the parameters of the configuration based on a perturbation factor [42]. The algorithm moves from a global search towards a local search as the iterations progress by dynamically and probabilistically reducing the number of parameters that are perturbed.
3. ***Particle Swarm Optimization*** (PSO) [32] works by iteratively improving the candidate solution with regard to the objective function. Each candidate solution is known as a particle and all particles together form a swarm. The particles are moved around the search-space based on the best value the particle has seen so far (exploration) and the global best value seen by the whole swarm (exploitation).
4. ***Genetic Algorithms*** (GA) [34] start with an initial random population of candidate configurations which is then pruned based on the value of the objective function at these configurations. This pruned subset is used to generate a new set of candidates through mutation (randomly changing the configurations of some parameters) and crossover (combining configurations of the candidates).
5. ***Bayesian Optimization*** (BO) starts with a prior distribution of the search space guided by the surrogate; we experiment with the popular Gaussian Process (GP) [13], Gradient Boosted Regression Trees (GBRT) [22], and Random Forests (RF) [23] surrogate models. The posterior distribution is updated at each step of exploration using Bayesian method.
6. ***Tree-structured Parzen Estimator*** (TPE) is similar to BO, but models the likelihood and prior instead of the posterior [13].

3.2.2 Dimensionality reduction strategies. If an application has m microservices each with p_i parameters (for $i = 1, 2, \ldots, m$), 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 not all optimization algorithms work well in high dimensions (number of tunable parameters, in our case), for example, Bayesian Optimization (BO) is known to not perform well when the number of parameters to optimize is more than 20 [35].

1. ***Critical path.*** In the call graph of a request, the critical path is the path formed by microservices that determines the latency of the request. We employ standard practices [37] to determine the critical path of a request and only consider configuration tuning for these microservices. We rely on the service time (or span) measurements provided by Jaeger for each microservice to determine the critical path. We also exclude all microservices on the critical path whose service time is less than 1ms; we find that such microservices do not contribute significantly to latency and can be omitted to reduce the configuration search space (by as much as 33% in our experiments).
2. ***Known bottlenecks.*** Prior work on performance diagnosis of microservices applications conducted thorough empirical analysis to identify performance bottlenecks [26]. We thus investigate configuration tuning only for the 8 bottleneck microservices identified by these works. Since this approach *requires prior knowledge of bottlenecks*, we consider it an unrealistic approach but one that serves as ground truth for comparison.
3. ***Performance variance.*** Prior works [20, 39, 41] demonstrated the improvement in performance that can be obtained by redesigning components that cause high performance variability. Inspired by this approach, we consider configuration tuning only for the 7 microservices that have a significant service time coefficient of variation [17] (above 0.5 in our experiments).

3.3 Experimental results

In practice, the optimization algorithms cannot be run indefinitely. Unless otherwise specified, we thus limit the number of configurations to be explored for each optimization algorithm to 15. For initialization, the optimization algorithms typically start with a random configuration. 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.

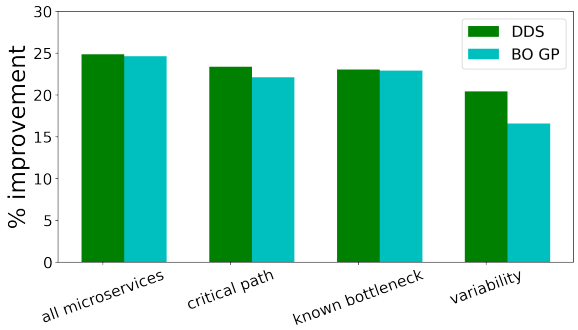

Figure 3. Evaluation of different dimensionality reduction techniques with respect to improvement in latency over the default configuration under DDS and Bayesian optimization.

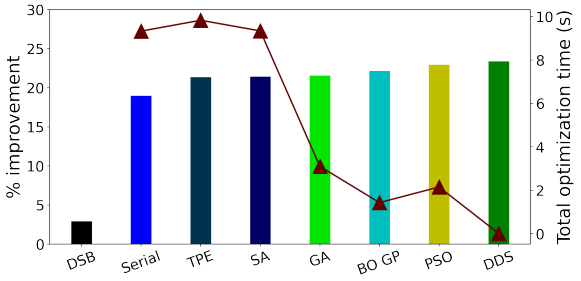

Figure 4. Comparison of improvement in latency compared to default configuration (left y-axis) and the time incurred by the optimization (right y-axis) for all algorithms when tuning the microservices on the critical path.

Efficacy of dimensionality reduction. Figure 3 shows the percentage improvement in tail (95^{th} percentile) latency of the social networking application under different dimensionality reduction techniques, compared to the tail latency when using the default configuration for all parameters. For ease of illustration, we show the results for two specific optimization algorithms. We see that tuning all 28 microservices of the social networking application provides about 25% improvement in tail latency. However, tuning only the microservices on the critical path (12 microservices) provides 22–23% improvement. Tuning the known bottlenecks provides similar improvements, suggesting that the critical path approach correctly identifies the microservices that have the most impact; note that the known bottlenecks approach requires prior, offline knowledge of bottlenecks, which is not always feasible and requires significant additional effort. Finally, by focusing on the variability causing microservices, the afforded improvement in latency is about 17–20%.

To further contrast the three different dimensionality reduction techniques, we consider the overlap in subsets of microservices chosen by the techniques. 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 three techniques have the ability to identify important, performance-impacting microservices.

Performance of different optimization algorithms. The bars in Figure 4 show the (sorted) percentage improvement (on left y-axis) in tail latency over the default configuration afforded by different optimization algorithms using the critical path approach. For comparison, we also show (as DSB) the improvement afforded (about 3%) by the configuration employed by the DeathStarBench benchmark developers [2]. We see that DDS provides the best improvement of 23.4%, followed closely by PSO (22.9%) and BO (22.1%). We note that for Bayesian Optimization, we experimented with various surrogate models; we found Gaussian Process (with the Expected Improvement acquisition function), referred to as BO GP in our figures, to provide the best results.

We also show with Serial the performance improvement afforded by the approach that sequentially tunes (using BO GP) each microservice on the critical path, as opposed to jointly tuning all microservices on the critical path. The Serial approach is similar to one of the baselines proposed in Vanir [15] for the cloud resource allocation problem (see Section 4). We find that Serial provides about 18.9% improvement, suggesting that independently tuning microservices can be sub-optimal. Further, we find that the order of optimization is also important for Serial; when sequentially optimizing microservices in the reverse order of the critical path, the percentage improvement is better.

To evaluate the overhead of different optimization algorithms, we plot (solid line with right y-axis) the time taken by the optimization across all iterations in Figure 4. We find that DDS requires the least amount of time (7ms), followed by BO (1.4s) and PSO (2.2s). SA and TPE incur a high overhead. Serial employs BO but takes a significant amount of time (9.3s) because it involves tuning each of the 12 microservices on the critical path sequentially with 15 iterations each.

Based on the above results, we conclude that, for our evaluation, *DDS is the best performing optimization algorithm.* An additional advantage of DDS is that it is designed for optimizing in high dimensions [42], making it suitable for enterprise applications that are composed of a large number of microservices.

Analysis of best configurations chosen by all the algorithms. As seen in Figure 4, different optimization algorithms provide comparable latency benefits. An important question in this context is whether the different algorithms are converging to the same globally optimal or different locally optimal configurations. Interestingly, we find that the best configuration chosen by different algorithms is indeed different. However, we do see some similarities and differences in the parameter settings in these configurations. The default *memory limit* value for Memcached is

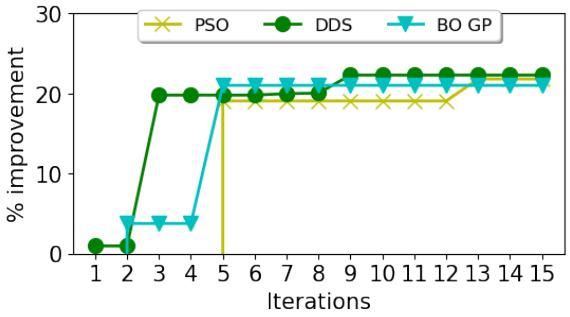

Figure 5. Comparison of tuning efficiency for the top 3 algorithms for 15 iterations when tuning on the critical path.

64MB; however, the best configurations suggested by the algorithms have a *memory limit* value of at least 4GB for the *post-storage-memcached* microservice. The threading and execution model in MongoDB is *synchronous* by default, but the best performing algorithm, DDS, sets this parameter to *adaptive* for all the MongoDB microservices on the critical path, suggesting that adaptive is a better option. Similarly, DDS sets the number of threads in *write-home-timeline-service* to 18, which is higher than the setting assigned by other algorithms, enabling it to exploit the host's processing power.

Convergence analysis of algorithms. The results shown in Figure 4 are based on the best configuration picked by the algorithms from among 15 iterations. To analyze the significance of number of iterations, we plot the best improvement afforded until different iterations for the top 3 algorithms in Figure 5. We see that DDS quickly finds good configurations as compared to BO and PSO. 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 4, suggesting that the optimization algorithms converge relatively quickly. This is a useful feature in practice given that each additional iteration imposes certain overhead and unavailability on the application.

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 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.

4 Related Work

Microservices configuration tuning. μTune [40] is a framework that reduces the tail latency of On-Line Data Intensive (OLDI) applications implemented as microservices. μTune builds empirically-derived piece-wise linear models for different threading models and thread pool sizes at various loads, which then guides the online tuning stage. However, as discussed in Section 3.3, microservices have numerous other parameters that can impact performance.

Application configuration tuning. There has been considerable research in parameter tuning for individual applications, such as Apache web server [44], Memcached [43], database [46] and storage systems [19], etc. While the above works can be used to tune individual microservices in isolation, the dependencies between microservices necessitates global optimization across microservices.

SmartConf [45] 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 [40].

BestConfig [47] 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 configuration tuning.

Cloud configuration tuning. Bilal et al. [16] perform an exhaustive comparison of existing black-box techniques for the problem of finding the best cloud configuration that minimizes objective functions like execution time or execution cost. Vanir [15] optimizes the cloud configuration for analytics clusters using Mondrian forest-based performance model and transfer learning. OPTIMUSCLOUD [33] jointly optimizes VM configurations and database configurations for cloud-deployed database systems by training a performance prediction model. Kaminski et al. [30] employ black-box optimization algorithms to find cost-effective resource assignments while meeting performance targets for a multi-tenant, container-based cloud environment. CherryPick [11] uses Bayesian Optimization (BO) to build a performance model for Big Data systems, which is then used to find the best cloud Configurations 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. [31] employ learning-based techniques to find the most important parameters of database systems that impact performance.

Carver [18] employs Latin Hypercube Sampling to explore the effect of different parameters on storage system performance and use the variance in performance caused by a parameter as an indicator of the parameter's importance. As discussed in Section 3, focusing on microservices on the critical path is a more effective approach than focusing on microservices that cause the most performance variation.

5 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 workshop paper *makes the case for configuration tuning of microservices*. We formulate and investigate the problem, identify the key challenges (large state space and inter-dependent parameters), and evaluate different techniques to address these challenges. Our experimental results on a popular benchmark application show that, with moderate effort, the tail latency of microservices applications can be improved by as much 23% by tuning the configuration parameters of specific microservices.

Acknowledgment: This work was supported by NSF grants CSN-1750109 and CNS-1717588.

References

[1] [n.d.]. Beginner's Guide. http://nginx.org/en/docs/beginners_guide.html.

[2] [n.d.]. DeathStarBench. https://github.com/delimitrou/DeathStarBench.

[3] [n.d.]. Elastic Search: The heart of the free and open Elastic Stack. https://www.elastic.co/elasticsearch/.

[4] [n.d.]. Jaeger: open source, end-to-end distributed tracing. https://www.jaegertracing.io/.

[5] [n.d.]. memcached(1) - Linux man page. https://linux.die.net/man/1/memcached.

[6] [n.d.]. MongoDB Server Parameters. https://docs.mongodb.com/manual/reference/parameters/.

[7] [n.d.]. Redis configuration. https://redis.io/topics/config.

[8] [n.d.]. Set wiredTigerConcurrentReadTransactions based on machine specs? https://jira.mongodb.org/browse/SERVER-19911.

[9] [n.d.]. SkOpt. https://scikit-optimize.github.io.

[10] Randy Abernethy. 2018. *The Programmer's Guide to Apache Thrift*. Manning publications.

[11] 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 *Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation* (Boston, MA, USA) *(NSDI'17)*. USENIX Association, USA, 469–482.

[12] Luiz André Barroso, Jimmy Clidaras, and Urs Hölzle. 2013. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines, Second Edition*. http://dx.doi.org/10.2200/S00516ED2V01Y201306CAC024

[13] James Bergstra, Rémi Bardenet, Yoshua Bengio, and Balázs Kégl. 2011. Algorithms for Hyper-Parameter Optimization. In *Advances in Neural Information Processing Systems*, J. Shawe-Taylor, R. Zemel, P. Bartlett, F. Pereira, and K. Q. Weinberger (Eds.), Vol. 24. Curran Associates, Inc. https://proceedings.neurips.cc/paper/2011/file/86e8f7ab32cfd12577bc2619bc635690-Paper.pdf

[14] J. Bergstra, D. Yamins, and D. D. Cox. 2013. 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 - Volume 28* (Atlanta, GA, USA) *(ICML'13)*. JMLR.org, I–115–I–123.

[15] Muhammad Bilal, Marco Canini, and Rodrigo Rodrigues. 2020. Finding the Right Cloud Configuration for Analytics Clusters. In *Proceedings of the 11th ACM Symposium on Cloud Computing* (Virtual Event, USA) *(SoCC '20)*. Association for Computing Machinery, New York, NY, USA, 208–222. https://doi.org/10.1145/3419111.3421305

[16] Muhammad Bilal, Marco Serafini, Marco Canini, and Rodrigo Rodrigues. 2020. Do the Best Cloud Configurations Grow on Trees? An Experimental Evaluation of Black Box Algorithms for Optimizing Cloud Workloads. *Proc. VLDB Endow.* 13, 12 (July 2020), 2563–2575. https://doi.org/10.14778/3407790.3407845

[17] Charles E. Brown. 1998. *Coefficient of Variation*. Springer Berlin Heidelberg, Berlin, Heidelberg, 155–157. https://doi.org/10.1007/978-3-642-80328-4_13

[18] Zhen Cao, Geoff Kuenning, and Erez Zadok. 2020. Carver: Finding Important Parameters for Storage System Tuning. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*. USENIX Association, Santa Clara, CA, 43–57. https://www.usenix.org/conference/fast20/presentation/cao-zhen

[19] Zhen Cao, Vasily Tarasov, Sachin Tiwari, and Erez Zadok. 2018. Towards Better Understanding of Black-box Auto-Tuning: A Comparative Analysis for Storage Systems. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. USENIX Association, Boston, MA, 893–907. https://www.usenix.org/conference/atc18/presentation/cao

[20] Jeffrey Dean and Luiz André Barroso. 2013. The Tail at Scale. *Commun. ACM* 56, 2 (Feb. 2013), 74–80. https://doi.org/10.1145/2408776.2408794

[21] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. 2007. Dynamo: Amazon's Highly Available Key-Value Store. In *Proceedings of Twenty-First ACM SIGOPS Symposium on Operating Systems Principles* (Stevenson, Washington, USA) *(SOSP '07)*. Association for Computing Machinery, New York, NY, USA, 205–220. https://doi.org/10.1145/1294261.1294281

[22] J. Elith, J. R. Leathwick, and T. Hastie. 2008. A working guide to boosted regression trees. *Journal of Animal Ecology* 77, 4 (2008), 802–813. https://doi.org/10.1111/j.1365-2656.2008.01390.x arXiv:https://besjournals.onlinelibrary.wiley.com/doi/pdf/10.1111/j.1365-2656.2008.01390.x

[23] Khaled Fawagreh, Mohamed Medhat Gaber, and Eyad Elyan. 2014. Random forests: from early developments to recent advancements. *Systems Science & Control Engineering* 2, 1 (2014), 602–609. https://doi.org/10.1080/21642583.2014.956265 arXiv:https://doi.org/10.1080/21642583.2014.956265

[24] Y. Gan and C. Delimitrou. 2018. The Architectural Implications of Cloud Microservices. *IEEE Computer Architecture Letters* 17, 2 (2018), 155–158. https://doi.org/10.1109/LCA.2018.2839189

[25] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayantara Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, Kelvin Hu, Meghna Pancholi, Brett Clancy, Chris Colen, Fukang Wen, Catherine Leung, Siyuan Wang, Leon Zaruvinsky, Mateo Espinosa, Yuan He, and Christina Delimitrou. 2019. 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)* (Providence, RI).

[26] Yu Gan, Yanqi Zhang, Kelvin Hu, Yuan He, Meghna Pancholi, Dailun Cheng, and Christina Delimitrou. 2019. 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)* (Providence, RI).

[27] Robert Heinrich, André van Hoorn, Holger Knoche, Fei Li, Lucy Ellen Lwakatare, Claus Pahl, Stefan Schulte, and Johannes Wettinger. 2017. Performance Engineering for Microservices: Research Challenges and Directions. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering Companion* (L'Aquila, Italy) *(ICPE '17 Companion)*. Association for Computing Machinery, New York, NY, USA, 223–226. https://doi.org/10.1145/3053600.3053653

[28] Kasun Indrasiri and Danesh Kuruppu. 2020. *gRPC: Up and Running*. O'Reilly Media.

[29] P. Jamshidi, C. Pahl, N. C. Mendonça, J. Lewis, and S. Tilkov. 2018. Microservices: The Journey So Far and Challenges Ahead. *IEEE Software* 35, 3 (2018), 24–35. https://doi.org/10.1109/MS.2018.2141039

[30] Matthijs Kaminski, Eddy Truyen, Emad Heydari Beni, Bert Lagaisse, and Wouter Joosen. 2019. A Framework for Black-Box SLO Tuning of Multi-Tenant Applications in Kubernetes. In *Proceedings of the 5th International Workshop on Container Technologies and Container Clouds* (Davis, CA, USA) *(WOC '19)*. Association for Computing Machinery, New York, NY, USA, 7–12. https://doi.org/10.1145/3366615.3368352

[31] Konstantinos Kanellis, Ramnatthan Alagappan, and Shivaram Venkataraman. 2020. 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)*. USENIX Association. https://www.usenix.org/conference/hotstorage20/presentation/kanellis

[32] J. Kennedy and R. Eberhart. 1995. Particle swarm optimization. In *Proceedings of ICNN'95 - International Conference on Neural Networks*, Vol. 4. 1942–1948 vol.4. https://doi.org/10.1109/ICNN.1995.488968

[33] Ashraf Mahgoub, Alexander Michaelson Medoff, Rakesh Kumar, Subrata Mitra, Ana Klimovic, Somali Chaterji, and Saurabh Bagchi. 2020. OPTIMUSCLOUD: Heterogeneous Configuration Optimization for Distributed Databases in the Cloud. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, 189–203. https://www.usenix.org/conference/atc20/presentation/mahgoub

[34] Seyedali Mirjalili. 2019. *Genetic Algorithm*. Springer International Publishing, Cham, 43–55. https://doi.org/10.1007/978-3-319-93025-1_4

[35] R. Moriconi, M.P. Deisenroth, and K.S. Sesh Kumar. 2020. High-dimensional Bayesian optimization using low-dimensional feature spaces. *Mach Learn 109, 1925–1943* (2020). https://doi.org/10.1007/s10994-020-05899-z

[36] Panos M. Pardalos and Thelma D. Mavridou. 2009. *Simulated annealingSimulated Annealing*. Springer US, Boston, MA, 3591–3593. https://doi.org/10.1007/978-0-387-74759-0_617

[37] Haoran Qiu, Subho S. Banerjee, Saurabh Jha, Zbigniew T. Kalbarczyk, and Ravishankar K. Iyer. 2020. FIRM: An Intelligent Fine-grained Resource Management Framework for SLO-Oriented Microservices. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 805–825. https://www.usenix.org/conference/osdi20/presentation/qiu

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

[39] D. Skinner and W. Kramer. 2005. Understanding the causes of performance variability in HPC workloads. In *IEEE International. 2005 Proceedings of the IEEE Workload Characterization Symposium, 2005.* 137–149. https://doi.org/10.1109/IISWC.2005.1526010

[40] Akshitha Sriraman and Thomas F. Wenisch. 2018. µTune: Auto-Tuned Threading for OLDI Microservices. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, Carlsbad, CA, 177–194. https://www.usenix.org/conference/osdi18/presentation/sriraman

[41] Amoghavarsha Suresh and Anshul Gandhi. 2019. Using Variability as a Guiding Principle to Reduce Latency in Web Applications via OS Profiling. In *The World Wide Web Conference* (San Francisco, CA, USA) *(WWW '19)*. Association for Computing Machinery, New York, NY, USA, 1759–1770. https://doi.org/10.1145/3308558.3313406

[42] Bryan A. Tolson and Christine A. Shoemaker. 2007. Dynamically dimensioned search algorithm for computationally efficient watershed model calibration. *Water Resources Research* 43, 1 (2007). https://doi.org/10.1029/2005WR004723 arXiv:https://agupubs.onlinelibrary.wiley.com/doi/pdf/10.1029/2005WR004723

[43] Muhammad Wajahat, Salman Masood, Abhinav Sau, and Anshul Gandhi. 2017. 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 (IGSC '17)*. Orlando, FL, USA.

[44] Qingyang Wang, Shungeng Zhang, Yasuhiko Kanemasa, Calton Pu, Balaji Palanisamy, Lilian Harada, and Motoyuki Kawaba. 2019. Optimizing N-Tier Application Scalability in the Cloud: A Study of Soft Resource Allocation. *ACM Trans. Model. Perform. Eval. Comput. Syst.* 4, 2, Article 10 (June 2019), 27 pages. https://doi.org/10.1145/3326120

[45] Shu Wang, Chi Li, Henry Hoffmann, Shan Lu, William Sentosa, and Achmad Imam Kistijantoro. 2018. Understanding and Auto-Adjusting Performance-Sensitive Configurations. *SIGPLAN Not.* 53, 2 (March 2018), 154–168. https://doi.org/10.1145/3296957.3173206

[46] Bohan Zhang, Dana Van Aken, Justin Wang, Tao Dai, Shuli Jiang, Jacky Lao, Siyuan Sheng, Andrew Pavlo, and Geoffrey J. Gordon. 2018. A Demonstration of the Ottertune Automatic Database Management System Tuning Service. *Proc. VLDB Endow.* 11, 12 (Aug. 2018), 1910–1913. https://doi.org/10.14778/3229863.3236222

[47] Yuqing Zhu, Jianxun Liu, Mengying Guo, Yungang Bao, Wenlong Ma, Zhuoyue Liu, Kunpeng Song, and Yingchun Yang. 2017. BestConfig: Tapping the Performance Potential of Systems via Automatic Configuration Tuning. In *Proceedings of the 2017 Symposium on Cloud Computing* (Santa Clara, California) *(SoCC '17)*. Association for Computing Machinery, New York, NY, USA, 338–350. https://doi.org/10.1145/3127479.3128605