BestConfig: Tapping the Performance Potential of Systems via Automatic Configuration Tuning*

Yuqing Zhu, Jianxun Liu, Mengying Guo, Yungang Bao, Wenlong Ma,
Zhuoyue Liu, Kunpeng Song, Yingchun Yang
Institute of Computing Technology, Chinese Academy of Sciences
Huawei
{zhuyuqing,liujianxun,guomengying,baoyg,mawenlong}@ict.ac.cn
{liuzhuoyue,songkunpeng,yingchun.yang}@huawei.com

ABSTRACT

An ever increasing number of configuration parameters are provided to system users. But many users have used one configuration setting across different workloads, leaving untapped the performance potential of systems. A good configuration setting can greatly improve the performance of a deployed system under certain workloads. But with tens or hundreds of parameters, it becomes a highly costly task to decide which configuration setting leads to the best performance. While such task requires the strong expertise in both the system and the application, users commonly lack such expertise.

To help users tap the performance potential of systems, we present BestConfig, a system for automatically finding a best configuration setting within a resource limit for a deployed system under a given application workload. BestConfig is designed with an extensible architecture to automate the configuration tuning for general systems. To tune system configurations within a resource limit, we propose the divide-and-diverge sampling method and the recursive bound-and-search algorithm. BestConfig can improve the throughput of Tomcat by 75%, that of Cassandra by 63%, that of MySQL by 430%, and reduce the running time of Hive join job by about 50% and that of Spark join job by about 80%, solely by configuration adjustment.

CCS CONCEPTS

• **General and reference** → **Performance**; *Evaluation*; • **Software and its engineering** → **Software performance**; *System administration*; • **Information systems** → Database utilities and tools;

KEYWORDS

automatic configuration tuning, ACT, performance optimization

*Yuqing Zhu is the corresponding author.

SoCC '17, September 24–27, 2017, Santa Clara, CA, USA

ACM ISBN 978-1-4503-5028-0/17/09…$15.00
https://doi.org/10.1145/3127479.3128605

ACM Reference Format:
Yuqing Zhu, Jianxun Liu, Mengying Guo, Yungang Bao, Wenlong Ma, Zhuoyue Liu, Kunpeng Song, Yingchun Yang. 2017. BestConfig: Tapping the Performance Potential of Systems via Automatic Configuration Tuning. In *Proceedings of SoCC '17, Santa Clara, CA, USA, September 24–27, 2017,* 13 pages.
https://doi.org/10.1145/3127479.3128605

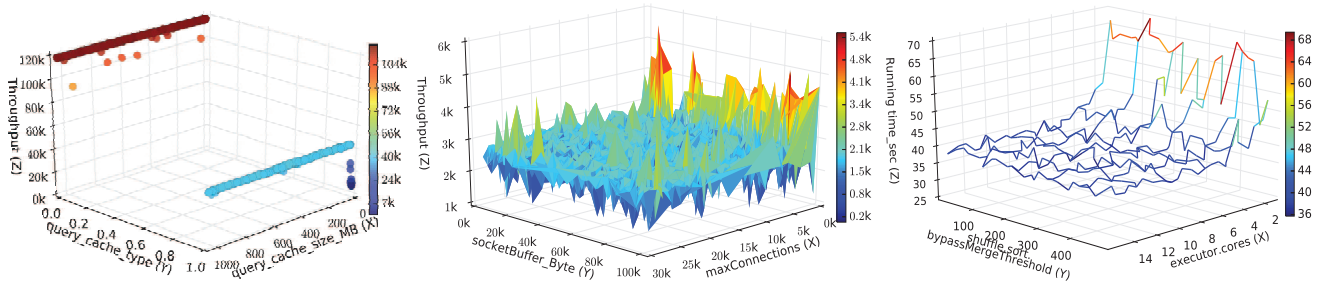
1 INTRODUCTION

More and more configuration parameters are provided to users, as systems in the cloud aim to support a wide variety of use cases [42]. For example, Hadoop [18], a popular big data processing system in the cloud, has more than 180 configuration parameters. The large number of configuration parameters lead to an ever-increasing complexity of configuration issues that overwhelms users, developers and administrators. This complexity can result in configuration errors [5, 27, 34, 35]. It can also result in unsatisfactory performances under atypical application workloads [6, 11, 19, 41]. In fact, configuration settings have strong impacts on the system performance [9, 26, 32, 36]. To tap the performance potential of a system, system users need to find an appropriate configuration setting through configuration tuning.

A good configuration setting can greatly improve the system performance. For instance, changing the *query_cache_type* parameter of MySQL from zero to one can result in more than **11 times** performance gain for an application workload, as shown in Figure 1(a). This performance gain can be significant if the workload is recurring on a daily base—this is very likely, especially for systems like databases or web servers. Nevertheless, configuration tuning for general systems is difficult due to the following three matters.

Variety. Systems for configuration tuning can be data analytic systems like Hadoop [18] and Spark [31], database systems like MySQL [28], or web servers like Tomcat [37]. Various deployments for a system are possible in the cloud. Performance goals concerning users can be throughput, latency, running time, etc. Among the variety of performance goals, some need to be maximized, while some minimized. The configuration tuning process must also take the application workload into account, and there are a variety of possible workloads. Furthermore, various combinations of systems, performance goals and workloads are possible.

Complexity. Given different performance goals and applied different workloads, a deployed system has different performance surfaces for a given set of configuration parameters. Different systems can have highly diverse and complex performance surfaces.



(a) MySQL under uniform reads (b) Tomcat under webpage navigation workload (c) Spark under HiBench-KMeans workload

Figure 1: Diverging performance surfaces of *MySQL*, *Tomcat* and *Spark*. (Best view in color)

Take Figure 1(a), 1(b) and 1(c) for example, MySQL has no performance surface but only two lines, while Tomcat has a bumpy performance surface and Spark has a relatively smooth performance surface. Previously, an unexpected performance surface is reported for PostgreSQL [11], which even costs system developers months of efforts to reason about the underlying interactions.

Overhead. Configuration tuning involves solving a problem with a high-dimensional parameter space, thus a large sample set is commonly needed to find a solution [16]. However, collecting a large set of performance-configuration samples is impractical for configuration tuning. As no performance simulator exists for general systems, the samples can only be generated through real tests on the deployed system. Hence, configuration tuning must restrain the overhead of sample collection. Besides, the time overhead of the optimization process must also be considered.

Existing solutions do not fully address all the above challenges. Though sporadic proposals are found on automatically suggesting configuration settings for Web servers [6, 41, 43], databases [11] and Hadoop [4, 19] respectively, these solutions are not generally applicable to the variety of systems in the cloud. A few statistical or machine learning models are proposed for distributed systems [8, 14], but these models are not applicable to the complicated cases as shown from Figure 1(a) to 1(c). Configuration tuning is related to the problem of optimizing the performance for systems with high-dimensional parameters [21], but previous research typically studies the problem based on simulations [16]; the overhead aspect is rarely considered to the extent as required by configuration tuning for general systems.

In this paper, we present BestConfig—an automatic configuration tuning system that can optimize performance goals for general systems in the cloud by adjusting configuration parameters and that can recommend the best configuration setting found within a given resource limit. A typical resource limit is the number of tests allowed for configuration tuning. To address the resource limit challenge, BestConfig adopts an effective sampling method with wide space coverage and this coverage will be improved as more resources are provided. With the variety of systems and workloads, as well as the complexity of their interactions, it is impossible to build a useful performance model on a limited number of samples. Hence, BestConfig adopts a search-based optimization algorithm and exploits the general properties of performance models. To facilitate the usage with the variety of deployed systems and workloads, we design for BestConfig a software architecture that has loosely coupled but extensible components and that adopts a sample-test-optimize process in closed loop.

In the evaluation with extensive experiments, BestConfig can improve the throughput of Tomcat by 75%, that of Cassandra [7] by 63%, that of MySQL by 430%, and reduce the running time of Hive-over-Hadoop [20] join job by about 50% and that of Spark join job by about 80%, as compared to the default configuration setting, simply by adjusting configuration settings.

In sum, this paper makes the following contributions.

- To the best of our knowledge, we are the first to propose and the first to implement an automatic configuration tuning system for general systems. And, our system successfully automates the configuration tuning for six systems widely deployed in the cloud.
- We propose an architecture (§3.4) that can be easily plugged in with general systems and any known system tests. It also enables the easy testing of other configuration tuning algorithms.
- We propose the divide-and-diverge sampling method (§4.1) and the recursive-bound-and-search method (§4.2) to enable configuration tuning for general systems within a resource limit.
- We demonstrate the feasibility and the benefits of BestConfig through extensive experiments (§5), while refusing the possibility of using common model-based methods such as linear or smooth prediction models for general systems (§5.1).
- We have applied BestConfig to a real use case (§6) showing that, even when a cloud deployment of Tomcat has a full resource consumption rate, BestConfig can still improve the system performance solely by configuration tuning.

2 BACKGROUND AND MOTIVATION

In this section, we describe the background and the motivation of automatic configuration tuning for general systems. We also analyze the challenges in solving this problem.

2.1 Background

Configuration tuning is crucial to obtaining a good performance from a deployed system. For an application workload, a configuration setting leads to the best performance of the system, but it might not be optimal given another application workload. Take Figure 1(a) for example. Under the *uniform-read* workload, the value of *query_cache_type* is key to a good performance; but, as shown in

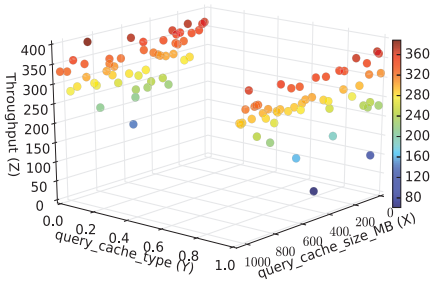


Figure 2: The performance surface for MySQL under the Zipfian read-write workload. (Best view in color)

Figure 2, the *query_cache_type* value has no obvious relation with the system performance for a *Zipfian read-write* workload. In fact, the default setting generally cannot achieve the best performance of a system under all workloads.

Configuration tuning is highly time-consuming and laborious. It requires the users: (1) to find the heuristics for tuning; (2) to manually change the system configuration settings and run workload tests; and, (3) to iteratively go through the second step many times till a satisfactory performance is achieved. Sometimes, the heuristics in the first step might be misguiding, as some heuristics are correct for one workload but not others; then, the latter two steps are in vain. In our experience of tuning MySQL, it has once taken five junior employees about half a year to find an appropriate configuration setting for our cloud application workloads.

Configuration tuning is even not easy for experienced developers. For example, it has been shown that, although PostgreSQL's performance under the workload of a TPC-H query is a smooth surface with regard to the configuration parameters of cache size and buffer size [11], the cache size interacts with the buffer size in a way that even takes the system developers great efforts to reason about the underlying interactions. In fact, the system performance models can be highly irregular and complicated, as demonstrated by Figure 1(a) to 1(c) and Figure 2. How the configuration settings can influence the system performance can hardly be anticipated by general users or expressed by simple models.

Benefits. Automatic configuration tuning can greatly benefit system users. First, a good configuration setting will improve the system performance by a large margin, while automatic configuration tuning can help users find the good configuration setting. Second, a good configuration setting is even more important for repetitive workloads, and recurring workloads are in fact a common phenomenon [1, 13]. Third, automatic configuration tuning enables fairer and more useful benchmarking results, if the system under test is automatically tuned for a best configuration setting before benchmarking—because the system performance is related to both the workload and the configuration setting.

2.2 Challenges

Several challenges exist for automatic configuration tuning for general systems. These challenges must be addressed *simultaneously*.

Variety of performance goals: Users can have different performance goals for configuration tuning. For a data analytical job on Spark, the performance goal is normally to reduce the running time, while for a data-access workload on MySQL, it would be to increase the throughput. Sometimes, users can have multiple performance goals, e.g., increasing the throughput and decreasing the average latency of individual operations for MySQL. Some users would also require to improve the performance goal such as throughput but not to worsen other metrics such as memory usage. Besides, some performance goals need to be maximized, while some need to be minimized.

Variety of systems and workloads: To tune the variety of systems and workloads, we cannot build or have users build performance models for the tuning purpose as previously done for Hadoop [19]. Some deployed systems are distributed, e.g., Spark and Hadoop, while some are standalone, e.g., Tomcat or one-node Hadoop. A system's performance model can be strongly influenced by the hardware and software settings of the deployment environment [46]. Hence, the automatic configuration tuning system must handle the variety of deployment environments. It must enable an easy usage with the deployed systems and workloads. The heterogeneity of deployed systems and workloads can have various performance models for tuning, leading to different best configuration settings and invalidating the reuse of samples across different deployments [39, 46].

High-dimensional parameter space: As mentioned previously, many systems in the cloud now have a large number of configuration parameters, i.e., a high-dimensional parameter space for configuration tuning. On the one hand, it is impossible to get the complete image of the performance-configuration relations without samples covering the whole parameter space. On the other hand, collecting too many samples is too costly. The typical solutions to the optimization problem over high-dimensional spaces generally assume the abundance of samples. For example, some solve the optimization problem with around *10* parameters using about *2000* samples [16, 43]. Except through simulations, it is too costly to collect such an amount of samples in practice; thus, such solutions are not applicable to the configuration tuning problem of real systems.

Limited samples: It is impractical to collect a large number of performance-configuration samples in practice. Besides, it is impossible to build a performance simulator for every system in the cloud, thus making the simulation-based sample collection infeasible. We have to collect samples through real tests against the deployed systems. Thus, methods used for configuration tuning cannot rely on a large sample set. Rather, it should produce results even on a limited number of samples. And, as the number of samples is increased, the result should be improved.

3 BESTCONFIG DESIGN

BestConfig is designed to automatically find, within a given resource limit, a configuration setting that can optimize the performance of a deployed system under a specific application workload. We call the process of adjusting configuration settings as configuration tuning (or just tuning) and the system to adjust as SUT (System Under Tune).

3.1 Design Overview

To satisfy users' various needs on performance optimization and simultaneously simplify the optimization problem, we adopt the

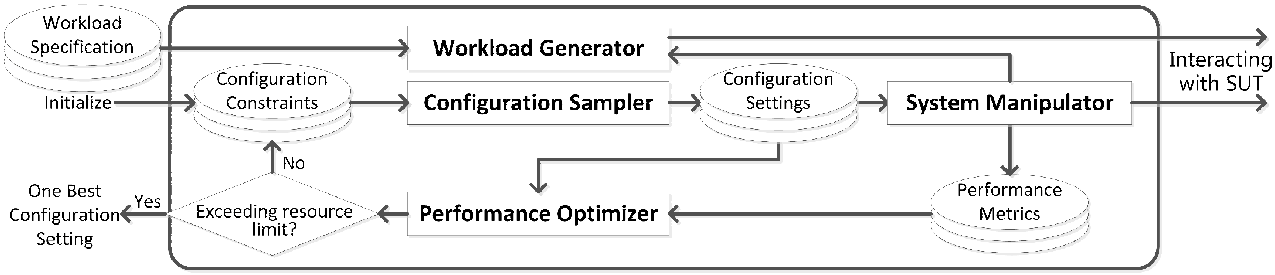


Figure 3: The automatic configuration tuning process and the major components of BestConfig.

utility function approach to amalgamating multiple performance optimization needs into a single maximization goal. BestConfig exposes an interface for users to express their performance optimization goals (§3.3).

To handle the variety of deployed systems and workloads, BestConfig is designed with a flexible architecture that has loosely coupled components, as sketched in Figure 3. These components are connected through data flows. The system manipulator component is an interface to interact with an SUT deployed in the target environment, while the workload generator component allows the easy plug-in of any target workload.

With limited samples, the tuning process must collect samples by careful choices. Different combinations of deployed systems and workloads can require different sampling choices. Thus, we design the BestConfig architecture with a sampler component that interacts with the system manipulator on sample collection. Besides, the performance optimization process can introduce more information on which configuration settings to sample; thus, the performance optimizer component of BestConfig is designed to interact with the sampler to pass on such knowledge. The resulting architecture of BestConfig is detailed in Section 3.4.

To address the configuration tuning problem, we must solve the two subproblems of **sampling** and **performance optimization (PO)** simultaneously. Due to the challenges of high-dimensional parameter space and limited samples, the sampling subproblem differs from the random sampling in related works. It must be solved with additional conditions as detailed in Section 3.6. The PO subproblem also faces similar conditions (§3.6). A feasible solution to automatic configuration tuning for general systems must address all the conditions for the two subproblems.

BestConfig exploits the sampling information when solving the PO subproblem, and vice versa. In contrast, sampling and performance optimization are generally addressed **separately** in related works. We combine the sampling method DDS (Divide and Diverge Sampling) with the optimization algorithm RBS (Recursive Bound and Search) as a complete solution. DDS will sample for later RBS rounds in subspaces that are not considered in early RBS rounds. In this way, the requirement on wide space coverage for sampling is better satisfied. Furthermore, RBS exploits DDS in the bounded local search to reduce the randomness and increase the effectiveness of the search. In comparison, sampling was rarely considered and exploited for the local search step in related works. We detail DDS and RBS in Section 4. In the following, we first describe the key steps for automatic configuration tuning (§3.2).

3.2 Key Steps for Configuration Tuning

Figure 3 sketches the automatic configuration tuning process of BestConfig. The tuning process is in closed loop. It can run in as many loops as allowed by the resource limit. The resource limit is typically the number of tests that are allowed to run in the tuning process. It is provided as an input to the tuning process. Other inputs include the configuration parameter set and their lower/upper bounds (denoted as *configuration constraints*). The output of the process is a configuration setting with the optimal performance found within a given resource limit.

Given *configuration constraints*, the configuration sampler generates a number of configuration settings as allowed by the resource limit. The configuration settings are then used to update the configuration setting of the SUT. For each configuration setting, a test is run against the SUT; and, the corresponding performance results are collected. The performance results are then transformed into a scalar performance metric through the utility function.

All the sample pairs of the performance metric and the corresponding configuration setting are used by the performance optimization (PO) algorithm. The PO algorithm finds a configuration setting with the best performance. If the resource limit permits more tests and samples, the PO algorithm will record the found configuration setting and output a new set of configuration constraints for the next tuning loop. Otherwise, the tuning process ends and BestConfig outputs the configuration setting with the best performance found so far.

3.3 Performance Metric by Utility Function

BestConfig optimizes towards a scalar performance metric, which has only a single value. The scalar performance metric is defined by a *utility function*, with user-concerned performance goals as inputs. If only one performance goal is concerned, e.g., the throughput or the latency, the utility function is the identity function, i.e., $f(x) = x$, where x is the performance goal. If multiple performance goals are concerned simultaneously, we can define the utility function as a weighted summation. For example, if a user wants to increase the throughput and decrease the latency, the utility function can be defined as $f(x_t, x_l) = x_t/x_l$, where x_t is the throughput and x_l the latency. In case that the throughput must be increased and the memory usage must not exceed the threshold c_m, an example utility function is $f(x_t, x_m) = x_t \times S(c_m - x_m - 5)$, where x_m is the memory usage and $S(x)$ is the sigmoid function $S(x) = \frac{1}{1+e^{-x}}$. BestConfig allows users to define and implement their own utility functions through a `Performance` interface [45].

During the configuration tuning process, BestConfig maximizes the performance metric defined by the utility function. Although users can have performance goals that need to be minimized, we can easily transform the minimization problem into a maximization one, e.g., taking the inverse of the performance metric.

3.4 Highly-Extensible Architecture

BestConfig has a highly flexible and extensible architecture. The architecture implements the flexible configuration tuning process in closed loop. It allows BestConfig to be easily used with different deployed systems and workloads, requiring only minor changes.

The main components in BestConfig's architecture include *Configuration Sampler*, *Performance Optimizer*, *System Manipulator* and *Workload Generator*. *Configuration Sampler* implements the scalable sampling methods. *Performance Optimizer* implements the scalable optimization algorithms. *System Manipulator* is responsible for updating the SUT's configuration setting, monitoring states of the SUT and tests, manipulating the SUT, etc. *Workload Generator* generates application workloads. It can be a benchmark system like YCSB [10] or BigOP [10] running a benchmarking workload; or, it can be a user-provided testing system regenerating the real application workloads. The system manipulator and the workload generator are the only two components interacting with the SUT.

For extensibility, the components in the architecture are loosely coupled. They only interact with each other through the data flow of configuration constraints, configuration settings and performance metrics. The configuration sampler inputs the system manipulator with sets of configuration settings to be sampled. The system manipulator inputs the performance optimizer with the samples of performance-configuration pairs. The performance optimizer adaptively inputs new configuration constraints to the configuration sampler. With such a design, BestConfig's architecture allows different scalable sampling methods and scalable PO algorithms to be plugged into the configuration tuning process. On coping with different SUTs or workloads, only the system manipulator and the workload generator need to be adapted. With the extensible architecture, BestConfig can even optimize systems emerging in the future, with only slight changes to the system manipulator and the workload generator.

3.5 An Example of Extending BestConfig

With the current Java implementation of BestConfig [45], one can define a new sampling method by implementing the `ConfigSampler` interface. The `ConfigSampler` interface accepts the sample set size limit and a list of configuration constraints as inputs, and returns a list of configuration settings.

Similarly, to plug in a new PO algorithm, one can implement the `Optimization` interface. The implementation of the `Optimization` interface can accept a list of configuration settings and their corresponding performance metrics from the system manipulator. It must decide whether to continue the automatic configuration process or not, based on the given resource limit, e.g., the number of tests allowed. The best configuration setting can be output to a file, while the new configuration constraints are directly passed to the configuration sampler.

At present, extending the system manipulator requires only changing a few shell scripts that interact with the SUT and the workload generator. The workload generator is loosely coupled with other system components. Thus, it is highly convenient to integrate user-provided workload generation systems, e.g., YCSB and HiBench bundled in the BestConfig source [45]. Thanks to the highly extensible architecture, we have already applied BestConfig to six systems as listed in Table 1. The corresponding shell scripts for these systems are provided along with the BestConfig source.

3.6 Subproblems: Sampling and PO

The subproblem of sampling must handle all types of parameters, including boolean, enumeration and numeric. The resulting samples must have a wide coverage of the parameter space. To guarantee resource scalability, the sampling method must also guarantee a better coverage of the whole parameter space if users allow more tuning tests to be run. Thus, the sampling method must produce sample sets satisfying the following three conditions: (1) the set has a wide coverage over the high-dimensional space of configuration parameters; (2) the set is small enough to meet the resource limit and to reduce test costs; and, (3) the set can be scaled to have a wider coverage, if the resource limit is expanded.

The subproblem of performance optimization (PO) is to maximize the performance metric based on the given number of samples. It is required that the output configuration setting must improve the system performance than a given configuration setting, which can be the default one or one manually tuned by users. To optimize the output of a function/system, the PO algorithm must satisfy the following conditions: (1) it can find an answer even with a limited set of samples; (2) it can find a better answer if a larger set of samples is provided; and, (3) it will not be stuck in local sub-optimal areas and has the possibility to find the global optimum, given enough resources. Two categories of PO algorithms exist, i.e., model-based [8, 19, 39] and search-based [16, 41, 43]. In the design of BestConfig, we exploit the search-based methods.

We do not consider model-based PO methods for the following reasons. First, with the large number of configuration parameters, model-based methods would require a large number of samples to construct a useful model, thus violating the first condition of the PO subproblem. Second, model-based methods require the user to have a priori knowledge about the model, e.g., whether the model should be linear or quadratic, but it is mission impossible for general users to input such a priori information for each combination of SUT, deployment setting and workload. Third, model-based methods have hyper-parameters, which strongly impact how the model works; but setting these hyper-parameters is as hard as tuning the configuration setting of the SUT. Without enough samples, precise a priori knowledge or carefully-tuned hyper-parameters, model-based methods will not even work. In Section 5.1, we experiment with two model-based methods using limited samples. We demonstrate that these model-based methods hardly work in the configuration tuning problem with a resource limit.

4 DDS & RBS IN COOPERATION

To address the two subproblems of automatic configuration tuning, we propose the divide-and-diverge sampling (DDS) method and the recursive bound-and-search (RBS) algorithm. Although the sampling and PO methods can work separately, DDS and RBS in cooperation enables the effective tuning process of BestConfig.

4.1 DDS: Divide & Diverge Sampling

Parameter space coverage. To guarantee a wide coverage over the high-dimensional parameter space, we **divide** the space into subspaces. Then we can randomly select one point from each subspace. Thus, each subspace is represented by one sample. In comparison to the random sampling without subspace division, it is very likely that some subspaces are not represented, especially when the dimension of the space is high. Given n parameters, we can divide the range of each parameter into k intervals and collect combinations of the intervals. There are k^n combinations, thus k^n subspaces and samples. This way of sampling is called *gridding* or *stratified sampling*. Thanks to subspace division, gridding guarantees a complete coverage of the whole parameter space. But it also results in a sample set with a large cardinality, which is in exponential relation to the number of parameter dimensions. Hence, it violates the second requirement of the sampling subproblem.

Resource limit. To meet the second requirement, we reduce the number of subspaces to be sampled. We observe that, **the impact of an influential parameter's values on the performance can be demonstrated through comparisons of performances, disregard of other parameters' values.** For example, consider the performance model of MySQL as plotted in Figure 1(a). If the value of a parameter has great impacts on the performance like *query_cache_type*, we actually do not need to examine all combinations of the parameter's values with every other parameter's values. Instead, we need only examine each potentially outstanding value of the parameter once and compare the resulting performance with other samples. Thus, given a limited resource, we consider each interval of a parameter once, rather than making full combinations of all intervals.

After dividing parameter ranges into k intervals, we do not make a full combination of all intervals. Rather, we take a permutation of intervals for each parameter; then, we align the interval permutation for each paremeter and get k samples. For example, with two parameters X and Y divided into 6 range intervals respectively, we can take 6 samples as demonstrated in Figure 4. Each range interval of X is represented exactly once by the sample set. So is that of Y. For a given sample-set size, we **diverge** the set of sample points the most by representing each interval of each parameter exactly once.

Scalability. The third requirement for sampling is to be scalable with regard to the resource limit, e.g., the number of tests allowed, while meeting the previous two requirements. In fact, the above **divide-and-diverge sampling** (DDS) method directly meets the third requirement. The value of k is set according to the resource limit, e.g., k being equal to the number of tests allowed. Increasing the number of allowed tests, the number of samples will increase equally; thus, the parameter space will be divided more finely and the space coverage will be increased.

Furthermore, as the configuration tuning process is in closed loop, multiple times of sampling can be run. For the sake of scalability and coverage, DDS do not complete restart a new sampling process by redividing the whole space. Rather, on a request of re-sampling, DDS reuses its initial division of the whole parameter space and samples in subspaces not considered previously, while diverging the sample points as much as possible.

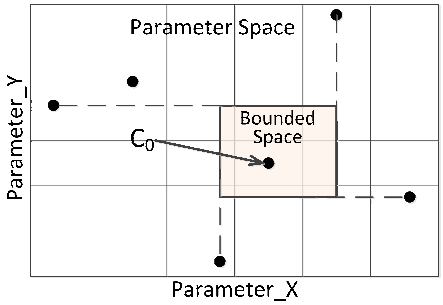


Figure 4: An example of running DDS and RBS for a 2D space.

Heterogeneity of parameters. Although DDS considers the continuous range of a parameter, DDS can be applied to parameters of boolean or categorical types by transforming them into parameters with continuous numeric ranges. Take the boolean type for example. We can first represent the parameter value of *true* and *false* by 1 and 0 respectively. Then, we let the values be taken from the range of $[0, 2)$, to which the DDS method can be directly applied. We can map a sampled value within ranges of $[0, 1)$ and $[1, 2)$ respectively to the values of 0 or 1, which are equal to *false* and *true* respectively. Similar mappings can be carried out for categorical or enumerative parameters as well.

4.2 RBS: Recursive Bound & Search

Consider the performance surfaces in Figure 1(b) and Figure 1(c). These performance plots have parameters with numeric values and continuous ranges, thus the performance surfaces are continuous surfaces. **Given a continuous surface, there is a high possibility that we find other points with similar or better performances around the point with the best performance in the sample set.** Even if the continuous performance surface might not be smooth, e.g., that in Figure 1(b), or if the performance surface is continuous only when projected to certain dimensions, e.g., that in Figure 1(a) when constrained to specific subspaces, the above observation still applies. Based on this observation, we design the RBS (Recursive Bound and Search) optimization algorithm.

Bound step. Given an initial sample set, RBS finds the point C_0 with the best performance. Then, it asks for another set of points sampled in the *bounded space* around C_0. Based on the observation in the last paragraph, there is a high possibility that we will find another point (say C_1) with a better performance. We can again sample in a bounded space around C_1. We can recursively carry out this bound-and-sample step until we find no point with a better performance in a sample set.

Here, there is a problem of how large the *bounded space* should be. According to the observation in Section 4.1, if a performance value shall have influential and positive impacts on the performance, it shall lead to a high performance in the sample set. For the initial sample set, parameter values other than those represented by C_0 are actually not having positive impacts as influential as C_0 on the performance, thus we should not consider them again given the limited resource. In other words, the *bounded space* around C_0 shall not include parameter values represented by any other points in the sample set. In addition, a high performance might be

achieved by any parameter value of those around C_0 but unrepresented in the sample set.

RBS fixes the bounds of the *bounded space* as follows. For each parameter p_i, RBS finds the largest value p_i^f that is represented in the sample set and that is smaller than that of C_0. It also finds the smallest value p_i^c that is represented in the sample set and that is larger than that of C_0. For the dimension represented by the parameter p_i, the bounded space has the bounds of (p_i^f, p_i^c). Figure 4 demonstrates this bounding mechanism of RBS. The same bounding mechanism can be carried out for every $C_j, j = 0, 1, \ldots$ in each bound-and-sample step.

By now, RBS has addressed the first two requirements for the PO subproblem. It finds an answer even with a limited set of samples by recursively taking the bound-and-sample step around the point C_j, which is the point with the best performance in a sample set. Let each bound-and-sample step called a *round*. RBS can adjust the size of the sample set and the number of rounds to meet the resource limit requirement. For example, given a limit of nr tests, RBS can run in r rounds with each sample set sized n. Given more resources, i.e., a larger number of allowed tests, RBS can carry out more bound-and-sample steps to search more finely in promising bounded subspaces.

Recursion step. To address the third requirement and avoid being stuck in a sub-optimal bounded subspace, RBS restarts from the beginning of the search by having the sampler to sample in the complete parameter space, if no point with a better performance can be found in a bound-and-sample step. This measure also enables RBS to find a better answer if a larger set of samples is provided. This is made possible through searching around more promising points scattered in the huge high-dimensional parameter space.

4.3 Why Combining DDS with RBS Works

In this section, we discuss about why combing DDS with RBS works in the configuration tuning problem. The performance of a system can be represented by a measurable objective function $f(x)$ on a parameter space D. In DDS, D is divided into orthogonal subspaces D_i. We define the distribution function of objective function values as:

$$\phi_{D_i}(y_0) = \frac{m(\{x \in D_i | f(x) \leq y_0\})}{m(D)} \quad (1)$$

where y_0 is the performance for the default configuration setting P_0 and $m(\cdot)$ denotes *Lebesgue measure*, a measure of the size of a set. For example, *Lebesgue measure* is area for a set of 2-dimensional points, and volume for a set of 3-dimensional points, and so on. The above equation thus represents the portion of points that have no greater performance values than P_0 in the subspace. The values of $\phi_{D_i}(y_0)$ fall within the range of $[0, 1]$. If a subspace has no points with greater performance values than y_0, it will have a zero value of $\phi(y_0)$. When all points in a subspace have higher performances, the subspace will have $\phi(y_0)$ evaluated to one.

DDS divides the whole high-dimensional space into subspaces and then samples in each subspace. A sample can be either greater or no greater than the default performance y_0. Assume all points with no greater performances are in set s_{i0} and those with greater ones are in set s_{i1}, we have $m(s_i) = m(s_{i0}) + m(s_{i1})$. Given $\phi_{D_i}(y_0)$ for subspace D_i, randomly sampling according to the *uniform distribution* will result in a $\phi_{D_i}(y_0)$ probability of getting points with no better performances and a $1 - \phi_{D_i}(y_0)$ probability of getting points with better performances.

Randomly sampling according to the uniform distribution, DDS will output samples with greater performances after around $n = 1/(1 - \phi_{D_i}(y_0))$ samples for subspace D_i. Although the exact value of n is not known, the principle underlying the uniform-random number generation guarantees that more samples will finally lead to the answer. In other words, given enough resources (i.e., samples), DDS will get a point with a greater performance than P_0.

RBS bounds and samples around the point with the best performance in a sample set. This key step works because, if a point C_j in a subspace D_i is found with the best performance, it is highly probable that the subspace D_i has a larger value of $1-\phi_{D_i}(y_0)$ than the other subspaces, as all subspaces are sampled for the same number of times in all rounds of RBS. According to the definition of $\phi_{D_i}(y_0)$, subspaces with larger values of $1-\phi_{D_i}(y_0)$ shall have more points that lead to performances greater than y_0, as compared to subspaces with smaller values of $1 - \phi_{D_i}(y_0)$. Thus, RBS can scale down locally around C_j to search again for points with better performances. As a result, the bound step of RBS, recursively used with DDS, will lead to a high probability of finding the point with the optimal performance. If the small probability event happens that the bound step runs in a subspace with a relatively small value of $1-\phi_{D_i}(y_0)$, the phenomenon of trapping in the local sub-optimal areas occurs. The recursion step of RBS is designed to handle this situation by sampling in the whole parameter space again.

5 EVALUATION

We evaluate BestConfig on six widely deployed systems, namely Hadoop [18], Hive [20], Spark [31], Cassandra [7], Tomcat [37], and MySQL [28]. These systems are deployed for Huawei's applications named Cloud+ and BI. To generate workloads towards systems under tune, we embed widely adopted benchmark tools in the workload generator. We use HiBench [22] for Hive+Hadoop and Spark, YCSB [10] for Cassandra, SysBench [24] for MySQL and JMeter [23] for Tomcat. Table 1 summarizes the evaluated systems along with the corresponding numbers of tuned parameters respectively. The detailed lists of the tuned parameters, as well as the detailed descriptions of the SUTs and the evaluated workloads, are accessible on the Web [45].

Table 1: The evaluated systems and parameters.

Software	Description	Language	# Parameters Tuned
Spark	Distributed computing	Scala	**30**
Hadoop	Distributed computing	Java	**109** (in all)
Hive	Data analytics	Java	
Cassandra	NoSQL database	Java	**28**
MySQL	Database server	C++	**11**
Tomcat	Web server	Java	**13**

Our experimental setup involves multiple local clusters of servers to deploy the six systems. If not specifically mentioned, the server is equipped with two 1.6GHz processors that have two physical cores, and 32GB memory, running CentOs 6.0 and Java 1.7.0_55. To avoid interference and comply with the actual deployment, we run the system under tune, the workload generator and other components of BestConfig on different servers. Further details can be found on the Web [45].

In the evaluation, we answer five questions:

(1) Why the configuration tuning problem with a resource limit is nontrivial (§5.1);
(2) How well BestConfig can optimize the performance of SUTs (§5.2);
(3) How effective the cooperation of DDS and RBS is (§5.3);
(4) How the sample set size and the number of rounds affect the tuning process (§5.4);
(5) Whether the configuration setting found by BestConfig will maintain its advantage over the given setting in tests outside the tuning process (§5.5).

5.1 Infeasibility of Model-based Methods

The difficulty of the configuration tuning problem can be demonstrated by the infeasibility of model-based methods. Common machine learning methods are model-based methods. They were previously used in optimization problems that have only a limited number of parameters. Based on the highly extensible architecture of BestConfig, we implement two PO algorithms, adopting the machine learning approach.

One is based on the COMT (Co-Training Model Tree) method [17], which assumes a linear relation between parameters and the performance. COMT divides the parameter space into subspaces and builds linear models for each subspace. Therefore, many common linear models can be taken as special cases of COMT. Besides, COMT is a semi-supervised machine learning method, which is designed and expected to work with a limited number of samples.

We train the COMT model using training sets of 100, 200 and 300 samples respectively. Each training set is randomly selected from a pool of 4000 samples, which are generated in a BestConfig tuning experiment over the Tomcat deployment described above. According to the COMT algorithm, the training not only exploits the training set, but also another set of unsampled points to reduce generalization errors. We validate the three learned models on the testing set with all samples in the sample pool. We summarize the prediction errors in Table 2, where *Avg. err. rate* is the average error rate and *Max. err. rate* the maximum error rate. Here, *error rate* is computed as the actual performance dividing the difference between the predicted performance and the actual performance.

Table 2: Linear-model based performance predictions.

Sample set size	Avg. err. rate	Max. err. rate
100	14%	240%
200	15%	1498%
300	138%	271510%

From Table 2, we can see that the predictions are in fact very much inaccurate. Although the first two average error rates look small, the corresponding models can make highly deviated predictions. The reason that more samples lead to worse predictions is twofold. One is because of model overfitting, and the other is due to the highly irregular performance surface of the SUT.

The other machine learning model we have tried is the GPR (Gaussian Process Regression) method [11], which assumes a differentiable performance function on parameters. It is the state-of-the-art model-based method adopted in a recent work on database tuning [39]. GPR does not predict the performance for a given point. Rather, it constructs the model based on the covariances between sample points and outputs points that are most probably to increase the performance the most, i.e., to achieve the best performance.

We experiment GPR using training sets with 100, 200 and 300 samples respectively. These sample sets are also collected from BestConfig tuning experiments over the Tomcat deployment described above. Among all the provided samples, GPR make a guess on which point would lead to the best performance (*best guess*). We then run a test on the best-guess point to get the actual performance. We compare the actual performance for GPR's best guess with that for the default configuration setting (*default*). Besides, we compare GPR's best guess with the real best point that has the optimal performance among all the provided samples, denoted as *real best*. The results are given in Table 3. We can see that, although the prediction is improving as the number of samples increases, GPR's predictions about best points are hardly accurate.

Table 3: GPR-based predictions on best points.

Sample set size	Bst. guess/dflt.	Bst. guess/rl. bst.
100	93%	56%
200	104%	63%
300	121%	58%

Because of the complexity of performance models, common model-based optimization methods, e.g. COMT and GPR, do not work well in the configuration tuning problem. In essence, the assumptions of such algorithms do not hold for the SUTs. As a result, methods like COMT and GPR cannot output competitive configuration settings. Moreover, their results do not guarantee to improve as the number of samples is increased, i.e., not scalable with the resource limit; instead, their results might worsen because of overfitting, violating the conditions of the PO subproblem.

5.2 Automatic Configuration Tuning Results

Figure 5 presents the automatic configuration tuning results for Cassandra, MySQL, Tomcat, Spark and Hive+Hadoop using BestConfig. For the latter three systems, we ran two tuning experiments with different benchmark workloads on each system. In all experiments, we set the sample set size to be 100 and the round number to be one. As demonstrated by the results, BestConfig improves the system performances in all experiments. Even though the Hive+Hadoop system has 109 parameters to tune, BestConfig can still make a performance gain. In comparison to the other SUTs,

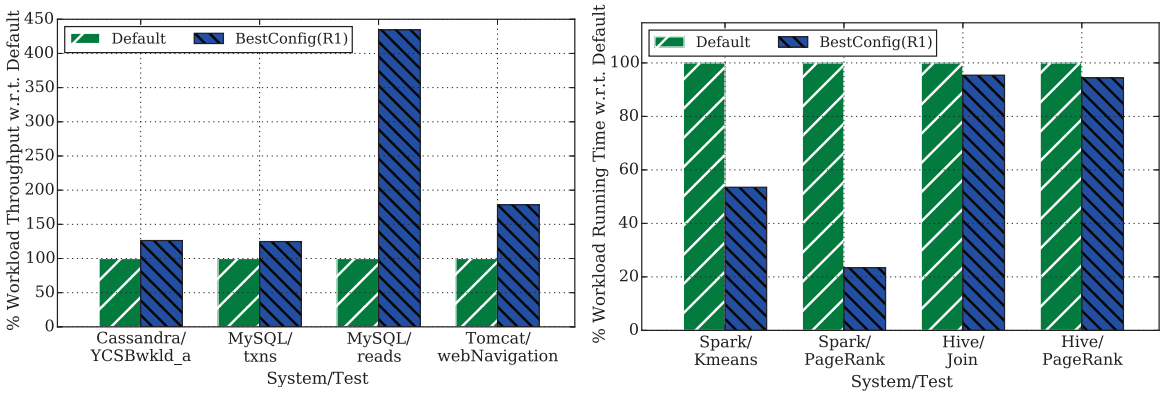


Figure 5: BestConfig's optimization capability with regard to the default configuration setting.

the performance gain for Hive+Hadoop is relatively small. The underlying reason is that this SUT has almost 10 times as many configuration parameters as the other SUTs.

However, **BestConfig can improve the tuning result as the size of the sample set is increased**. Setting the sample set size to be 500, we carry out another experiment of Hive+Hadoop under the HiBench Join workload. The result is demonstrated in Figure 6. The BestConfig setting reduces 50% running time of the Join job.

To sum up the results, BestConfig has improved the throughput of Tomcat by about 75%, that of Cassandra by about 25%, that of MySQL by about 430%, and reduced the running time of Hive join job by about 50% and that of Spark join job by about 80%, solely by configuration adjustments.

Invalidating manual tuning guidelines. The results produced by BestConfig have invalidated some manual tuning rules. For example, some guideline for manually tuning MySQL says that the value of *thread_cache_size* should never be larger than 200. However, according to BestConfig's results demonstrated in Figure 7, we can set the parameter to the large value of 11987, yet we get a much better performance than following the guideline.

5.3 Cooperation of DDS and RBS

We have evaluated the DDS (divide-and-diverge sampling) method of BestConfig as compared to uniform random sampling and gridding sampling. We carry out the comparisons based on Tomcat through tuning two configuration parameters. In the experiments, we use all sampling methods with RBS. We set the initial sample set size to be 100 and the round number to be 2. Thus, after sampling for the first round, each sampling method will sample around a promising point in the second round, denoted as *bound and sample*. The results are plotted in Figure 8.

In the initial round, the three sampling methods have sampled points with similar best performances. The effectiveness and advantage of DDS is demonstrated in the bound-and-sample round. As shown in Figure 8, DDS have sampled points with the best performance as much as three times more than those of the gridding and the uniform sampling. The advantage of DDS over the gridding is due to its diverging step, while that over the uniform sampling is due to the complete coverage of the sampling space. DDS considers 100 diversities for each parameter, while the gridding considers only 10. And, there is a possibility that the uniform sampling will take samples locating at some restricted area of the space, while DDS is guaranteed to scatter samples across the space and with divergence.

We also compare DDS with LHS (Latin Hypercube Sampling) [25]. LHS can produce the same sample sets as DDS in one-time sampling. However, DDS differs from LHS in that DDS remembers previously sampled subspaces and resamples towards a wider coverage of the whole parameter space. This difference leads to the DDS method's advantage of coverage and scalability over LHS. This advantage is demonstrated in Figure 9 through a configuration tuning process for Tomcat. In this tuning process, we set the sample

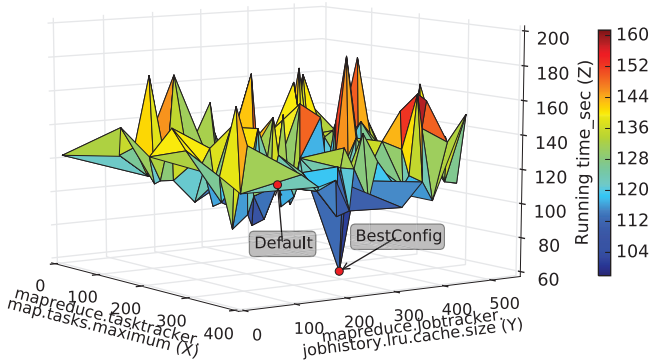


Figure 6: BestConfig reduces 50% running time of HiBench-Join on Hive+Hadoop within 500 tests.

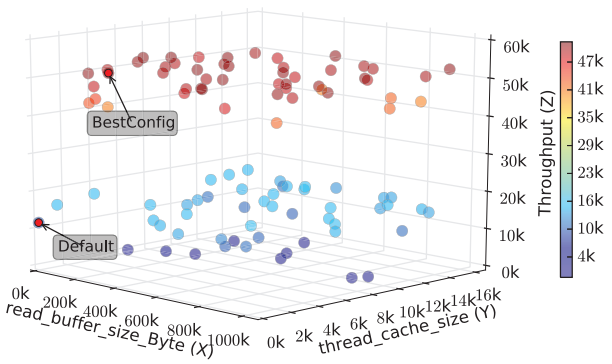


Figure 7: Throughputs for varied *thread_cache_size* of MySQL, invalidating the manual tuning guideline.

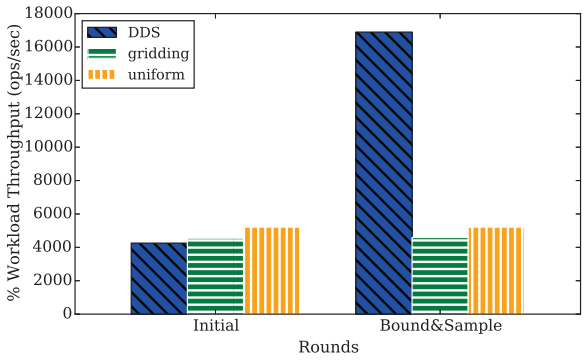


Figure 8: Sampling method comparisons: DDS outperforms gridding and uniform in the latter round.

set size for each round as 100 (according to the experimental results of Table 4). We can see that DDS in cooperation with RBS makes progress in earlier rounds than LHS with RBS.

Furthermore, we try replacing RBS with RRS (Recursive Random Search) [43]. RRS is a search-based optimization method with the exploitation and exploration steps, similar to the bound and recursion steps of RBS. However, RBS are designed with space coverage and scalability, while RRS has no such preferred properties. Hence, when we compare RBS+DDS with RRS+LHS in experiments on a Tomcat deployment, the former outperforms the latter given the same resource limit. The results are demonstrated in Figure 10.

5.4 Varied Sample-Set Sizes & Rounds

To understand how the sample set size and the number of rounds affect the optimization process, we limit the number of tests to 100 and carry out five sets of experiments with varied sample-set sizes and rounds. We vary the sample-set sizes from 5 to 100 and the number of rounds from 20 to 1 accordingly. The experiments are run upon Tomcat using the webpage navigation workload, tuning 13 parameters.

The first five rows of Table 4 summarizes the results for each set of experiments, regarding the performance gains for the initial sampling-search round and the whole tuning process. As the size of the sample set increases, both performance gains are increasing. This fact implies that DDS is scalable. In the meantime, given a limited resource, we should set a sample-set size as large as possible, before we increase the number of rounds.

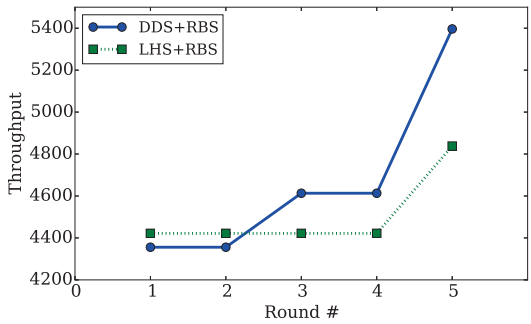


Figure 9: DDS+RBS makes progress in earlier rounds than LHS+RBS.

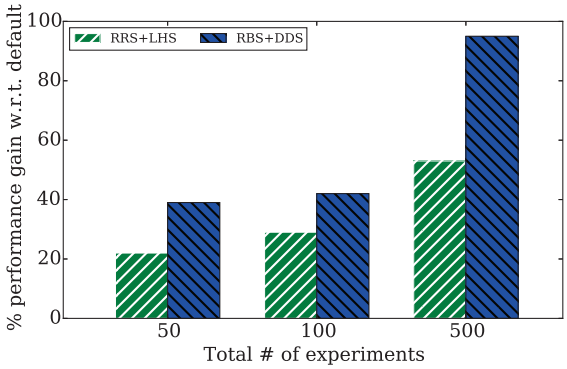


Figure 10: RBS+DDS vs. RRS+LHS.

However, **a larger sample-set size for one round does not necessarily always indicate a better tuning result.** We have experimented with 500 samples for one round, tuning the Tomcat deployment. We find that little performance gain is obtained over the tuning process with 100 samples for one round, as demonstrated by the last row of Table 4. In comparison, the tuning process of Figure 6, which also uses 500 samples for one round, makes much more performance gain than when using 100 samples for one round. The reason behind the difference lies in **the number of parameters**. In our experiments, Tomcat has only 13 parameters, while Hive+Hadoop has 109. The more parameters, the larger sample-set size is required.

Rounds matter. Despite that a large initial sample-set size is important, more rounds are necessary for better tuning results. Consider Figure 9 again. Because of randomness, it is not guaranteed that more rounds mean definitely better results. For example, the second round of DDS+RBS does not actually produce a better result than the first round. However, more rounds can lead to better results, e.g., the third round and the fifth round of DDS+RBS in Figure 9. In fact, the third round and the fifth round are executing the recursion step of RBS. This step is key to avoiding suboptimal results. Thus, by searching in the whole parameter space again, the third and fifth rounds find configuration settings with higher performances. How much BestConfig can improve the performance of a deployed system depends on factors like SUT, deployment setting, workload and configuration parameter set. But BestConfig can usually tune a system better when given more resource and running more rounds than when given less resource and running fewer rounds.

Table 4: Performance gains on varied sample-set sizes and rounds.

Exps (SetSize*rounds)	Initial gain	Best gain
5*20	0%	5%
10*10	0%	14%
20*5	26%	29%
50*2	39%	39%
100*1	42%	42%
500*1	43%	43%

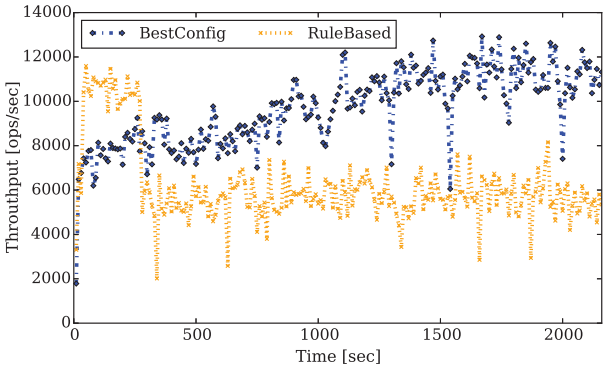


Figure 11: A long-running test of the manually-tuned and BestConfig settings for Cassandra under Huawei's Cloud+ application workloads.

5.5 Stable Advantage of BestConfig Setting

BestConfig generally adopts short tests in the tuning process. For the long running application workloads, it might seem that the iterative testing and configuration tuning process cannot work. We argue that, even though the automatic configuration tuning process for such workloads might be long, the performance gained from the long tuning process is still worthwhile. Besides, as the benchmarking community has proved theoretically and practically [2, 12, 30, 33, 38, 47], the long-running application workloads can be represented by some short-running workloads.

As demonstrated by our experience with an actual BestConfig deployment, short tests can represent long-running workloads, if the tests are specified and generated properly. We have deployed BestConfig to tune the Cassandra system for Huawei's Cloud+ applications. For confidentiality reasons, we simulated the application workloads using the YCSB benchmark, which is then integrated to the workload generator of BestConfig. In the automatic configuration tuning process, the simulated workload is run for about ten minutes. We set the sample set size to be 60 and the round number as one. As output by BestConfig, a configuration setting was found with about 29% performance gain than the setting tuned by Huawei engineers. Later, we applied the configuration setting found by BestConfig to Cassandra and ran the workload for about 40 minutes. A similar long-running test is carried out with the Huawei-tuned configuration setting as well. The resulting throughput is demonstrated in Figure 11.

As shown in Figure 11, BestConfig's configuration setting keeps its advantage over the one set by Huawei engineers. In fact, BestConfig's setting has an average throughput of 9679 ops/sec, while Huawei's rule-based setting can only achieve one of 5933 ops/sec. Thus, BestConfig has actually made a 63% performance gain. This performance improvement is made merely through configuration adjustments.

In fact, BestConfig can usually tune a system to a better performance than the manual tuning that follows common guidelines recommended for general system deployments. The reasons are twofold. First, such manual tuning might achieve a good performance for many system deployments under many workloads, but the tuned setting is usually not the best for a specific combination of SUT, workload and deployment environment. As demonstrated in Section 5.2, some tuning guidelines might work for some situations but not the others. Second, the number of configuration parameters is too large and the interactions within a deployed system are too complex to be comprehended by human [3].

5.6 Discussion

Users ought to specify a resource limit in proportion to the number of parameters for tuning. Although BestConfig can improve the performance of a system based on a small number of samples, there is a minimum requirement on the number of samples. Consider Table 1. If the user allows only 5 samples in a round for tuning 13 parameters, the user is not likely to get a good result. When the number of samples exceeds that of parameters, e.g., from the second row of Table 1, the performance of the system gets improved obviously. Similarly, for a system with more than 100 parameters to tune, BestConfig can only improve the system performance by about 5%, if only 100 samples are provided (Figure 5). However, when the sample set size is increased to 500, BestConfig can improve the system performance by about 50% (Figure 6).

If we can reduce the number of parameters to tune, we can reduce the number of tuning tests and fasten the tuning process, since the number of parameters is related to the number of samples needed for tuning. A recent related work on configuration tuning has proposed to reduce the number of parameters through a popular linear-regression-based feature selection technique called Lasso [39]. We consider integrating similar parameter reduction methods into BestConfig as future work.

BestConfig can generally do a great job if given the whole set of parameters. Even if the set of parameters is not complete, BestConfig can generally improve the system performance as long as the set contains some parameters affecting the system performance. In case that the set of parameters to tune are totally unrelated to an SUT's performance, BestConfig will not be able to improve the SUT's performance. Besides, BestConfig cannot improve an SUT's performance if (1) the SUT is co-deployed with other systems, which are not tuned by BestConfig and which involve a performance bottleneck affecting the SUT's performance; or, (2) the SUT with the default configuration setting is already at its optimal performance.

However, if the above situations occur, it means that the SUT's performance cannot be improved merely through configuration adjustments. Instead, other measures must be taken such as adding influential parameters for tuning, removing bottleneck components or improving the system design.

6 USE CASE: TOMCAT FOR CLOUD+

BestConfig has been deployed to tune Tomcat servers for Huawei's Cloud+ applications. The Tomcat servers run on virtual machines, which run on physical machines equipped with ARM CPUs. Each virtual machine is configured to run with 8 cores, among which four are assigned to process the network communications. Under the default configuration setting, the utilizations of the four cores serving network communications are fully loaded, while the utilizations of the other four processing cores are about 80%. With such CPU behaviors, Huawei engineers have considered that the current throughput of the system is the upper bound and no more improvement is possible.

Table 5: BestConfig improving performances of a fully-loaded Tomcat.

Metrics	Default	BestConfig	Improvement
Txns/seconds	978	1018	**4.07**% ↑
Hits/seconds	3235	3620	**11.91**% ↑
Passed Txns	3184598	3381644	**6.19**% ↑
Failed Txns	165	144	**12.73**% ↓
Errors	37	34	**8.11**% ↓

Using BestConfig and setting the overall throughput as the performance metric for optimization, we then found a configuration setting that can improve the performance of the deployment by 4%, while the CPU utilizations remain the same. Later, the stability tests demonstrate that the BestConfig setting can guarantee the performance improvement stably. The results of the stability tests are demonstrated in Table 5. We can observe improvements on every performance metric by using the BestConfig setting.

Thus, BestConfig has made it possible to improve the performance of a fully loaded system by simply adjusting its configuration setting. It has expanded our understanding on the deployed systems through automatic configuration tuning.

7 RELATED WORK

The closest related works for BestConfig are the classic Latin Hypercube Sampling (LHS) method [25] and the recursive random search (RRS) algorithm [43]. DDS differs from LHS in that DDS remembers previously sampled subspaces and resamples towards a wider coverage of the whole parameter space. This difference leads to the DDS method's advantage of coverage and scalability over LHS. RBS differs from RRS in two aspects. First, RRS requires the users to set multiple hyper-parameters, which have strong impacts on the optimization results, but setting hyper-parameters is as hard as configuration tuning. Second, RRS searches a local subspace by examining one sample after another. Such a design is efficient only if the local search is limited to a small space, but this is generally not true for high-dimensional spaces. If the hyper-parameters are carefully set as for a narrow local search, then the space not examined would be too large. Such trade-off is difficult. Besides, searching a local space by taking one sample at a time involves too much randomness; in comparison, the local search of BestConfig takes advantage of the sampling method. One more crucial difference is that BestConfig exploits RBS and DDS together.

Quite a few past works have been devoted to automatic configuration tuning for Web systems. These works either choose a small number of parameters to tune, e.g., smart hill climbing [41], or require a huge number of initial testings [16, 44], e.g., simulated annealing [40] and genetic algorithms [15]. Although constructing performance models might help finding appropriate configuration settings [29], a large number of samples will be required for modeling in a large configuration parameter space. But collecting a large set of samples requires to test the SUT for many times. This is a highly costly process. A related work uses reinforcement learning in the same tuning problem [6]. It formulates the performance optimization process as a finite Markov decision process (MDP), which consists of a set of states and several actions for each state. The actions are increasing or decreasing the values of individual parameters. As mentioned previously, the performance functions of deployed systems can be complicated, e.g., with many sudden ups and downs on the surface. A seemingly wise step with some performance gain might result in a bad final setting, due to the local suboptimal problem.

Works on automatic configuration tuning for database systems also exist. iTuned [11] assumes a smooth performance surface for the SUT so as to employ the Gaussian process regression (GPR) for automatic configuration tuning. But the assumption can be inapplicable to other SUTs, e.g., Tomcat or MySQL given some specific set of configuration parameters. The recent work of OtterTune [39] also exploits GPR. It additionally introduces a feature-selection step to reduce the number of parameters. This step reduces the complexity of the configuration tuning problem. We are examining the possibility of integrating similar feature selection methods into BestConfig to reduce the number of configuration parameters before starting the tuning process.

Automatic configuration tuning is also proposed for the Hadoop system. Starfish [19] is built based upon a strong understanding of the Hadoop system and performance tuning. Thus, the method used in Starfish cannot be directly applied to other systems. Aloja [4] adopts the common machine learning methods, exploiting a large database of samples. But, samples are costly to obtain. As analyzed in Sectio 3.6 and 5.1, Aloja's approach is not applicable to the configuration tuning of general systems.

8 CONCLUSION

We have presented the automatic configuration tuning system BestConfig. BestConfig can automatically find, within a given resource limit, a configuration setting that can optimize the performance of a deployed system under a specific application workload. It is designed with a highly flexible and extensible architecture, the scalable sampling method DDS and the scalable performance optimization algorithm RBS. As an open-source package, BestConfig is available for developers to use and extend in order to effectively tune cloud systems. We have used BestConfig to tune the configuration settings of six widely used systems and observed the obvious performance improvements after tuning. Furthermore, tuning the Tomcat system on virtual machines in the Huawei cloud, BestConfig has actually made it possible to improve the performance of a fully loaded system by simply adjusting its configuration settings. These results highlight the importance of an automatic configuration tuning system for tapping the performance potential of systems.

ACKNOWLEDGMENTS

We would like to thank our shepherd, Ennan Zhai, and the anonymous reviewers for their constructive comments and inputs to improve our paper. We would like to thank the Huawei Cloud+ and the Huawei BI teams in helping us verify BestConfig towards their online applications. This work is in part supported by the National Natural Science Foundation of China (Grant No. 61303054 and No. 61420106013), the State Key Development Program for Basic Research of China (Grant No. 2014CB340402) and gifts from Huawei.

REFERENCES

[1] Sameer Agarwal, Srikanth Kandula, Nicolas Bruno, Ming-Chuan Wu, Ion Stoica, and Jingren Zhou. 2012. Re-optimizing data-parallel computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*. USENIX Association, 21–21.

[2] Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, et al. 2006. *The landscape of parallel computing research: A view from Berkeley*. Technical Report. UCB/EECS-2006-183, EECS Department, University of California, Berkeley.

[3] Phil Bernstein, Michael Brodie, Stefano Ceri, David DeWitt, Mike Franklin, Hector Garcia-Molina, Jim Gray, Jerry Held, Joe Hellerstein, HV Jagadish, et al. 1998. The Asilomar report on database research. *ACM Sigmod record* 27, 4 (1998), 74–80.

[4] Josep Lluís Berral, Nicolas Poggi, David Carrera, Aaron Call, Rob Reinauer, and Daron Green. 2015. Aloja-ml: A framework for automating characterization and knowledge discovery in hadoop deployments. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, 1701–1710.

[5] Jon Brodkin. 2012. Why Gmail went down: Google misconfigured load balancing servers. (2012). http://arstechnica.com/information-technology/2012/12/why-gmail-went-down-google-misconfigured-chromes-sync-server/.

[6] Xiangping Bu, Jia Rao, and Cheng-Zhong Xu. 2009. A reinforcement learning approach to online web systems auto-configuration. In *Distributed Computing Systems, 2009. ICDCS'09. 29th IEEE International Conference on*. IEEE, 2–11.

[7] Cassandra.Apache.Org. 2017. Apache Cassandra Website. (2017). http://cassandra.apache.org/.

[8] Haifeng Chen, Wenxuan Zhang, and Guofei Jiang. 2011. Experience transfer for the configuration tuning in large-scale computing systems. *IEEE Transactions on Knowledge and Data Engineering* 23, 3 (2011), 388–401.

[9] Cloudera.Com. 2017. Tuning YARN. http://www.cloudera.com/documentation/enterprise/5-6-x/topics/cdh_ig_yarn_tuning.html. (2017).

[10] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st SoCC*. ACM.

[11] Songyun Duan, Vamsidhar Thummala, and Shivnath Babu. 2009. Tuning database configuration parameters with iTuned. *Proceedings of the VLDB Endowment* 2, 1 (2009), 1246–1257.

[12] Anon et al, Dina Bitton, Mark Brown, Rick Catell, Stefano Ceri, Tim Chou, Dave DeWitt, Dieter Gawlick, Hector Garcia-Molina, Bob Good, Jim Gray, et al. 1985. A measure of transaction processing power. *Datamation* 31, 7 (1985), 112–118.

[13] Andrew D Ferguson, Peter Bodik, Srikanth Kandula, Eric Boutin, and Rodrigo Fonseca. 2012. Jockey: guaranteed job latency in data parallel clusters. In *Proceedings of the 7th ACM european conference on Computer Systems*. ACM, 99–112.

[14] Adem Efe Gencer, David Bindel, Emin Gün Sirer, and Robbert van Renesse. 2015. Configuring Distributed Computations Using Response Surfaces. In *Proceedings of the 16th Annual Middleware Conference*. ACM, 235–246.

[15] David E Goldberg and John H Holland. 1988. Genetic algorithms and machine learning. *Machine learning* 3, 2 (1988), 95–99.

[16] Bilal Gonen, Gurhan Gunduz, and Murat Yuksel. 2015. Automated network management and configuration using Probabilistic Trans-Algorithmic Search. *Computer Networks* 76 (2015), 275–293.

[17] Qi Guo, Tianshi Chen, Yunji Chen, Zhi-Hua Zhou, Weiwu Hu, and Zhiwei Xu. 2011. Effective and efficient microprocessor design space exploration using unlabeled design configurations. In *IJCAI Proceedings-International Joint Conference on Artificial Intelligence*, Vol. 22. Citeseer, 1671.

[18] Hadoop.Apache.Org. 2017. Apache Hadoop Website. (2017). http://hadoop.apache.org/.

[19] Herodotos Herodotou, Fei Dong, and Shivnath Babu. 2011. No one (cluster) size fits all: automatic cluster sizing for data-intensive analytics. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*. ACM, 18.

[20] Hive.Apache.Org. 2017. Apache Hive Website. (2017). http://hive.apache.org/.

[21] Holger H Hoos. 2011. Automated algorithm configuration and parameter tuning. In *Autonomous search*. Springer, 37–71.

[22] S. Huang, J. Huang, J. Dai, T. Xie, and B. Huang. 2010. The hibench benchmark suite: Characterization of the mapreduce-based data analysis. In *Proc. of ICDEW 2010*. IEEE, 41–51.

[23] JMeter.Apache.Org. 2017. Apache JMeter^{TM}. http://jmeter.apache.org. (2017).

[24] Launchpad.Net. 2017. SysBench: System evaluation benchmark. http://github.com/nuodb/sysbench. (2017).

[25] Michael D McKay, Richard J Beckman, and William J Conover. 2000. A comparison of three methods for selecting values of input variables in the analysis of output from a computer code. *Technometrics* 42, 1 (2000), 55–61.

[26] Aurimas Mikalauskas. 2017. 17 KEY MYSQL CONFIG FILE SETTINGS (MYSQL 5.7 PROOF). http://www.speedemy.com/17-key-mysql-config-file-settings-mysql-5-7-proof/. (2017).

[27] Rich Miller. 2012. Microsoft: Misconfigured Network Device Led to Azure Outage. (2012). http://www.datacenterknowledge.com/archives/2012/07/28/microsoft-misconfigured-network-device-caused-azure-outage/.

[28] MySQL.Com. 2017. MySQL Website. (2017). http://www.mysql.com/.

[29] Takayuki Osogami and Sei Kato. 2007. Optimizing system configurations quickly by guessing at the performance. In *ACM SIGMETRICS Performance Evaluation Review*, Vol. 35. ACM, 145–156.

[30] Kim Shanley. 2010. History and Overview of the TPC. (2010).

[31] Spark.Apache.Org. 2017. Apache Spark Website. (2017). http://spark.apache.org/.

[32] Spark.Apache.Org. 2017. Tuning Spark. (2017). http://spark.apache.org/docs/latest/tuning.html

[33] Spec.Org. 2017. Standard Performance Evaluation Corporation (SPEC). http://www.spec.org/. (2017).

[34] Chunqiang Tang, Thawan Kooburat, Pradeep Venkatachalam, Akshay Chander, Zhe Wen, Aravind Narayanan, Patrick Dowell, and Robert Karl. 2015. Holistic configuration management at Facebook. In *Proceedings of the 25th Symposium on Operating Systems Principles*. ACM, 328–343.

[35] Keir Thomas. 2011. Amazon: The Cloud Crash Reveals Your Importance. (2011). http://www.pcworld.com/article/226033/thanks_amazon_for_making_possible_much_of_the_internet.html.

[36] Tobert. 2017. Al's Cassandra 2.1 tuning guide. https://tobert.github.io/pages/als-cassandra-21-tuning-guide.html. (2017).

[37] Tomcat.Apache.Org. 2017. Apache Tomcat Website. (2017). http://tomcat.apache.org/.

[38] TPC.Org. 2017. Transaction Processing Performance Council (TPC). http://www.tpc.org/. (2017).

[39] Dana Van Aken, Andrew Pavlo, Geoffrey J Gordon, and Bohan Zhang. 2017. Automatic Database Management System Tuning Through Large-scale Machine Learning. In *Proceedings of the 2017 ACM International Conference on Management of Data*. ACM, 1009–1024.

[40] Peter JM Van Laarhoven and Emile HL Aarts. 1987. Simulated annealing. In *Simulated Annealing: Theory and Applications*. Springer, 7–15.

[41] Bowei Xi, Zhen Liu, Mukund Raghavachari, Cathy H Xia, and Li Zhang. 2004. A smart hill-climbing algorithm for application server configuration. In *Proceedings of the 13th international conference on World Wide Web*. ACM, 287–296.

[42] Tianyin Xu, Long Jin, Xuepeng Fan, Yuanyuan Zhou, Shankar Pasupathy, and Rukma Talwadker. 2015. Hey, you have given me too many knobs!: understanding and dealing with over-designed configuration in system software. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ACM, 307–319.

[43] Tao Ye and Shivkumar Kalyanaraman. 2003. A recursive random search algorithm for large-scale network parameter configuration. *ACM SIGMETRICS Performance Evaluation Review* 31, 1 (2003), 196–205.

[44] Wei Zheng, Ricardo Bianchini, and Thu D. Nguyen. 2007. Automatic Configuration of Internet Services. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*. ACM, New York, NY, USA, 219–229.

[45] Yuqing Zhu and Jianxun Liu. 2017. Better Configurations for Large-Scale Systems (BestConf). (2017). http://github.com/zhuyuqing/bestconf

[46] Yuqing Zhu, Jianxun Liu, Mengying Guo, Wenlong Ma, and Yungang Bao. 2017. ACTS in Need: Automatic Configuration Tuning with Scalability Guarantees. In *Proceedings of the 8th SIGOPS Asia-Pacific Workshop on Systems*. ACM.

[47] Yuqing Zhu, Jianfeng Zhan, Chuliang Weng, Raghunath Nambiar, Jinchao Zhang, Xingzhen Chen, and Lei Wang. 2014. Bigop: Generating comprehensive big data workloads as a benchmarking framework. In *International Conference on Database Systems for Advanced Applications*. Springer, 483–492.