EVADyR: a new dynamic resampling algorithm for optimizing noisy expensive systems

S. Robert-Hayek[1], S. Zertal[2], and P. Couvée[1]

[1] Atos BDS RD Data Management
Echirolles, France
sophie.robert@atos.net
philippe.couvee@atos.net
[2] University of UPSacaly-UVSQ
Guyancourt, France
soraya.zertal@uvsq.fr

Abstract. Black-box auto-tuning methods have been proven to be efficient for tuning configurable computer appliance. However, because of the shared nature and the complexity of the software and hardware stack of some systems such as cloud or HPC systems, the measurement of the performance function can be tainted by noise during the tuning process, which can reduce and sometime prevent, the benefit of auto-tuning. An usual choice is to add a resampling step at each iteration to reduce uncertainty, but this approach can be time-consuming. In this paper, we propose a new resampling and filtering algorithm called EVADyR (Efficient Value Aware Dynamic Resampling). This algorithm is able to tune efficiently a prefetching strategy in the case of multiple parallel accesses. Because it finds a better exploration versus exploitation trade-off by resampling only promising parametrizations and increases the level of confidence around the suggested solution as the tuning process advances, it outperforms state of the art dynamic resampling by reducing the distance to the optimum by 93.5%, as well as speed-up the experiment duration by 45.8% because less iterations are needed to reach the found optimum. An additional proof of this study is the demonstration of the importance of using noise reduction strategies for the optimization of highly shared resources such as HPC or cloud systems.

Mots-Clefs. Auto-tuning, Black-box optimization, High Performance Computing, Stochastic Optimization, Resampling

1 Introduction

Most of the software of modern computer systems come with many configurable parameters that control the system's behavior and its interaction with the underlying hardware. These parameters are challenging to tune by solely relying on field insight and user expertise, due to huge parametric spaces and complex, non-linear system behavior and environment variations. Consequently, users often have to rely on the default parameters and do not take advantage of the possible performance that the system could deliver for their applications. As users are not easily able to take adequate decisions for the parametrization of complex systems, new tuning methods, usually called *auto-tuning* methods, have emerged from the optimization and machine learning fields to automate parameter selection depending on the current workload. They have been successfully applied to a wide range of systems, such as storage systems, database management systems and compilers.
Among existing auto-tuning methods, black-box optimization has been demonstrated to be successful for tuning a wide range of computer systems. However, these studies were done under the assumption that the tuned system is deterministic, *i.e.* a given parametrization will always yield the same execution time. While this hypothesis is valid when working in a controlled and exclusive test environment, it does not always hold for systems that rely on shared resources, such as cloud or HPC systems. Due to the high cost of dedicating exclusive resources, complex system auto-tuning often occurs in shared environments, requiring automatic tuning methods to consider potential interference, which we will call "noise", that can degrade the performance of traditional auto-tuning heuristics, as the performance of the system being tuned can vary due to fluctuations in resource availability or the presence of competing tasks, independently from the tested parameters. The black-box optimizer risks matching the input parameters with the random variations in the performance, rather than the performance itself. Sub-optimal parameter configuration that appears good due to

favorable conditions, such as low system activity, can be retained, while good parameter configurations may be rejected if they were evaluated under challenging circumstances, such as high usage or storage backup events. Consequently, to be used on modern shared systems, auto-tuning algorithms must be able to distinguish between the true system behavior and random variations, while remaining as sparse as possible in terms of iterations.
In this paper, we consider the noise present when auto-tuning a highly configurable appliance aimed at reducing I/O contention in High Performance Computing (HPC) systems, called the *Small Read Optimizer* (SRO), and suggest a new resampling algorithm, called EVADyR (**E**fficient **V**alue **A**ware **Dy**namic **R**esampling), and show that it outperforms already existing state-of-the-art.
The main contributions of this paper are:
– The proposition of the original EVADyR resampling algorithm based on dynamic resampling but specifically tailored to the optimization of noisy and expensive function;
– The highlighting of the importance of using noise reduction when tuning systems in highly shared production environment;
– The validation of this new algorithm on a real-life I/O accelerator with different noisy context, which compared to state-of-the-art dynamic resampling provides an improvement of the quality of convergence of 93.5% and a speed-up of the experiment duration by 45.76% for the tuned system.
This paper is organized as follows. Section 2 covers related works and section 3 the main principles of Bayesian Optimization and some of the state of the art noise reduction techniques. Section 4 presents the proposed EVADyR algorithm to improve resampling algorithms, section 5 the experiment plan and section 6 the obtained results. Section 7 concludes this study by giving some insights into some future work.

2 Related works

In the literature, black-box optimization is a popular choice for tuning different reconfigurable systems, and it has been particularly helpful in computer science to look for the optimal configurations of various software and hardware components. In the field of Big Data Processing systems, software that are notoriously hard to tune such as Hadoop, Spark and Storm, have benefited from black-box optimization. It is for example the methodology chosen by Liao et al. in their Gunther framework [24] which relies on Genetic Algorithms to find the optimal parametrization of Hadoop. In [20], Jamshidi et al. use Bayesian Optimization to optimize the stream processing system Storm. In [9], Desani et al. compare two derivative-free methods (Bounded Optimization BY Quadratic Approximation method and Constrained Optimization BY Linear Approximation method) to find the optimal configuration of the Hadoop framework. In Database Management Systems (DBMS), tuning relying on surrogate modeling with expected improvement and pruning strategies have been suggested through the iTuned framework designed by Duan et al. in [11]. This framework has significantly improved the performance of PostgreSQL for different workloads. Also, using black-box optimization to optimize query has also proven its efficiency using simulated annealing [19] and genetic algorithms [5].
Storage systems are hard to tune as they have both a very large parametric space with a strong dependence on the running workload [7]. The efficiency of black-box optimization for tuning storage systems when faced with different workloads has been explored by Cao et al. in [8], where they propose a thorough analysis of the behavior of each heuristic. Reinforcement learning has also been successfully used as an auto-tuner to optimize the performance of the Lustre filesystem in data center storage systems by Li et al. in [23] and an optimal parametrization for the several layers of HDF5 library was found using genetic algorithms by Behzad et al. in [4]. An extension of this auto-tuner which selects the best parameters according to the I/O pattern is described in [3] by the same authors.
Within the HPC community, auto-tuning has gained a lot of attention for tuning particular HPC application and improve their portability across architectures. In [35], Seymour et al. provide a comparison of several random-based heuristic searches (Simulated annealing, genetic algorithms ...) that have provided some good results when used for code auto-tuning. Bergstra has had good results as well in this field with surrogate modeling using boosted regression trees [6]. In [26], Menon et al. use Bayesian Optimization and suggest the framework HiPerBOT to tune application parameters as well as compiler runtime settings. HPC systems energy consumption can also benefit from Bayesian Optimization, as Miyazaki et al. have shown in [27] where an auto-tuner based on a combination of Gaussian Process regression and the Expected Improvement acquisition function has raised their cluster to the Green500 list. A scheduling algorithm using genetic algorithms has been introduced by

Kassab et al. in [21] and manages to schedule jobs under a limited energy power constraint. The MPI community has also shown the superiority of a hill-climbing black-box algorithm over an exhaustive sampling of the parametric space in [12].

The literature addressing the problem of noise when tuning real systems is surprisingly sparse, as most of the works detailed in the previous section do not study the potential interference on the tuned system and do not mention their tuner resilience to possible interference in shared settings. While this is not critical when working on single user systems, such as local hard drives for personal computers [8], it cannot be ignored when working on highly parallel shared systems [28]. Several studies do acknowledge their system's noise [7], but do not provide any practical solution to make the tuner resilient, other than computing the mean or the median of the found best parametrization repeated several times [34] [17]. To our knowledge, the only notable exception is the Baloo framework [16] developed by Grohmann et al. for tuning distributed database systems, which performs adaptive sampling until the confidence interval around the mean is smaller than a set threshold, or until the maximum number of reevaluation allowed per parametrization is reached.

3 Bayesian Optimization for noisy systems

3.1 Problem formalization

Formally, let S be the system to optimize and θ_i its parametrization from the possible parametrizations of the tunable system $\Theta = \{\theta_i\}_{i\in\mathbb{N}}$. Let $\mathcal{A}$ be the optimized application and $\mathcal{E}$ the execution context (the underlying hardware, the number of nodes...) for which we optimize the application.

Let $f : (\mathcal{A},\ \mathcal{E}, \Theta) \longrightarrow \mathbb{R},\ \theta \longrightarrow f(\theta)$ represent the execution time for parametrization θ. Because of the possible random interference depending on the system's state, we only have access to the observed execution time $F(\theta)$. These interference, which we will refer to as noise, can depend on the parametrization or the optimization step and is thus a function of θ and n, represented by $\epsilon(\theta, n)$. The vector corresponding to the different sampled values at parameter θ will be denoted as $F(\theta)^j_{1\leq j\leq j_\theta}$, j_θ being the number of samples for parameter θ. The estimation of f at θ is denoted $\hat{f}(\theta)$.

With these notations, the optimizer must solve the following problem:

$$\begin{gathered} min\ \mathbb{E}(F(\theta)),\ \theta \in \Theta \\ F(\theta) = f(\theta) + \epsilon(\theta, n) \end{gathered}$$

This formula holds in the case of minimization (for example when minimizing the execution time) and in the case of maximization (for example of the throughput), one can simply minimize $-f$. From this definition, we find that in this noisy framework the optimal parametrization is not the one leading to the quickest run but the one corresponding to the optimal execution time on average. This ensures that the optimum is not a result of chance. Throughout the remainder of this paper, the term "optimum" will refer to this average optimal parametrization.

3.2 Bayesian optimization for auto-tuning

Black-box optimization is a method for optimizing a function with unknown properties that can only be evaluated a limited number of times. In the context of tuning computer systems, it involves optimizing the performance of an application based on the relationship between input parameters (configurable component and execution context) and the output (application performance) without having any insight into the internal workings of the system.

Two distinct steps are necessary for black-box optimization: the selection of initial parameters, such as Latin Hypercube Designs [39], and an optimization heuristic selecting at each iteration the most promising configuration.

Many heuristics have been designed and tested, among which Bayesian Optimization (also known as Sequential Model Based Optimization). It is an efficient and versatile method for tuning complex systems, which uses a cheaper to evaluate probabilistic model to approximate the performance function. Bayesian Optimization requires two components: an acquisition function that selects the next data point to evaluate and a probabilistic model that represents the performance function. In our experiments, based on previous study results in [33], we will use Expected Improvement [39] as an acquisition function and Gaussian processes [31] as a probabilistic model. The optimization process will continue until the improvement from the current best objective function value is below a set threshold over a number of iterations.

3.3 Stochastic black-box optimization

While almost non-existent in the system's tuning community, black-box optimization with noisy fitness is a very proficient field when it comes to theoretical research on synthetic benchmarking function. The available research can be broadly split into two main categories : *heuristic specific noise reduction* and *heuristic agnostic noise reduction*.
In the case of Bayesian Optimization, a possible improvement is the modification of acquisition functions in order to make them handle noisy observations better. For instance, Gramacy et al. in [15] and Vàsquez et al. in [40] use respectively the mean and a quantile as an estimation of the performance function through Gaussian Processes when using Bayesian Optimization. Letham et al. propose in [22] a novel way of defining expected improvement, called *Noisy Expected Improvement*, using Quasi Monte Carlo simulation. In [18], Huang et al. suggest using the *Augmented Expected Improvement*, which uses a robust estimation of the best performing parametrization by defining the best solution as the one with the lowest β-quantile, with β a configurable value. A similar quantile based approach is proposed by Picheny et al. in [30] where they suggest using the *Expected Quantile Improvement* to select the next data point to evaluate. In [14], Forrester et al. suggest using a reinterpolation procedure which uses the results of a Gaussian Process regressor on noisy observations into another interpolation model which will be used to compute the Expected Improvement. While these different methods have proven to be efficient when facing different noises on different benchmarking functions [29], they have the major drawback of being specific to the selected optimization and cannot be generalized to other heuristics, such as genetic algorithms or simulated annealing, even though there is no single best performing heuristic for every optimization problem [33, 7, 41]. For this reason, we decide to take another, more versatile, approach that can be used with any optimization heuristic: resampling [2] [13] [37] [36] which is one of the most popular method in this category. We focus on it in this paper as there is no single best performing heuristic for every optimization problem [42], as demonstrated in some of our previous works [33], and we aim to improve noise reduction methods that allow switching heuristics depending on the context and optimization use-case.

3.4 Resampling techniques

Resampling consists in adding a "resampling filter" by using a set logical rule to select which parametrization to reevaluate, as shown in figure 1. Its goal is to reduce the standard deviation of the mean of an objective value in order to augment the knowledge of the impact of the parameter on the performance. Resampling is a trade-off between having a better knowledge of the space and wasting some computing times on re-evaluation. Many strategies exist in order to efficiently reevaluate a parametrization, and we present here two of the most popular: simple and dynamic resampling using SEDR (Standard Error Dynamic Resampling).

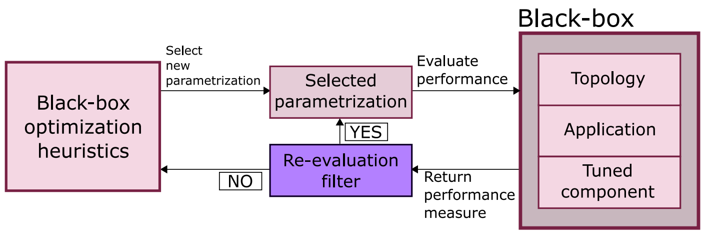

Fig. 1. Schematic representation of resampling algorithms

Simple resampling Simple resampling consists in evaluating for a fixed number of times the fitness value of the selected parametrization [13], regardless of the parametrization and its fitness. In our case, it consists in launching a fixed number of times the application and the tuned system with the same selected parametrization. Its main drawbacks is its lack of adaptivity to the noise.

Dynamic resampling Because the need for resampling is often not homogeneously distributed throughout the search space [36] (especially for shared and distributed systems), dynamic resampling methods that consider the performance variance around each data point have been introduced. Among the most popular, *Standard Error Dynamic Resampling* (SEDR) [10] adapts the number of samples to the noise strength measured at each parametrization by using a different number of evaluations n_θ for each one. The parametriza-

tion is re-evaluated until the 95% confidence interval around the mean is below a fixed threshold τ_{se}.
SEDR has proven its efficiency in theoretical [10] and practical problems [38] where the fitness value is not taken into account whenever setting a confidence interval threshold and require the same certainty regardless of the performance value. However, when working with real-systems, especially for applications which can take a long time to run, the length of the confidence interval should take into account the application's performance. Because of this, the authors in [32] suggest the definition of an interval width proportional to the currently measured mean for this parametrization, and it is this version of SEDR resampling that will be the basis for our EVADyR algorithm.

4 The EvaDyr algorithm: improving resampling methods

Existing resampling methods have proven to be efficient but still present several drawbacks, because:

1. **Resampling can take too many iterations on a single parametrization**, causing a waste of resources, because methods based on noise values can become focused on a particular parametrization with large deviation, even if the noise is temporary (data backup, unusual traffic on the system,..). This variation is already included in [16].
2. **Dependence on hyperparameters**: The success of dynamic and static resampling methods depends greatly on their hyperparameters (the number of resamples for simple resampling and the interval confidence width for dynamic resampling). If the threshold is set too high, no resampling will occur, and if it is too low, excessive resampling will occur, hindering exploration of the parametric space.
3. **No comparison with already tested parametrization**: Resampling does not consider the comparison between the current parametrization's fitness and the previously tested ones. This can lead to resampling slow parametrizations multiple times, slowing the convergence process and wasting the systems' resource.

To address these limitations, we suggest the EVADyR algorithm (**E**fficient **VA**lue **A**ware **Dy**namic **R**esampling), with several improvements:

Setting a limit to the number of resamples We add a floor limit N to the allowed number of resamples on a given parametrization. The number of resamples n_θ for parametrization θ is thus located between 2 (because each parametrization is resampled at least twice in order to measure the noise on this data point) and N. We set for the experiments the maximum number of resamples per parametrization to 10% of the total maximum allowed budget.

Dynamic confidence intervals Another improvement is the removal of the dependence on the hyperparameter threshold. To do so, we make the size of the confidence interval inversely proportional to the number of elapsed steps and use a bounded decreasing exponential to reduce the ratio of the confidence interval at each step. We also make it proportional to the mean measured for this parametrization, to consider the fact that the required precision on the mean estimator is dependent on the value of the mean.
The required confidence around the mean becomes smaller as the number of iterations raises, as we verify:

$$ci\_width(\theta) = 2 \times 1.96 \times \frac{\hat{\sigma}(\theta)}{\sqrt{j_\theta}} \leq FC(j_\theta) \times \hat{\mu}(\theta) = max(0.99^{j_\theta}, 0.1) \times \hat{\mu}(\theta)$$

This ensures that at the beginning of the optimization process, the algorithm is more lax about the precision of the true performance value to test many different parametrization, ensuring a satisfactory exploration of the parametric space. However, at the end of the optimization process, it is more precise about the knowledge we have about the parametrization ensuring an adequate exploitation of already known parametrizations.

Performance based resampling filter The last change we suggest is to introduce a dynamic filtering component before performing the resampling. Its goal is to filter the most promising parametrizations before submitting them to the resampling which reduces its time cost. The filtering process runs as follow:

1. Each time the heuristic suggests a new parametrization, it is evaluated at least twice
2. If the median of the related performance is inferior to a certain ratio of the current median, move to the next step, otherwise move to step 4
3. Keep this parametrization and submit it to the resampling process.
4. Discard this unpromising parametrization, go back to step 1 if the budget is not empty.

The ratio of the median used in step 2 can either be a fixed value or can be computed dynamically as a decreasing function of the number of elapsed iterations, similarly to dynamic interval definition. As the optimization progresses, this filter ensures that the algorithm is more and more strict about the quality of the resampled solutions, so that last iterations are not wasted on parametrization that are not promising. As the optimization process draws to an end, we make sure that we do not waste any of the remaining resources.

5 Evaluation of EVADyR's performance

To evaluate the relevance of noise reduction and compare EVADyR to the state-of-the-art on real-life test case, we perform the tuning on a notoriously noisy and highly shared environment by tuning an I/O accelerator, aimed at reducing I/O contention on large scale HPC systems. This I/O accelerator, called the Small Read Optimizer (`SRO`) is a dynamic data preload strategy that prefetches frequently accessed file chunks into the memory of the compute node. It detects repeatedly accessed zones in a file and loads the entire zone into memory. This method takes into account both temporal and spatial factors, making it more effective than the Linux read-ahead strategy, which only loads one zone at a time. The behavior of the software can be greatly impacted by the four positive and discrete dimensions in its parametric space [33] [32].

5.1 Interference generation

To validate the relevance of our algorithm and compare it to state of the art, we run an I/O heavy benchmark performing pseudo-random read accesses, in a noisy setting created by performing concurrent accesses on the file read by the benchmark. This type of direct interference is common when running applications with different concurrent nodes that require using the same data, which is common in shared systems [25]. It creates an intense, localized noise, that affects periodically and negatively the system's performance.

Tuned application and potential auto-tuning improvement An I/O intensive application was selected that performs 4k operations on a 500GB file with 500 hotspots of width 50MB. The hotspots have 10000 random accesses before moving on to the next file zone. This access pattern is common in the case of applications accessing different zones of a data file (such as in a 3D model), performing many random accesses within a zone if it's considered interesting, before moving on to the next zone.
The benchmark runs on a single compute node. It consists in an Intel(R) Xeon(R) CPU E5-2650 0 @ 2.00GHz with 16 physical cores (32 logical cores), bi-socket, and 82 GB of DDR4-DRAM. The back-end parallel filesystem is a Lustre bay of 40TB. The storage system is isolated from the rest of the users in order to have a fine control over the generated noise and ensure the precision of our study.
To determine the optimal parameters of the accelerator for the application to use as the ground truth for comparison in a noiseless settings, auto-tuning experiments were run 20 times using Bayesian Optimization on isolated nodes and storage systems to eliminate interference. The maximum number of iterations is set to 100 and the optimization process stops if there is less than 5% improvement in the optimum over 15 iterations.
The collected statistical estimators over the 20 noiseless experiments show that all optimization runs converge towards the same optimal performance and the average potential tuning improvement, corresponding to the distance to the default parametrization, is located around 75.33%, with a standard error of 1.15%. The retained best performance, used as ground truth for the optimum performance of the application, is taken to be the average value of the best performance over all 15 experiments: 8.88 seconds, for an average improvement of 75.33%.

Noise generation methodology Noise is introduced by running parallel applications on other computing nodes that perform random operations on the same data file. Three nodes are used, with two nodes performing write operations and one performing read operations, corresponding to concurrent random accesses (reads or writes) across the input data file. The elapsed time between the arrival of each interference is defined as the time interval between the beginning and the end of the concurrent noise application and is expressed in seconds. Three different noise frequencies were tested, selected to match a certain frequency of arrival relative to the duration of the I/O benchmark with default parameters. We denote the experiments as `SRO.n`, n being the noise arrival time. For example, `SRO.60` will refer to the SRO experiment with interference coming every 60 seconds.
The impact of noise on the constant default parametrization is shown in figure 2. The noise profile varies depending on the frequency of arrival. At 60 seconds, the noise is constant, with each run translated to a higher value. At 120 seconds, the noise is less constant but the value

of the application does not return to its original value between runs, creating a challenging environment for the optimizer. At 300 seconds, a Cauchy-type noise is observed, with a run taking longer when hit by the noise but returning to the original value between runs. This test environment provides different types of challenges for the optimizer to validate its relevance in dealing with constant noise (60 seconds arrival), unstable noise (120 seconds arrival), and impulse-type noise (300 seconds arrival).

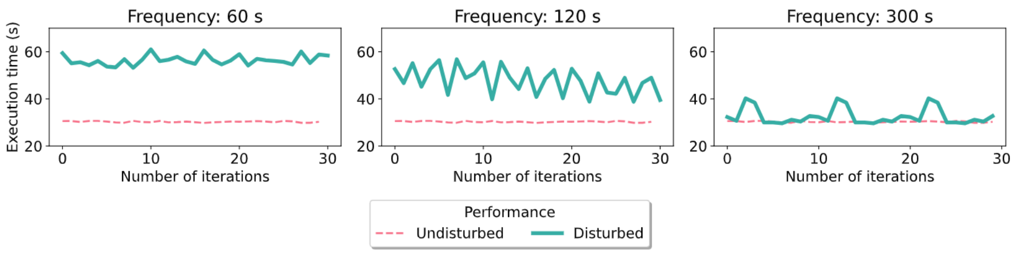

Fig. 2. Impact of noise on constant parametrization

Tested methods The black-box optimization heuristic is SMBO, using Gaussian Processes as the regression method and Expected Improvement as the acquisition function. The initialization method uses LHS, the number of initialization runs is set to 10. The maximum number of optimization runs is set to 100, resulting in a maximum total number of iterations of 110. The stop criterion stops the experiment automatically when there is less than a 5% improvement over the last 15 iterations. Each optimization process is repeated 5 times to average some of its random behavior.
The EVADyR algorithm is compared to static resampling (testing 3 and 5 resamples) and SEDR (using an interval width of 10% and 30% of the mean), as well as in the absence of any noise reduction strategy. As one of the major advantage of our methods is the absence of hyperparameters, we do not test different hyperparameters values for our solution.
All the code used to run the experiment is bundled in the SHAMan optimization framework, fully available as an Open-Source framework [1], and our experiments are thus fully reproducible.

5.2 Evaluation metrics

To evaluate the relevance of each optimization heuristic, we compute the different metrics described in table 1.

Table 1. Evaluation metrics for noisy optimization

Name	Abbreviation	Description
Distance to optimum parametrization	AvgDistOpt	Difference between the *average* of the found parametrization to the *best time on average*
Improvement compared to the default parametrization	ImprovDefault	Difference between the *average* of the found parametrization to the *average performance found for the default parametrization*
Total duration	Duration	Total duration of the experiment as the sum of the execution times at each iteration

The first metric *AvgDistOpt* is computed by measuring the distance between the average performance corresponding to the found parametrization and the optimum. This metric corresponds to measuring the asymptotic quality of the optimizer. It consists in running 15 times the parametrization found in the noisy case on the noiseless cluster and computing the mean. The metric *ImprovementDefault* corresponds to the distance between the asymptotic value of the parametrization returned by the tuner and the average performance measured

at the default parametrization. It represents the interest of using an auto-tuner rather than simply using the default parametrization.
The third metrics *Duration* reflect the time taken by the optimization experiment before the automatic stop criterion stops it in terms of elapsed time. There is a trade-off between the convergence speed and the quality of the optimization, as the more we perform evaluations and resampling, the more we gain insight on the system's behavior, but the more expensive resources are spent. We are thus looking for an equilibrium between having a solution close to the optimum and minimizing the resources cost.

6 Results and discussion

6.1 On the importance of noise reduction for tuning noisy systems

The values of the different metrics for the experiments when using Bayesian Optimization without any noise reduction technique are presented in table 2.

Table 2. Optimization results without noise reduction

Experiment ID	Avg Dist Opt (%)	Improv. Default (%)	Duration (s)
SRO.60	75.54	56.70	1200.98
SRO.120	319.92	-3.58	1079.57
SRO.300	66.25	58.99	827.03

These experiments show that stochasticity makes the auto-tuner ineffective without noise reduction, resulting in wasted time and resources: in the case of `SRO.120`, the parametrization returned by the optimization process performs worse than the default parametrization, and in the case of `SRO.60` and `SRO.300` an improvement is observed compared to the default parametrization of respectively 56.70% and 58.99%, but nothing as high as the 75% that can be expected. The experiments show that the noise makes the optimization problem more difficult, even in the case of lower frequency noise, such as `SRO.300`, and affects the quality of the optimizer, causing the stop criterion to be reached and the optimization to be stopped, as can be seen by looking at the convergence speed in table 2. The noise impact on the regression model and the acquisition function is also noted as a reason for the decreased optimization results.

6.2 Using state of the art's algorithms

Impact of static resampling The metrics computed for the different number of resamples in the case of static resampling are available in table 3. The results of using static resampling show improvement in the quality of optimization performed by the auto-tuner compared to not using any noise reduction. For every experiment, there is a number of resamples that improves significantly the performance of the tuner when compared to not using any resampling process in terms of distance to the ground truth. In the case of the less noisy experiment `SRO.300`, it even comes close to the optimum and the maximum observed potential improvement. However, the optimization process is slowed down by the re-evaluation of parametrizations that are not very impacted by noise, which are re-evaluated the same number of times as others that are more subject to noise. The experiments confirm the drawbacks of using static resampling, as it is hard to find the right exploration-exploitation trade-off and some iterations are wasted on parametrization with low noise.

Table 3. Metrics on static resampling

Resamp.	Exp.	Avg DistOpt (%)	Improv. Default (%)	Duration (s)
3	**SRO.60**	23.57	69.52	2430.698
	SRO.120	30.92	67.71	1411.23
	SRO.300	38.65	65.80	1211.79
5	**SRO.60**	15.98	71.39	1946.672
	SRO.120	27.37	68.58	1843.915
	SRO.300	2.84	74.63	1632.083

Impact of dynamic resampling Dynamic resampling does not show any benefits compared to static resampling and in some cases performs worse than without using any noise reduction strategy, as shown in figure 4. The reason behind this is the difficulty of finding the right parametrization of the algorithm depending on the noise setting. Optimization trajectories show in the case of a 10% interval width that the parametrization can be resampled too much or too little depending on the frequency of the noise. In the case of a 30% interval width, the resampling interval is too large and the algorithm does not have enough data to have a precise estimation of the performance function. These results highlight the advantage of dynamic resampling over static resampling in the case of regular Cauchy noise and confirm some of the results found in the state of the art.

Table 4. Metrics on dynamic resampling

Width (%)	Exp.	Avg Dist Opt (%)	Improv. Default (%)	Duration (s)
10%	**SRO.60**	53.99	62.02	4304.41
	SRO.120	112.40	47.61	3131.82
	SRO.300	28.21	68.37	1620.60
30%	**SRO.60**	309.31	-0.96	2240.62
	SRO.120	308.15	-0.68	1994.496
	SRO.300	205.76	24.58	2014.72

6.3 Using EVaDyR

The values of the metrics computed for the experiments using the EVADyR algorithm with all the improvements suggested in section 4 are available in table 5. Using dynamic intervals and bounded dynamic resampling combines the benefits of both dynamic and static resampling. It eliminates the need for selecting a specific hyperparameter for the algorithm and adjusts the resampling rate to the requirements of the noisy setting. This leads to a significant improvement in the distance to the optimum compared to standard dynamic resampling: it results in a decrease of the distance from 53.99% to 8.91% for `SRO.60`, from 112.40% to 5.12% for `SRO.120` and from 28.21% to 2.39% for `SRO.300`. This is reflected as well in the improvement compared to the default parametrization, which is above 73%.

Table 5. Metrics value using EVaDyR

Exp.	Avg Dist Opt (%)	Improv. Default (%)	Duration (s)
SRO.60	5.81	73.90	2045.52
SRO.120	4.27	74.28	1556.82
SRO.300	2.57	74.70	1311.26

In terms of distance to the ground truth, adding a resampling filter allows to find results very close to the ground truth optimum, which brings a strong improvement compared to the default parametrization. Even in the case of the most noisy optimization problem `SRO.60`, the performance measured at the returned parametrization is only 5.81% away from the ground truth, with an absolute difference of 0.5 seconds, reaching almost the full optimization potential of the auto-tuner. In the case of `SRO.120`, the performance measured at the returned parametrization is 4.27% away from the optimum, corresponding to a 0.36 seconds difference between the ground truth and the found performance. For `SRO.300`, the returned performance is 2.57% away, for an absolute difference of 0.15 seconds. Overall experiments, we find that the distance to the optimum is very small and almost negligible to users.
The most notable advantage of adding a resampling filter to dynamic bounded intervals is the improvement of the convergence speed, by preventing the evaluation of uninteresting parametrizations. This leads to a reduction in the number of iterations and total duration of the experiment, as we observe a time gain of 22% for `SRO.60`, 16% for `SRO.120`, 14% for `SRO.300`.

6.4 Summary of results

The results summary of using EVADyR algorithm compared to static resampling and SEDR, as well as not using any noise reduction strategy are available in table 6. They show that

EVADyR outperforms the state-of-the-art algorithms in terms of distance to the optimum and improvement from the default parametrization: static resampling is outperformed by 72.6% and dynamic resampling by 93.5%. The same improvement is seen when looking at the distance of the turned optimum from the default parametrization. EVADyR also provides a significant improvement in convergence quality, with a 97.46% improvement compared to not using any noise reduction. EVADyR thus both improves the convergence property of noise reduction algorithms and removes the need for hyperparameters, primarily by adding bounded dynamic interval resampling to the tuning process.
Another important result of our study is the importance of noise reduction: when not using any, black-box optimization is unable to perform the tuning in a noisy environment, as the returned parametrization brings little to no improvement compared to the default parametrization. EVADyR thus improves the convergence quality of the auto-tuner by 97.46% rather than when not taking the noise into account. In terms of convergence speed and experiment duration, the results show a time gain in terms of experiment duration compared to the state-of-the-art, because of the added resampling decision filter. Indeed, compared to static resampling, EVADyR brings a time gain of 9.38 %, and compared to dynamic resampling, EVADyR brings a time gain of 45.76 %.

Table 6. Comparison of our solution to the state of the art in terms of:

	None	**Static**	**Dynamic**	**EVADyr**
Distance to the ground truth (%)	153.90	15.39	64.86	4.21
Improvement to default (%)	37.37	71.53	59.33	73.88
Experiment duration (s)	1035.33	1807.55	3018.94	1637.86

7 Conclusion

In conclusion, this paper discusses the impact of noisy interference on the performance of black-box optimization auto-tuners. We prove that the noise cannot be neglected in production systems, as it significantly impacts the performance of the optimizer. To increase resilience, we suggest a new resampling technique called EVADyR and show that it improves convergence by 93.5% and reduces experiment duration by 45.76% compared to state of the art dynamic resampling. We also emphasize the importance of using noise reduction strategies in the case of highly shared systems, as we found a 97.46% increase in the optimum performance.
Planned improvement of this work consists in testing noise with a random time of arrival rather than the fixed tested intervals. This would allow to study the behavior of the noise reduction resampling algorithms when faced with more uncertainty in terms of noise arrival. Further works also include the comparison of our results to heuristic-specific noise resilient Bayesian Optimization, by using acquisition functions tailored for stochastic optimization, such as *Noisy Expected Improvement*, *Augmented Expected Improvement* and *Expected Quantile Improvement.* -–

References

1. The SHAMan application, https://github.com/bds-ailab/shaman
2. Aizawa, A.N., Wah, B.W.: Scheduling of Genetic Algorithms in a Noisy Environment. In: Evolutionary Computation. vol. 2, pp. 97–122 (1994)
3. Behzad, B., Byna, S., Prabhat, M., Snir, M.: Pattern-driven parallel i/o tuning. In: Proceedings of the 10th Parallel Data Storage Workshop. pp. 43–48 (2015)
4. Behzad, B., Luu, H.V.T., Huchette, J., Byna, S., Prabhat, Aydt, R., Koziol, Q., Snir, M.: Taming parallel i/o complexity with auto-tuning. In: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis. SC '13
5. Bennett, K., Ferris, M., Ioannidis, Y.: A genetic algorithm for database query optimization. In: Proceedings of the fourth International Conference on Genetic Algorithms. pp. 400 – 407 (1991)
6. Bergstra, J., Pinto, N., Cox, D.: Machine learning for predictive auto-tuning with boosted regression trees. In: 2012 Innovative Parallel Computing (InPar). pp. 1–9 (2012)

7. Cao, Z.: A Practical , Real-Time Auto-Tuning Framework for Storage Systems. Ph.D. thesis, State University of New York at Stony Brook (2018)
8. Cao, Z., Tarasov, V., Tiwari, S., Zadok, E.: Towards better understanding of black-box auto-tuning: A comparative analysis for storage systems. In: Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference. pp. 893–907. USENIX ATC '18 (2018)
9. Desani, D., Costa, V.G., Marcondes, C.A.C., Senger, H.: Black-box optimization of hadoop parameters using derivative-free optimization. 2016 24th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP) pp. 43–50 (2016)
10. Di Pietro, A., While, L., Barone, L.: Applying evolutionary algorithms to problems with noisy, time-consuming fitness functions. In: Proceedings of the 2004 Congress on Evolutionary Computation. vol. 2, pp. 1254–1261 (2004)
11. Duan, S., Thummala, V., Babu, S.: Tuning database configuration parameters with ituned. In: Proceedings of the VLDB Endowment. pp. 1246 – 1257 (2009)
12. Faraj, A., Yuan, X.: Automatic generation and tuning of mpi collective communication routines. In: Proceedings of the 19th Annual International Conference on Supercomputing. pp. 393 – 402 (2005)
13. Fitzpatrick, J.M., Grefenstette, J.J.: Genetic algorithms in noisy environments. In: Machine Learning. vol. 3, pp. 101–120 (1988)
14. Forrester, E.I.J., Keane, A.J., Bressloff, N.W.: Design and analysis of 'noisy' computer experiments. AIAA Journal pp. 2331–2339
15. Gramacy, R., Lee, H.: Optimization under unknown constraints. In: Bayesian Statistics. vol. 9 (2010)
16. Grohmann, J., Seybold, D., Eismann, S., Leznik, M., Kounev, S., Domaschka, J.: Baloo: Measuring and modeling the performance configurations of distributed dbms. In: 28th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS). pp. 1–8 (2020)
17. Gur, Y., Yang, D., Stalschus, F., Reinwald, B.: Adaptive multi-model reinforcement learning for online database tuning. In: International Conference on Extending Database Technology (2021)
18. Huang, D., Allen, T., Notz, W., Miller, R.A.: Sequential kriging optimization using multiple-fidelity evaluations. In: Structural and Multidisciplinary Optimization. vol. 32, pp. 369–382 (2006)
19. Ioannidis, Y., Wong, E.: Query optimization by simulated annealing. In: SIGMOD Record. pp. 9 – 22 (1987)
20. Jamshidi, P., Casale, G.: An uncertainty-aware approach to optimal configuration of stream processing systems. In: arXiv (2016)
21. Kassab, A., Nicod, J.M., Philippe, L., Rehn-Sonigo, V.: Assessing the use of genetic algorithms to schedule independent tasks under power constraints. In: 2018 International Conference on High Performance Computing Simulation (HPCS). pp. 252–259 (2018)
22. Letham, B., Karrer, B., Ottoni, G., Bakshy, E.: Constrained bayesian optimization with noisy experiments. In: ArXiv (2017)
23. Li, Y., Chang, K., Bel, O., Miller, E.L., Long, D.D.E.: Capes: Unsupervised storage performance tuning using neural network-based deep reinforcement learning. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. pp. 42:1–42:14. SC '17 (2017)
24. Liao, G., Datta, K., Willke, T.: Gunther: Search-based auto-tuning of mapreduce. In: Euro-Par. vol. 8097, pp. 406 – 419 (2013)
25. Melo Alves, M., de Assumpção Drummond, L.M.: A multivariate and quantitative model for predicting cross-application interference in virtual environments. Journal of Systems and Software **128**, 150–163 (2017)
26. Menon, H., Bhatele, A., Gamblin, T.: Auto-tuning parameter choices in hpc applications using bayesian optimization. In: 2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS). pp. 831 – 840 (2020)
27. Miyazaki, T., Sato, I., Shimizu, N.: Bayesian optimization of hpc systems for energy efficiency. In: High Performance Computing. pp. 44–62 (2018)
28. Ozer, G., Netti, A., Tafani, D., Schulz, M.: Characterizing hpc performance variation with monitoring and unsupervised learning. In: Jagode, H., Anzt, H., Juckeland, G., Ltaief, H. (eds.) High Performance Computing. pp. 280–292 (2020)
29. Picheny, V., Wagner, T., Ginsbourger, D.: A benchmark of kriging-based infill criteria for noisy optimization. In: Structural and Multidisciplinary Optimization. vol. 48, pp. 607–626 (2013)

30. Picheny, V., Ginsbourger, D., Richet, Y., Caplin, G.: Quantile-based optimization of noisy computer experiments with tunable precision. In: Technometrics. vol. 55 (2012)
31. Rasmussen, C.E., Williams, C.K.I.: Gaussian Processes for Machine Learning (Adaptive Computation and Machine Learning) (2005)
32. Robert, S., Zertal, S., Vaumourin, G.: Using genetic algorithms for noisy systems' auto-tuning: an application to the case of burst buffers. In: Proceedings of the International Conference on High Performance Computing Simulation (HPCS) (2020)
33. Robert, S., Zertal, S., Vaumourin, G., Couvée, P.: A comparative study of black-box optimization heuristics for online tuning of high performance computing i/o accelerators. In: Concurrency and Computation: Practice and Experience (2021)
34. Schmied, T., Didona, D., Döring, A., Parnell, T., Ioannou, N.: Towards a general framework for ML-based self-tuning databases (2020)
35. Seymour, K., You, H., Dongarra, J.: A comparison of search heuristics for empirical code optimization. In: 2008 IEEE International Conference on Cluster Computing. pp. 421–429 (2008)
36. Siegmund, F., Ng, A., Deb, K.: A comparative study of dynamic resampling strategies for guided evolutionary multi-objective optimization. In: 2013 IEEE Congress on Evolutionary Computation. pp. 1826–1835 (2013)
37. Stagge, P.: Averaging efficiently in the presence of noise. In: Eiben, A.E., Bäck, T., Schoenauer, M., Schwefel, H.P. (eds.) Proceedings of the 5th Parallel Problem Solving from Nature. pp. 188–197. Springer Berlin Heidelberg (1998)
38. Syberfeldt, A., Ng, A., John, R.I., Moore, P.: Evolutionary optimisation of noisy multi-objective problems using confidence-based dynamic resampling. In: European Journal of Operational Research. vol. 204, pp. 533–544 (2010)
39. V. K. Ky, C. D'Ambrosio, Y.H., Liberti, L.: Surrogate-based methods for black-box optimization. International Transactions in Operational Research (24) (2016)
40. Vázquez, E., Villemonteix, J., Sidorkiewicz, M., Walter, E.: Global optimization based on noisy evaluations: An empirical study of two statistical approaches. In: Journal of Physics Conference Series. vol. 135 (2008)
41. Wolpert, D.H., Macready, W.G.: No free lunch theorems for optimization. IEEE Transactions on Evolutionary Computation **1**(1), 67–82 (1997)
42. Zheng, W., Fang, J., Juan, C., Wu, F., Pan, X., Wang, H., Sun, X., Yuan, Y., Xie, M., Huang, C., Tang, T., Wang, Z.: Auto-tuning mpi collective operations on large-scale parallel systems. In: 2019 IEEE 21st International Conference on High Performance Computing and Communications; IEEE 17th International Conference on Smart City; IEEE 5th International Conference on Data Science and Systems (HPCC/SmartCity/DSS). pp. 670 – 677 (2019)