EVADyR: a new dynamic resampling algorithm for auto-tuning noisy High Performance Computing systems

Sophie Robert-Hayek[b], Soraya Zertal[b], Philippe Couvée[a]

[a]*Atos BDS R&D Data Management, Echirolles, 38130, France*
[b]*Li-PaRAD, UPSaclay-UVSQ, Guyancourt, 78297, France*

Abstract

Black-box auto-tuning methods have been proven to be efficient for tuning configurable computer hardware, including those encountered within the High Performance Computing (HPC) ecosystem. However, because of the shared nature of HPC clusters and the complexity of the software and hardware stacks, the measurement of the performance function can be tainted by noise during the tuning process, which can reduce and sometimes prevent the benefit of the tuning approach. A usual choice for performing the tuning in spite of these interference is to add a resampling step at each iteration to reduce uncertainty, but this approach can be time-consuming and must be done carefully. In this paper, we propose a new resampling and filtering algorithm called EVADyR (Efficient Value Aware Dynamic Resampling). Compared to the state of the art, it finds a better exploration versus exploitation trade-off by resampling only promising parametrization and increases the level of confidence around the suggested solution as the tuning process advances. This algorithm was able to tune efficiently two I/O accelerators highly sensitive to interference, in two different scenarios. Compared to Standard Error Dynamic Resampling (SEDR), a state of the art noise reduction strategy, we show that EVADyR is able to reduce the distance to the optimum by 93.5% and 24.7% for the two I/O accelerators respectively, as well as speed-up the experiment duration by 45.8% and 58.1% because less iterations are needed to reach the found optimum. Our results prove the importance of using noise reduction strategies whenever tuning systems running in production.

Keywords: Resampling, auto-tuning, noise reduction, I/O accelerators, HPC

PACS: 0000, 1111

2000 MSC: 0000, 1111

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 can have a huge impact on the systems performance [1] and they are challenging to tune by solely relying on field insight and user expertise, due to huge parameter spaces and complex, non-linear system behavior. Besides, the optimal configuration often depends on the current workload, and parameters must be changed at each environment variations. Consequently, users often have to rely on the default parameters given by the vendor, and do not take advantage of the possible performance of running their application on a tuned system. This problem is particularly observed within highly shared systems, such as public clouds and HPC systems, as the hundreds of assembled devices create very complex and highly configurable stacks. Faced with the inability of relying solely on users to choose the parametrization of complex computer systems, new tuning methods have emerged from various computer science communities to automate parameter selection depending on the current workload. Because they do not require any human intervention, these approaches are called *auto-tuning* methods. Throughout the years, they have been successfully applied to a wide range of systems, such as storage systems, database management systems and compilers.

Previous works have successfully demonstrated the efficiency of black-box optimization for tuning different computing components [2, 3]. However, these comparative studies were achieved 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 controlled and exclusive test environments, it does not always hold for resources shared across many users.

Complex system auto-tuning often occurs in shared noisy environments, requiring automatic tuning methods to consider potential interference that can degrade the performance of traditional auto-tuning heuristics. This phenomenon can severely affect the optimization algorithm and slow down the convergence process, even though the tuning process must stay as sparse as possible because of the cost of computing resources.

In this paper, we focus on the case of *High Performance Computing*, which is particularly affected by the problem of auto-tuning in noisy environments, due to its highly shared nature. We suggest a new resampling algorithm, called EVADyR (**E**fficient **V**alue **A**ware **Dy**namic **R**esampling), that takes into account the possible noise present when auto-tuning a shared system.

This method can be applied to any tunable systems within a noisy setting, and we choose to evaluate its performance by tuning two highly configurable appliances aimed at reducing I/O contention, called I/O accelerators. We show that on these use-cases our algorithm outperforms existing resampling state-of-the-art algorithm.

The main contributions of this paper are:

- A survey of existing noise reduction methods for Bayesian Optimization and a discussion of their limitations;
- A new resampling algorithm, EVADyR, adapted to the optimization of noisy and expensive functions;
- Validation of the new algorithm on two I/O accelerators with different noisy contexts compared to static and dynamic resampling.

This paper is organized as follows. We describe in section 2 the main principles of black-box optimization and its application to the case of system auto-tuning. Section 3 introduces the major works related to ours. Section 4 details several of the most effective methods to perform auto-tuning in a noisy setting, and section 5 introduces resampling methods for stochastic optimization. In section 6, we describe some of the limitations of the existing resampling algorithms and address them by proposing the EVADyR algorithm in order to make them more efficient. We then detail in section 7 the two tuned systems as well as the experiment plan designed to validate this method, and discuss the results in section 8. Section 9 concludes this paper and gives some insights into further works on the topic.

2. System tuning using Black-Box optimization

2.1. Black-box optimization for auto-tuning in shared systems

Black-box optimization refers to the optimization of a function of unknown properties, most of the time costly to evaluate, which requires a limited number of possible evaluations. As these methods are oblivious to the tuned system, they have emerged as a promising solution for tuning diverse systems. When applied to the tuning of computer systems, they consist in treating the tuned system as a black-box, deriving insight only from the relationship between the input and the output parameters, as described in figure 1. The input parameter θ_k of the black-box are the parameters of the tuned component and the output $F(\theta_k)$ measures the applications performance, which can for example be the elapsed time or a throughput.

The first step of any black-box optimization algorithm is the selection of the initial parameters to start the optimization process. An acceptable initialization starting plan must respect at least two properties [4, 5]: the space constraints and the non-collapsible property. The space constraints are shaped by the feasible set. The non-collapsible property ensures that every configuration has a different value for each dimension. Latin Hypercube Designs [4] are a usual choice as an initialization strategy, because of their simplicity and efficiency, and we select them for the initialization throughout all of our experiment.

The second step of black-box optimization is a feedback step. It consists in iteratively selecting a parametrization, evaluating the black-box function at this point and selecting accordingly the next data point to evaluate by using a higher procedure for searching an optimal solution in a parameter space,

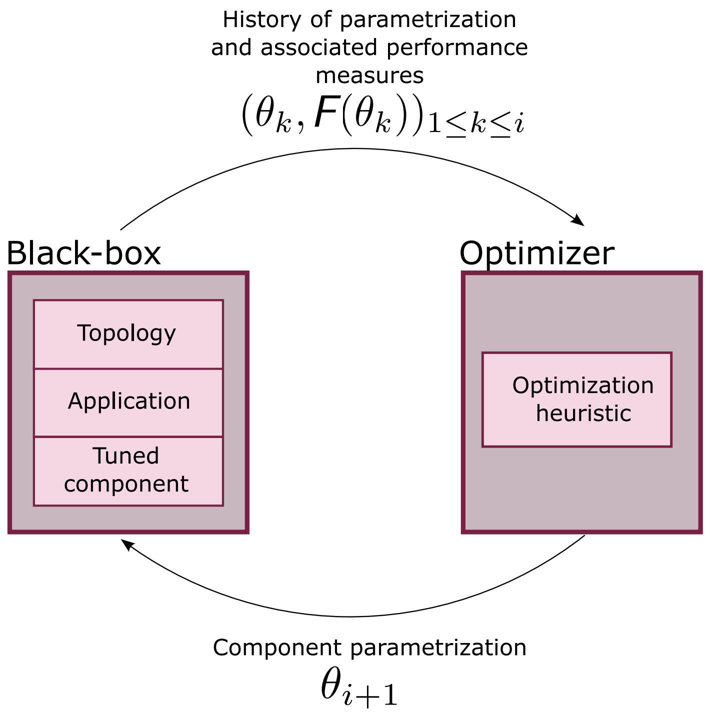

History of parametrization
and associated performance
measures
$(\theta_k, F(\theta_k))_{1\leq k\leq i}$
Black-box
Topology
Application
Tuned component
Optimizer
Optimization heuristic
Component parametrization
θ_{i+1}

Figure 1: Schematic representation of the black-box

called an optimization *heuristic*. Several optimization heuristics are available in the literature, such as genetic algorithms [6], simulated annealing [7] and Bayesian Optimization [8].
The optimization process then runs until the improvement of the best objective function value is below a threshold t for a number of iterations g.

Auto-tuning highly shared systems presents several challenges, as the resources are shared among multiple users and their applications and the delivered performance fluctuates according to the ressources availability. Consequently, the auto-tunning process must take place in a shared and noisy setting to be able to distinguish between the true system behavior and random variations, while remaining as sparse as possible in terms of iterations.

2.2. Problem formulation

Formally, the problem to solve in this paper can be defined as follows and is closely related to the goals of stochastic approximation [9].

Let θ_i the parametrization of the system to optimize, which belongs to a discrete subset of the possible parametrizations of the tunable system $\Theta = \{\theta_i\}_{i \in \mathbb{N}}$.

Let $\mathcal{A}$ be the application or workload for which we choose to optimize the parameters and $\mathcal{E}$ the execution context for which we want to perform this optimization.

Let $f : (\mathcal{A},\ \mathcal{E}, \Theta) \longrightarrow \mathbb{R},\ \theta \longrightarrow f(\theta)$ represent the performance measure of application $\mathcal{A}$ running in context $\mathcal{E}$ for parametrization θ. Because of the noise, we only have access to the "observed" or "sampled" execution times $F(\theta)$. The additional random variations, which we call "noise", can depend on the parametrization as well as the optimization step and is a function of θ and n. We represent it by $\epsilon(\theta, n)$.

We treat the execution time as a random variable which realization are the elapsed times for each run. 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:

$$min\ \mathbb{E}(F(\theta)),\ \theta \in \Theta$$

$$F(\theta) = f(\theta) + \epsilon(\theta, n)$$

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 $-f$.
We deduce 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 considering all the parametrizations. This ensures that the optimum is not a result of chance and actually due to the system's parametrization. Throughout the remainder of this paper, the term "optimum" will refer to the optimal parametrization on average.

3. Related works

In the literature, black-box optimization is a popular choice for tuning different tunable systems, and it has been particularly helpful in computer science to look for the optimal configuration of various software and hardware components. It is for example the methodology chosen by Liao et al. in their Gunther framework [10] which relies on Genetic Algorithms to find the optimal parametrization of Hadoop. In [11], Jamshidi et al. use Bayesian Optimization to optimize the stream processing system Storm. 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 [12] to improve the performance of a PostgreSQL database for different workloads and in [13], Xi et al. have used a variation of Simulated Annealing to find the optimum configuration of Web servers.

Storage systems are notoriously hard to tune, as they have a very large parameter space and a strong dependence on the running workload [3, 14]. The efficiency of black-box optimization for tuning such systems when faced with different workloads has been explored by Cao et al. in [15]. Reinforcement learning has also been successfully used as an auto-tuner to optimize the performance of the Lustre filesystem by Li et al. in [14] and an optimal parametrization for the several layers of HDF5 library was found using genetic algorithms by Behzad et al. in [16] with an extension to select the best parameters according to the I/O pattern in [17].

Within the HPC community, auto-tuning has gained a lot of attention for tuning particular HPC application and improve their portability across architectures [18]. In [19] and [20], Seymour et al. and Knijnenburg et al. provide a comparison of several random-based heuristic searches for code auto-tuning. In [21], 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 [22] that 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 [23]. A scheduling algorithm using genetic algorithms has been introduced by Kassab et al. in [6] 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 parameter space in [24] and [25].

However, the literature addressing the problem of noise affecting performance measurement whenever tuning real systems is surprisingly sparse, as most of the works previously cited do not study the potential interference on the tuned system and do not mention their tuner resilience to noise either. While this is not critical when working on single user systems, such as local hard drives for personal computers [15], it cannot be ignored when working on highly parallel shared systems [26, 27, 28]. Several studies do acknowledge their system's noise [3], 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 [29, 30]. To our knowledge, the only notable exception is the Baloo framework [31] 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. This type of adaptive resampling is explored in section 5.2, and its main drawbacks are addressed in section 6.

4. Existing noise reduction algorithms

While almost non-existent in the system's auto-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 separated into two main categories:

Heuristic specific noise reduction. Heuristic specific optimization consists in modifying the heuristic itself in order to make it more resilient to noise. For example, 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 the case of Bayesian Optimization, we can the work Vàsquez et al. [32] who use quantiles as estimations of the performance function through Gaussian Processes, Letham et al. *Noisy Expected Improvement* for using Quasi Monte Carlo simulation [33] and Huang et al. who suggest the *Augmented Expected Improvement* as a robust estimation of the best solution based on the lowest β-quantile, with β a configurable value [34]. Picheny et al. propose the *Expected Quantile Improvement* for selecting the next data point [35] and Forrester et al. suggest a reinterpolation procedure using Gaussian Process regression for computing Expected Improvement [36]. Additionally, genetic algorithms can benefit from the GASAC algorithm [37]. While these different methods have proven to be efficient when facing different noises on different benchmarking functions [38, 39], they have the major drawbacks of being very specific to the selected optimization heuristic, and cannot be generalized to other heuristics. As discussed in works such as [2, 3, 15, 40], there is no single best performing heuristic for every optimization problem, and we wish to deal with noise reduction methods that allow switching heuristics depending on the context, focusing ourselves on heuristic agnostic noise reduction.

Heuristic agnostic noise reduction. Heuristic agnostic optimization methods only modify the way the data points are acquired and the way the fitness function is fed into the algorithm. Two of the most efficient agnostic heuristic noise reduction methods are fitness aggregation and resampling:

(a) **Fitness aggregation**: Fitness regression uses parameter combinations that have already been evaluated to estimate the execution time at each parametrization using regression techniques [41], using either the previous history or only neighboring parameters. Possible solution include approximating the fitness function as a weighted average of evaluated parameters (Memory-based Fitness Evaluation Genetic Algorithms, MFEGA [42, 43]), using neighboring data points to model noise in different zones of the parameter space [44], fitting a Kriging model for fitness approximation in evolutionary algorithms [45] or using a local quadratic regression model for fitness estimation [46].

(b) **Resampling**: Resampling consists in reevaluating several times the same parametrization in order to get a more precise idea of its impact on the execution time [47, 48, 49], which will detail thoroughly in section 5.

While fitness aggregation methods has yielded some good results for improving convergence speed on theoretical problems, as shown by [43, 42], a study of ours in [50] has shown it to be less efficient than resampling methods on our particular application for tuning I/O accelerators. We thus focus this work on resampling methods and their possible improvement.

5. Resampling methods

Resampling consists in adding a ”resampling filter” by using a set logical rule to select which parametrization should be reevaluated. A detailed schematic representation of the integration of resampling within the black-box optimization tuning loop is available in figure 2. The general goal of resampling is to reduce the bias in using the mean estimator as the value of the performance metric.

Algorithmically, we define a resampling filter as a function $\mathcal{RF}$ which takes as input an optimization's trajectory already evaluated fitness and corresponding parameters $(\theta_i \in \Theta, F(\theta_i) \in \mathbb{R})_k$, for the optimization trajectory

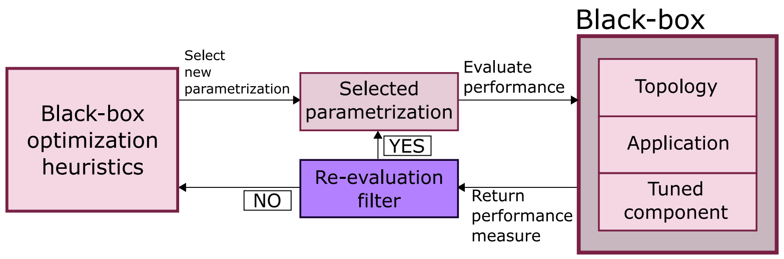


Figure 2: Schematic representation of resampling algorithms

at step k, and outputs a boolean on whether or not an already evaluated parametrization should be re-evaluated.

$$\mathcal{RF} : (\Theta, \mathbb{R}) \longrightarrow \{0, 1\}$$

This filter can be integrated for every draws including initialization draws, but we decide to exclude initialization configuration as we want the initialization plan to remain able to test as many parametrization as possible.

Resampling is a trade-off between having a better knowledge of the space and wasting some computing time on re-evaluation. Many strategies exist in order to efficiently reevaluate a parametrization, and we will present two of the most popular and efficient resampling algorithms in the literature. For a more exhaustive description of resampling methods, the reader can refer to the thorough classification proposed by Siegmund et al. in [51].

5.1. Simple resampling

Simple resampling computes a fixed number of times the fitness value of the selected parametrization [48], regardless of the parametrization and its associated fitness. In our case, it consists in launching a fixed number of times the application and the tuned system with the same selected parametrization.

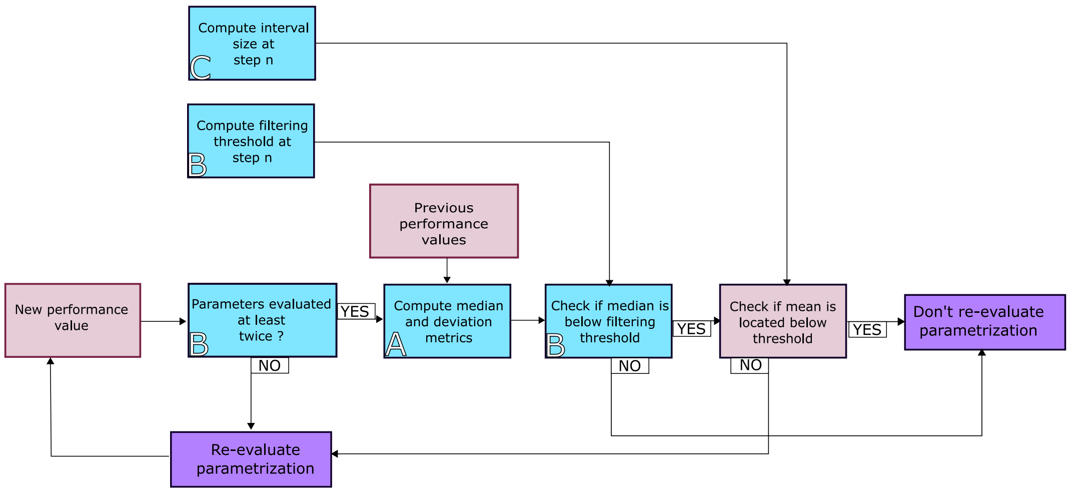
The main drawbacks of simple resampling is their lack of adaptivity to the noise present on the parameter space: the need for resampling is most of the time not homogeneously distributed throughout the search space [51]. This is especially true for shared and distributed computer systems, where the noise is often concentrated in a short time span, for example when there is a backup or another user performing concurrent accesses on the storage systems.

5.2. Dynamic resampling

Standard Error Dynamic Resampling (SEDR), as introduced by Di Pietro et al. in [52], adapts the number of samples to the noise strength measured at each parametrization. The parametrization is re-evaluated until the width of the 95% confidence interval for normally distributed data around the mean for the studied parametrization $ci : \Theta \to \mathbb{R}$ is below a fixed threshold τ_{se}.

$$ci(\theta) = 2 \times 1.96 \times \frac{\hat{\sigma}(\theta)}{\sqrt{j_\theta}} = 2 \times 1.96 \times \frac{\sqrt{\frac{1}{j_\theta - 1} \sum_{j=1}^{j_\theta} (F(\theta)^{j_\theta} - \hat{f}(\theta))^2}}{\sqrt{j_\theta}}$$

While simple averaging reevaluates each parametrization n times, SEDR introduces a variable number of evaluations n_θ for each parametrization. SEDR has proven to be efficient on theoretical problems, as described in the works of Di Pietro et al. [52], as well as practical ones as tested by Syberfeldt et al. in [53]. It is also included in the Baloo framework [31], under the name of *Adaptive Resampling*, where it has yielded some good results for tuning distributed databases.



Compute interval size at step n
C
Compute filtering threshold at step n
B
Previous performance values
New performance value
Parameters evaluated at least twice ?
B
YES
NO
Compute median and deviation metrics
A
Check if median is below filtering threshold
B
YES
NO
Check if mean is located below threshold
YES
NO
Don't re-evaluate parametrization
Re-evaluate parametrization

Figure 3: Schematic representation of the EVADyR resampling algorithm

6. Improving resampling methods

While existing methods have proven to be efficient on several theoretical problems [51] as well as practical ones [54], we have identified the following drawbacks which we address in our EVADyR algorithm, described in figure 3.

6.1. Too many iterations on a single parametrization

When the noise is very high for a single parametrization, resampling methods based on noise measurements, such as dynamic resampling, can spend all their budget on a single parametrization because the deviation is large. However, when working with real system, this strong noise is often temporary, for example because the system is going through a particularly high traffic such as a database backup, and the optimizer would benefit more from moving on to another parametrization instead of getting stuck in a locally noisy parametrization.

We suggest to include in our algorithm **a ceiling limit N to the allowed**

number of resamples on a given parametrization, the optimizer being able to come back later to explore more thoroughly this parametrization. Each parametrization is resampled at least twice in the case of dynamic resampling in order to measure the noise on this data point and see if resampling is required, this means that the number of resamples n_θ for parametrization θ is located between 2 and N. This is represented by **component A** in is below a fixed threshold τ_{se}.figure 3. Throughout the rest of this paper, we set the maximum number of resamples per parametrization to 10% of the total maximum allowed budget. This variation is already included in Baloo in [31].

6.2. Dependence on hyperparameters

Dynamic and static resampling have a strong dependence on their hyperparameters (number of resamples in the case of simple resampling and the interval confidence width in the case of dynamic resampling). Indeed, a too high threshold would not trigger any resampling, while a low threshold would trigger too many resampling and prevent the algorithm from exploring the feasible region.

We suggest **dynamic confidence intervals** to remove the dependence of the algorithm on the hyperparameter threshold, by making the size of the confidence interval dependent on the number of iterations. The size of the confidence interval becomes inversely proportional to the number of elapsed steps by using a bounded decreasing exponential to select the ratio of the confidence interval at each step. We also make it proportional to the mean measured for this parametrization, to take into account the fact that the required precision on the mean estimator is dependent on the magnitude of

the mean.
The width of the confidence interval (ci) around the mean thus gets smaller and smaller as the number of iterations raises, as we verify:

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

This ensures that at the beginning of the optimization process, the algorithm is more lax about the precision we want for a parametrization, as we would rather test many different parametrizations to explore the parameter space. However, at the end of the optimization process, we want to increase the precision of the estimate of the performance function for a given parametrization, ensuring adequate exploitation of already known ones. This has three benefits: we keep a strong exploration component of the algorithm at the beginning of the optimization process, we ensure a final knowledge of final parametrizations and we do not have to set any external hyperparameters. We reflect this change in **component C** of figure 3.

6.3. No comparison with already tested parametrization

The existing algorithms **do not consider how the measured value of the fitness at this data point ranks when compared to the other tested fitness during the resampling process**. Because of this, it can resample several times a slow parametrization (compared to the already tested ones) and slows the convergence process of the tuning algorithm.
We thus introduce a dynamic filtering component before performing the resampling, to address drawbacks (c). Its goal is to filter the most promising

parametrizations before submitting them to the resampling which reduces its time cost, so that no optimization budget is wasted on slow parametrizations compared to the rest of the evaluated ones. 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 performance for this parametrization is inferior to a certain ratio of the current median (for example if the ratio is set to 1, we ensure that the new parametrization is better than at least 50% of the already tested parametrizations), move to the next step, otherwise move to step 4.
3. Keep this parametrization and submit it to the resampling process.
4. Discard this parametrization as its evaluations are not promising, 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. Any decreasing function can be used, and we selected a bounded decreasing exponential function. 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. This filtering component is reflected by the **components B** in figure 3.

More formally, if we denote as FC the function selected to perform the filtering, we ensure that each time a new parametrization θ is selected at step n, it is evaluated at least twice and re-evaluated only if the median performance $med(F(\theta_i)^j_{1 \leq j \leq n_\theta})$ for parametrization θ_i verifies at step n:

$$med(F(\theta_i)^j_{1 \leq j \leq n_\theta}) \leq FC(n) \times med(F(\theta_i)_{1 \leq i \leq n})$$

$$med(F(\theta_i)^j_{1 \leq j \leq n_\theta}) \leq max(0.99^n, 0.5) \times med(F(\theta_i)_{1 \leq i \leq n})$$

where $med(F(\theta_i)_{1 \leq i \leq n})$ corresponds to the median of the whole other performance measures up to this new parametrization. Although this filtering mechanism is not unique in the resampling literature (see, for example, the works suggested by Barry et al. in [55]), it is both simple and effective, as demonstrated by our experimental results.

6.4. Implementation

An implementation of the algorithm is available within the Open-Source SHAMan auto-tuning framework [56], which ensures the reproducibility of our experiments.

7. Experiment plan

To evaluate the relevance of noise reduction and compare EVADyR to the state-of-the-art, we perform the tuning on two different I/O accelerators, a smart prefetch strategy and a burst buffer. For each accelerator, we design two different experiment plans, each with its own noisy context encountered when auto-tuning in production. These I/O accelerators have been selected as relevant noisy tunable systems as they are real-life use-cases and tuning them in production is a particularly difficult issue.

7.1. Selected optimization heuristic: Bayesian Optimization

Even though our resampling algorithm is heuristic agnostic, we select for testing purpose an optimization heuristic and we choose Bayesian Optimization, basing ourselves on a previous study of ours [2]. Bayesian Optimization, also referred to as Sequential Model Based Optimization (SMBO), consists in using a probabilistic model to approximate the function to optimize and use this model to select the next data point that will be evaluated by the real function. This model is less costly to evaluate than the actual function and thus many data points can be evaluated in order to have a better knowledge of the original function. Bayesian Optimization requires two inputs to be instantiated: the acquisition function and the probabilistic model used to represent the performance function.

7.1.1. The acquisition function

An acquisition function indicates for each configuration its potential performance improvement by being evaluated next, given the input of the probabilistic model. It should offer a trade-off between *exploration* of parametric zones where the model is uncertain (*i.e.* zones with a high variance) and the *exploitation* of already promising well-explored zones (*i.e.* zones where the mean is low). We select the most common acquisition method: the *Expected Improvement* (EI), which computes the expected improvement $EI(\theta)$

from switching from the best configuration seen so far θ^* to the examined parametrization θ:

$$I(\theta) = \begin{cases} f(\theta) - f(\theta^*) & f(\theta) < f(\theta^*) \\ 0 & f(\theta) \geq f(\theta^* \end{cases}$$

$$EI(\theta) = \mathbb{E}(I(\theta))$$

7.1.2. The probabilistic model

A suitable probabilistic model should be able to give an estimation of the mean and the standard deviation for each possible parametrization. The most popular choice is Gaussian processes, a probabilistic modeling technique used for non-parametric regression and uncertainty estimation. We will focus solely on Gaussian Processes, because of their efficiency in low-dimensional numerical input spaces [57].

7.2. Tuned systems

I/O accelerators are software or hardware components which aim is to reduce the increasing performance gap between compute nodes and the back-end storage, that can slow down I/O intensive applications [58]: on large supercomputers, many nodes performing reads or writes stress the back-end storage and can make the application wait while it performs its I/O, generating I/O bottlenecks. This is especially true for HPC applications that periodically save their current state by performing checkpoints, which causes many writes during a short timeframe [59]. To mitigate these problems, several I/O accelerators have been developed over the years. In this paper, we present and tune two I/O accelerators: a smart prefetch strategy called *Small Read Optimizer* (SRO) [60] and the *Smart Burst Buffer* (SBB) [61].

7.2.1. Small Read Optimizer (SRO)

The Small Read Optimizer (`SRO`) consists of a dynamic data preload strategy to prefetch chunks of files that are regularly accessed on the compute node memory, in order to reduce the latency gap between the computing on the microprocessors and data access on the main storage system. It consists in automatically detecting zones in the file that are accessed several times in order to load into memory the whole file zone. The file is divided in several zones (`binsize`), and for each of these zones, the number of accesses is recorded. Once a certain number of access (`cluster_threshold`) has been recorded for a given bin, a zone `prefetch_size` is loaded in the cache. The parameter `sequence_length` tracks the number of accesses that should be kept within the internal memory of the algorithm.

7.2.2. Smart Burst Buffer (SBB)

The Smart Burst Buffer (SBB) is the Atos' commercial implementation of a burst buffer, acts as a temporary cache between computing processes and permanent storage. It redirects bursts of I/O to this cache to prevent I/O bottlenecks. The SBB allocates two cache levels on a node, called datanode, connected to compute nodes with high bandwidth RDMA connections: a RAM cache and an NVMe cache. Parametrizing the SBB is complex and application-specific, with considerations like bursty writing behavior requiring more RAM and Flash destagers threads to ensure smooth I/O processing.

7.3. Hardware

All of the used computes nodes over these experiments are Intel(R) Xeon(R) CPU E5-2650 0 @ 2.00GHz with 16 physical cores (32 logical cores), dual-socket, and 82 GB of DDR4-DRAM. The back-end parallel filesystem is Lustre [62] with 40TB. In the case of the SRO, a single compute node is used. In the case of SBB, the benchmark runs on three compute nodes and one I/O node (where the burst buffer's cache is allocated). The compute nodes are connected to the data node through FDR RDMA (56Gb/s) network.

7.4. Tuned applications

SRO case. The selected application is an I/O benchmark performing operations of size 4kB, with 500 hotspots of width 50MB, on a file of size 500GB. Within each hotspot, 10000 operations are performed randomly before moving on to the next file zone. This type of pattern is representative of an application accessing different zones of the data file and performing many random accesses around this file zone if found interesting, before moving on to the next zone : a possible example are applications exploring different geographical zones within a map. To find the optimal parameters of the accelerator for the benchmark in order to use them as the ground truth, we run 20 auto-tuning optimization experiments on isolated nodes and storage system to remove any possible interference. We use Bayesian optimization and the initialization plan uses Latin Hypercube Sampling (LHS) as described in section 2.1, with 10 iterations for initialization. The maximum number of iterations is set to 100 optimization steps. The selected stop criterion is improvement based, and the optimization process stops if there is less than

5% of improvement in the optimum over 15 iterations. The optimization grid is available in table 1.

Table 1: Optimization grid for the SRO experiments

Parameter name	**Minimum value**	**Maximum value**	**Step**	**Unit**
Cluster threshold	2	102	20	N/A
Binsize	262144	1048576	262144	Byte
Prefetch	1048576	10485760	1048576	Byte
Sequence length	50	100	10	N/A

We measure that the optimal execution time remains stable across all experiments with a standard error of less than 1 second, indicating convergence to the same optimal performance. The average potential tuning improvement is around 75.33% with a standard error of 1.15%, confirming control over the tuning environment. The retained best performance, which we will use as the ground truth for the optimum, is the average of the best performance over all 15 experiments, equals to 8.88 seconds.

SBB case. The application used for evaluation is the standard I/O benchmark IOR [63], commonly used for evaluating the performance of parallel file systems. We use it to perform a write sequential pattern, writing a total of 174 GB using 32 parallel I/O processes, each process writing to its own file by blocks of 1MB. This scenario simulates a write-intensive phase of a standard HPC application's checkpoint creation. The default parametrization, which sets the number of workers and destagers to 10, the cache to its maximum size and the cache threshold to 90% yields a mean execution time of 172.69s, which is 12% slower than the optimal configuration. The optimal configuration in terms of cache size is, as expected, the biggest possible (60GB). The default number of workers is also the optimal possible value. However, setting the number of destagers to the maximum possible value, as would be the most intuitive decision based on field-expertise, is actually counter-productive, as the found optimum is equal to 8, and emptying the data node once its 50% full yields the best results.

7.5. Noise generation

7.5.1. The SRO case: concurrent accesses

To test the behavior of noise reduction methods in different use-cases, we create some noise during the tuning process by performing some concurrent accesses on the same file used by the application during the optimization run.

Generation methodology. The noise is created by running in parallel of the main application concurrent applications on other computing nodes, which perform some random operations on the same data file. Three nodes are used, two nodes are performing write operations and one read operations. 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. It is set to a constant value and is expressed in seconds in table 2. Three different noise frequency have been tested and selected to match a certain frequency of arrival relative to the duration of the execution time of the I/O benchmark with default parameters. We will denote each SRO experiment as `SRO.n`, n being the time of arrival of the noise. For example, `SRO.60` will refer to the SRO experiment with interference coming every 60 seconds.

Table 2: Frequency of the noise

Elapsed time between arrivals	**Frequency relative to the application run**
60s	Twice per default optimization run
120s	Every 4 default optimization runs
300s	Every 10 default optimization runs

Noise characterization. The noise profile varies depending on the frequency of arrival. For a time of arrival of 60 seconds, a constant noise pattern is observed, causing each run to be translated to a higher value. However, with a time of arrival of 120 seconds, the noise becomes less constant, and the application's value does not return to its true value between each run, creating a confusing landscape for the optimizer. On the other hand, for a time of arrival of 300 seconds, we notice a Cauchy-type noise, where some runs take longer due to the noise impact but return to the original value between each run.

We can see that this noisy environment provides different types of challenge for the optimizer: dealing with a constant noisy environment, each measured performance being translated to a higher value (60 seconds arrival), an unstable noisy environment (120 seconds arrival) and an impulse-type noise (300 seconds arrival).

7.5.2. Tested methods

For the optimization process, we select the same heuristic parametrization as described in paragraph 7.1: SMBO, using Gaussian Processes as the regression method and Expected Improvement as the acquisition function. The initialization method uses LHS and 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 is set to stop 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.

We compare EVADyR as suggested in section 6 to two methods from the described state-of-the-art, static resampling and adaptive resampling as described in section 5, as well as when not using any noise reduction strategy. The tested hyperparameters for these methods are available in table 3. For our suggested improvements, we compare the impact of adding a dynamic confidence interval and a performance resampling filter to the state-of-the-art. As one of the major advantage of our methods is the absence of hyperparameters, we do not test different hyperparameters values for our solution.

Table 3: Tested hyperparameters for state of the art algorithms for the SRO experiment

Method	Hyperarameter name	Tested values
Static resampling	Number of resamples	$\{1, 3, 5\}$
Dynamic resampling	Confidence interval width	$\{10\%, 30\%\}$

7.5.3. The SBB case: concurrent accesses

Noise generation. In the case of SBB, the noise generation strategy is different and instead of generating it manually, we use the noise observed on a cluster shared by twenty users, over a twenty days period. During this time frame, we exhaustively measured the performance corresponding to different parameters repeated several times, in order to obtain noisy measurements, and build an ex-situ dataset with several performance measurements per parametrization. To build the exhaustive test dataset, a total of 432 tested distinct parametrizations are tested and available in table 4.

Table 4: Tested values for the burst buffer parameters

Parameter name	**Range**
Number of workers	$\{2, 4, 6, \ldots, 12\}$
Number of destagers	$\{2, 4, 6, \ldots, 12\}$
Cache size	$\{40GB, 50GB, 60GB\}$
Cache threshold	$\{50\%, 60\%, 70\%, 90\%\}$

Each parametrization is sampled 16 times to evaluate the effect of noise on the system. This number was chosen as a trade-off between statistical significance and realistic experimental time constraints. According to the notations introduced in section 2.2, this corresponds to collecting m runtimes $F(\theta_i)^j_{1\leq j\leq m}$ for each of the n parametrizations $(\theta_i)_{1\leq i\leq n}$ of the burst buffer. In total, the collection of this exhaustive test dataset required 474 hours.

Loop simulation. To compute the metrics on realistic trajectories, we use another approach than running them on a real system as in the SRO case: the auto-tuning loop is simulated offline to confront the heuristics to real-life conditions. The generated dataset is used as a black-box: at each iteration, the heuristic suggests a new parametrization and the dataset is queried to return a randomly selected execution time among those available for this parametrization. The dataset is then used to replace the black-box (*i.e.* the combination between the burst buffer, the application, and the execution context) in the real-life loop. This approach has the advantage of being very fast to use for tests, because of the ex-situ dataset, as opposed to running the noise reduction optimization experiments directly as for the SRO. It has however the drawbacks of being a simplified version of reality, because of the limited 16 sampled performance measure per parametrization, instead of the thousands of possible measures observed in practice.

Noise characterization. From table 5, we deduce that the coefficient of variation is approximately 25% in the case of the raw elapsed times and 17% on the averaged elapsed times. This indicates a strong noise in the dataset to challenge the optimization heuristic.

Table 5: Statistics on raw and averaged elapsed times within each parametrization

	Raw elapsed time	**Average elapsed time**
Mean (s)	197.33	197.33
Standard error (s)	49.49	34.60
Minimum (s)	136.00	151.75
Maximum (s)	625.00	327.37
Median (s)	185.00	188.81

7.6. Evaluation metrics

To evaluate the relevance of each optimization heuristic, we compute different metrics described in table 6.

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. In the case of the SBB dataset, it consists in computing the mean over the 16 repetitions, while in the case of the SRO, it consists in running 15 times the parametrization found in the noisy case on the

Table 6: Evaluation criteria 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*
Convergence speed	Convergence	Number of elapsed iterations when the stop criterion stops the experiment
Total duration	Duration	Total duration of the experiment as the sum of the execution times at each iteration

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 and fourth metrics *Convergence* and *Duration* reflect the time taken by the optimization experiment before the automatic stop criterion stops it, both in terms of steps and 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 wanting to have a solution close to the optimum, while minimizing the resources cost.

8. Results and discussion

The results are organized as follows: first we give a preambule quantative study on the importance of noise reduction in our context, then we present the optimization trajectories without any noise reductionand using state-of-the-art resampling methods, followed by the results obtained when using the improvements we suggested in section 6.

8.1. On the importance of noise reduction

The values of the different metrics for the two I/O accelerators when using Bayesian Optimization without any noise reduction technique are presented in table 7 and illustrated in figure 4.

These results show that noise renders the auto-tuner ineffective when no noise reduction is applied. In the SBB and `SRO.120` experiments, the optimization process even returns parametrizations worse than the default,

Table 7: Optimization results without noise reduction for the two I/O accelerators

Experiment	**Time of arrival (s)**	**Avg Dist Opt (%)**	**Improv. Default (%)**	**Convergence**	**Duration (s)**
SRO	**60**	75.54	56.70	22.00	1200.98
	120	319.92	-3.58	21.00	1079.57
	300	66.25	58.99	26.00	827.03
SBB		12.90	-0.60	30.73	5728.82

making it a waste of time and resources. This behavior results in faster convergence but yields suboptimal parametrizations in noisy environments.

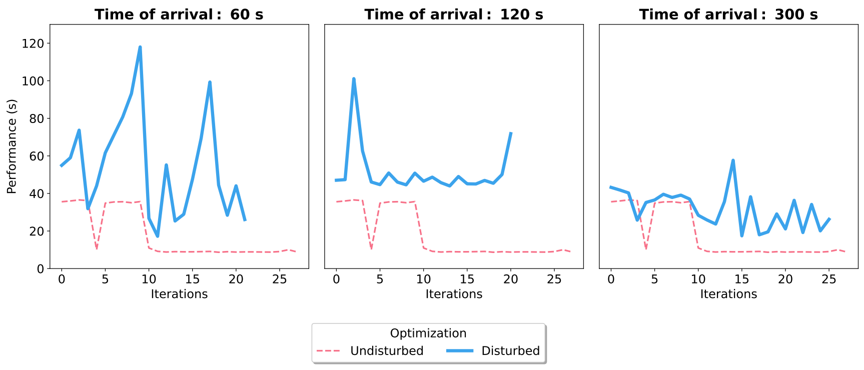


Figure 4: Impact of noise on optimization trajectories for the SRO experiment

These results highlight the importance of taking into account the noise whenever performing an optimization in a noisy setting, both in the case of a constant noise as with the SBB experiment and in the case of a regularly arriving noise with the SRO experiment.

8.2. Performance of existing methods

8.2.1. Static resampling

The metrics computed in the case of static resampling are available in table 8 for the SBB and in table 9 for the SRO.

Table 8: Results for static resampling on the SBB dataset

Number of resamples	**Avg DistOpt (%)**	**Improv. Default (%)**	**Convergence**	**Duration (s)**
2	8.2	3.6	49.1	8924.9
5	6.9	4.8	100.3	17948.2
10	6.7	4.9	160.0	28204.7
15	13.4	-1.1	160.0	29340.2

Table 9: Results for static resampling on SRO experiment

Number of resamples	**Time of arrival (s)**	**Avg DistOpt (%)**	**Improv. Default (%)**	**Convergence**	**Duration (s)**
3	**60**	23.6	69.5	78.0	2430.7
	120	30.9	67.7	51.0	1411.2
	300	38.7	65.8	51.0	1211.8
5	**60**	16.0	71.4	70.0	1946.7
	120	27.4	68.6	70.0	1843.9
	300	2.8	74.6	90.0	1632.1

When comparing these results to not using any noise reduction, we see a clear improvement for every experiment in terms of the quality of the optimization performed by the auto-tuner. For example, in the case of `SRO.300`, it even comes close to the optimum and the maximum observed potential improvement compared to default. However, using static resampling slows down the optimization process because the stop criterion does not efficiently end the experiment once resampling starts. We find for the SBB case, the optimal number of resamples is 10, while for SRO, it's 5: this difference underscores the challenge of selecting the right hyperparametrization, especially as noise levels vary with the system's state. Too many resamples waste resources in less noisy systems, while too few resamples lead to wasted optimization attempts in highly noisy systems. These experiments confirm the drawbacks of static resampling as discussed in section 6 and align with findings in [51].

8.2.2. Dynamic resampling

Dynamic resampling in the SBB experiment improves convergence speed compared to static resampling, as displayed in table 10. Using dynamic resampling with a 30% interval width reduces the optimization experiment from 160 to 82 steps, resulting in a shorter duration of approximately 4 hours and 30 minutes instead of 7 hours and 45 minutes.

Table 10: Results for dynamic resampling on the SBB experiment

Percentage (%)	**Avg Dist Opt (%)**	**Improv. Default (%)**	**Convergence**	**Duration (s)**
1	12.5	-0.3	160.0	31363.4
3	14.4	-2.0	160.0	29243.6
5	9.8	2.2	160.0	29485.6
10	11.6	0.6	160.0	30601.4
30	6.6	5.0	82.0	15657.7
50	8.9	3.0	51.2	9349.0

When it comes to SRO experiments, displayed in table 11, the performance of dynamic resampling is less pronounced than in the case of the SBB experiment, especially for noisy problems (`SRO.60` and `SRO.120`), with a respective distance of 112% and 54% away from the ground truth.

Table 11: Results of dynamic resampling for the SRO experiment

Percentage (%)	**Time of arrival (s)**	**Avg Dist Opt (%)**	**Improv. Default (%)**	**Convergence**	**Duration (s)**
10%	**60**	54.0	62.0	100	4304.4
	120	112.4	47.6	100	3131.8
	300	28.2	68.4	100	1620.6
30%	**60**	309.3	-1.0	50	2240.6
	120	308.2	-0.7	32	1994.5
	300	205.8	24.6	58	2014.7

In deed, the algorithm resamples excessively, limiting exploration. On the opposite, lower-frequency noise prevents resampling until strong noise appears, leading to suboptimal performance. Dynamic resampling proves advantageous when noise occurs regularly, compared to static resampling, as it triggers resampling only periodically when noise affects optimization instead of systematically.

These experiments also confirm the necessity of setting a floor limit N per parametrization, as dynamic resampling can spend all of its budget resampling only a few parametrization whenever the noise arrives.

8.3. Using dynamic intervals and setting resampling bounds

The metrics associated with adding the resampling filter, as well as limiting the maximum number of runs, as described in section 6 are displayed in table 12.

Table 12: Metrics values for using a dynamic interval definition

Exp.	**Time of arrival (s)**	**Avg Dist Opt (%)**	**Improv. Default(%)**	**Convergence**	**Duration (s)**
SRO	**60**	8.9	73.1	100.0	2641.2
	120	5.1	74.1	100.0	1803.1
	300	2.4	74.7	100.0	1517.3
SBB	▨	7.4	4.3	160.0	28064.5

The first notable advantage of this approach is the removal of the need of selecting a particular hyperparameter for the algorithm, which is a difficult task as their values vary throughout the run leading to a variation of the resampling rate to adapt to the requirements of the noisy settings. Bounding the number of resampling iterations is also key to not wasting too many iterations on a single parametrization.
This change in behavior results in a strong improvement of the distance to the optimum compared to using standard dynamic resampling in the case of the SRO. Figure 5 provides examples of optimization trajectories: the raw and the aggregated per parametrization execution times are displayed, as well as the number of resamples per parametrization and the width of the interval per optimization iteration. These figures display the effect of bounding the number of iterations per parametrization, as we see that more different

parametrization are tested than in the unbounded case, while still resampling noise-tainted parametrization. In the case of the SBB, the improvement com-

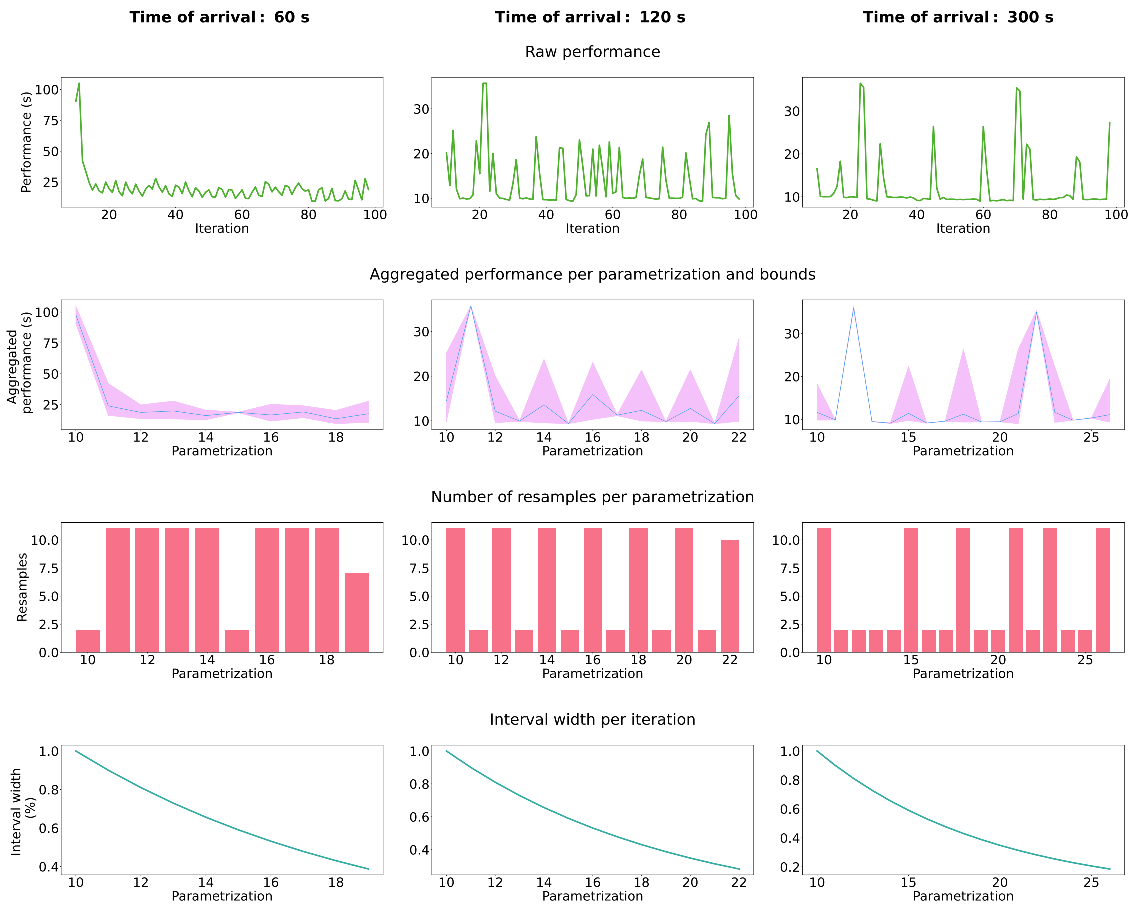


Figure 5: Example of optimization trajectory for the SRO experiment using dynamic confidence interval

pared to using unbounded dynamic resampling does not bring improvement compared to using a 30% resampling dynamic interval, as the distance to the optimum increases from 6.64% to 7.38%. However, the suggested method still removes the burdensome task of finding this hyperparameter. Similarly to what was observed on the SRO experiment, the optimizer spends the whole optimization budget. An example of optimization trajectory for the SBB experiment, containing the same information than in the case of the SRO experiment, is available in figure 6.

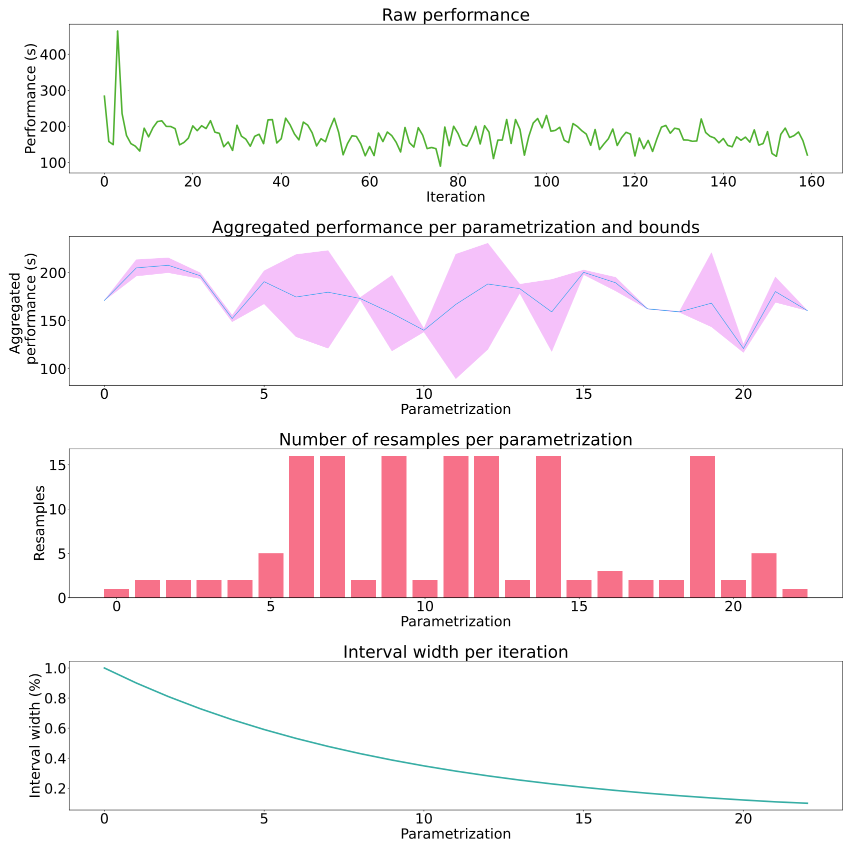


Figure 6: Example of optimization trajectory for the SBB experiment using dynamic confidence interval

8.4. Using the full EVADyR algorithm

The values of the metrics calculated when using the full EVADyR algorithm are available in table 13. In terms of distance to the ground truth,

Table 13: Metrics value when using all add-ons

Experiment	**Time of arrival (s)**	**Avg Dist Opt (%)**	**Improv. Default (%)**	**Convergence**	**Duration (s)**
SRO	**60**	5.8	73.9	100	2045.5
	120	4.3	74.3	81.1	1556.8
	300	2.6	74.7	75.9	1311.3
SBB		5.0	5.1	36.1	6565.9

adding a resampling filter allows to find results very close to the 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 and it is comparable to the case of SBB. For the cases of `SRO.120` and `SRO.300`, the distance from the grownd truth is almost negligible to users (5.3% and 2.6% respectively).

The most notable difference of adding a resampling filter to dynamic bounded intervals is the improvement in convergence speed, as the filter prevents the evaluation of parametrization that are not interesting in terms of performance. Indeed, when not using any filter, the stop criterion did not stop the experiment before the entire budget was spent, as reported in table 12, but using a filter makes the experiment faster in the case of `SRO.120`, with a 18.89% speed-up, `SRO.300`, with a 24.11% speed-up, and `SBB`, with a 77.45% speedup, in terms of number of iterations before the run stops. Even when the number of iterations is not reduced as in the case of `SRO.60`, this speed-up is also reflected on the total duration of the experiment which is reduced for every tested experiment. Indeed, we observe for `SRO.60` a time gain of 22%, for `SRO.120` a time gain of 16%, for `SRO.300` a time gain of 14% and for `SBB` a time gain of 76%. All the improvements obtained by using EVADyR in term of distance to the ground truth, in term of improvement over the default parametrization and in term of experiment duration are summarized in table 14. It shows clearly the superiority of EVADyR compared to the state of the art algorithms.

9. Conclusion

In conclusion, we presented in this paper a study of the impact of noisy interference on the performance of black-box optimization auto-tuner. By studying two types of different noise and the two different IO accelerators, we demonstrate that the noise cannot be neglected whenever performing black-box optimization on systems running in production, as it severely impacts the performance of the optimizer.

To increase the resilience of the auto-tuner to the noise observed when running in production, we introduced the concept of resampling and suggest three possible improvements to dynamic resampling and called this algorithm EVADyR (Efficient Value Aware Dynamic Resampling). Over four different experiments, we show that our solution improves the convergence of state-of-the-art dynamic resampling by 93.5% and 24.7% for respectively the SRO and the SBB accelerator, as well as shortens by 45.76% and 58.07% for these same accelerators. We also prove the importance of using noise reduction strategies whenever tuning systems running in production, as we find that using noise reduction strategies increases the found optimum by

Table 14: Comparison of our solution to the state of the art in terms of:

(a) Distance to the ground truth (%)

	SRO	SBB
No noise reduction	153.90	12.90
Static resampling	15.39	6.71
Dynamic resampling	64.86	6.64
EVADyR	4.21	5.00

(b) Improvement compared to the default parametrization (%)

	SRO	SBB
No noise reduction	37.37	-0.60
Static resampling	71.53	4.92
Dynamic resampling	59.33	4.98
EVADyR	73.88	5.05

(c) Experiment duration (s)

	SRO	SBB
No noise reduction	1035.33	5728.82
Static resampling	1807.55	28204.71
Dynamic resampling	3018.94	15657.69
EVADyR	1637.86	6565.91

respectively 97.46% and 61.24%.

A possible 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.

-–

References

[1] Z. Cao, V. Tarasov, H. P. Raman, D. Hildebrand, E. Zadok, On the performance variation in modern storage stacks, FAST'17, USENIX Association, 2017, p. 329–343.

[2] S. Robert, S. Zertal, G. Vaumourin, P. Couvée, A comparative study of black-box optimization heuristics for online tuning of high performance computing I/O accelerators, Concurrency and Computation: Practice and Experience (2021).

[3] Z. Cao, A practical , real-time auto-tuning framework for storage systems, Ph.D. thesis, State University of New York at Stony Brook (2018).

[4] A. Keane, A. Sóbester, Engineering design via surrogate modelling, Wiley Online Library, 2008, Ch. 1, pp. 1–31.

[5] J. Sacks, W. J. Welch, T. J. Mitchell, H. P. Wynn, Design and Analysis of Computer Experiments, in: Statistical Science, Vol. 4, 1989, pp. 409 – 423.

[6] A. Kassab, J.-M. Nicod, L. Philippe, V. Rehn-Sonigo, Assessing the use of genetic algorithms to schedule independent tasks under power constraints, in: 2018 International Conference on High Performance Computing Simulation (HPCS), 2018, pp. 252–259.

[7] S. Kirkpatrick, C. D. Gelatt, M. P. Vecchi, Optimization by simulated annealing (1983).

[8] B. Shahriari, K. Swersky, Z. Wang, R. P. Adams, N. D. Freitas, Taking the human out of the loop: A review of bayesian optimization (2015).

[9] H. Robbins, S. Monro, A Stochastic Approximation Method, The Annals of Mathematical Statistics 22 (3) (1951) 400 – 407. `doi:10.1214/aoms/1177729586`.
URL `https://doi.org/10.1214/aoms/1177729586`

[10] G. Liao, K. Datta, T. Willke, Gunther: Search-based auto-tuning of mapreduce, in: Proc. of International European conference On Parallel and distributed Computing (Euro-Par), Vol. 8097, 2013, pp. 406 – 419.

[11] P. Jamshidi, G. Casale, An uncertainty-aware approach to optimal configuration of stream processing systems (2016). `doi:10.48550/ARXIV.1606.06543`.
URL `https://arxiv.org/abs/1606.06543`

[12] S. Duan, V. Thummala, S. Babu, Tuning database configuration parameters with ituned, in: Proceedings of the VLDB Endowment, 2009, pp. 1246 – 1257.

[13] B. Xi, Z. Liu, M. Raghavachari, C. Xia, L. Zhang, A smart hill–climbing algorithm for application server configuration, in: Thirteenth International World Wide Web Conference Proceedings (WWW'2004), 2004, pp. 10 – 15.

[14] Y. Li, K. Chang, O. Bel, E. L. Miller, D. D. E. Long, 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, SC'17, 2017.

[15] Z. Cao, V. Tarasov, S. Tiwari, E. Zadok, 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, USENIX ATC '18, 2018, pp. 893–907.

[16] B. Behzad, H. V. T. Luu, J. Huchette, S. Byna, Prabhat, R. Aydt, Q. Koziol, M. Snir, Taming parallel I/O complexity with auto-tuning, in: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, SC '13, 2013, pp. 1–12.

[17] B. Behzad, S. Byna, M. Prabhat, M. Snir, Pattern-driven parallel I/Otuning, in: Proceedings of the 10th Parallel Data Storage Workshop, 2015, pp. 43 – 48.

[18] P. Balaprakash, J. Dongarra, T. Gamblin, M. Hall, J. K. Hollingsworth, B. Norris, R. Vuduc, Autotuning in high-performance computing applications, Proceedings of the IEEE 106 (11) (2018) 2068–2083.

[19] K. Seymour, H. You, J. Dongarra, A comparison of search heuristics for empirical code optimization, in: 2008 IEEE International Conference on Cluster Computing, 2008, pp. 421–429.

[20] P. Knijnenburg, T. Kisuki, M. O'Boyle, Combined selection of tile sizes and unroll factors using iterative compilation, The Journal of Supercomputing 24 (2003) 43–67.

[21] H. Menon, A. Bhatele, T. Gamblin, Auto-tuning parameter choices in HPC applications using bayesian optimization, in: IEEE International Parallel and Distributed Processing Symposium (IPDPS), 2020, pp. 831 – 840.

[22] T. Miyazaki, I. Sato, N. Shimizu, Bayesian optimization of HPC systems for energy efficiency, in: Proceedings of International conference on High Performance Computing, 2018, pp. 44–62.

[23] Green_list, `https://www.top500.org/lists/green500/2023/06`.

[24] A. Faraj, X. Yuan, Automatic generation and tuning of mpi collective communication routines, in: Proceedings of the 19th Annual International Conference on Supercomputing, 2005, pp. 393 – 402.

[25] S. Vadhiyar, G. Fagg, J. Dongarra, Automatically tuned collective communications, in: Proceedings of the 2000 ACM/IEEE Conference on Supercomputing, 2000.

[26] M. Stoffel, F. Broquedis, F. Desprez, A. Mazouz, Phase-ta: Periodicity detection and characterization for HPC applications, in: International Conference on High Performance Computing Simulation (HPCS), 2021.

[27] O. Mondragon, P. Bridges, K. Ferreira, S. Levy, P. Widener, Understanding performance interference in next-generation HPC systems, in: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, 2016.

[28] G. Ozer, A. Netti, D. Tafani, M. Schulz, Characterizing HPC performance variation with monitoring and unsupervised learning, in: Pro-

ceedings of the international conference on High Performance Computing, 2020, pp. 280–292.

[29] T. Schmied, D. Didona, A. C. Döring, T. P. Parnell, N. Ioannou, Towards a general framework for ml-based self-tuning databases, CoRR (2020). `arXiv:2011.07921`.
URL `https://arxiv.org/abs/2011.07921`

[30] Y. Gur, D. Yang, F. Stalschus, B. Reinwald, Adaptive multi-model reinforcement learning for online database tuning, in: International Conference on Extending Database Technology, 2021.

[31] J. Grohmann, D. Seybold, S. Eismann, M. Leznik, S. Kounev, J. Domaschka, 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), 2020, pp. 1–8.

[32] E. Vázquez, J. Villemonteix, M. Sidorkiewicz, E. Walter, Global optimization based on noisy evaluations: An empirical study of two statistical approaches, in: Journal of Physics Conference Series, Vol. 135, 2008.

[33] B. Letham, B. Karrer, G. Ottoni, E. Bakshy, Constrained bayesian optimization with noisy experiments (2017). `doi:10.48550/ARXIV.1706.07094`.
URL `https://arxiv.org/abs/1706.07094`

[34] D. Huang, T. Allen, W. Notz, R. A. Miller, Sequential kriging optimization using multiple-fidelity evaluations, in: Structural and Multidisciplinary Optimization, Vol. 32, 2006, pp. 369–382.

[35] V. Picheny, D. Ginsbourger, Y. Richet, G. Caplin, Quantile-based optimization of noisy computer experiments with tunable precision, in: Technometrics, Vol. 55, 2012.

[36] E. I. J. Forrester, A. J. Keane, N. W. Bressloff, Design and analysis of 'noisy' computer experiments, AIAA Journal 2331–2339.

[37] V. Rodehorst, O. Hellwich, Genetic algorithm sample consensus (gasac) - a parallel strategy for robust parameter estimation, in: 2006 Conference on Computer Vision and Pattern Recognition Workshop (CVPRW'06), 2006, pp. 103–103.

[38] V. Picheny, T. Wagner, D. Ginsbourger, A benchmark of kriging-based infill criteria for noisy optimization, in: Structural and Multidisciplinary Optimization, Vol. 48, 2013, pp. 607–626.

[39] H. Jalali, I. Van Nieuwenhuyse, V. Picheny, Comparison of kriging-based algorithms for simulation optimization with heterogeneous noise, in: European Journal of Operational Research, Vol. 261, 2017, pp. 279–301.

[40] D. H. Wolpert, W. G. Macready, No free lunch theorems for optimization, IEEE Transactions on Evolutionary Computation 1 (1) (1997) 67–82.

[41] Yaochu Jin, J. Branke, Evolutionary optimization in uncertain

environments-a survey, in: IEEE Transactions on Evolutionary Computation, Vol. 9, 2005, pp. 303–317.

[42] H. Kita, Y. Sano, Genetic algorithms for optimization of noisy fitness functions and adaptation to changing environments, in: Joint Workshop of Hayashibara Foundation and Workshop on Statistical Mechanical Approach to Probabilistic Information Processing (SMAPIP), 2003.

[43] Y. Sano, H. Kita, Optimization of noisy fitness functions by means of genetic algorithms using history of search with test of estimation, in: Proceedings of the 2002 Congress on Evolutionary Computation (CEC'02), Vol. 1, 2002, pp. 360–365.

[44] J. Branke, C. Schmidt, H. Schmec, Efficient fitness estimation in noisy environments, in: Proceedings of Genetic and Evolutionary Computation, 2001, pp. 243–250.

[45] A. Ratle, Accelerating the convergence of evolutionary algorithms by fitness landscape approximation., in: Proceedings of the 5th International Conference on Parallel Problem Solving from Nature, 1998.

[46] I. Paenke, J. Branke, Y. Jin, Efficient search for robust solutions by means of evolutionary algorithms and fitness approximation, in: IEEE Transactions on Evolutionary Computation, Vol. 10, 2006, pp. 405–420.

[47] A. N. Aizawa, B. W. Wah, Scheduling of genetic algorithms in a noisy environment, Evolutionary Computation 2 (2) (1994) 97–122. `doi:10.1162/evco.1994.2.2.97`.

[48] J. M. Fitzpatrick, J. J. Grefenstette, Genetic algorithms in noisy environments, Machine Learning 3 (2) (1988) 101–120.

[49] P. Stagge, Averaging efficiently in the presence of noise, in: Proceedings of the 5th Parallel Problem Solving from Nature, 1998, pp. 188–197.

[50] S. Robert, S. Zertal, G. Vaumourin, 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.

[51] F. Siegmund, A. Ng, K. Deb, A comparative study of dynamic resampling strategies for guided evolutionary multi-objective optimization, in: IEEE Congress on Evolutionary Computation, 2013, pp. 1826–1835.

[52] A. Di Pietro, L. While, L. Barone, Applying evolutionary algorithms to problems with noisy, time-consuming fitness functions, in: Proceedings of the 2004 Congress on Evolutionary Computation, Vol. 2, 2004, pp. 1254–1261.

[53] A. Syberfeldt, A. Ng, R. I. John, P. Moore, Evolutionary optimisation of noisy multi-objective problems using confidence-based dynamic resampling, in: European Journal of Operational Research, Vol. 204, 2010, pp. 533–544.

[54] H. Anahideh, J. Rosenberger, V. Chen, High-dimensional black-box optimization under uncertainty, in: Computers Operations Research, 2021.

[55] Hong, Nelson, An indifference-zone selection procedure with minimum switching and sequential sampling, in: Proceedings of the 2003 Winter Simulation Conference, 2003., Vol. 1, 2003, pp. 474–480.

[56] The SHAMan application. https://github.com/bds-ailab/shaman.

[57] F. Hutter, H. H. Hoos, K. Leyton-Brown, Sequential model-based optimization for general algorithm configuration, in: International conference On Learning and Intelligent Optimization, 2011, pp. 507–523.

[58] D. A. Reed, J. Dongarra, Exascale computing and big data, Communications of the ACM 58 (2015) 56–68.

[59] F. Cappello, A. Geist, W. Gropp, S. Kale, B. Kramer, M. Snir, Toward exascale resilience: 2014 update, Supercomputing Frontiers and Innovations 1 (1) (2014).

[60] Atos, Tools to improve your efficiency, `https://atos.net/wp-content/uploads/2018/07/CT_J1103_180616_RY_F_TOOLSTOIMPR_WEB.pdf`.

[61] Atos steps up with nvme of flash accelerator solutions, `https://atos.net/wp-content/uploads/2018/07/CT_J1103_180616_RY_F_TOOLSTOIMPR_WEB.pdf`.

[62] Lustre filesystem, `http://lustre.org/`.

[63] Ior benchmark description, `http://wiki.lustre.org/IOR`.