Using genetic algorithms for noisy systems' auto-tuning: an application to the case of burst buffers

Sophie ROBERT*†, Soraya ZERTAL†, Gregory VAUMOURIN*
*Atos BDS R&D Data Management
†Li-PaRAD, University of Versailles
Email: sophie.robert@atos.net, soraya.zertal@uvsq.fr, gregory.vaumourin@atos.net

***Abstract*—In this paper, we propose a methodology for finding the optimum parameters of complex systems whose performance measure can be affected by noise. We build this method on genetic algorithms and test several different possible noise reduction strategies to make them more resilient to interference. We define several benchmark metrics to provide a comprehensive comparison of the strategies and rank them according to their efficiency in a production setting. We validate this methodology by tuning a commercial implementation of a burst buffer, an intermediate high-bandwidth Input/Output (I/O) accelerator positioned between the compute nodes and the storage system. Auto-tuning in this context must deal with the challenges of the highly dynamic and noisy context of High Performance Computing (HPC) clusters' storage and the strong dependence between the running application and the burst buffer optimal configuration. We show that adding a noise reduction strategy is critical to the optimization process, as it improves the convergence speed of conventional genetic algorithms by 23% and their success rate in finding the optimal parametrization by 78.9%. Our methodology is able to improve the execution time of the IOR benchmark by 10.5% compared to the default burst buffer parametrization, in less than 21 iterations.**

***Index Terms*—Input/Output; Burst buffer; Auto-tuning; Optimization; Genetic algorithms; Noise resiliency;**

I. Introduction

Most modern software and hardware components come with a wide range of parameters that have a high impact on their behavior and the system's performance. Finding the optimal parametrization for each application on a system is key to make the most of its resources. This problematic is particularly important in the field of High Performance Computing (HPC) which handles a wide range of applications with heavy computation and Input/Output (I/O) workloads. To tackle this optimization challenge, the simplest solution is to set a default value that would be used for every run on the system but this can cause suboptimal performance. Another solution could be to perform an exhaustive sampling of the parametric space by testing for each parametrization its impact on the performance function of the application but this is impractical as it is too time consuming. An alternative could be to design a theoretical explicit model to describe the software or hardware to be tuned, but the complexity of the relationship between each component and the lack of insight on the system's behavior make this model complicated to find. To avoid the complex task of tuning the system manually or through theoretical insights, we proposed in [1] a novel approach for system auto-tuning by adding an on-line optimization step to the application launching process. Each time that an application already known by the system is submitted, the most appropriate parametrization is selected by analyzing the application previous history. This creates a feedback loop which iteratively adjusts the parameters of the application using a black-box optimization heuristic, as shown in figure 1. We apply this approach to find the optimal configuration of I/O accelerators, and more precisely to burst buffers for HPC systems.

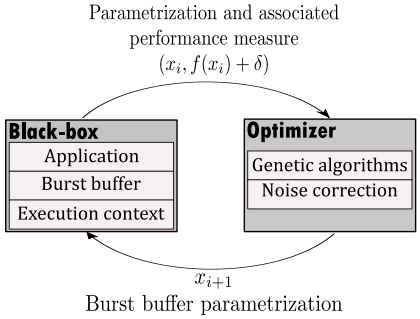

Fig. 1: Schematic view of the auto-tuning loop

Because of their highly dynamic nature and the complexity of applications and software stacks, HPC systems are subjected to many interference when running in production, which results in a different performance measure for each run even with the same system's parametrization. If this variability is not taken into account, the efficiency of black-box heuristics can be significantly reduced because of a slow down of the optimization process.

In this work, we suggest a generalist approach for auto-tuning of configurable hardware and software components that can be subjected to interference, like I/O accelerators, power savings strategies [2] or storage system [3]. We build this method on genetic algorithms whose self-averaging nature makes them good candidates for finding the optimum of a noisy function, and provide a comprehensive comparison of several noise reduction strategies found in the literature to improve their resilience to noise. The effectiveness of our methodology and its implementation for tuning a burst buffer running in production is evaluated and discussed.

The main contribution of this paper is the proposition of an optimization engine that:

- Does not require any assumption or previous knowledge on the system to optimize
- Automatically sets the parametrization of the burst buffer when a job is submitted using the job's history of runs

- Applies noise reduction strategies to deal with noisy performance measures
- Has been validated on a production HPC cluster, using Atos' burst buffer [4] and the IOR [5] benchmark.

In the remainder of the paper, section II presents relevant related works for auto-tuning of computer systems and selection of optimal burst buffer parametrization. Section III describes our motivation and the several challenges that come with tuning a burst buffer. Section IV formally introduces the addressed problem and defines the notation we use in the paper. Section V presents genetic algorithms to perform the auto-tuning and section VI describes noise correction strategies. The different metrics used to compare these strategies are available in section VII and the description of the experiment to evaluate them comes in section VIII. Results are presented and discussed in section IX and our contribution is summarized with some hints for future works in section X.

II. Related works

Auto-tuning using black-box optimization has been successfully used in several domains in the last years, and has been particularly helpful in computer science for finding optimal configurations of various software and hardware systems. Within the HPC community, it has been used to improve the portability of applications across architectures using random-based heuristic searches in [6]. Surrogate modeling has also successfully been used to reduce clusters' energy consumption in [2]. Genetic algorithms in particular have proven successful for tuning the parameters of particular benchmarks [7]. The I/O community has also successfully tuned highly configurable storage systems. In [3] and [8], the authors present a comparative study of four optimization heuristics (surrogate models, genetic algorithms, simulated annealing, and reinforcement learning) for tuning the storage systems for particular workloads. Reinforcement learning has also proven to be efficient as an auto-tuner to optimize the performance of the Lustre filesystem in data center storage systems [9]. Genetic algorithms in particular have proven to be an efficient choice of heuristic for tuning different parameters of the I/O stack of different HPC platforms [10], for finding the optimal parametrization of HDF5 applications [11] and to find at run-time the best I/O configuration of a system given an I/O pattern [12]. However, to our knowledge, none of these authors address the problem of potential noise when tuning configurable systems and do not offer any insight on the stability of the performance measured while performing the tuning. Consequently, none of these papers offer methods to improve the noise-resilience of black-box optimization heuristics and their efficiency when used for shared systems like HPC platforms.

When it comes to finding the optimal parametrization of burst buffers specifically, [13] has suggested to use a Markov chain based model to describe the behavior of the burst buffer. In [14], a polynomial time algorithms for the special case of finding the optimal buffer size, both when considering static and dynamic resource allocation was proposed. While very successful for describing a particular conception of a burst buffer architecture, this model lacks flexibility and does not take into account the parameters specific to each burst buffer's commercial implementation.

III. Motivation

In this paper, we adapt genetic algorithms to deal with noisy performance measures and test them by auto-tuning burst buffers, new storage tiers placed between the compute nodes and the parallel file system that serves as a high bandwidth temporary store. They have emerged as a promising solution to cope with limited I/O bandwidth and improve the system's performance [15] [16] [17], especially for data intensive applications [18].

In this work, we use the Atos commercial implementation of a burst buffer [4], called the *Smart Burst Buffer*, which works as a high bandwidth temporary RAM cache allocated on a specific node called the Data node. This particular burst buffer implementation comes with a dozen of different parameters. The four ones that have the strongest influence on execution time, according to an ANalysis Of VAriance (ANOVA) (p-value < 0.05) [19], are described in Table I and their signification for the burst buffer is described in figure 2. A strong difficulty when looking for the optimal value for these parameters is their dependence on the many characteristics of the application. It is the case of the cache threshold, which is strongly dependent on the volume of I/O performed by the application. Indeed, applications whose data can fit and be re-used in the burst buffer's cache benefit from as little reclaim as possible and applications that are very write intensive require to empty the burst buffer as often and as fast as possible.

The other main difficulty of burst buffer tuning is the variability of I/Os execution time [20], because of shared resources contentions (Lustre caches, RDMA network, *etc.*) and optimizations (prefetching, collective I/O). Not taking into account these possible interference greatly decreases the efficiency of black-box heuristics.

To emphasize on the different challenges of burst buffer tuning in a production environment, the execution times of an IOR benchmark scenario that writes sequentially 174GB through the burst buffer is illustrated in figure 3. These runs are made on a production HPC cluster in use. Each configuration is run 16 times to illustrate the effect of noise versus the parametrization, and we observe strong interference within identical parametrization (coefficient of variation of 17%). The strong impact of the parameters on the execution time is also illustrated by this example: even if the cache size is set to the maximum, there is an execution time variation of 41% between the worse and the best parametrization. The strong interaction between each parameter also increases the task difficulty: as the worker and destager threads run on the same set of Data node cores, they compete for CPU time and cause destaging contentions.

Tuning our selected system presents several challenges for the optimizer, which must deal with a very noisy performance

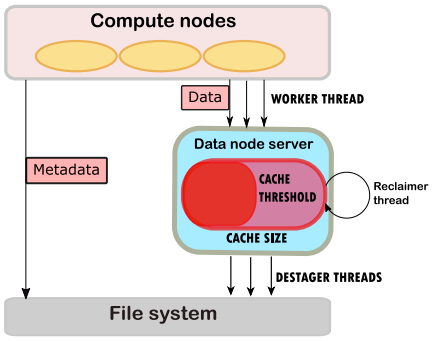

Fig. 2: The Atos burst buffer architecture

TABLE I: The application's specific burst buffer parameters

Name of parameter	Description
Number of workers	Size of the worker threads pool used to transfer the data from the compute nodes to the Data node
Number of destagers	Size of the destager threads pool used to transfer the data from the Data node to the parallel filesystem
Cache size	The size of the allocated RAM cache on the Data node
Cache threshold	The fraction of used RAM cache before we start removing clean data from the burst buffer's cache in order to make room for the other incoming I/O.

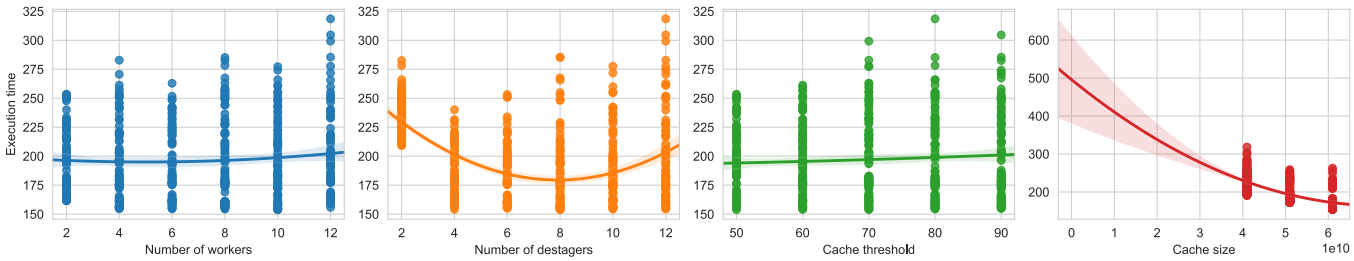

Fig. 3: Impact of the burst buffer parameters on the IOR benchmark execution time.

function, a non-homogeneous noise landscape over the parametric space, a four-dimensional space and strong interactions between parameters.

IV. Problem formulation and notations

We are looking for the optimum of a performance function that can only be accessed through its observations that are tainted with noise. Formally, the optimization component must solve the following problem:

$$min\ \mathbb{E}(F(x)),\ x \in \mathcal{P}$$
$$F(x) = f(x) + \epsilon(x)$$

with $f(x)$ representing the "true" execution time for parametrization x, while we only have access to the "observed" or "sampled" execution times $F(x)$. The noise, which can depend on the parametrization, is a function of x and is represented by $\epsilon(x)$. We treat the execution time as a random variable whose realization are the elapsed times for each run.

The vector corresponding to the different sampled value at parameter x will be denoted as $F(x)^j_{1\leq j\leq j_x}$, j_x being the number of samples for parameter x. The estimation of f at x is denoted $\hat{f}(x)$. $\mathcal{P}$ represents the entire possible set of parametrizations for the burst buffer. Its size will be denoted as n, so that all the possible parametrization of the accelerator can be described by the vector $(x_i)_{1\leq i\leq n}$.

From this definition, we find that in our case 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 (for example because the cluster was idle) and actually due to the burst buffer parametrization. Throughout the remainder of this paper, the term "optimum" will refer to the optimal parametrization on average.

V. Optimizing with genetic algorithms

Genetic algorithms are evolutionary algorithm that mimic natural selection. They consist in selecting two parametrizations among the set of already tested ones, according to a process based on the execution time of each parametrization. These two parameters are then combined to create a new one which can undergo a mutation and provide a new parametrization. The selection step exploits the parametric space and the performance function, while the mutation step brings some random exploration into the optimization process.

A. Initialization step

It is fundamental for any black-box optimization to select the parameters to launch the initialization starting plan that must respect the space's constraint and the non-collapsible property [21]. Such a design is called a *Latin Hypercube Design*. The non-collapse property specifies that no parametrization can have the same value on any dimension . This is especially important as we have no insight on the individual effect of each parameter.

B. Selection step

The goal is to select the two fittest parametrizations that will be combined to generate a new combination. Many different methods exist [22] and we focus on two popular ones: the probabilistic and the tournament pick. The former consists in picking out the best parametrization whilst the second consists in randomly drawing two pools of parametrization of a given size and selecting the candidates with the best execution time in each pool.

C. Crossover step

It consists in merging two parametrizations to create a new one as inspired by biology. Single-point crossover randomly splits each parent parametrization in two and concatenates the two parts. The *n*-points crossover cuts the parents into n parts and alternatively concatenates their split parts. We experimented with both.

D. Mutation step

It is a random event that can happen to the newly created parametrization that modifies some of its value. This enforces the exploration property of the genetic algorithm. There are an infinite number of ways to mutate a parametrization. In our case, mutation is implemented as a random walk.

VI. Noise correction methodology

This section presents different noise reduction strategies we suggest to make our auto-tuner resilient to the interference HPC systems are subjected to. While the self-averaging nature of genetic algorithms makes them very successful at finding the optimum of a noisy function [23] [24], the noise present in the data slows down the convergence process which requires more evaluations to reach the optimum. There exists in the literature two categories of methods to enhance genetic algorithms' effectiveness at finding the optimum in the presence of noise: resampling policies [25] and fitness regression [26] [27] [28]. Their goal is to give a more precise estimation of the true value $f(x)$ at parametrization x so that the algorithm can perform its selection process with better knowledge of the parametric space. Throughout the following section, the selection process of the algorithm uses the estimated value $\hat{f}(x)$ instead of the raw sampled values.

A. Resampling policies

Resampling consists in reevaluating several times the same parametrization in order to get a more precise idea of its impact on the real execution time. Here, $f(x)$ is estimated by the mean estimator $\bar{F}(x)$, which preciseness improves as the sample size for parametrization x increases. The genetic algorithm thus performs its selection process using the mean value for each parametrization.

$$\hat{f}(x) = \bar{F}(x) = \frac{1}{j_x} \sum_{j=1}^{j_x} F(x)^j$$

The measure of the dispersion around the estimator can be estimated by the standard error of the mean at parametrization x, equals to $\frac{\sigma(x)}{\sqrt{n}}$ (n being the number of evaluations at this parametrization). As $\sigma(x)$ is in practice unknown, it is estimated by the unbiased standard deviation estimator $\hat{\sigma}(x)$.

$$\hat{\sigma}(x) = \sqrt{\frac{1}{j_x - 1} \sum_{j=1}^{j_x} (F(x)^j - \bar{F}(x))^2}$$

Several strategies exist in order to efficiently reevaluate a parametrization. We focus on two of them because of their simplicity and their effectiveness.

a) Simple averaging

Simple averaging computes several times the fitness value of the selected parametrization [23], regardless of the parametrization and its associated fitness. In our case, it consists in launching a fixed number of times the application and the burst buffer with the same selected parametrization.

b) Standard Error Dynamic Resampling

Standard Error Dynamic Resampling (SEDR), as introduced in [29], adapts the number of samples to the noise strength measured at each parametrization. The parametrization is reevaluated until the standard error of the mean is below a fixed threshold th_{se}. While simple averaging reevaluates each parametrization n times, SEDR introduces a variable number of evaluations n_x for each parametrization.

c) Setting the hyperparameters

Each of the two methods described above requires a parameter: the number of repetitions. We propose an original method to set this parameter, by estimating the population mean $\hat{\mu}$ and the standard deviation $\hat{\sigma}$ on the initial Latin Hypercube Design and setting the standard error of the mean threshold to a certain percentage p of the estimated mean $th_{se} = p \times \hat{\mu}$. In the case of simple averaging, as $se = \frac{\sigma}{\sqrt{n}}$, we set $n = \left\lfloor \frac{\hat{\sigma}^2}{p^2 \times \hat{\mu}^2} \right\rfloor$. The initial sampling using Latin Hypercube Design ensures that the initial parametrizations are representative of the true population mean and variance.

B. Fitness regression

Fitness regression uses parameter combinations that have already been evaluated to estimate the execution time at each parametrization using regression techniques [27]. Either all the previous history or only neighboring parameters can be used for regression. One of the advantages of using only neighboring parametrizations is that any assumptions made by the regressor on the execution function only has to hold locally. If $\mathcal{N}_x$ is a neighborhood of x and $g_{\mathcal{N}_x}$ a regressor trained on this neighborhood, $f(x)$ can be estimated by:

$$\hat{f}(x) = g_{\mathcal{N}_x}(x)$$

While any regression technique can be used, we chose to focus on two types of regressors: weighted average and quadratic regression. This choice is motivated by the increase of convergence speed in a noisy context as proven in [30].

For each model, the neighborhood of parametrization x defined as the $\lfloor p \times n \rfloor$ closest to x according to the Euclidean distance, p being the percentage of the population size that should be used and n the total population size. This neighborhood definition is referred to as *dynamic sampling* in [30].

a) Weighted average

A weighted average is a type of average where each observation in the data set is multiplied by a predetermined weight before calculation.

$$\hat{f}(x) = \sum_{y \in \mathcal{N}_x} \frac{\bar{F}(y) \times w(y)}{\sum_{y \in \mathcal{N}_x} w(y)}$$

In our case, weights reflect the distance between the parametrization x and the other previously estimated parametrizations: the closer the data point is to x, the higher is its weight and thus its importance in the estimation of x. While any weight function with this property could be used, we used the weight function suggested in [30]:

$$w(y) = \begin{cases} (1 - (\frac{d(x,y)}{max_{y \in \mathcal{N}_x} d(x,y)})^3)^3 \text{ if } y \in \mathcal{N}_x \\ 0 \end{cases}$$

b) Quadratic regression

As described in [30], quadratic regression consists in fitting a second degree polynomial on the parameters to predict the corresponding execution time. The coefficients of the regressors are estimated using least square optimization.

VII. Algorithms' evaluation

The goal of conventional optimization is to find the optimum of a function. Generally, the suitability of the algorithm is evaluated by comparing the distance between the solution and the true optimum and the number of iterations required to reach it.

However, these metrics are not sufficient to reflect the constraints due to on-line learning, because the user is aware of the duration of each run. To be used in production, the burst buffer's optimizer should be kept as transparent as possible to the end-user. To compare and rank the methods' behavior presented in section VI, we propose 7 metrics that reflect both the constraints of classical noisy optimization and on-line learning. They are described in table II and structured in three categories. The first one gathers the metrics that are common to every optimization problem. The second category gathers the metrics for measuring the variability in the optimization trajectory. The third and last one describes the computation cost of the algorithm.

TABLE II: Description of the comparison metrics

Metric name	**Abbr.**	**Description**
Number of success (%)	SuccessRate	The number of times the optimal parametrization has been correctly found, as percentage of the total number of trials
Distance to the optimum (%)	DistOptim	The distance between the optimum found and the optimum in percentage
Number of steps to reach optimum	NbrSteps	The number of steps to reach the optimum
Average distance to the optimum (%)	AvgDistOptim	The distance to the optimum in percent averaged at each round
Regression cost (sec)	RegCost	The average regressions per iteration
Percentage of execution times below the execution time corresponding to the default (%)	InferiorDefault	The percentage of iterations that yield a time inferior to the one corresponding to the default parametrization
Elapsed time (sec)	ElTime	The elapsed time to output the trajectory

In order to provide a single evaluation scale for the algorithms, we define three scoring functions to rank each algorithm. The first score, $\text{score}_{\text{optim}}$ measures the quality of the optimization, by quantifying the convergence speed The second score, $\text{score}_{\text{stability}}$ quantifies the variability of the trajectory. The third score is the sum of the two. Each metric is normalized between 0 and 1 to account for different magnitudes. The higher the score, the better the algorithm behaves according to our criteria. Each metric is given the same weight, but these can easily be modified by adding weights to give different importance to the heuristics' qualities.

$$\text{score}_{\text{optim}} = \text{SuccessRate} - \text{NbrSteps} - \text{DistOptim}$$
$$\text{score}_{\text{stability}} = -\text{AvgDistOptim} + \text{InferiorDefault} - \text{RegCost}$$
$$\text{score} = \text{score}_{\text{optim}} + \text{score}_{\text{stability}}$$

VIII. Validation of the methodology

A. Loop simulation

To compute the metrics on realistic trajectories, the autotuning loop must be simulated offline to confront the heuristics to real-life conditions. To do so, we use the dataset of IOR runs described in section III which contains the execution times that correspond to runs of an application with several parametrizations of the burst buffer. This dataset is necessary in order to find the optimal parametrization of the accelerator.

This known optimum acts as the ground truth and allows us to compare the results given by our methodology to a reference. Without this knowledge, the quality of the optimization cannot be quantified.

The dataset is then used to replace the black-box (*i.e.* the combination between the burst buffer, the application, and the execution topology) in the real-life loop. 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.

B. Experiment design

1) Test application

The application used for evaluation is the standard I/O benchmark IOR, commonly used for evaluating the performance of parallel file systems [5]. We choose 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.

2) Test parameters

To build the exhaustive test dataset described in section VIII-A, different values for the four parameters described in section III are tested. This results in a total of 540 tested parametrizations.

3) Experiment execution

Each parametrization is sampled 16 times (*i.e.* $m = 16$) to evaluate the effect of noise on the system. This number was chosen as a trade-off between statistical significance and

TABLE III: 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	$\{50\%, 60\%, 70\%, 90\%\}$

realistic experimental time constraints. According to the notations introduced in section IV, this corresponds to collecting m runtimes $F(x_i)_{1\leq j\leq m}^{j}$ for each of the n parametrizations $(x_i)_{1\leq i\leq n}$ of the burst buffer. In total, the collection of this exhaustive test dataset required 474 hours.

4) Hardware specifications

The IOR benchmark uses three compute nodes and one Data node (where the burst buffer's cache is allocated). Each node consists of an Intel(R) Xeon(R) CPU E5-2670 with 16 physical cores (32 logical cores), bi-socket, and 62 GB of DDR4-DRAM. The compute nodes are connected to the Data node through FDR RDMA (56Gb/s) network. The back-end parallel filesystem is a Lustre bay[31] of 40TB. The storage bay is shared with other users to allow interference and I/O variability.

C. Description of the dataset

1) Main characteristics

The main statistical estimators and the statistical distribution of the raw collected times and the elapsed times averaged for each parametrization are shown in table IV.

TABLE IV: Statistics on raw and averaged elapsed times within each parametrization

	Raw elapsed time	Averaged elapsed time
Size	8640	540
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

2) Noise characterization

From table IV, 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 genetic algorithms. The distribution of this noise is not homogeneous within each parametrization as determined using Levene's and Bartlett's variance homogeneity test ($p - value < 10^{-5}$ in both cases) [32] [33]. These two tests assess the null hypothesis that the variance is homogeneous within each group. This validates the assumption that the noise is not constant over the parametric space and is thus a function of x. An ANOVA [19] which determines if the mean of the noise is constant across each parametrization, shows that some parametrizations are more sensitive to interference than others (p-value < 0.05) .

On average, one outlier run within each parametrization can be detected using Tukey's fences [34], but the distribution of the outliers are even across each parametrization (ANOVA, p-value < 0.05): no parametrization is more susceptible to have outlier runs more than another. These outliers can thus be attributed to the cluster and not to the burst buffer.

3) Optimal configuration

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% away from the optimal configuration.

The optimal configuration in terms of cache size is, as can be 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 is counter-productive, as the optimum is set to 8, and emptying the data node once its 50% full yields the best results.

D. Tested methods

We conducted tests for the whole 200 possible combinations of optimization methods and regression techniques described in section VI. Resampling methods require as hyperparameter the percentage of the mean of the initial Latin Hypercube sample to use as bias threshold. Fitness regressions require the percentage of neighbors to use for regressing the fitness function. The mutation rate is set to 0.2 for all heuristics. The maximum number of iterations is set to 50, the first 15 being reserved for initialization. Each optimization process is repeated 30 times to average any random behavior. The different hyperparameters and abbreviations are available in table V. The score associated with a random sampler (abbreviated as **RS**) is given for comparison.

TABLE V: Summary of tested heuristics

Heuristic class	Abbr.	Nbr	Hyperparameters
Conventional GA	CGA	4	
Resampled GA	RGA	24	$p_{resamp} \in 10, 20, 40$
Fitness regression GA	FGA	28	$p_{fit} \in 5, 10, 20, 50$
Resampled + fitness reg. GA	FRGA	44	$p_{resamp} \in 10, 20, 40$ $p_{fit} \in 5, 10, 20, 50$

IX. Results and discussion

We begin this section by highlighting how essential taking into account the noise is when tuning the burst buffer. We then provide a comprehensive comparison of the different noise reduction strategies we proposed in section VI. We conclude by focusing on the advantages of using our tuner compared to the default parametrization.

A. Importance of noise reduction strategy

The goal of the different noise reduction strategies is to improve the score in the presence of noise and figure 4 shows that each one of them improves it over conventional genetic algorithm. As seen in figure 5, conventional genetic algorithm bring no improvement in median execution time compared to using the default parametrization. However, using a noise reduction strategy greatly reduces the median execution time (figure 5) and improves the convergence and stability scores (figure 4). Taking into account the noise is thus critical to

TABLE VI: Best heuristic in terms of global score for each category and the corresponding metrics

	CGA	**RGA**	**FGA**	**FRGA**	**RS**
Aggregation			Quadra ($p = 5\%$)	Quadra ($p = 5\%$)	
Resampling		SEDR ($p = 10\%$)		SEDR ($p = 40\%$)	
SuccessRate	13.3	63.3	50.	56.6	2
DistOptim	0.55	0.20	0.44	0.23	0.77
NbrSteps	29.5	22.7	24.1	20.83	34.3
AvgDistOptim	18.1	14.0	14.71	12.28	27.9
RegCost	13.6	8.2	9.81	7.02	20.2
InferiorDefault	48	64	64	70	27.8
ElTime	222.6	265.2	1084.1	836.8	146.5
ScoreOptim	**0.65**	**0.98**	**0.68**	**0.89**	0
ScoreStability	**0.54**	**0.84**	**1**	**0.89**	0
Score	**0.54**	**1**	**0.7**	**0.96**	0

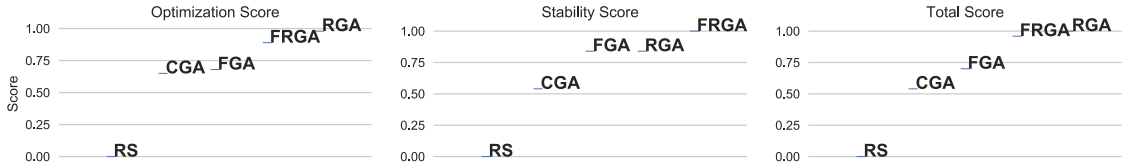

Fig. 4: Ranking of noise reduction techniques according to their best score in each category

perform an efficient optimization of the burst buffer, as regular optimization is no better than using its default parametrization.

B. Comparison of the different heuristics

The different noise reduction strategies described in section VI can be compared:

1) In terms of convergence score

For the convergence speed, resampling using SEDR is the most efficient technique, with a bias threshold of 10% around the mean of the initialization plan. Increasing the bias (and thus reducing the number of iterations) lowers score values. The main impact of resampling is the increase in the precision of the algorithm and especially the number of times the correct optimum is found, with a 78.9% increase in success rate compared to conventional genetic algorithms. Because of a more thorough exploitation of each data point, the number of steps required to find the optimal parametrization is reduced by 23%. While adding a fitness aggregation component improves the convergence quality compared to conventional genetic algorithms, it is still less efficient than adding a resampling strategy. We find that modeling the area around the target fitness as a quadratic surface yields better results than using a weighted model, and the smaller the neighborhood, the better the results. This means that modeling a larger zone as a quadratic surface is an oversimplification of the optimization problem. Combining resampling and fitness aggregation yields a score improvement compared to conventional genetic algorithms and single fitness regression but not as good as using only resampling, in terms of precision of the optimum found. However, using the combination of the two methods lowers the number of iterations required for convergence by 29%. Quadratic regression over a small neighborhood performs best and the combination of both methods reduces the number of repetitions when resampling (the bias threshold goes up to 40% for SEDR). The main benefits of noise reduction strategies are the lowering of the convergence speed (especially with FRGA) and the increase in the success rate (especially with RGA) of the algorithm. Selecting the best algorithm depends on the importance given to each metric, and when giving the same importance to both, RGA is the best choice.

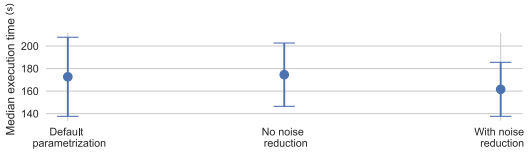

Fig. 5: Comparison of median execution time when using default parametrization, conventional genetic algorithms and genetic algorithms with noise reduction strategy (RGA)

2) In terms of stability

Every noise reduction strategy improves the stability of the optimization process compared to conventional genetic algorithms, by reducing the regression cost and the average distance to the optimum. The most stable behavior is offered by the combination of fitness regression and resampling strategies. With a regression cost of 7 seconds on average (3.55% compared to the mean execution time), the perception of the regression cost by the user is negligible. Using either of these components separately gives a similar, lower score.

C. Increase of burst buffer performance

When combining the scores, resampling strategies give the best results, as can be seen in figure 4. When using this heuristic, we propose an optimization engine that finds in 21 iterations a value performing 10.5% better than the default parametrization.

The heuristic provides in median over each iteration an improvement of 5.18% compared to the default parametrization (figure 5): our tuner is more efficient than using the default parametrization, even when the optimization process is still running and convergence has not been reached.

X. Conclusion and perspectives

We present in this paper an auto-tuner based on genetic algorithms for finding the optimal configuration of any configurable system whose performance measure is affected by noise. We applied our methodology to the case of burst buffer tuning, which presents many challenges for the auto-tuner. It must deal with a very noisy performance function, a non-homogeneous noise landscape over the parametric space, a four-dimensional space and strong interactions between parameters. This engine takes into account the high variability in I/O execution time by adding noise reduction strategies on top of conventional genetic algorithms. The conducted experiments show that adding these strategies provides in 21 iterations a value performing 10.5% better than the default parametrization. Using a noise reduction strategy along with genetic algorithms increases their success rate by 78.9% and their convergence speed by 23%.

As future work, we plan to confront our methodology to use-cases other than burst buffers, such as power saving strategies and data placement on storage systems. Another perspective is the development of an auto-tuning framework, specific to the HPC ecosystem, which integrates the different methods described in this paper.

References

[1] S. Robert, S. Zertal, and G. Goret, "Auto-tuning of io accelerators using black-box optimization," in *Proceedings of the International Conference on High Performance Computing Simulation (HPCS)*, 2019.

[2] T. Miyazaki, I. Sato, and N. Shimizu, "Bayesian optimization of hpc systems for energy efficiency," in *High Performance Computing*, (Cham), pp. 44–62, Springer International Publishing, 2018.

[3] Z. Cao, "A practical , real-time auto-tuning framework for storage systems," 2018.

[4] "Atos steps up with nvmeof flash accelerator solutions." https://atos.net/wp-content/uploads/2018/07/CT$_J1103_180616_RY_{FT}OOLSTOIMPR_WEB.pdf.Online;accessed:$ $02-03-2020$.

[5] "Ior benchmark description." http://wiki.lustre.org/IOR.

[6] P. Balaprakash, J. Dongarra, T. Gamblin, M. Hall, J. K. Hollingsworth, B. Norris, and R. Vuduc, "Autotuning in high-performance computing applications," *Proceedings of the IEEE*, vol. 106, no. 11, pp. 2068–2083, 2018.

[7] D. Dunlop, S. Varrette, and P. Bouvry, "On the use of a genetic algorithm in high performance computer benchmark tuning," pp. 105 – 113, 07 2008.

[8] Z. Cao, V. Tarasov, S. Tiwari, and 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, pp. 893–907, 2018.

[9] Y. Li, K. Chang, O. Bel, E. L. Miller, and 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, (New York, NY, USA), pp. 42:1–42:14, 2017.

[10] B. Behzad, S. Byna, Prabhat, and M. Snir, "Optimizing i/o performance of hpc applications with autotuning," *ACM Trans. Parallel Comput.*, vol. 5, no. 4, pp. 15:1–15:27, 2019.

[11] B. Behzad, H. V. T. Luu, J. Huchette, S. Byna, Prabhat, R. Aydt, Q. Koziol, and 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.

[12] B. Behzad, S. Byna, M. Prabhat, and M. Snir, "Pattern-driven parallel i/o tuning," in *Proceedings of the 10th Parallel Data Storage Workshop*, pp. 43–48, 11 2015.

[13] G. Aupy, O. Beaumont, and L. Eyraud-Dubois, "What Size Should your Buffers to Disks be?," in *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS)*, IEEE, 2018.

[14] G. Aupy, O. Beaumont, and L. Eyraud-Dubois, "Sizing and Partitioning Strategies for Burst-Buffers to Reduce IO Contention," in *Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2019.

[15] J. Bent, G. Grider, B. Kettering, A. Manzanares, M. McClelland, A. Torres, and A. Torrez, "Storage challenges at los alamos national lab," in *Proceedings of the IEEE 28th Symposium on Mass Storage Systems and Technologies (MSST)*, pp. 1–5, 2012.

[16] N. Liu, J. Cope, P. Carns, C. Carothers, R. Ross, G. Grider, A. Crume, and C. Maltzahn, "On the role of burst buffers in leadership-class storage systems," in *Proceedings of IEEE 28th Symposium on Mass Storage Systems and Technologies (MSST)*, pp. 1–11, IEEE, 2012.

[17] T. Wang, S. Oral, Y. Wang, B. Settlemyer, S. Atchley, and W. Yu, "BurstMem: A high-performance burst buffer system for scientific applications," in *Proceedings of the IEEE International Conference on Big Data (Big Data)*, pp. 71–79, IEEE, 2014.

[18] D. A. Reed and J. Dongarra, "Exascale computing and big data," *Communications of the ACM*, vol. 58, pp. 56–68, 2015.

[19] L. Sthle and S. Wold, "Analysis of variance (anova)," *Chemometrics and Intelligent Laboratory Systems*, vol. 6, no. 4, pp. 259 – 272, 1989.

[20] M. Mubarak, P. Carns, J. Jenkins, J. K. Li, N. Jain, S. Snyder, R. Ross, C. D. Carothers, A. Bhatele, and K. Ma, "Quantifying i/o and communication traffic interference on dragonfly networks equipped with burst buffers," in *2017 IEEE International Conference on Cluster Computing (CLUSTER)*, pp. 204–215, Sep. 2017.

[21] K.-T. Fang, R. Li, and A. Sudjianto, *Design and Modeling for Computer Experiments (Computer Science & Data Analysis)*. Chapman & Hall/CRC, 2005.

[22] N. Saini, "Review of selection methods in genetic algorithms," *International Journal Of Engineering And Computer Science*, vol. 3, 2017.

[23] J. M. Fitzpatrick and J. J. Grefenstette, "Genetic algorithms in noisy environments," *Machine Learning*, vol. 3, no. 2, pp. 101–120, 1988.

[24] D. V. Arnold and H. georg Beyer, "On the effects of outliers on evolutionary optimization," in *International Conference on Intelligent Data Engineering and Automated Learning*, 2003.

[25] P. Stagge, "Averaging efficiently in the presence of noise," in *Proceedings of the 5th Parallel Problem Solving from Nature* (A. E. Eiben, T. Bäck, M. Schoenauer, and H.-P. Schwefel, eds.), pp. 188–197, Springer Berlin Heidelberg, 1998.

[26] 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.

[27] Y. Jin and J. Branke, "Evolutionary optimization in uncertain environments-a survey," *IEEE Transactions on Evolutionary Computation*, vol. 9, no. 3, pp. 303–317, 2005.

[28] Y. Sano and 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, pp. 360–365 vol.1, 2002.

[29] A. Di Pietro, L. While, and L. Barone, "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.

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

[31] "Lustre filesystem." http://lustre.org/.

[32] D. H. Arsham and M. Lovric, "Bartlett's test," *International Encyclopedia of Statistical Science*, vol. 2, pp. 20–23, 2011.

[33] M. B. Brown and A. B. Forsythe, "Robust tests for the equality of variances," *Journal of the American Statistical Association*, vol. 69, no. 346, pp. 364–367, 1974.

[34] N. C. Schwertman and R. de Silva, "Identifying outliers with sequential fences," *Computational statistics & data analysis*, vol. 51, no. 8, 2007.