SHAMan: a flexible framework for auto-tuning HPC systems

Sophie ROBERT[1,2], Soraya ZERTAL[2], and Philippe COUVEE[1]

[1] Atos BDS R&D Data Management
sophie.robert@atos.net philippe.couvee@atos.net
[2] Li-PaRAD, University of Versailles
sophie.robert2@uvsq.fr soraya.zertal@uvsq.fr

ORCID(Sophie ROBERT):https://orcid.org/0000-0003-4359-9124
ORCID(Soraya ZERTAL):https://orcid.org/0000-0002-3390-9240
ORCID(Philippe COUVEE):https://orcid.org/0000-0002-8989-9408

Abstract. Modern computer components, both hardware and software, come with many tunable parameters and their parametrization can have a strong impact on their performance. Auto-tuning methods relying on black-box optimization have delivered good results for finding the optimal parametrization of complex computer systems. In this paper, we present a new optimization framework, called the **S**mart **H**PC **MAN**ager. It provides an out-of-the-box Web application to perform black-box auto-tuning of computer components running on a distributed system for an application submitted by the user. This framework integrates three state-of-the-art heuristics, as well as resampling strategies to deal with the noise due to resource sharing, and pruning strategies to speed-up the convergence process. We demonstrate a possible use-case of this framework by tuning a software I/O accelerator.

Keywords: Auto-tuning · Black-box optimization · Optimization heuristics · I/O accelerators · HPC

1 Introduction

Modern computer components, both hardware and software, come with many tunable parameters and their parametrization can have a significant impact on their performance. Making sure that the parametrization of the system is optimal is thus crucial for maximizing the performance of computer systems. This is especially important for HPC systems, as many components of a cluster can be parametrized, such as Message Passing Interface libraries (MPI), storage bays or I/O (Input/Output) accelerators. For optimal performance, the most adapted parametrization for each application running on the cluster must be determined and used. But, this parametrization is difficult to find because of the complexity of the relationship between each component and the lack of insight on the system's behavior.

Auto-tuning methods relying on black-box optimization are a promising solution to find the optimal parameters of complex systems in various fields, by removing the complex task of tuning the system manually or through theoretical models.

In this paper, we present a new optimization framework, called the **S**mart **H**PC **MAN**ager, which provides an out-of-the-box Web application to perform black-box auto-tuning of custom computer components running on a distributed system, for an application submitted by the user. This framework integrates three state-of-art heuristics, as well as noise reduction strategies to deal with the possible interference of shared resources for large scale HPC systems, and pruning strategies to limit the time spent by the optimization process.

The rest of the paper is organized as follows. Section 2 describes works and tools related to ours and section 3 highlights the advantages of our software compared to already existing ones. Section 4 introduces the principle of black-box optimization and its application to auto-tuning of computer systems. The main features available in SHAMan are described in section 5 and the architecture of the software is described in section 6. An example of use of the software to tune a software I/O accelerator is described in section 7. We conclude and discuss further directions in section 8.

2 Related work

Auto-tuning using black-box optimization has been used in several domains in the last years. It has yielded good results in very diverse situations and has been particularly helpful in computer science for finding optimal configurations of various software and hardware systems [6][8], especially in the HPC [2] [14] and I/O communities [11][4][3]. When it comes to auto-tuning frameworks, several have been proposed recently. Frameworks like *Google Vizier* presented in [7] provide black-box optimization as a service. Several innovative frameworks, like Optuna [1] and Autotune [9], have been developed to find the optimal parametrization of Machine Learning models. However, to our knowledge, no framework specific to HPC clusters' running in production has been suggested.

3 Advantages

Compared to already existing solutions, we provide these main advantages:

- *Accessibility*: the optimization engine is accessible through a Web Interface
- *Easy to extend*: the optimization engine uses a plug-in architecture and the development of the heuristic is thus the only development cost
- *Integrated within the HPC ecosystem*: the framework relies on the Slurm workload manager to run HPC applications. The microservice architecture enables it to have no concurrent interactions with the cluster and the application itself.

- *Flexible for a wide range of use-cases*: new components can be registered through a generalist configuration file.
- *Integrates noise reduction strategies*: because of their highly dynamic nature and the complexity of applications and software stacks, HPC systems are subject to many interference when running in production, which results in a different performance measure for each run even with the same system's parametrization. Noise reduction strategies are included in the framework to perform well even in the case of strong interference.
- *Integrates pruning strategies*: runs with unsuited parametrization are aborted, to speed-up the convergence process

4 Auto-tuning of computer systems using black-box optimization

4.1 What is black-box optimization?

Black-box optimization refers to the optimization of a function f, possibly stochastic, with unknown properties in a minimum of evaluations, without making any assumption on f. The only available information is the history, which consists in the previously evaluated parameters and their corresponding objective value. Given a budget of n iterations, the problem can be transcribed as:

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

- f the function to optimize, F the observed values of the function
- $\mathcal{P}$ the parameter space and ϵ a possible noise function

Black-box optimization process consists in iteratively selecting a parametrization, evaluating the black-box function at this point and selecting accordingly the next data point to evaluate.

4.2 Adaptation for auto-tuning of computer systems

When used for auto-tuning of computer systems, black-box optimization consists in treating the combination of the configurable component and the running application as a black-box, which takes as input the parametrization of the tuned component and outputs a performance measure, as depicted in figure 1. Given this new information, the optimization heuristic decides the next parametrization to evaluate by the system. This feedback loop is repeated until a convergence criterion is reached and the best found parametrization is deemed to be the best parametrization for the system.

4.3 Heuristics available in the framework

Among the black-box heuristics available in the literature, we have implemented surrogate models [10], simulated annealing [13] and genetic algorithms [5]. Our selection is motivated by their simplicity of implementation and their proven efficiency for various computer systems' tuning[12][4].

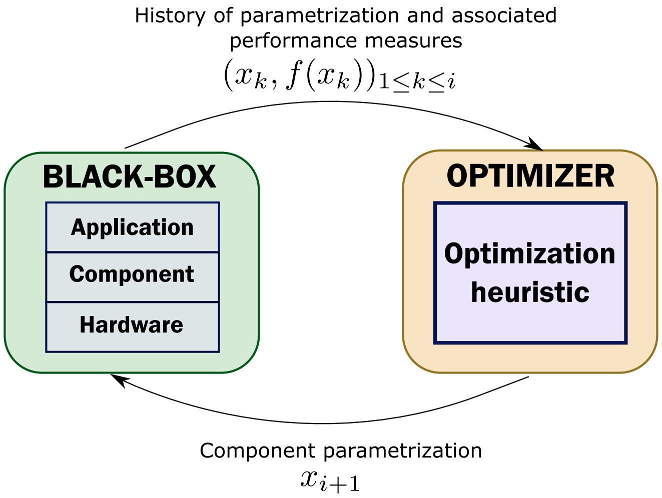

History of parametrization and associated performance measures
$(x_k, f(x_k))_{1\leq k\leq i}$
BLACK-BOX
Application
Component
Hardware
OPTIMIZER
Optimization heuristic
Component parametrization
x_{i+1}

Fig. 1. Schematic representation of the black-box

5 Main features

SHAMan is a framework to perform auto-tuning of configurable component running on HPC distributed systems. It performs the auto-tuning loop by parametrizing the component, submitting the job through the Slurm workload manager, and getting the corresponding execution time. Using the combination of the history (parametrization and execution time), the framework then uses black-box optimization to select the next most appropriate parametrization, up until the number of allocated runs is over.

5.1 Terminology

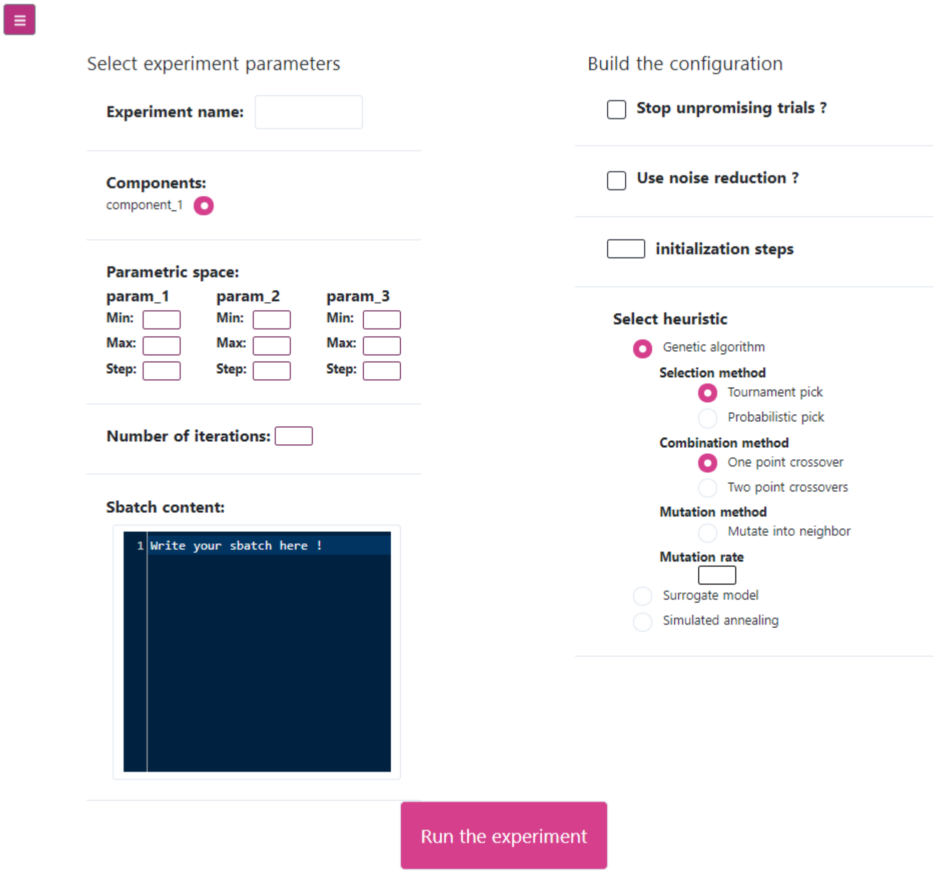
Throughout this section, we will use the following terms:

- **Component**: the component which optimum parameters must be found. It must be configurable, either through environment variable or command line arguments.
- **Target value**: the measurement that needs to be optimized. For now SHAMan only deals with the execution time.
- **Parametric grid**: the possible parametrization tested by SHAMan, defined as a minimum, maximum and a step value.
- **Application**: a program that can be run on the clusters' nodes through Slurm and for which we want to find the optimal parametrization of the component.
- **Budget**: the maximum number of evaluations the optimization algorithm can make to find the optimum value.

- **Experiment**: A combination of a component, a target value, an application and a budget that will output the best parametrization for the application and the component.

5.2 Declaring a new component

Running the command `shaman-install` with a YAML file describing a component registers it to the application. This YAML file must describe how the component is launched and declares its different parameters and how they must be used to parametrize the component. After the installation process, the components are available in the launch menu, as seen in figure 2. This component can be activated through options passed on the job's command line, a command



Select experiment parameters

Experiment name: □

Components:
component_1 ●

Parametric space:

param_1	**param_2**	**param_3**
Min: □	**Min:** □	**Min:** □
Max: □	**Max:** □	**Max:** □
Step: □	**Step:** □	**Step:** □

Number of iterations: □

Sbatch content:

```
Write your sbatch here !
```

Build the configuration

□ **Stop unpromising trials ?**

□ **Use noise reduction ?**

□ **initialization steps**

Select heuristic

● Genetic algorithm

Selection method
● Tournament pick
○ Probabilistic pick

Combination method
● One point crossover
○ Two point crossovers

Mutation method
○ Mutate into neighbor

Mutation rate
□

○ Surrogate model
○ Simulated annealing

Fig. 2. Page for designing and launching an experiment

called at the top of the script or the setting of the LD_PRELOAD variable. The `header` variable is a command written at the top of the script and that is called between each optimization round, before running the job. For instance, a clear cache command is called when tuning I/O accelerators to ensure independence between runs. The parameters with a default value, can be either passed:

- As an environment variable (`env_var=True`)
- As a variable appended to the command line variable with a flag (`cmd_var=True`)
- As a variable passed on the job's command line (`cli_var=True`)

```
components:
  component_1:
    cli_command: example_1
    header: example_header
    command: example_cmd
    ld_preload: example_lib
    parameters:
      param_1:
        env_var: True
        type: int
        default: 1
      param_2:
        cmd_var: True
        type: int
        default: 100
        flag: test
      param_3:
        cli_var: True
        type: int
        default: 1

  component_2:
  ...
```

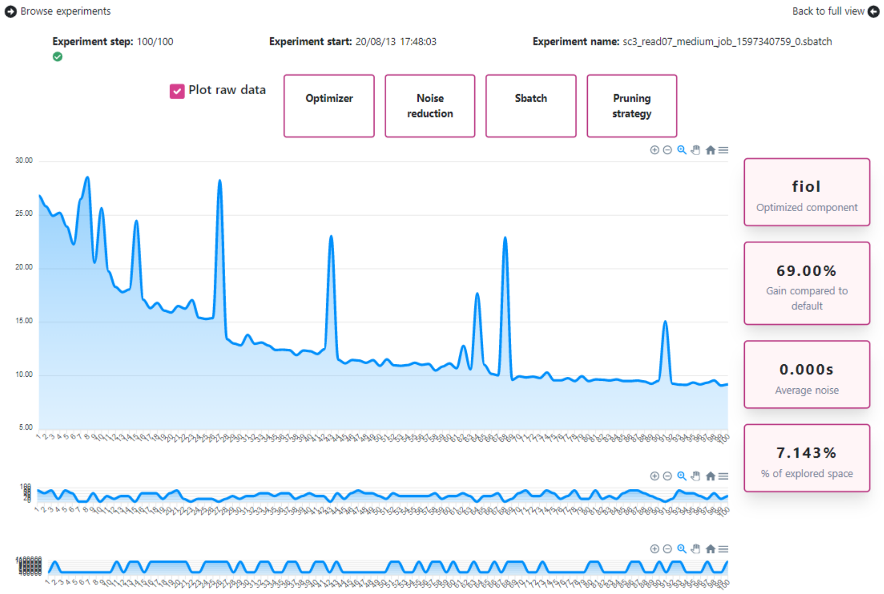
5.3 Launching an experiment

To launch an experiment through the menu depicted in figure 2, the user has to configure the black-box by:

1. Write an application according to Slurm sbatch format.
2. Select the component and the parametric grid through the radio buttons (minimal, maximal and step value).
3. Configure the optimization heuristic, chosen freely among available ones. Resampling parametrization and pruning strategies can also be activated.

5.4 Visualization of an experiment

After the submission, the evolution of the running experiments can be visualized in real-time, by displaying the different tested parameters and the corresponding execution time, as well as the improvement brought by the best parametrization. If noise reduction is enabled, the user can either visualize a raw or aggregated views of the performance as shown on figure 3.



Experiment step: 100/100

Experiment start: 20/08/13 17:48:03

Experiment name: sc3_read07_medium_job_1597340759_0.sbatch

fiol
Optimized component

69.00%
Gain compared to default

0.000s
Average noise

7.143%
% of explored space

Fig. 3. Result page after FIOL experimentation.

6 Architecture

6.1 General architecture

SHAMan relies on a microservice architecture (see figure 4) and integrates:

- A front-end Web application
- A back-end storage database
- An optimization engine, reached by the API through a message broker, and which uses runners to perform optimization tasks.

The several services communicate through a REST API.

6.2 Optimization engine

The optimization engine is a stand-alone Python library. It is installed on a node, separate from the compute nodes, so that it does not interfere with the running application, but is able to communicate with the Slurm workload manager and share the same filesystem as the compute nodes.

The engine, schematically described in figure 5, is composed of a generic black-box library and a wrapper.

The black-box optimizer The black-box optimizer module performs the black-box optimization process. It can optimize any Python object with a `.compute` method that takes as input a vector corresponding to a parametrization and outputs a scalar corresponding to the target value.

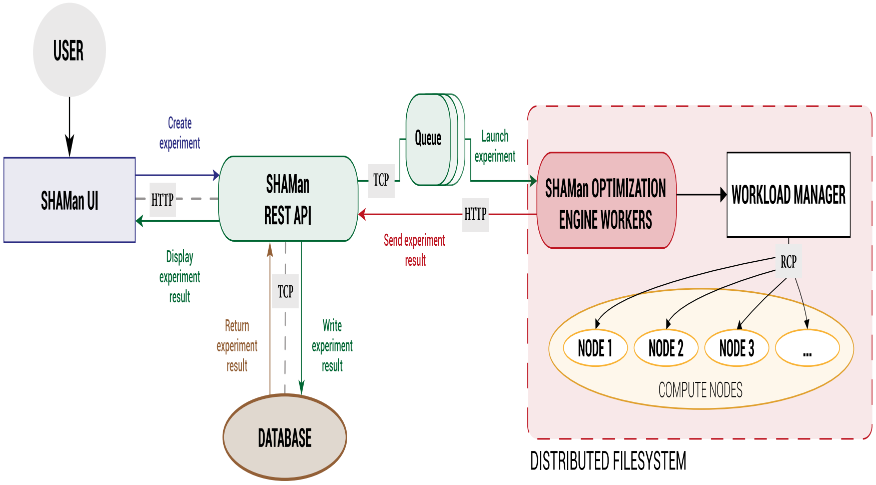


Fig. 4. General architecture of the tuning framework.

The black-box wrapper The black-box wrapper module transforms the component undergoing tuning into a black-box. It is a Python object (called `bb_wrapper`) that configures and launches the component according to the specification given by the user in the configuration file. It builds a `.compute` method that takes as input the parameters of the component and the parameters of the experiment, edits the virtual environment and the sbatch file, submits the job through Slurm, parses the output of the Slurm program to get the execution time, and sends this information to the optimizer so that it can make the decision of the next parameters to use.

6.3 Implementation choices

SHAMan uses *Nuxt.js* as a frontend framework. The optimization engine is written in Python. The database relies on the NoSQL database management system *MongoDB*. The message broker system uses Redis as a queuing system, manipulated with the ARQ Python library. The API is developed in Python, using the FastAPI framework.

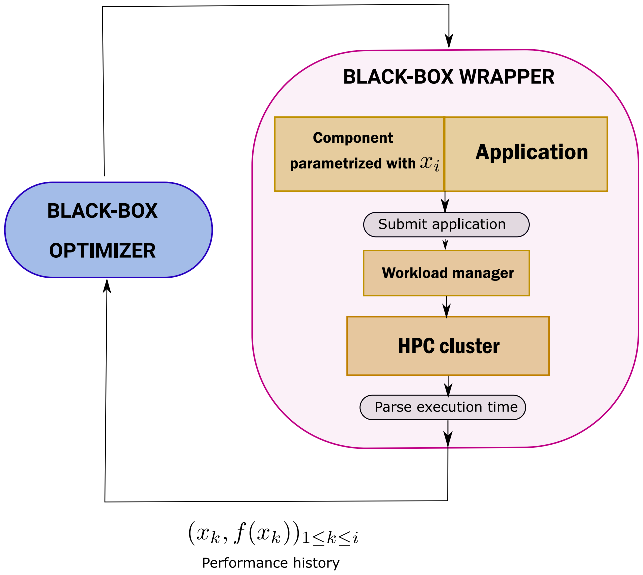


Fig. 5. Schematic representation of the optimization engine.

7 An application to auto-tuning an I/O accelerator

We illustrate how a pure software I/O acceleration library developed by Atos, called the Fast I/O Libraries, `FIOL` has been optimized by the framework when used with an I/O benchmark.

7.1 The Small Read Optimizer accelerator

This accelerator relies on a dynamic data preload strategy to prefetch in memory chunks of files that are regularly accessed, in order to speed-up pseudo-random file accesses. The file is conceptually cut into different zones of size `binsize` and the number of accesses within each is counted, up until a maximum value of `sequence_length`. Once the number of accesses in a zone reaches a certain threshold `cluster_threshold`, the pre-fetching mechanism loads a file zone of size `prefetch_size` in memory. These parameters are summarized in table 1. The FIOL accelerator is highly sensitive to its parametrization, as a poorly set parametrization can trigger too many prefetches, slowing down the application, or too little, which limits the speed-up potential of the accelerator. It is important to find the right balance, in order to prefetch only zones of the file that will be accessed again.

Name of parameter	Description
`sequence_length`	Number of past accesses kept for counting
`binsize`	Size of the file zones
`cluster_threshold`	Number of operations required to trigger the pre-fetching mechanism
`prefetch_size`	Size of the zone that will be prefetched

Table 1. Fast I/O libraries parameters

7.2 Configuration file

The configuration file for registering the SRO with SHAMan is:

```
components:
  small_read_optimizer:
    cli_command: fastio=yes
    parameters:
      sequence_length:
        env_var: True
        type: int
        default: 100
      binsize:
        env_var: True
        type: int
        default: 1048576
      cluster_threshold:
        env_var: True
        type: int
        default: 2
      prefetch_size:
        env_var: True
        type: int
        default: 20971520
```

7.3 Test experiment

Based on the results found in [12], we perform the following experiment:

– Test application: The test application is an in-house benchmarking application, running on a single node. This application performs a series of reads using a pseudo-random pattern.
– Initialization budget: The experiment is initialized with 20 iterations.
– Experiment parametric grid: The selected parametric grid is described in table 2.
– Test black-box heuristic: The test is conducted using genetic algorithms with a tournament pick method for population selection and single point crossover, for a budget of 80 iterations. Noise reduction strategy is disabled and the pruning strategy uses the current median execution time as a threshold time.

Parameter name	**Lower limit**	**Upper limit**	**Step**
Cluster threshold	2	100	20
Binsize	524288	1048576	262144
Sequence length	50	700	50
Prefetch size	1048576	10485760	1048576

Table 2. Description of the parametric grid

7.4 Results

The result screen is displayed in figure 3. Compared to default parametrization, we find an improvement of 69%, after exploring only 7.14% of the specified parametric space. This result shows the strong impact of the parameters on the system's performance and the usefulness of auto-tuning to maximize the effect of this I/O accelerator.

8 Conclusion and further works

We present in this paper a new optimization software, called SHAMan, aimed at tuning configurable component of HPC systems for particular applications. It comes with several state-of-the-art heuristics, as well as resampling and pruning strategies, particularly adapted to the tuning of noisy and costly system. It is also easily configurable for a wide range of different components and use-cases, as we demonstrated by tuning an I/O accelerator. In terms of future works, we are planning on adding more sophisticated resampling and pruning strategies, to further enhance the efficiency of the optimization engine. We are also looking at adding stop criteria different than a budget-based one.

References

1. Takuya Akiba, Shotaro Sano, Toshihiko Yanase, Takeru Ohta, and Masanori Koyama. Optuna: A next-generation hyperparameter optimization framework, 2019.
2. 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*, 106(11):2068–2083, 2018.
3. 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, pages 68:1–68:12, 2013.
4. 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, pages 893–907, 2018.
5. L. Davis. *Handbook of Genetic Algorithms.* Van Nostrand Reinhold, 1991.

6. D. Desani, V. Gil Costa, C. A. C. Marcondes, and H. Senger. Black-box optimization of hadoop parameters using derivative-free optimization. *2016 24th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP)*, pages 43–50, 2016.
7. Daniel Golovin, Benjamin Solnik, Subhodeep Moitra, Greg Kochanski, John Elliot Karro, and D. Sculley, editors. *Google Vizier: A Service for Black-Box Optimization*, 2017.
8. P. Jamshidi and G. Casale. An uncertainty-aware approach to optimal configuration of stream processing systems. *CoRR*, abs/1606.06543, 2016.
9. Patrick Koch, Oleg Golovidov, Steven Gardner, Brett Wujek, Joshua Griffin, and Yan Xu. Autotune. *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, Jul 2018.
10. V. K. Ky, C. D'Ambrosio, Y. Hamadi, and L. Liberti. Surrogate-based methods for black-box optimization. *International Transactions in Operational Research*, (24), 2016.
11. 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, pages 42:1–42:14, New York, NY, USA, 2017.
12. 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.
13. C. D. Gelatt S. Kirkpatrick and M. P. Vecchi. *Optimization by Simulated Annealing*, volume 220. Science, 1983.
14. K. Seymour, H. You, and J. Dongarra. A comparison of search heuristics for empirical code optimization. In *2008 IEEE International Conference on Cluster Computing*, pages 421–429, 2008.