UNIVERSITÉ CATHOLIQUE DE LOUVAIN

UNIVERSIDADE DE LISBOA
INSTITUTO SUPERIOR TÉCNICO

UNDER ERASMUS MUNDUS PROGRAMME

Automatic Configuration for Cloud Workloads

Muhammad Bilal

Supervisor: Prof. Peter Van Roy
Co-Supervisor: Prof. Rodrigo Seromenho Miragaia Rodrigues

Thesis approved in public session to obtain the PhD Degree in

Computer Science and Engineering

Jury

Prof. Charles Nicolas Lucien Pecheur, UCLouvain, Belgium
Prof. Peter Van Roy, UCLouvain, Belgium
Prof. Olivier Bonaventure, UCLouvain, Belgium
Prof. Marco Canini, KAUST, Saudi Arabia
Prof. Paolo Romano, IST ULisboa, Portugal
Prof. Shivaram Venkataraman, University of Wisconsin-Madison, USA
Doctor Neeraja J. Yadwadkar, Stanford University, USA

2022

This thesis is dedicated to everyone who has ever taught me even the smallest of things.

Acknowledgments

I would like to deeply thank everyone who shared this journey with me and helped me along the way. First of all, I would like to thank my academic advisors Marco Canini and Rodrigo Rodrigues, without them this thesis would not have been possible. They were always helpful, supportive, and encouraging. To Marco Serafini and Rodrigo Fonseca for their collaboration with me and useful input on my work. To all the people that took care of all the administrative matters at both of the universities. Special thanks to Peter Van Roy, Vanessa Maons, and Sophie Renard who were always there to help me with the extensive amounts of paperwork.

Many thanks to my sister Hina and my father Jawed who encouraged me to pursue higher education. To my childhood friends Abdul Moeed and Bilal Khan for always being there for more than 25 years. To Sana Imtiaz, with whom I have spent the majority of my academic life. She helped me through countless things, always believed in me, and always had words of encouragement.

Thanks to my colleagues from EMJD-DC, INESC-ID, UCL, and KAUST. Especially to my EMDC fellows João, Igor, Fotis, Ken, Mario, and Daniel. I am grateful to them for all discussions about work, life, and everything in between, and for the good times we spent together. Lastly, thank you to everyone who ever taught me anything and helped me with something, regardless of how small it was. There are too many to name here but they have all contributed to my personal and professional growth.

I have lived in Louvain-la-Neuve, Lisbon, Barcelona, Stockholm, and Thuwal during this journey and I learned something in each city. The people in each country and city contributed to my overall growth. Finally, I would like to thank the committee members for their helpful comments and questions that helped improve this thesis.

This work was supported in part by the Erasmus Mundus Joint Doctorate in Distributed Computing (EMJD-DC) funded by the Education, Audiovisual and Culture Executive Agency (EACEA) of the European Commission under the FPA 2012-0030; the Fundação para a Ciência e Tecnologia (FCT) via projects PTDC/ EEI-SCR/ 1741/ 2014 (Abyss); and Funding from KAUST.

Abstract

With the increase in the adoption of Cloud computing and big data processing systems, a common way to deploy analytics workloads is to acquire on-demand resources in a Cloud environment whenever the workloads need to execute. Public Cloud providers offer numerous infrastructure choices to Cloud users. For example, the number of different Virtual Machine (VM) instances offered by the Cloud providers has been steadily increasing, now numbering in dozens. This enables Cloud users to have access to the type of instance that fits their use case. However, this blessing can also be a curse since selecting the right resources is often not straightforward. When executing analytics workloads in the Cloud, ensuring the allocation of the right resources is paramount to achieve cost efficiency and satisfy strict service-level objectives (SLOs) on job completion times. This requires users to decide the number of instances (VMs) and type of instances for their deployments (together forming a Cloud configuration). In addition, the software platforms for big data processing have system parameters that need to be appropriately set. The choice of these cloud configurations and system parameter values dictate the performance and cost of the whole deployment.

This thesis aims to reduce the decision burden on the users when deploying workloads in the Cloud. Our primary focus is to allow users to concentrate on defining the performance objectives rather than determining the resource allocation for their Cloud workloads. Specifically, we investigate ways to automatically determine configurations for Cloud workloads given user-specified performance and cost objectives. These workloads include jobs running on batch and stream processing systems and serverless functions. The configurations under consideration in this thesis include cloud configurations and system parameters.

More concretely, in this thesis, we discuss automatic Cloud configuration optimization in two contexts. First, for distributed batch data processing systems, a Cloud configuration is the number of VMs and the type of VMs to use for deployment of the workload. Second, for serverless systems, a Cloud configuration is the type of VM instance, CPU share, and memory share given to a serverless function. System parameters, in turn, are the parameters of the distributed software system being used to run analytics workloads. These parameters include parallelism levels, buffer and queue sizes, etc. In this thesis, we perform system parameter tuning for Apache Storm. However, the methodology applies to other similar distributed stream processing systems as well.

To attain this broad set of goals, we present four research pillars that comprise this thesis. First, we discuss an empirical evaluation of several popular black-box optimization algorithms in the context of automatic cloud configuration and determine the best choice under different conditions. Next, we propose Vanir, a framework to automatically optimize cloud configurations for multi-framework analytics clusters while reducing the number of offline benchmarking runs required. Third, we discuss the benefits of decoupling memory and CPU resources and enabling the use of different instance types for serverless functions, showing that black-box optimization algorithms can handle the resulting increase in resource configuration options. Lastly, we propose and evaluate three methods for automatically tuning the system parameters of stream processing systems to efficiently deploy stream processing workloads

in the Cloud.

Keywords: Cloud computing, Distributed systems, Serverless computing, Black-box optimization, Parameter tuning, Resource configuration

Contents

Acknowledgments . . . iii
Abstract . . . v
List of Tables . . . xi
List of Figures . . . xiii

1 Introduction **1**
1.1 Thesis goal and scope . . . 2
1.2 Thesis Contributions . . . 3
1.3 Summary of Results . . . 4
1.4 Publications . . . 5
1.5 Thesis Outline . . . 5

2 Background and Literature Review **7**
2.1 Background . . . 7
2.1.1 Cloud computing and big data frameworks . . . 7
2.1.1.1 Cloud computing . . . 7
2.1.1.2 Big data processing frameworks . . . 8
2.1.1.3 Distributed Storage Systems . . . 9
2.1.2 Problem Formulation . . . 9
2.1.3 Black box optimization algorithms . . . 10
2.1.3.1 Random Search . . . 10
2.1.3.2 LHS Sampling . . . 11
2.1.3.3 Bayesian Optimization (BO) . . . 11
2.1.3.4 Tree-structured Parzen Estimator (TPE) . . . 13
2.1.3.5 Stochastic Hill Climbing (SHC) . . . 14
2.1.3.6 Simulated Annealing (SA) . . . 14
2.2 Literature Review . . . 15
2.2.1 Cloud configuration optimization . . . 15
2.2.2 Resource allocation in data center environments . . . 15
2.2.3 System configuration optimization . . . 16

3 Evaluation of Black Box Algorithms for Optimizing Cloud Workloads 19
3.1 Introduction . . . 19
3.2 Background and Related Works . . . 21
3.2.1 Problem Formulation . . . 21
3.2.2 Related works . . . 22
3.3 Optimization algorithms . . . 22
3.4 Experimental methodology . . . 23
3.4.1 Implementation . . . 24
3.4.2 Datasets . . . 24
3.4.3 Evaluation metrics . . . 25
3.5 Experimental Results . . . 26
3.5.1 Configuring optimization algorithms . . . 26
3.5.1.1 Importance of hyper-parameters . . . 26
3.5.1.2 Best algorithm-specific hyper-parameters . . . 28
3.5.2 Narrowing down the choices for BO . . . 29
3.5.2.1 Best acquisition function for BO . . . 29
3.5.2.2 Best BO variant . . . 30
3.5.3 Which optimization algorithm is better? . . . 33
3.5.3.1 Optimizing for execution time . . . 33
3.5.3.2 Optimizing for execution cost . . . 35
3.5.4 What is the best method for online optimization? . . . 35
3.5.5 When are more optimization runs worth the extra cost? . . . 36
3.5.6 Summary . . . 37
3.6 Choosing a Black-Box Optimization algorithm . . . 38
3.7 Discussion . . . 38
3.8 Summary . . . 39

4 Finding the Right Cloud Configuration for Analytics Clusters 41
4.1 Introduction . . . 41
4.2 Analytics clusters config . . . 42
4.2.1 The cloud configuration problem . . . 43
4.2.2 Selecting good configurations matters . . . 43
4.2.3 Challenges . . . 44
4.2.4 Baseline solutions . . . 44
4.3 Overview of Vanir . . . 45
4.4 Offline optimizer . . . 46
4.4.1 Metrics-based algorithm . . . 47
4.4.2 Tuning the offline optimizer . . . 48
4.5 Online optimizer . . . 48

4.5.1 Mondrian forests . . . 49
4.5.2 Acquisition method . . . 49
4.5.2.1 Centroid selection . . . 50
4.5.2.2 Candidate set selection . . . 50
4.5.2.3 Configuration selection . . . 51
4.5.3 Tuning the acquisition method . . . 51
4.6 Learning from other jobs . . . 52
4.6.1 Determining if jobs are similar . . . 52
4.6.2 Bootstrapping models . . . 53
4.7 Evaluation . . . 53
4.7.1 Applications . . . 54
4.7.2 Results . . . 54
4.7.2.1 Quality of optimization . . . 54
4.7.2.2 Cost of optimization . . . 56
4.7.2.3 Speed of search . . . 56
4.7.2.4 Constraint violations . . . 57
4.7.2.5 Contribution of different optimizers . . . 58
4.7.2.6 Contribution of similarity and transfer learning . . . 58
4.7.2.7 Unbounded search . . . 59
4.7.2.8 In-depth look at the best configurations . . . 59
4.7.2.9 Generalizing to a pipeline of batch jobs . . . 60
4.7.2.10 Handling input data size changes . . . 61
4.8 Related Work . . . 61
4.9 Discussion . . . 62
4.10 Summary . . . 63

5 Rethinking Resource Allocation for Serverless Functions 65
5.1 Introduction . . . 65
5.2 Motivation . . . 67
5.3 Experimental setup . . . 68
5.3.1 Benchmark functions . . . 69
5.3.2 Cost calculation . . . 69
5.4 How beneficial is flexible resource allocation? . . . 70
5.4.1 Larger search spaces yield advantages... . . . 70
5.4.2 ... and potential to reduce idleness . . . 71
5.5 Is automatically discovering good configurations possible? . . . 72
5.5.1 Background on optimization techniques . . . 73
5.5.2 Optimization techniques are effective . . . 74
5.5.3 Input data variation has modest influence . . . 76

5.5.4 Online optimization is feasible . . . 78
5.5.5 Resource allocation models can predict performance of untested configurations . 79
5.6 Automatic resource allocation from the provider's perspective . . . 80
5.6.1 On the interface between user and provider . . . 80
5.6.2 Cost-performance trade-off while using different instance types . . . 84
5.7 Design space . . . 85
5.8 Related work . . . 85
5.9 Discussion . . . 86
5.10 Summary . . . 87

6 Automatic Parameter Tuning of Streaming Processing Systems 89
6.1 Introduction . . . 89
6.2 Preliminaries . . . 91
6.2.1 Stream-Processing Systems . . . 91
6.2.2 Problem Formulation . . . 91
6.2.3 Our Approach . . . 92
6.2.4 Background on Latin Hypercube Sampling . . . 93
6.3 Impact of Parameter Tuning . . . 94
6.4 Optimization Algorithms . . . 97
6.4.1 Hill-Climbing Algorithm (HC) . . . 97
6.4.2 Modified Hill-Climbing Algorithm (mHC) . . . 98
6.4.3 The Heuristics-based Approach (HB) . . . 99
6.5 Evaluation . . . 100
6.5.1 Applications . . . 101
6.5.2 Experimental Setup . . . 101
6.5.3 Performance of Parameter Tuning . . . 102
6.5.4 Impact of Parameter Ranking on HB . . . 105
6.5.5 Comparison with Other Approaches . . . 105
6.5.6 Extrapolation to Large Clusters . . . 106
6.5.7 Cost-Benefit Analysis . . . 107
6.6 Related Work . . . 107
6.7 Discussion . . . 109
6.8 Summary . . . 110

7 Conclusions 111
7.1 Summary and Insights . . . 111
7.2 Future directions . . . 112

Bibliography 115

List of Tables

3.1 Prior work, corresponding optimization algorithms and cloud configurations that they optimize. . . . 22
3.2 Workloads and configuration search space for each dataset. . . . 24
3.3 Algorithm-specific hyper-parameters and their values. For BO with LCB acquisition function κ is used; for EI and PI, ξ is used. . . . 28
3.4 Number of extra periodic repetitions of the workload required in production to break-even with the time and cost of optimization when optimizing for execution time and execution cost, respectively. . . . 37

4.1 Max to Min ratio for execution time and execution cost of the tested configurations. . . . 44
4.2 Possible values of configuration parameters. . . . 53

5.1 Resource allocation search space. A configuration is a value for CPU share, memory limit and instance family. For instance family names, a prefix 'c' means compute-optimized, and 'm' is a general-purpose instance; a suffix 'g' means Graviton2 ARM-based, 'a' AMD-based, and no suffix corresponds to Intel-based instances. The search space has 288 configurations. . . . 67
5.2 Benchmark serverless functions. Each benchmark has certain resources that are more important than others, either for performance or to avoid function failure. Resources: C: CPU, P: parallelism, M: memory, N: network. . . . 69
5.3 Number of alternative instance types with one or more resource allocation configurations with the performance metric within threshold (θ) of the best configuration in the *Decoupled* search space. The cells in red indicate cases where there is no alternative instance type able to reach within θ% of the best configuration. The cells in blue denote cases where all instance types have at least one configuration that provides performance within θ% of the best configuration. . . . 72

6.1 Selected parameters and their value ranges. . . . 95
6.2 Algorithms used in our framework for parameter tuning. . . . 97
6.3 Cost-benefit analysis contrasting the cost of running parameter tuning versus the cost of scaling out the stream processing cluster. The benefits of automated parameter tuning always occur within a few hours. . . . 107

List of Figures

2.1 Increase in the number of instance available in AWS EC2 over the years. 8

2.2 Example of an objective function being modeled by Bayesian optimization. The objective function is being minimized. The utility function for each acquisition function is shown along with the recommendation for the next point to evaluate, based on the highest value of the utility function. 12

3.1 (a, b) Execution time and execution cost for two selected workloads in DS_1 (see Table 3.2) shown as a heatmap normalized by the best configuration for each respective metric. The blank boxes are configurations that were either not present in the dataset or do not execute the workload successfully. 20

3.2 Example search space and optimization samples. 22

3.3 Importance (through Functional ANOVA) of each hyper-parameter of the optimization algorithms. 27

3.4 Comparison of acquisition function for different Bayesian optimization algorithms. All three cases shown here are scenarios in which the objective function is execution cost. In all other scenarios, there is less than 5% difference in the performance of different acquisition functions. 27

3.5 Comparison of different Bayesian optimization algorithms. 30

3.6 Execution time and execution cost of different configurations of $\mathsf{lda}_{\mathsf{H}}$ and km_{B}. In (a) and (b), the actual number of nodes can be obtained from the cluster size using multiplying factors of 1, 2, 4, 8 for 4xlarge, 2xlarge, xlarge and large instance sizes, respectively. . . . 31

3.7 Number of instance families explored by GBRT and GP when optimizing for execution cost. 32

3.8 Configurations tested by GBRT and GP when optimizing for execution cost km_{B}. 32

3.9 Comparison of optimization algorithms when optimizing for execution time using tuned or default algorithm-specific hyper-parameters. 33

3.10 Comparison of optimization algorithms when optimizing for execution cost using tuned or default algorithm-specific hyper-parameters. 34

3.11 Aggregate number of SLO violations for varying thresholds (as a factor of the objective function value of the best configuration in the search space). Each optimization algorithm is using the default algorithm-specific hyper-parameter values. 36

3.12 Decision tree for selecting an optimization algorithm. 38

4.1 Example data analytics cluster. 42
4.2 Configuration optimization workflow in Vanir. 46
4.3 Acquisition method used by the online optimizer. 48
4.4 Example candidate set based on 20% maximum distance from a centroid (purple cross) for the middle framework (⟨20, m5.l⟩). The valid configurations in the neighborhood are those with a variation of ±10% memory and ±10% number of CPUs (combined ±20%). A selected configuration based on instance type variation doubles the number of CPUs and maintains the same memory. 51
4.5 (a) Best execution time (left). (b) Number of benchmarking iterations (right). 54
4.6 Cost of optimization. Baseline 1 and 2 are generally 1.5-6 times more expensive than Vanir. 55
4.7 Progression of optimization for target jobs. 55
4.8 Speed of optimization. On average, Vanir takes only half the number of iterations compared to Baseline 2 to reach within 15% of the best execution time found. 56
4.9 Progression of Vanir optimization across eight jobs. Only one ET constraint violation occurred in online phase. 57
4.10 Contribution of different components of Vanir. Compared to offline-only optimization, the online phase lowers execution time by up to 36%. 58
4.11 Comparing Vanir's performance on unbounded search space vs. Baseline 2 using sp. Vanir automatically determines the right upper resource limits and avoids incorrect user-defined search space bounds. 59
4.12 Comparing Vanir's performance on pipeline of batch jobs (pp) vs. Baseline 2. Performance is comparable; Vanir spends just 6 runs in benchmarking phase. 60
4.13 Handling changes in input data size for cc. Vanir improves the execution time by 30% within 10 optimization runs for input data size increase of 10% and 50%. 61

5.1 Execution time and cost of each function across the entire configuration space defined in Table 5.1, normalized w.r.t. the best configuration of each function. 68
5.2 Four strategies for configuring serverless functions. Each strategy defines a search space, corresponding to a subset of the possible resource allocation configurations. . . . 70
5.3 Potential gains within each search space. The graphs show the best (a) Execution Time (ET) and (b) Execution Cost (EC) of each function across different search spaces, normalized to the overall best configuration. 71
5.4 Performance of the best-found configurations for sampling-based and model-based algorithms. 74
5.5 Execution time performance of the best found configuration as optimization by different BO variants progresses. 75
5.6 Execution cost performance of the best found configuration as optimization by different BO variants progresses. 76

5.7 Execution time of the best configurations found by a generic optimization process (blue), a data-specific optimization process (orange), and the overall ideal configuration (green). 77
5.8 Average number violations during online optimization for (a) execution time (ET) and (b) execution cost (EC). 78
5.9 MAPE for different benchmarks and optimization methods when optimizing for (a) execution time and (b) execution cost. 79
5.10 MAPE for different benchmarks and optimization methods when comparing the best configuration for each instance type, when optimizing for (a) execution time, and (b) execution cost. 80
5.11 Method for measuring the distance between the configurations in the predicted and actual Pareto front. The distance is split into execution time and execution cost components and measured between each configuration in the predicted Pareto front and the nearest configuration in actual Pareto front. 80
5.12 Normalized absolute average distance between the points in the predicted Pareto front and the actual front for the default input. We show normalized execution cost (d_c) and execution time (d_t) components of the distance separately. 81
5.13 Convergence of the optimization process (BO with GP) for all the benchmarks and different weights for execution time and execution cost. 82
5.14 Normalized values for execution time and execution cost after a hierarchical optimization. ET/EC and ideal-ET/ideal-EC represent the best configuration using the prediction model and with oracle-like knowledge, respectively. The normalization is done w.r.t the best configuration observed during optimization process 83
5.15 Reduction in cloud provider costs while keeping the objective function value within 10% of the best found configuration in the search space. Cost reduction is based on model predictions for the best configurations of each instance type, assuming an 80% discount in pricing for idle resources. 84
5.16 Systematic set of design choices of a system providing automatic resource allocation for serverless functions. 85

6.1 Configuration optimization framework. 93
6.2 Variation of the 90^{th}-ile latency for different configurations and workloads in three benchmark topologies. 94
6.3 Latency and throughput for different values of executors for Rolling Count and Split bolt for the RC topology. 96
6.4 Latency results for different duration of experiments 102
6.5 Performance of the three algorithms on the RC topology with book workload and throughput objective of 150k (a,b) and 300k (c,d) tuple/s. 103
6.6 Performance of the three algorithms on RTW topology with book workload and 150k throughput/s objective. 104

6.7 Performance of the three algorithms on SA topology with book workload and 100k throughput/s objective. 104
6.8 Performance variation due to different parameter rankings for mentioned throughput limit. 105

Chapter 1

Introduction

Data analytics is part of the computational ecosystem of virtually every organization that seeks to extract value from the data it collects. Data analytics provides numerous benefits to organizations, ranging from improved efficiency, to optimized customer acquisition and retention, or the creation of new products and services [1]. Processing such large volume of data often requires a distributed data processing system, given the size of the data sets and the rate at which data is produced.

Cloud computing is the delivery of computing services, including servers, storage, databases, networking, software, analytics, and intelligence over the Internet ("the Cloud"). There is a steady increase in the adoption of Cloud computing as a primary source of compute and storage for data analytics jobs. For example, spending on Cloud Infrastructure-as-a-Service (IaaS) worldwide has gone from 44.4 billion dollars in 2019 to 51 billion dollars in 2020 and it is expected to grow by 60% to 82.2 billion dollars by 2022 [2]. Under this computing paradigm, an increasingly common way to deploy the analytics workloads is to *acquire resources* in a Cloud environment and do so *on demand*, whenever (typically recurring) jobs need to execute. In this scenario, the characteristics of Cloud computing provide great flexibility to scale computation up (by picking appropriately sized computing instances) and out (by choosing how many instances to use).

As Cloud computing gains adoption, Cloud providers are offering an increasing number of choices to Cloud users. For example, the number of different types of compute instances has been steadily increasing. This enables Cloud users to have access to the type of instance that fits their use case. However, this blessing can also be a curse since selecting the best configuration is often not straightforward. When executing jobs in the Cloud, ensuring that the *right resources* are allocated is paramount, not only due to cost efficiency but also to satisfy strict service-level objectives (SLOs) on job completion times. Furthermore, achieving these objectives is particularly complex, since it requires the user to decide the number of instances (typically, Virtual Machines or VMs) and type of instance to use for the deployment. These parameters are part of what we will call a Cloud configuration throughout the thesis. In addition, software systems usually have their own system parameters that need to be appropriately set. The choice of these Cloud configurations and parameter values dictate the performance of the whole deployment.

To add to the difficulty of this problem, different applications and workloads have different behavior and resource requirements, and therefore their optimal configuration is not the same. This choice can significantly impact the performance (execution time, latency, or throughput) and cost of the deployment: as we will show in subsequent chapters, the difference between the best and the worst performing configurations could be anywhere between 2x to one order of magnitude. In the absence of systematic methods for finding the right configuration, deploying these computations in the Cloud is often a matter of trial and error, which is tedious, time-consuming, and impractical in cases where there is a large number of workloads that need optimization.

As the big data landscape becomes an ever-growing, richer ecosystem, the challenges in finding good configurations are increasing. Many types of frameworks exist in the modern big data landscape, such as batch processing systems, stream processing systems, data query frameworks, AI frameworks, distributed databases, and distributed file systems, with dozens of open-source options for each category. Different frameworks have different performance characteristics and resource requirements. Finding good configurations for a single framework is already a challenging task but data analytics has evolved from using a single system or a pair of systems (e.g., HDFS for storage + Hadoop for computation) to have many different frameworks connected in a pipeline. Additionally, new paradigms within Cloud computing have emerged, such as serverless computing, which brings in its own problems in terms of finding the right configuration for the infrastructure that supports those paradigms.

Therefore, having automatic methods to optimize the configurations given the performance requirements and user-specific constraints is becoming increasingly important to improve performance and drive down the cost of Cloud deployments while achieving desired performance goals.

1.1 Thesis goal and scope

This thesis aims to reduce the decision burden on the users when deploying workloads in the Cloud. Our primary focus is to allow users to concentrate on defining the performance objectives of their applications rather than determining the resource allocation for their Cloud workloads.

Specifically, we investigate ways to automatically determining configurations for Cloud workloads given the user-specified performance objectives. These workloads include jobs running on batch and stream processing systems and serverless functions. Our focus is on periodic workloads since a large fraction of the data analytics workloads is recurring, as supported by reports that more than 40% of the jobs in production clusters are recurring computations [3, 4, 5]. The configurations we are concerned with in this thesis include: **Cloud configurations** and **system parameters**.

More concretely, in this thesis, we discuss **automatic Cloud configuration optimization** in two contexts. First, for distributed batch data processing system, a Cloud configuration is the number of VMs and the type of VMs to use for deployment of the workload. Secondly, for serverless systems, Cloud configuration is the type of VM instance, the amount of CPU share, and the amount of memory share given to a serverless function. **System parameters** are the parameters of the distributed software system being used to run analytics workloads. These parameters include but are not limited to parallelism

levels, buffer and queue sizes, etc. In this thesis, we perform system parameter tuning for Apache Storm. However, the methodology can also be used for other similar distributed stream processing systems.

1.2 Thesis Contributions

In this thesis, we propose several techniques for Cloud configuration and parameter tuning for Cloud workloads. In particular, we demonstrate that it is possible to automatically find good configurations in a variety of modern and particularly challenging settings, ranging from multi-framework analytics clusters, serverless computing backends, or stream processing systems. In this context, we discuss many existing black-box algorithms from the literature, gaining a deep understanding of which ones are best suited for each setting, and describing improvements to these algorithms to make the optimization more efficient. This thesis is comprised of four main research contributions, which can be summarized as follows:

- **Chapter 3.** We conduct an extensive evaluation of the performance of several popular black-box optimization algorithms in the context of optimization of Cloud configurations for batch processing workloads. We evaluated 8 algorithms and 23 workloads using 2 objective functions. We also examine the problem of finding optimal Cloud configurations in an online setting and perform a break-even analysis to evaluate when performing more optimization runs is worth it.
- **Chapter 4.** We propose the design of Vanir, an automatic Cloud configuration optimization framework for analytics clusters comprising of multiple distributed systems. One of the key challenges in this setting is that the interconnection of several different frameworks leads to an explosion in the space of possible configurations. To address this, Vanir performs automatic Cloud configuration optimization by splitting the optimization task into benchmarking and production phases. This allows Vanir to handle the scalability challenges by finding a good, though not perfect configuration without requiring a long offline benchmarking or optimization, and then further improving the choice of configuration during the production phase.
- **Chapter 5.** We determine the benefits of decoupling memory and CPU resource allocations and using different instance types for hosting serverless functions. In this setting, one of the technical challenges is dealing with a large number of resource allocation choices, which we address using black-box optimization methods. Our findings lead to devising three types of interfaces to the user, based on an underlying optimization methodology, and allowing the user to select different cost-performance trade-off points in a principled way. Lastly, we showed that, even in the presence of prediction errors, we can utilize our performance models to enable the use of different instance types while providing predictable performance.
- **Chapter 6.** We conduct an evaluation of algorithms to automate parameter tuning of stream-processing systems. In addition to generic hill climbing, we created a modified hill-climbing algorithm and a heuristics-based approach in order to create a more efficient parameter tuning algorithm. The heuristics-based approach outperforms the basic hill-climbing algorithm in three of the

four test scenarios. Furthermore, while the evaluation of these algorithms was based on Apache Storm, the concepts apply to other distributed stream processing systems.

1.3 Summary of Results

The main results of this dissertation, structured according to each of the above mentioned contributions, can be summarized as follows:

- When comparing multiple black-box optimization algorithms for Cloud configuration optimization, our evaluation showed that tuning the algorithm-specific hyper-parameter black-box optimization algorithms is less important than the choice of the initial samples and the optimization budget. We found no clear set of optimal hyper-parameter values for any of the algorithms. While some existing works have preferred Bayesian optimization with Gaussian processes and Bayesian optimization with Extra Trees, our evaluation suggests that there is no silver bullet, since Bayesian Optimization (BO) with Gradient Boosted Regression Trees (GBRT) and Gaussian Processes(GP) perform better when optimizing for execution time and execution cost, respectively (§ 3.5).

- When optimizing Cloud configurations for analytics clusters comprised of multiple distributed frameworks, Vanir finds configurations comparable to benchmarking-based offline methods despite a 3$\times$ shorter benchmarking phase. Vanir has 1.3-24$\times$ lower optimization search cost, and it is 2$\times$ faster to reach within 15% of the execution time of the best configuration found (§ 4.7).

- Using the performance data we have collected for serverless functions, we show a potential to improve execution cost and execution time by up to 50% and 40% by decoupling CPU and memory resource allocations and utilizing different VM types for serverless functions. Our results showed that sampling-based and model-based black-box optimization methods could find configurations that are within 10-20% of the performance of the best configuration within 20 optimization trials (for a search space of 288 configurations). However, model-based methods are the obvious choice if predictions for untested configurations are needed. We found BO with GP to be the best model-based method in terms of reaching the best configuration within a limited number of optimization trials and providing a much lower prediction error than other BO variants we tested. Lastly, we show that our performance models can enable Cloud providers to use the idle resources of non-optimal instance types while minimizing the performance variation for the end user (§ 5.4, 5.5, 5.6).

- For the optimization of system parameters for distributed stream processing systems, we have shown that our modified hill-climbing algorithm and heuristics-based approach outperform the basic hill-climbing algorithm in three of the four experimental scenarios. In the worst case, the heuristics-based (HB) approach might provide up to 40% higher latency compared to hill-climbing algorithms. However, HB can converge two to five times faster to that point as compared with the other two approaches and thus might be suitable for online parameter tuning (§ 6.5).

1.4 Publications

The list of published and submitted publications relevant to this thesis are as follows:

- Muhammad Bilal and Marco Canini. 2017. Towards automatic parameter tuning of stream processing systems. In Proceedings of the 2017 Symposium on Cloud Computing (SoCC '17). ACM, New York, NY, USA, 189-200.
- Muhammad Bilal, Marco Canini, and Rodrigo Rodrigues. "Finding the right Cloud configuration for analytics clusters." Proceedings of the 11th ACM Symposium on Cloud Computing. 2020.
- Muhammad Bilal, Marco Serafini, Marco Canini, and Rodrigo Rodrigues. "Do the best Cloud configurations grow on trees? an experimental evaluation of black box algorithms for optimizing Cloud workloads." Proceedings of the VLDB Endowment 13.12 (2020): 2563-2575.
- Muhammad Bilal, Marco Canini, Rodrigo Fonseca, and Rodrigo Rodrigues. "With Great Freedom Comes Great Opportunity: Rethinking Resource Allocation for Serverless Functions." arXiv preprint arXiv:2105.14845 (2021). **Currently under submission**

Other Publications:
Muhammad Bilal, Hassan Alsibyani, and Marco Canini. 2018. Mitigating Network Side Channel Leakage for Stream Processing Systems in Trusted Execution Environments. In Proceedings of the 12th ACM International Conference on Distributed and Event-based Systems (DEBS '18). ACM, New York, NY, USA, 16-27. DOI: https://doi.org/10.1145/3210284.3210286

1.5 Thesis Outline

The remainder of this dissertation is organized as follows.
Chapter 2 provides the background on the problem of Cloud configuration optimization, black-box optimization algorithms and surveys the core body of related research.
Chapter 3 presents an empirical evaluation of several commonly used black-box optimization algorithms for the task of Cloud configuration optimization.
Chapter 4 presents our work on automatic Cloud configuration optimization for data analytics clusters comprising of multiple distributed systems.
Chapter 5 explores a more flexible resource allocation approach for serverless functions. We also discuss the methods and techniques that enable automatic resource allocation given the expanded resource options for serverless functions under this flexible resource allocation approach.
Chapter 6 presents the algorithms for automatic system parameter tuning of distributed stream processing systems.
Chapter 7 concludes this dissertation with takeaways, a discussion about the limitations of our research works, some potential further research opportunities, and a set of final remarks.

Chapter 2

Background and Literature Review

2.1 Background

This chapter presents the background information to aid the reader in contextualizing this thesis. First, in Section 2.1.1, we provide a brief introduction of cloud computing as well as distributed data processing frameworks – including batch and stream processing. Next, Section 2.1.2 provides the general problem formulation that we will be using throughout this thesis with some modifications specific to each individual research problem. Then, in Section 2.1.3, we provide a summary of the existing black-box optimization algorithms used in this thesis. Finally, we present an overview of the core related work in Section 2.2, with additional, more specific related work appearing subsequently at the end of each chapter.

2.1.1 Cloud computing and big data frameworks

2.1.1.1 Cloud computing

Cloud computing provides on-demand computing power and other IT resources via the internet through a pay-as-you-go pricing model. Users can quickly gain access to hundreds or thousands of machines for their data processing needs. Cloud computing helps users avoid significant upfront investment in hardware and removes the need for managing that hardware. It has enabled the proliferation of big data frameworks since now anyone can do big data processing while using the pay-as-you-go resources in the Cloud. Infrastructure as a Service (IaaS) is the most basic cloud computing service provided by cloud providers. With IaaS, users can rent IT infrastructure (servers, VMs, storage, networks, operating systems) from the cloud provider on a pay-as-you-go basis. The big benefit of IaaS is that it provides the easiest way to migrate existing in-house deployments to the Cloud. With the increasing adoption of IaaS, cloud providers have been increasing the number of different VM instances they offer to users over the last decade. More choices provide more flexibility but at the same time make it harder to find the most appropriate choice.

Figure 2.1 shows the number of instances available in AWS EC2 across the years. The number of instances has gone from 5 instances in 2008 to 326 instances in 2020. Note that we are ignoring

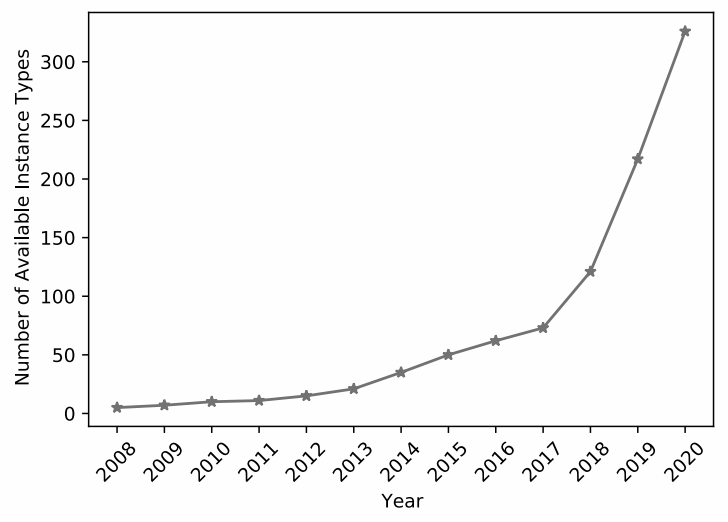


Figure 2.1: Increase in the number of instance available in AWS EC2 over the years.

the accelerated computing (GPU and FPGA) instances available in AWS since a user generally knows if their workload requires GPUs or FPGAs. We also exclude the instances listed as belonging to the previous generation [6].

Different instance types provide different performance and cost characteristics. The cost-performance trade-off provided by different instance types for different workloads is not obvious, thus making the process of finding the best choice of instance type non-trivial.

In addition to IaaS, cloud providers are now increasingly offering serverless computing. Serverless computing is a cloud computing execution model in which users can build and run applications and services without worrying about servers. The responsibility infrastructure management tasks such as provisions, configuring, and managing servers is handled by the cloud provider instead of the end-user. For example, services like AWS lambda come with automatic scaling, automatic provisioning of user-defined resources, built-in high availability, and a fine-grained pay-per-use billing model. Developers only have to specify the resource allocation per lambda (in terms of memory) and package their application code with dependencies. Cloud providers take care of when, where, and how to provision the necessary resources based on the input load to the serverless application.

While serverless enables developers to concentrate on their application code rather than the infrastructure code, the problem of resource management and configuration does not entirely vanish. Even when the cloud provider takes full control of resource provisioning, the responsibility is simply offloaded to the cloud provider, and it now has to make the appropriate decision while considering cost-performance trade-offs. Thus, the techniques discussed in this thesis are relevant even with the increasing prevalence of managed service offerings by cloud providers.

2.1.1.2 Big data processing frameworks

Big data processing generally requires distributed data processing systems that are capable of utilizing tens to thousands of machines. Users can create their data processing jobs on these systems using the

programming APIs that these systems provide. Broadly, these frameworks are split into two categories depending on how they operate on data: (i) Batch processing systems and (ii) Stream processing systems. However, increasingly, multiple frameworks incorporate both batch and stream processing APIs so that the same distributed data processing framework can be used for both batch or stream processing.

Batch Processing Systems. These distributed systems analyze a batch of data altogether. A Batch processing job will read data from a storage system or a database, perform the user-defined computation on the data and return the results to databases or a distributed storage system. Examples of well-known batch processing systems are Hadoop MapReduce [7], Apache Spark [8], and Apache Flink [9].

Stream-Processing Systems. These are software systems that analyze and act on (potentially unbounded) streams of data, generally in real-time or near real-time (considered as having latencies in the range of sub-second up to a few minutes). Stream-processing systems have the ability to perform streaming analytics, i.e., to continuously perform mathematical or statistical analyses on one or more data streams. Modern stream-processing systems have distributed, scalable, and fault-tolerant architectures that can handle high volumes of data in real-time.

There are several open-source stream-processing systems currently available, and the number is growing steadily. Apache Storm [10], Heron [11], Apache Flink [9] and Spark Streaming [8] are a few examples of production-grade stream-processing systems.

2.1.1.3 Distributed Storage Systems

In addition to data processing systems, other distributed systems that we refer to in this work are distributed storage systems. Distributed storage systems could either be distributed file systems or distributed databases. Just like distributed data processing frameworks, these storage systems store data on multiple machines. These systems often serve as input and/or output sources for data processing systems. Examples of these systems include HDFS, Apache Cassandra, MongoDB, and VoldemortDB.

These distributed big data frameworks are often not used in isolation but instead deployed by combining them into pipelines, which increases the size of the configuration search space that the optimization algorithm has to search, thus making the task more difficult.

2.1.2 Problem Formulation

The general problem of configuration optimization discussed in this thesis can be formulated as follows. Given a configuration x from the configuration search space X and the underlying objective function f, then $f(x)$ is the objective function value of a workload under consideration when executed using configuration x. The function f can represent an objective like execution time, execution cost or any performance metric of interest. The automatic configuration problem in the context of this thesis seeks to find x^* such that

$$\begin{aligned} x^* = & \arg\min_{x \in X} f(x) \\ \text{s.t.} \quad & c_i < t_i \quad \forall c \in C, t \in T \end{aligned} \tag{2.1}$$

Under this formulation, we will be minimizing objective functions (maximizing an objective function is an identical problem). Here, C and T are the set of constraints and their corresponding thresholds. Users can specify constraints on the value of the objective function or other metrics such that the x^* satisfies those constraints. For example, when optimizing for execution time, the user can specify that the execution cost should be lower than a specific value. Thus the goal of the optimization algorithm would be to find the configuration that has the lowest execution time but does not cost more than the user-specified execution cost constraint.

If a closed-form formula for the objective function is known, then we can use this mathematical formula to find the optimal configuration. However, since performance models of cloud workloads on different types of cloud instances and system parameter configurations are not generally available, we treat this setting as a black box. We model the unknown objective function f using black-box modeling techniques.

In the context of this thesis, X represents the space of possible cloud configurations or system parameter values. The specific configurations that X represents and any constraints considered in each research work in this thesis are defined in the respective chapter. We assume that a user has a limited optimization budget, i.e., the number of configurations that the optimization algorithm is allowed to test is limited. This is a common and practical assumption in existing research works on configuration optimization [12, 13, 14, 15].

2.1.3 Black box optimization algorithms

Black-box optimization (BBO) algorithms aim to optimize a given function in the absence of an analytical model of the system. BBO algorithms are used in cases where the analytical models are unavailable or intractable. The unknown optimization function f is considered a black box where only the relationship between input and output values of the function can be observed. Several algorithms are used for BBO; here, we briefly review a set of black-box optimization methods that we will use throughout this dissertation. These can be classified into three main classes: sampling-based (random search, LHS sampling), Sequential Model-Based (Bayesian Optimization variants, TPE), and search-based (Stochastic Hill Climbing and Simulated Annealing).

2.1.3.1 Random Search

The random search algorithm generates unique configurations by randomly sampling (without replacement) the configuration search space. In random search, new samples are generated without taking previously generated samples into account. Thus, the samples generated could be quite close to each other and concentrated in the same region of the search space in the worst-case scenario. However, random sampling provides the benefit of allowing users to generate new samples progressively, i.e., the number of to-be-tested configurations does not need to be defined ahead of time.

2.1.3.2 LHS Sampling

LHS samples the search space using a space-filling design. It generates near-random samples, but the difference between random sampling and LHS is that LHS simultaneously stratifies on all input dimensions. To generate N samples, each input dimension is split into N splits/strata of equal marginal probability $1/N$. Each stratum is then sampled once to generate N samples [16].

2.1.3.3 Bayesian Optimization (BO)

BO builds a surrogate for the objective function and quantifies the uncertainty in that surrogate model and then uses an acquisition function defined from this surrogate model to decide where to sample for the next optimization run. BO is particularly well-suited to global optimization problems where f is an expensive black-box function. In our setting, evaluating f entails deploying and running a workload in the cloud on the configuration to be tested, which is expensive.

BO reasons about f by starting with a prior $P(f)$. The prior is the initial probabilistic belief about the behavior of the objective function. This prior is based on the *surrogate method* which is used to build the surrogate model ($\mathcal{S}$) of the underlying unknown black-box function. Given the observations $\mathcal{D} = \langle x, f(x) \rangle$, it uses the Bayesian method to estimate a posterior distribution as:

$$P(f|\mathcal{D}) = \mathcal{S}(f; \mu_{f|\mathcal{D}}, \sigma_{f|\mathcal{D}})$$

where μ is the mean and σ is the variance or co-variance.

Surrogate methods:

There are many methods available to build a surrogate model; we focus on the most widely used methods: Gaussian Processes (GP) [17], Gradient Boosted Regression Trees (GBRT) [18], Random Forests (RF) [19] and Extra Trees (ET) [20].

Gaussian Processes (GP): Gaussian processes are generic supervised learning methods that can be used to solve regression as well as probabilistic classification problems. A GP defines a probability distribution over a set of functions based on the data available. A GP assumes that given a set of finite points x_1, ..., x_N, p(f(x_1), ..., f(x_N))) is jointly Gaussian with a mean function $\mu(x)$ and co-variance function $\kappa(x, x')$. The co-variance function defines the co-variance between points using a positive definite kernel function. Thus the unknown black-box function f can be approximated by:

$$f(x) \sim \mathcal{GP}(m(x), K(x, x')) \tag{2.2}$$

Gradient Boosted Regression Trees (GBRT): GBRT is a flexible non-parametric statistical learning technique. Just like GP, GBRT can be used for both regression and classification. Boosting works on the principle of ensemble by combining weak learners to deliver improved prediction accuracy. Gradient boosting involves three elements: 1) a loss function, 2) a weak learner to make predictions (regression trees in our case), and 3) an additive model (gradient descent) to add weak learners to minimize the loss function. The regression trees are built one at a time, sequentially. Each new tree helps to correct

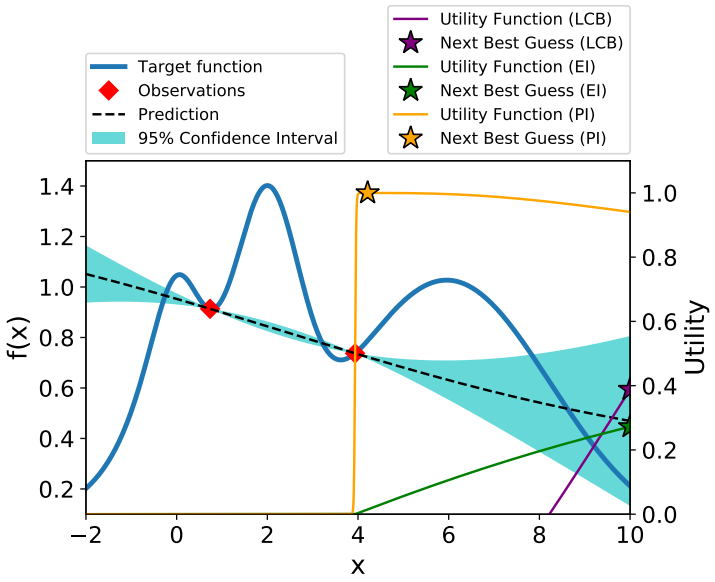


Figure 2.2: Example of an objective function being modeled by Bayesian optimization. The objective function is being minimized. The utility function for each acquisition function is shown along with the recommendation for the next point to evaluate, based on the highest value of the utility function.

errors made by a previously trained tree.

Random Forests (RF): In contrast to GBRT, RF use a bagging technique for ensemble learning. In bagging (or bootstrap aggregation), each model in the ensemble is created using a random subset of the original training dataset, which has the advantage of reducing variance. Unlike GBRT, where trees are built sequentially, in RF, each tree is built independently, and then the outputs of each tree are aggregated at the end.

Extra Trees (ET): ET (or Extremely Randomized Trees) introduce more variation into the ensemble. Unlike RF, all data is available for training of each tree and the split of each feature (to form a node in the tree) is chosen at random. Both features and splits are chosen randomly in contrast to RF, where only features are selected at random. This added randomness allows ET to have a lower variance than RF.

Acquisition functions:

In BO, the role of an acquisition function is to select the next configuration to test in order to update the surrogate model. The acquisition function can be interpreted as a function that evaluates the expected utility associated with updating the surrogate model at $\mathcal{S}(x)$. BO then selects the point with the highest expected utility, picking the most desirable point to evaluate f. Therefore, the acquisition function $A(x)$ can be represented as the expected utility $u(x)$, given the configuration x and the previous observations $\mathcal{D}$:

$$A(x) = \mathbb{E}[(u(x)|x, \mathcal{D})]$$

We now review the most commonly used acquisition functions and their utility functions. We first introduce them using a toy example. Figure 2.2 shows the surrogate model of an example objective function (shown in blue) built using two initial samples (observations). The utility function for each acquisition function discussed below is also shown in the figure. We can see that different acquisition

functions attribute completely different utility curves. The next best x to evaluate (illustrated as a star) is based on the highest value of the respective acquisition function.

Probability of Improvement (PI): Let f' be the minimum value of f observed so far; the PI function suggests a value of x that is most likely to improve upon this value, using the surrogate model $\mathcal{S}(x)$. The utility function for PI is represented as:

$$u(x) = \begin{cases} 0 & \mathcal{S}(x) > f' \\ 1 & \mathcal{S}(x) \leq f' \end{cases}$$

Expected Improvement (EI): The PI function provides a reward for improving upon the current minimum irrespective of the margin of improvement. Alternatively, EI also takes into account the amount of improvement. Hence, the utility function becomes:

$$u(x) = \max(0, f' - \mathcal{S}(x))$$

Thus, the reward is equal to the amount of improvement. Therefore, the EI acquisition function picks a point with the highest expected improvement.

Lower Confidence Bound (LCB): The LCB acquisition function takes the form:

$$A_{LCB}(x; \kappa) = \mu(x) - \kappa\sigma(x)$$

where $\kappa > 0$ is a trade-off hyper-parameter used to control exploration and exploitation. $\sigma(x)$ is the standard deviation and $\mu(x)$ is the mean. This acquisition function is also called Upper Confidence Bound (UCB), when used for maximizing an objective function.

GP hedge: GP hedge [21] chooses one of the three acquisition functions described above in a probabilistic fashion, based on gains g. Initially, gains are set to zero. At every iteration, each acquisition function is optimized independently to get a candidate point x_t. Out of these candidates, the next point x_{best} is chosen based on a softmax function $softmax(\eta g_t)$. After fitting the surrogate model with a new observation (x_{best}, y_{best}), the gain for each acquisition function is updated as $g_t = g_{t-1} + \mu(x_t)$. The hyper-parameter η controls the weights assigned to the gain for each acquisition function. This method is only available when using Gaussian processes for the surrogate model.

All the above acquisition functions provide a hyper-parameter to adjust the trade-off between exploitation (evaluating points with low mean) and exploration (evaluating points with high uncertainty).

2.1.3.4 Tree-structured Parzen Estimator (TPE)

Tree-structured Parzen Estimator and Bayesian optimization take a slightly different approach towards solving the same problem. While Gaussian process-based approaches model $p(y|x)$ directly (i.e., $y = f(x)$), TPE [22] models $p(x|y)$ and $p(y)$ separately. TPE essentially divides the sorted observations $\mathcal{D} = \langle x, f(x) \rangle$ according to the objective function value into two sets using a threshold y^*. Thus, TPE

defines $p(x|y)$ using two such densities:

$$p(x|y) = \begin{cases} l(x) & y < y^* \\ g(x) & y \geq y^* \end{cases}$$

where $l(x)$ is the density formed by observations that performed well and $g(x)$ is the density formed by observations that performed poorly. TPE algorithms choose y^* to be some quantile γ of the observed values y, such that $p(y < y^*) = \gamma$. The intuition is that good configurations will have a low probability in $g(x)$ and a high probability in $l(x)$. Thus, we can evaluate $\arg\min(g(x)/l(x))$ as the utility function for acquisition functions.

2.1.3.5 Stochastic Hill Climbing (SHC)

Stochastic hill climbing is a search-based method that does not rely on modeling the underlying black-box function. Stochastic hill climbing, as referred to in this thesis, works as follows. First the initial configurations X_0 are evaluated to get Y_0. The best configuration is picked from X_0 as the current configuration x_c. At each subsequent step, a random neighbor of the current configuration is generated. The current configuration (x_c) is then replaced by a random neighbor (x_n) given:

$$x_c = \begin{cases} x_n & p(x_n) \geq 1 \vee p(x_n) \geq \text{random}() \\ x_c & \text{otherwise} \end{cases}$$

where $p = e^{-(f(x_n)-f(x_c))/T}$. Here, T is a temperature value that is provided by the user. The formulation for p basically means that if $f(x_n) < f(x_c)$ then the neighbor is selected as x_c. Otherwise, the temperature hyper-parameter controls the probability of the neighbor being selected as x_c based on the difference between the objective function value of the two configurations. Having a higher temperature value enables configurations with a significantly worse objective function value than the current configuration can be selected. Therefore, a higher temperature allows for more exploration while a lower temperature allows for more exploitation. If $f(x_c)$ is less than the objective function value of the best configuration x_b (i.e., $f(x_b)$), then x_c is selected as the best configuration x_b. The algorithm terminates when one of the following two conditions are satisfied: a user-specified minimum is reached, or a specified budget is reached.

2.1.3.6 Simulated Annealing (SA)

Simulated Annealing is fairly similar to SHC, with the main difference being the introduction of a cool-down factor for the temperature T. In particular, for each step $i \rightarrow i+1$ a schedule constant α is used to lower the temperature as $T_{i+1} = \alpha T_i$, where $\alpha < 1$. Hence, simulated annealing lowers the temperature as the optimization progresses, therefore shifting the focus from exploration to exploitation later during optimization.

2.2 Literature Review

Our proposals are related to a range of existing research. In this section, we will survey the core set of related works, and, in each relevant chapter, we will contrast them against our work. We will divide the related works into two main categories as follows. Section 2.2.1 and 2.2.2 summarize existing proposals that are related to our work on cloud configuration optimization (Chapter 3, 4, 5). Section 2.2.3 survey work that is closely related to our work on parameter tuning described in Chapter 6.

2.2.1 Cloud configuration optimization

Several prior works have targeted the problem of cloud configuration optimization. In Ernest [23], the authors created an analytical model designed to predict the execution time of Spark jobs. They mainly target ML Applications. Elastisizer [24] uses a mix of black box and white box models to optimize cloud configurations as well as job-level configurations for map-reduce jobs. However, often the models presented in these works have limited general applicability.

Research works such as Cherrypick [12], Arrow [25], and Micky [13] treat the data analytics framework, such as Spark or Hadoop, as a black box. Cloud configuration optimization is done for the recurring jobs running on these frameworks using offline optimization trials. Cherrypick, Arrow and Micky use BO with GP, BO with ET, and a multi-arm bandit approach for optimization, respectively. Lynceus [26] uses Random Forests as a regression model that optimizes both cloud configuration and hyper-parameters for ML jobs with a budget-aware approach to reduce the monetary cost of the optimization process. Accordia [27] uses Gaussian Process with Upper Confidence Bound to tackle cloud configuration optimization for recurrent data processing workloads.

Scout [14], Selecta [28] and PARIS [29] can make use of historical data to quickly make cloud configuration decisions about new workloads. Unlike other works, Selecta also incorporates different storage options into the cloud configuration search space. These works require a significant amount of performance data for many workloads across different configurations. They then use some measure of workload similarity to determine good configurations for a new workload.

2.2.2 Resource allocation in data center environments

Resource configuration and allocation problems are not limited to cloud environments, but they are also present in data center environments. The big difference between cloud and data centers in this context is that there are a lot more options and heterogeneity in the cloud. Often data center resource allocation techniques balance resources across all the jobs running in the data center. This section discusses some of the relevant works targeted towards resource allocation in data center environments.

Paragon [30] is a data center heterogeneity and interference-aware scheduler. It uses collaborative filtering to classify an unknown incoming workload to assign it resources. Quasar [31] is a follow-up work on Paragon, which also uses collaborative filtering for unknown workload classification. However, this method requires an offline training set that is impractical to obtain when the configuration space is large.

Morpheus [32] provides high cluster utilization and predictable performance for enterprise clusters. It extracts SLOs implicitly and performs job placement and dynamic re-provisioning to achieve the SLO. Perforator [33] introduces a methodology to perform resource optimization for Hive. Several analytical models are designed and used in that work. However, those models are not directly applicable to our setting. DejaVu [34] is another work that tackles the problem of allocating resources to workloads in a data center setting.

In contrast to our setting, these works are designed for a data center environment. Thus, there are no cost constraints or explicit considerations towards heterogeneity, i.e., different types of machines.

2.2.3 System configuration optimization

There exists a large body of work on system configuration optimization or automatic parameter tuning, primarily in the fields of application servers, stream-processing systems, batch-processing systems, and databases.

iTuned [35] is a system for online parameter tuning of databases. It uses Bayesian optimization with Gaussian processes to find the best values for database parameters. iTuned's goal is to optimize for completion time, making it a single objective Bayesian optimization problem. OtterTune [36] also uses Gaussian processes with workload characterization to tune DBMS configurations. Another similar work that tackles parameter tuning for database systems is Rafiki [37]. Cao et. al. [38] provide a comparative analysis of a number of black-box optimization algorithms for the purpose of parameter tuning of storage systems. Metis [39] uses BO with GP and a customized acquisition function with data outlier removal to optimization parameters of a key-value store in order to tackle the challenges when using tail latency as the performance objective.

Dhalion [40] proposes a policy-based self-regulation system for Twitter Heron. Users can define policies comprising a set of symptoms, diagnosis, and resolution actions. Symptoms provide information about observed anomalies in the behavior of the stream processing system, which leads to the generation of a single diagnosis. As a result, a corresponding resolution action is carried out by the framework to bring the stream processing system back to its normal state. Thus, policies are essentially a flexible form of rule-based automation. With correct policies for each parameter and performance metric under consideration, their work can be used as an online tuning framework. Lin et al. [41] determined the degree of parallelism for each stage of the processing DAG based on the input data and the computational cost. Thus, they can adjust the parallelism level to accommodate the required throughput. Trotter et. el. use genetic algorithms and supervised learning to find optimal configurations for Apache Storm. Similarly, BO4CO [42] uses Gaussian processes for parameter tuning but in the context of stream-processing systems.

For optimizing batch processing jobs, MROnline [43] uses a hill-climbing algorithm along with custom rules to tune specific MapReduce parameters to optimize the average execution time of MapReduce jobs. Gunther [44] is a search method based on genetic algorithms to optimize parameters for MapReduce. Their primary goal is to decrease the execution time of the job and they focus on optimizing only for

one metric. In addition, there are other research works that tackle parameter tuning for Hadoop [45, 46] and Spark [47, 48].

Xi et al. [49] also used a hill-climbing algorithm to find appropriate configurations for application servers. Tao et al. [50] and Heinze et al. [51] used a recursive random search algorithm for parameter tuning in network protocols and an elastic scaling algorithm, respectively. Zhu et al. [52] proposed online tuning of parameters for high-data rate sensing and computer vision applications by learning a cost-based model using a parameter dependency graph. Cost-based models have also been used in parameter tuning for database systems as in [53]. However, cost-based models can have difficulty in accurately predicting performance across different types of hardware and application versions. For example, changes to the architecture of a database might render its cost-based model inapplicable.

Several surveys [54, 55, 56] provide an overview of several black-box optimization algorithms, including the ones discussed in this work. Tools like Vizier [57], BestConfig [58] and ParaOpt [59] provide general-purpose framework for parameter tuning. ConfAdvisor [60] uses what-if rules to provide adaptive configuration advice for container workloads running on kubernetes. Lastly, tools such as LearnConf [61] can be used to determine the most important system parameters that impact the performance and guide the tuning.

Chapter 3

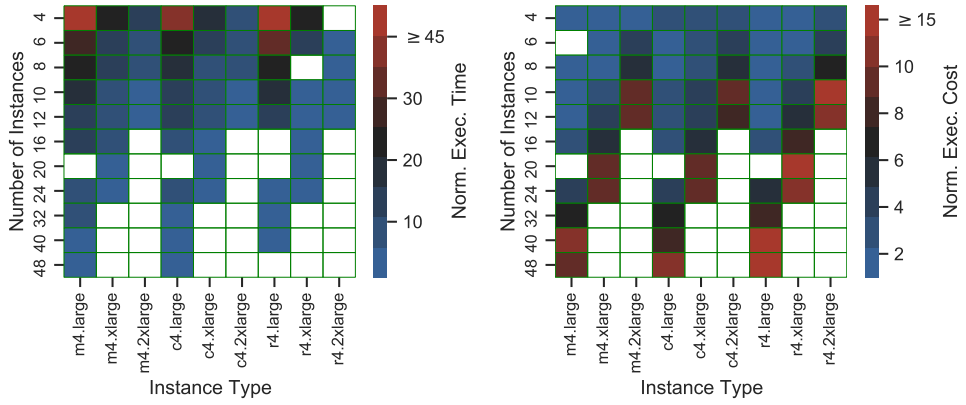
Evaluation of Black Box Algorithms for Optimizing Cloud Workloads

This chapter evaluates the performance of several black-box optimization algorithms to find good cloud configurations for distributed data processing workloads automatically. We compare these algorithms across several workloads and for two different objective functions to find the best algorithms for different scenarios. This analysis may enable researchers to select the best black-box optimization algorithms to use as a base for their optimization frameworks. Furthermore, they can add enhancements on top of these best generic algorithms for further improvements in the overall performance of their optimization frameworks.

3.1 Introduction

Different applications and workloads have different behavior and resource requirements, and therefore their optimal cloud configuration (number of instances and type of instances) is not the same. Furthermore, choosing the right configuration is crucial not only for cost efficiency but also to meet service-level objectives on the completion time for these jobs. To illustrate this, Figure 3.1 shows the heatmap of the normalized execution time and execution cost of two workloads on a cloud configuration search space. The configurations shown in orange are those with more than an order of magnitude higher runtime or execution cost compared to the best configuration in the search space. This highlights that choosing a bad configuration can lead to significantly higher execution time and execution cost than choosing the best configuration (up to 47 and 12.5 times higher, respectively).

Exhaustively testing the workload on all possible cloud configurations is very expensive and wasteful. This has recently led to several proposals for automatically finding a close to optimal configuration, quickly and at a low cost. Cherrypick [62], PARIS [29], Scout [14], Micky [63], and Arrow [25] are just some examples of approaches in this space (as we surveyed in Chapter 2). Each of these proposals employs a specific black-box optimization algorithm, such as random forests, Bayesian optimization with Gaussian processes, or Bayesian optimization with extra trees. By leveraging generic black-box



(a) Execution time for reg_{H}

(b) Execution cost for lr_{H}

Figure 3.1: (a, b) Execution time and execution cost for two selected workloads in DS_1 (see Table 3.2) shown as a heatmap normalized by the best configuration for each respective metric. The blank boxes are configurations that were either not present in the dataset or do not execute the workload successfully.

models, these systems avoid creating an analytical model, which would require capturing the complex relationship between the execution time of an application, the resources of the cloud instances, and the input workload.

Existing work on cloud configurations focuses on applying specific black-box optimization algorithms to solve the problem but falls short of providing a complete characterization of which algorithm is more suitable in which situation. As a consequence, it is not clear whether the optimization performance of these prior works is based on the selection of the optimization algorithms or on additional design decisions in their approaches. For example, Cherrypick uses Bayesian optimization with Gaussian processes while Arrow uses Bayesian optimization with extra trees. But it is not clear how much of the performance gain is because of the choice of the optimization algorithm itself, if any. A similar question arises when determining the advantages of techniques like augmenting Bayesian optimization with low-level performance metrics [25] or augmenting the search space with some prior knowledge about the performance of a similar workload [14].

This chapter fills this gap by performing an exhaustive comparison of 8 black-box optimization techniques using a total of 23 different workloads and two different search spaces with 69 and 140 different cloud configurations, respectively. Our evaluation highlights how searching for a cloud configuration has some peculiar characteristics that are different from other applications of black-box modeling, for example, hyper-parameter search [64]. On the one hand, running a cloud configuration setting is very expensive. It requires starting up a cluster, deploying the application, and running the workload. Therefore, it is typically only feasible to explore a limited number of solutions. On the other hand, the search space is relatively small, since a configuration consists of selected combinations of instance type and cluster sizes, and both parameters – instance type and the number of instances – have a relatively small range of possible values in practice. Furthermore, these parameters are integer and categorical, as opposed to continuous parameters.

Another contribution of this chapter is to produce an analysis of different variants of Bayesian Opti-

mization (BO), which is the most common technique used by existing work. Our analysis shows that, in several cases, BO with Gradient Boosted Regression Trees (GBRT) outperforms the methods used in prior research. Additionally, while prior work employs Expected Improvement (EI) as an acquisition function, we found the Probability of Improvement (PI) to be a better alternative for cloud configuration.

Overall, our results show that BO outperforms other algorithms that we analyzed in our empirical study. In terms of its variants, BO with Gaussian Processes (GP) and BO with GBRT provide the best performance when optimizing for execution cost and execution time, respectively. BO with GBRT provides up to 20% better execution time and BO with GP is able to provide up to 40% better execution cost. Unlike prior work, we also consider running optimization in an online mode, where different configurations are tested in the initial runs of a production setting with an upper bound on the execution time. We perform a break-even analysis to see when the cost of finding an optimal configuration can be amortized. Lastly, we provide a decision workflow for users to select an appropriate optimization algorithm based on their requirements.

Our experimental data and the implementation of black-box optimization algorithms are available [65] to facilitate further research on cloud configuration optimization.

3.2 Background and Related Works

3.2.1 Problem Formulation

In this chapter, the configuration x is a cloud configuration that is represented as a tuple $x = (N, I_F, I_S)$ where N is the number of instances, I_F is the instance family and I_S is the instance size. Instance family and instance size combine to form a particular instance type that can be requested from the cloud provider.

An instance type defines the CPU, memory, and storage capabilities of an instance. To define instance types, we follow the convention in use at many cloud providers that group their instance types based on their instance family (e.g., general-purpose, compute- or memory-optimized) and instance size (e.g., large, xlarge, 2xlarge). The instance family expresses the class of hardware specifications that may best meet the requirements of different applications as well as a CPU-memory ratio. The instance size, in turn, allows for choosing the number of virtual CPUs, the available memory size, local storage, and network bandwidth. We assume that there is a finite set of instance types and a finite number of choices for the number of instances. The set of all possible combinations of instance types and number of instances yields the *configuration search space*.

Thus, for this chapter, Equation 2.1 can be simplified into the following, where f is either execution time or execution cost:

$$x^* = \arg\min_{x \in X} f(x) \tag{3.1}$$

Figure 3.2 shows an example of the sequence of configurations explored in a search space for cloud configuration. The x-axis has the instance types (available in AWS) and the y-axis has the number of nodes in the cluster. The heatmap colors represent normalized runtime w.r.t. the best execution time in

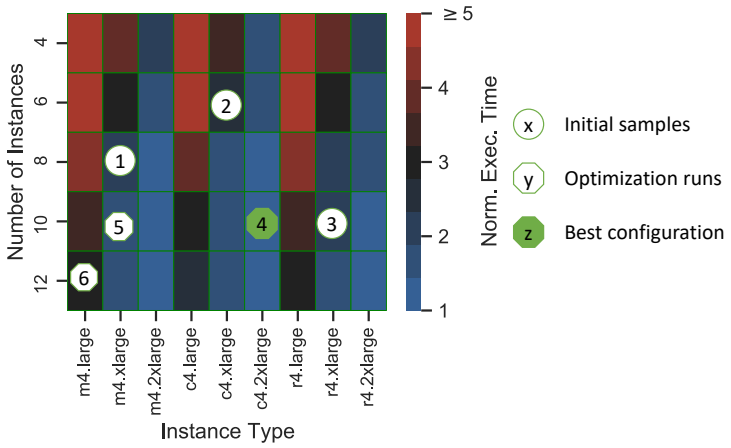


Figure 3.2: Example search space and optimization samples.

Table 3.1: Prior work, corresponding optimization algorithms and cloud configurations that they optimize.

Prior work	Black-Box method	Target Configurations
Ernest [23]	N/A (White-Box Model)	Cluster size
Cherrypick [62]	BO with GP	Instance type and Cluster size
PARIS [29]	Random Forests	Instance type
Arrow [25]	BO with ET	Instance type
Micky [63]	Multi-armed bandit	Instance type
Scout [14]	Custom search	Instance type and Cluster size

the search space: configurations with darker colors are closer to the best configuration (which is shown with the green hexagon). Most optimization algorithms start with a set of initial samples (shown with white circles) and then proceed with performing some optimization runs (shown with white hexagons), eventually reaching, or at least approaching, the best configuration in the search space.

3.2.2 Related works

In Chapter 2 (§2.2) we have summarized the prior research works in the area of automatic cloud configuration. Table 3.1 outlines the optimization algorithms that they use and the cloud configurations that they optimize for. In general, prior works compare against random search or coordinate descent or one other method but exclude most other alternative black-box optimization algorithms. Our work is aimed at filling this gap. In particular, our research complements these prior works by evaluating the effect of a larger number of alternative black-box methods and by devising criteria to select the best method in different circumstances.

3.3 Optimization algorithms

There is a wide variety of black-box optimization (BBO) algorithms that have been discussed and defined in the literature [64]. Most of these were created for hyper-parameter optimization. In this part of the

thesis, we pick a subset of these optimization algorithms based on both their popularity and, particularly, their optimization budget requirements. For example, we avoid population-based methods since they require a higher number of optimization steps.

The black-box optimization methods we evaluate empirically in this chapter can be classified into three main classes: Sampling-based (Random Search), Sequential Model-Based (Bayesian Optimization variants, TPE), and Search-based (Stochastic Hill Climbing and Simulated Annealing). These algorithms are listed below, and their detailed explanation can be found in § 2.1.3:

- Grid Search
- Random Search
- Bayesian Optimization using:
  - Gaussian Processes
  - Extra Trees
  - Random Forests
  - Gradient Boosted Regression Trees
- Tree-structured Parzen Estimator (TPE)
- Stochastic Hill Climbing (SHC)
- Simulated Annealing (SA)

Excluded BBO methods. The literature on black-box algorithms is vast and there are numerous black-box algorithms used for hyper-parameter optimization that we have not included in this study. In particular, CMA-ES [66] and other evolutionary algorithms such as GGA [67] and particular swarm optimization [68] are not suitable for the cloud configuration problem because of the higher budget required for evolutionary algorithms. Hyperband [69], which is essentially strategic random sampling, is designed for cases where only a small subset of the full dataset can be used for evaluating a configuration. Thus, it is better suited in cloud configuration scenarios where a single best configuration for multiple input datasets is the optimization goal.

3.4 Experimental methodology

To experimentally compare the black-box optimization algorithms mentioned in §3.3, we apply them to optimize the cloud configuration of a large set workloads. We describe our methodology in this section and present the results in the next section.

Our methodology is carefully designed to establish the statistical significance of the results and achieve a fair comparison across algorithms. We proceed as follows:

Decouple optimization from profiling: We compare optimization algorithms on the basis of their performance on pre-collected profiling datasets comprising the execution time of every workload for every configuration in the search space. Since our focus is exclusively on the comparison of optimization algorithms, for a given configuration, we hold its execution time constant and thus, we avoid biasing our results from the effects of the many sources of performance variability in cloud environments. Further, this enables us to determine the best configuration in the search space.

Repetitions: For each workload, we run every optimization algorithm 50 times, each time picking a different random set of initial samples. Repeating the optimization process multiple times is crucial to assess the overall optimization performance because the performance of many optimization algorithms depends on the initial set of random samples.

Initialization consistency: The same set of initial samples is used across all optimization algorithms in every repetition of an optimization run. This prevents differences in the initial sample sets from adding variance to the results across different optimization algorithms and is the basis for a fair comparison thereof.

Statistical significance: We use statistical significance tests to determine whether the mean values of the performance metrics (see below) vary in a meaningful way from one algorithm to another. In particular, we use the independent t-test [70] (with $p \leq 0.05$) from the family of parametric significance tests.

The remainder of this section provides further details regarding the algorithm implementations, datasets, and performance metrics.

3.4.1 Implementation

We implement the black-box optimization algorithms atop several widely used libraries. BO methods are built using the SkOpt library [71], since it provides a range of surrogate models and acquisition functions; TPE is implemented using the Hyperopt library [72, 73]; SHC and SA are based on the Solid library [74]. To provide a comparison on common ground, we modified the Solid library such that SHC and SA start from a set of n initial samples, instead of a single sample that Solid originally allows. We modified the optimization libraries to provide different methods with the same initial samples for a given repetition of the optimization process.

Table 3.2: Workloads and configuration search space for each dataset.

Dataset	**Workloads** [75]			**Configuration Search Space**		
	Framework	**Applications**	**Input Size**	I_F	I_S	**N**
DS_1	Hadoop	Pagerank (hpr), Terasort (ts), Wordcount (wc)	Huge (H), Bigdata (B)	m4, c4, r4	large	4, 6, 8, 10, 12, 16, 24, 32, 40, 48
	Spark 1.5	Kmeans (km), Naive-Bayes (nb), Regression (reg)			xlarge	4, 6, 8, 10, 12, 16, 20, 24
	Spark 2.1	Join (jn), Pagerank (spr), Logistic Regression (lr)			2xlarge	4, 6, 8, 10, 12
DS_2	Spark 2.4.4	Linear Regression (lnr), Latent Dirichlet Allocation (lda), Random Forest (rf)	Huge (H), Gigantic (G)	m5, m5a, c5n, c5, r5	large	16, 24, 32, 40, 48, 56, 64
					xlarge	8, 12, 16, 20, 24, 28, 32
					2xlarge	4, 6, 8, 10, 12, 14, 16
					4xlarge	2, 3, 4, 5, 6, 7, 8

3.4.2 Datasets

Table 3.2 shows an overview of the workloads and configuration search spaces of the two datasets used in this chapter, for a total of 23 workloads.

DS_1 is provided by the authors of Scout [76]. This dataset includes execution time information on a set of multi-node workloads, for a search space comprising of 69 configurations on which each workload has been executed. There are a total of 18 workloads comprised of 9 different applications and 2 input

data sizes per application. The applications consist of jobs executing within Spark or Hadoop as listed in Table 3.2. In DS_1, the best execution time and execution cost are within 150-2171s and 0.11-2.45 dollars, respectively.

To evaluate the performance of the optimization algorithms over a larger cloud configuration search space, we generated a second dataset (DS_2) by profiling runs of 5 additional workloads (based on 3 additional applications) in a configuration search space of 140 configurations. The size of this search space is more than twice the size of the search space used in Scout [76] (which had 69 configurations) and Cherrypick [62] (66 configurations). To reach this larger configuration search space, DS_2 includes execution time information for 20 instance types and 7 cluster sizes for each instance type, as shown in Table 3.2. For DS_2, the best execution time and execution cost are within 114-324s and 0.09-0.64 dollars, respectively.

The workloads for both datasets were obtained from Intel's HiBench big data benchmark suite [75]. The details about the characteristics of each input size (Huge, Gigantic, and Bigdata) as well as the applications are included in the public repository of HiBench [75]. Note that Random Forest failed to run successfully on a lot of cloud configurations in our search space with Gigantic input size and it is thus excluded.

Our experiments end up covering 63-69 configurations (out of 69) for DS_1 and 122-140 configurations (out of 140) for DS_2. The majority of optimization algorithms explore all configurations across 50 repetitions.

3.4.3 Evaluation metrics

Since we use 23 different workloads (18 in DS_1 and 5 in DS_2) for our evaluation, we must resort to aggregate metrics that summarize the results. In particular, we use the following two metrics in our evaluation to determine the performance of the optimization algorithms. Hereby, *objective function value* refers to either execution time or execution cost. Comparisons are made using the minimum of the objective function value (as per Eq. 2.1) within the given optimization budget.

Performance Score: The performance score is calculated by performing pairwise comparisons between all optimization algorithms. If the difference in the mean of the best objective function values achieved by two optimization algorithms is statistically significant, then the performance score of the algorithm with the lower objective function value is incremented. The pairwise comparison is performed for each optimization budget level and across all workloads. The performance score is aggregated across optimization budgets and workloads in each dataset.

Normalized Performance: The performance score provides an overall relative ranking of the optimization algorithms. Thus it helps us determine the best optimization algorithm in a given scenario. However, it does not give information about the actual difference in the best objective function value achieved by an algorithm compared to others. For that, we use normalized performance, which is the best objective function value of each optimization algorithm normalized w.r.t. the best algorithm according to the performance score. If there is no statistically-significant difference between the performance w.r.t. the best

algorithm, then both are assumed to have the same performance.

3.5 Experimental Results

Since we have a large number of optimization algorithms and choices for their hyper-parameters, we first narrow down the field of optimization algorithms to compare and then perform the comparison between those optimization algorithms using different scenarios. Thus, we split our analysis into three main parts: (i) the selection of hyper-parameter values for the optimization algorithms, (ii) the evaluation of BO variants, and (iii) the evaluation of selected optimization algorithms.

Henceforth, we use the following notation to represent a particular workload based on Table 3.2: $\mathsf{application}_{\mathsf{input\_size}}$. For instance, lr_{H} represents the linear regression application running with Huge input data size (preset size in HiBench).

The comparisons are performed across 5 different optimization budgets (6, 12, 18, 24, 30). The results are based on 50 repeated runs of the optimization algorithms for each workload using different randomized initial sample sets for each run.

To present results succinctly, we omit plots of performance score values. Instead, in any given scenario, we use the performance scores to determine the best algorithm. Then, we illustrate performance comparisons using normalized performance graphs (see Figure 3.4 as a reference). This kind of graph shows the normalized mean performance of algorithms w.r.t. the best algorithm as a bar plot associated with the left y-axis while the normalized performance of the best algorithm w.r.t. the best configuration in the search space is shown as a line plot associated with the right y-axis of the graph. The error bars represent 95% confidence interval (after statistical significance is accounted for during normalization).

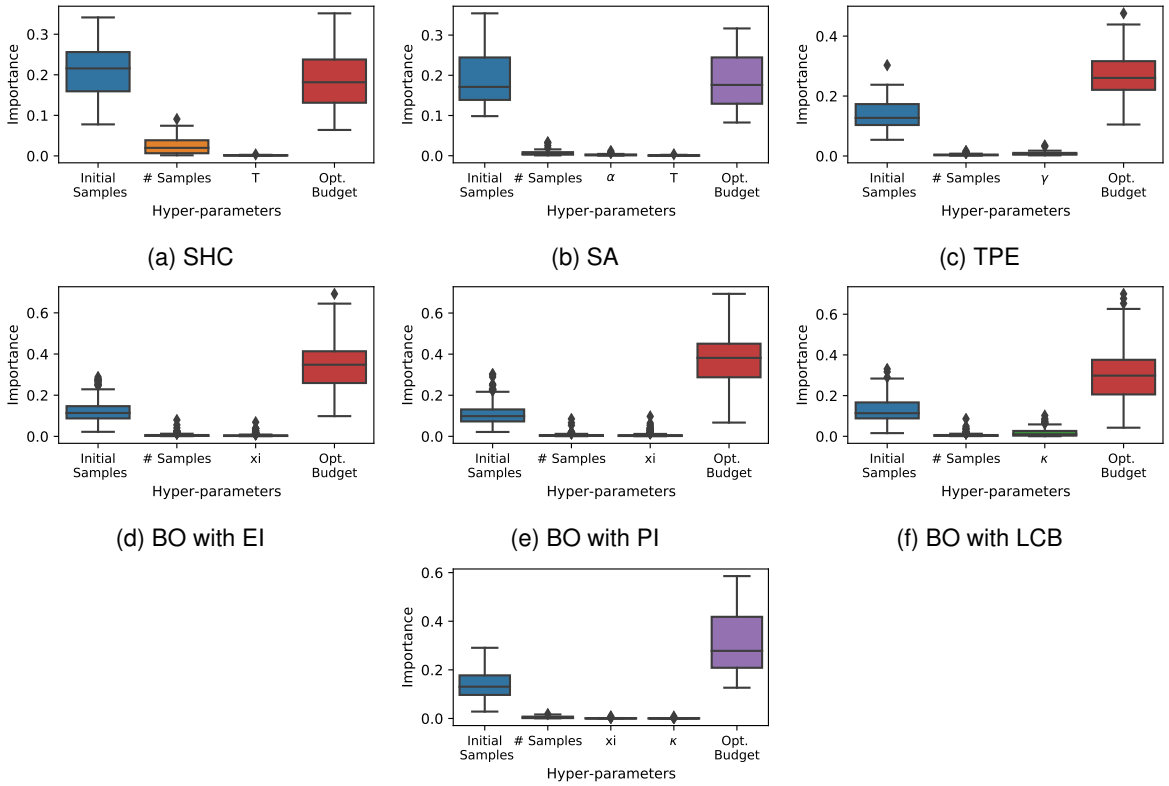
3.5.1 Configuring optimization algorithms

Every run of an optimization algorithm in a given scenario is associated with a certain configuration that comprises of the following hyper-parameters:[1] the number of initial samples, the seed to the random generator of initial samples, the optimization budget, and the algorithm-specific hyper-parameters (see Table 3.3). We now analyze the importance and impact of different hyper-parameter values across algorithms. We use performance scores to pick the best-performing hyper-parameter values for each algorithm.

3.5.1.1 Importance of hyper-parameters

We use Functional ANOVA [77] to quantify the importance of each hyper-parameter towards the performance variance of the optimization algorithm. Note that, in the context of Functional ANOVA, performance variance refers to the variability in the optimization's output – the (N, I_F, I_S) tuple – in terms of

[1] We use the term hyper-parameter to distinguish from the cloud configuration *parameters* (N, I_F, I_S), which are the output of the optimization process.



(a) SHC (b) SA (c) TPE

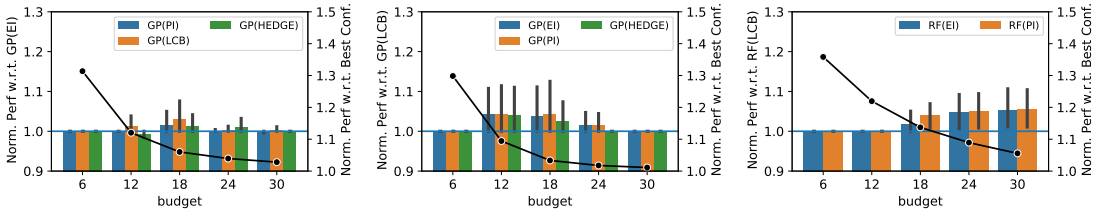
(d) BO with EI (e) BO with PI (f) BO with LCB

(g) BO with GP Hedge

Figure 3.3: Importance (through Functional ANOVA) of each hyper-parameter of the optimization algorithms.

the effects of variation due to different hyper-parameter. For each algorithm, in addition to the algorithm-specific hyper-parameters, we consider the importance of the initial samples selected (via random seed), the number of initial samples, and the optimization budget.

Figure 3.3 shows the fraction of variance (i.e., *importance*) in the performance introduced by each hyper-parameter for different optimization algorithms across all 23 workloads and for both objective functions (execution cost and execution time). For BO, the results also include the importance across all surrogate models.



(a) BO(GP) on workloads in DS_1 (b) BO(GP) on workloads in DS_2 (c) BO(RF) on workloads in DS_2

Figure 3.4: Comparison of acquisition function for different Bayesian optimization algorithms. All three cases shown here are scenarios in which the objective function is execution cost. In all other scenarios, there is less than 5% difference in the performance of different acquisition functions.

The individual contribution of algorithm-specific hyper-parameters towards variance in performance

Table 3.3: Algorithm-specific hyper-parameters and their values. For BO with LCB acquisition function κ is used; for EI and PI, ξ is used.

Algorithm	Hyper-parameters	Values
SHC	temperature (T)	50, 100, 250, 500, 750, 1000
SA	temperature (T)	50, 100, 250, 500, 750, 1000
	schedule constant (α)	0.3, 0.5, 0.7, 0.9
BO	kappa (κ) (LCB)	1.0, 1.5 1.96, 3.0, 5.0
	ξ (EI, PI)	0.01, 0.05, 0.1, 0.2
TPE	Threshold (γ)	0.1, 0.25, 0.4, 0.55, 0.7

is small compared to those of initial samples (via random seed) and optimization budget. On average, algorithm-specific hyper-parameters contribute to less than 5% to the overall variance. The exception is κ for BO with LCB acquisition function, in which case κ can contribute up to 10% to the variance in performance. On the other hand, the selection of initial samples and the optimization budget contributed up to 65% to performance variance, individually. The rest of the contribution is made by the pairwise marginals, which, for readability, are not shown in these plots.

3.5.1.2 Best algorithm-specific hyper-parameters

Table 3.3 shows the algorithm-specific hyper-parameters of different optimization algorithms and their respective values used to determine the best hyper-parameter settings. To find the best hyper-parameter setting, we evaluate all possible combinations of hyper-parameter values and the number of initial samples, for each algorithm on each workload and objective function. Using performance scores, we determine the best hyper-parameter values for each optimization algorithm across all workloads in a particular search space (DS_1 and DS_2) and for a particular objective function (execution time or execution cost). We see that the best hyper-parameter values change based on the workloads and type of objective function: there is no one winner in every scenario. For brevity, we list all of these values separately in the project repository [78].

In the remainder of this section, we present results based on the best hyper-parameter values as determined by the performance scores. Since in a real setting, tuning algorithm-specific hyper-parameters would require additional optimization runs and that might not always be practical or feasible, we also present results using selected default configuration values for the optimization algorithms. In our implementation, SA and SHC are the only algorithms that do not have a default value for their temperature and schedule constant hyper-parameters. Therefore, we use the default value of 0.5 for α and 50 for T. These values are based on overall performance across multiple scenarios in our results. These are not the best hyper-parameter values in every case but they work well for most cases in our experiments, among the limited set of hyper-parameter values we tested for α and T. In contrast, for BO and TPE, there exist default values either provided in the libraries or recommended in literature: HyperOpt uses a default value of 0.25 for γ, and the SkOpt library uses a default value of 0.01 for ξ and 1.96 for κ. By default, we use these values.

For the default number of initial samples, we use 3 initial samples for BO methods and 9 for the rest of the algorithms based on the hyper-parameter evaluation. This means that for an optimization budget

≤ 9, SHC, SA, and TPE are essentially equivalent to random sampling.

Takeaways: Across the algorithm-specific hyper-parameter values that we have tested, the hyper-parameters individually only contribute between 1-10% towards the performance variance of the optimization algorithms, based on a Functional ANOVA analysis. Additionally, there is no one-size-fits-all setting for the hyper-parameters of these optimization algorithms. This means that while algorithm-specific hyper-parameter tuning can provide some benefits, the effort for tuning these hyper-parameters might not be worth the marginal expected gains.

3.5.2 Narrowing down the choices for BO

We have several choices for acquisition functions as well as methods to build the surrogate model for Bayesian optimization. In this section, we narrow down our choices for the best candidate(s) among the BO variants. In this set of experiments, we fix the values for the hyper-parameters for each BO variant to the best values derived from the evaluation in the previous section.

3.5.2.1 Best acquisition function for BO

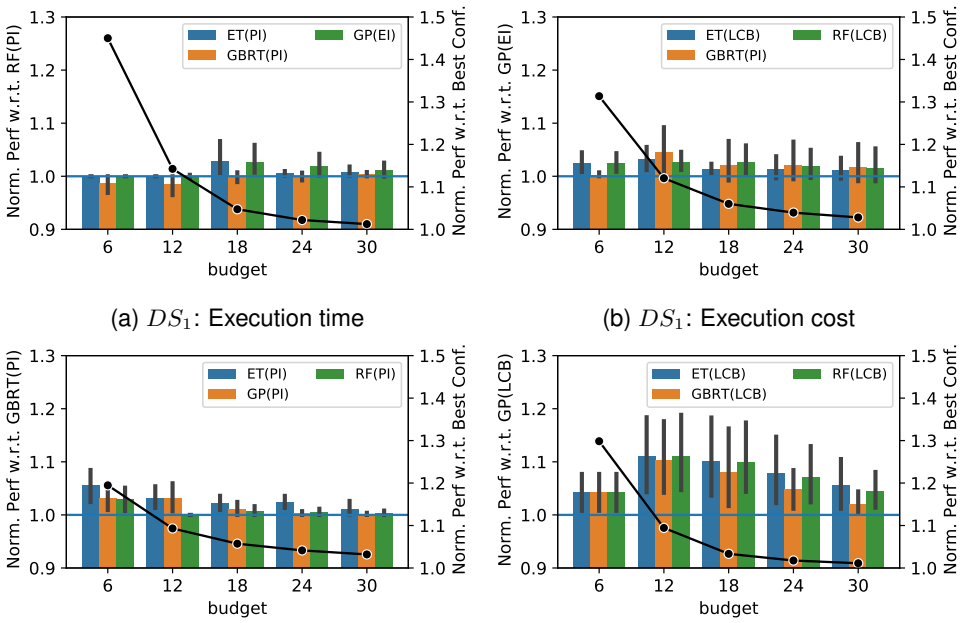
First, we evaluate the best acquisition function for each available method to build surrogate models. As mentioned before, acquisition functions are used to select the next configuration to test at each step in the optimization process. When using ET, GBRT, and RF as surrogate models, we have three choices for acquisition functions: EI, PI, and LCB. When using GP as a surrogate model, we have the additional choice of using GP hedge [79].

We compare all the choices of acquisition function for each BO method on the workloads in both datasets and for both objective functions. Then, we use the performance score to determine the best candidate for acquisition function and then normalize the performance of other acquisition functions based on that best candidate. Figure 3.4 presents only three cases in which there is a significant difference in the performance of different acquisition functions. The figure shows the performance of different acquisition functions normalized to the best acquisition function in the respective scenarios (left y-axis). Additionally, the performance of the best acquisition function w.r.t. the best configuration in the search space is shown as the line plot (right y-axis).

In all three scenarios shown in Figure 3.4, only one workload leads to a significant performance difference. When optimizing for execution cost on DS_1 with GP surrogate model (shown in Figure 3.4a), using the EI acquisition function can lead up to 25% lower execution cost when optimizing for km_B. In Figures 3.4b and 3.4c, we see the gap in performance because of a different workload, namely lda_H, where using LCB results in up to 17% and 14% better performance than other acquisition functions. In all other scenarios, the difference between the performance of acquisition functions is less than 5%.

These results suggest that the use of EI as the acquisition function is not always the best option, but in our experiments the difference was significant only in a few cases. Henceforth, for each scenario, we use the acquisition function with the best performance score for that scenario.

Takeaways: There are only a handful of instances where there is a statistically significant difference



(a) DS_1: Execution time

(b) DS_1: Execution cost

(c) DS_2: Execution time

(d) DS_2: Execution cost

Figure 3.5: Comparison of different Bayesian optimization algorithms.

in the performance of different acquisition functions. Generally, the performance difference is less than 10%, except for three cases out of 46 optimization scenarios where the difference in the performance between the best acquisition function and the rest was 14–25%.

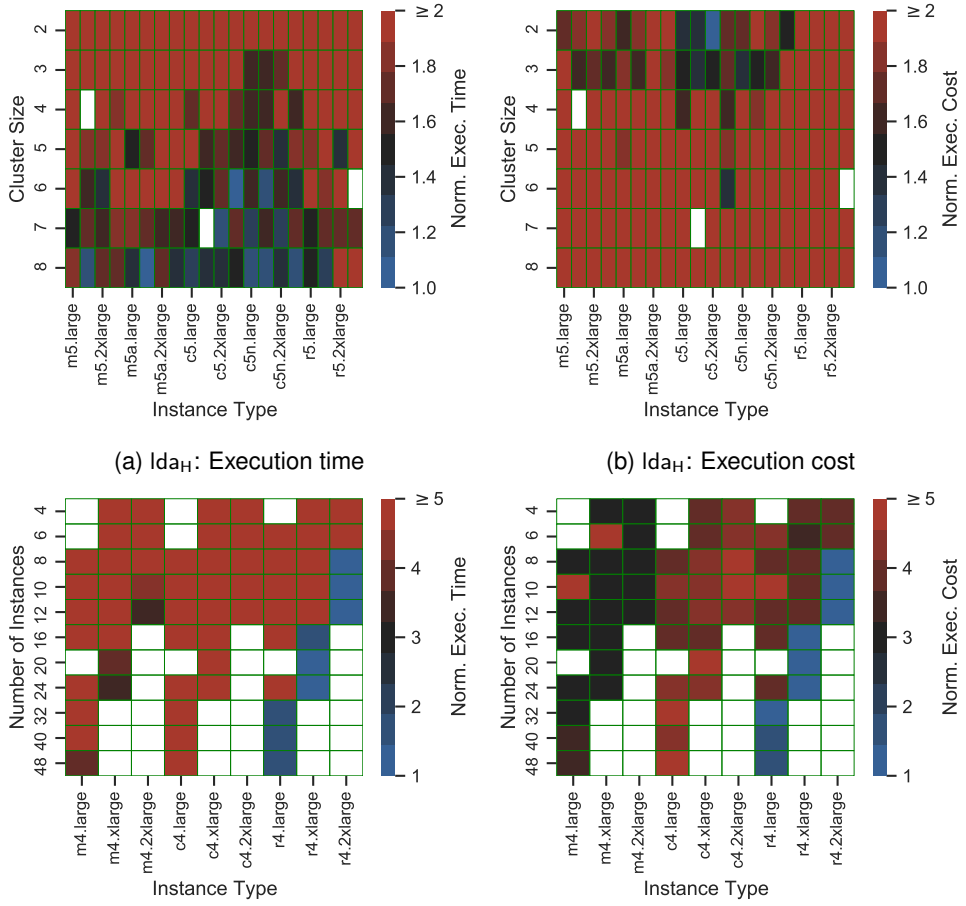
3.5.2.2 Best BO variant

Now that we have determined candidates for the acquisition function, we will compare the performance of different methods to build surrogate models using the chosen acquisition function. The normalized performance for different BO variants in different optimization scenarios is presented in Figures 3.5. Using the performance score, we determine that RF(PI) and GBRT(PI) are the best algorithms to optimize for execution time for DS_1 and DS_2, respectively. GBRT outperforms RF on lower budget for DS_1 but, on average, has a slightly higher minimum objective function value than RF(PI) for high optimization budget (≥ 24). On the other hand, When optimizing for execution cost, BO with GP performs better than other BO variants, with LCB and EI as the best acquisition functions for DS_2 and DS_1, respectively.

When optimizing execution time for DS_1 (Figures 3.5c RF(PI) provides up to 25% better execution time compared to GP and ET but overall it is comparable to GBRT(PI). For DS_2 (Figure 3.5a) using GBRT(PI) provides upto 9% better execution time, essentially when optimization budget is small (≤ 12).

However, choice of model is more important when optimizing for execution cost (Figures 3.5d and 3.5b). GP(EI) and GP(LCB) provides up to 22% and 26% lower execution for DS_2 and DS_1, respectively. The workloads where the choice of BO variant is most important are lda_H (in DS_2) and km_B (in DS_1).

Figure 3.6 shows the normalized execution time and execution cost of different configurations in the search space for the lda_H and km_B workloads. By observing the values of the objective function on the search space for these two workloads, we recognize that for these workloads, very few configurations in



(a) lda_H: Execution time

(b) lda_H: Execution cost

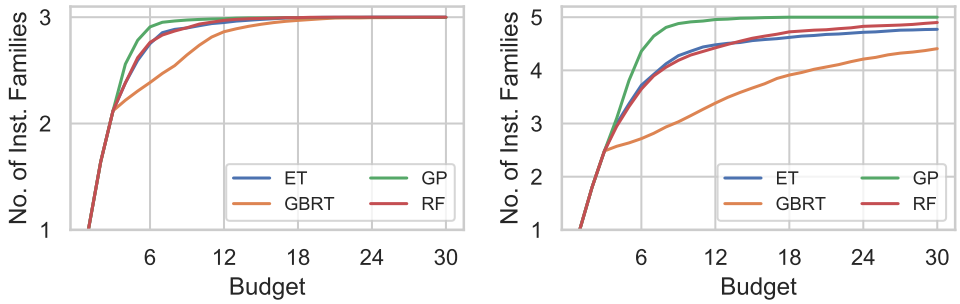
(c) km_B: Execution time

(d) km_B: Execution cost

Figure 3.6: Execution time and execution cost of different configurations of lda_H and km_B. In (a) and (b), the actual number of nodes can be obtained from the cluster size using multiplying factors of 1, 2, 4, 8 for 4xlarge, 2xlarge, xlarge and large instance sizes, respectively.

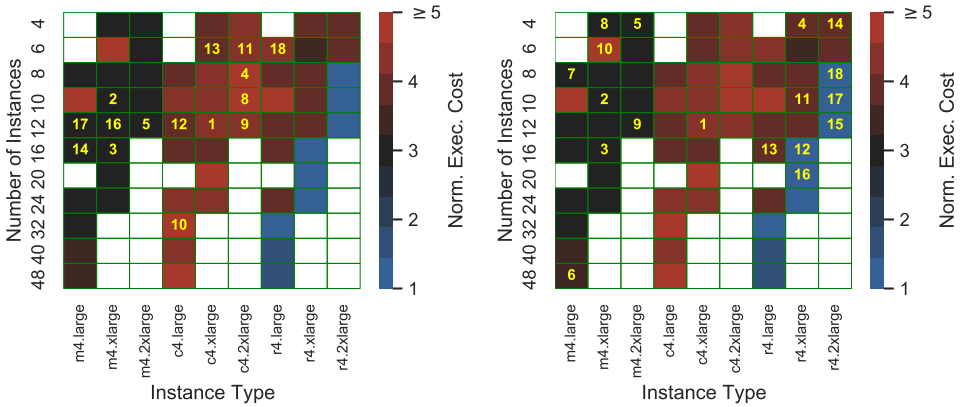
the search space are close to optimal in performance. Additionally, these handful of configurations that are close to optimal are in the same instance family. For lda_H, there are 10 configurations within 20% of the lowest execution time in the search space, but, when considering execution cost, the second-best configuration has 20% higher execution cost. In this case, we can conclude that optimizing for execution cost is much harder than optimizing for execution time and those are the scenarios in which BO(GP) performs better. By observing how BO(GP) explores the search space in this case, we deduce that compared to other BO variants, GP initially performs more exploration than exploitation and often tests cloud configurations with different instance families early on during the optimization process. Figure 3.7 shows the average number of instance families explored by each algorithm for workloads in both datasets when optimizing for execution cost. We can see that BO(GP) samples at least one configuration from each instance family more quickly than other BO variants. This leads BO(GP) to find a potentially better configuration when initial samples do not have good configurations in them. It also makes BO(GP) less susceptible to taking several steps near local minima.

The latter is exactly why GP performs better than GBRT on km_B when optimizing for execution cost. Figure 3.6c and 3.6d show that good configurations for execution cost and execution time are pretty



(a) DS_1 (b) DS_2

Figure 3.7: Number of instance families explored by GBRT and GP when optimizing for execution cost.



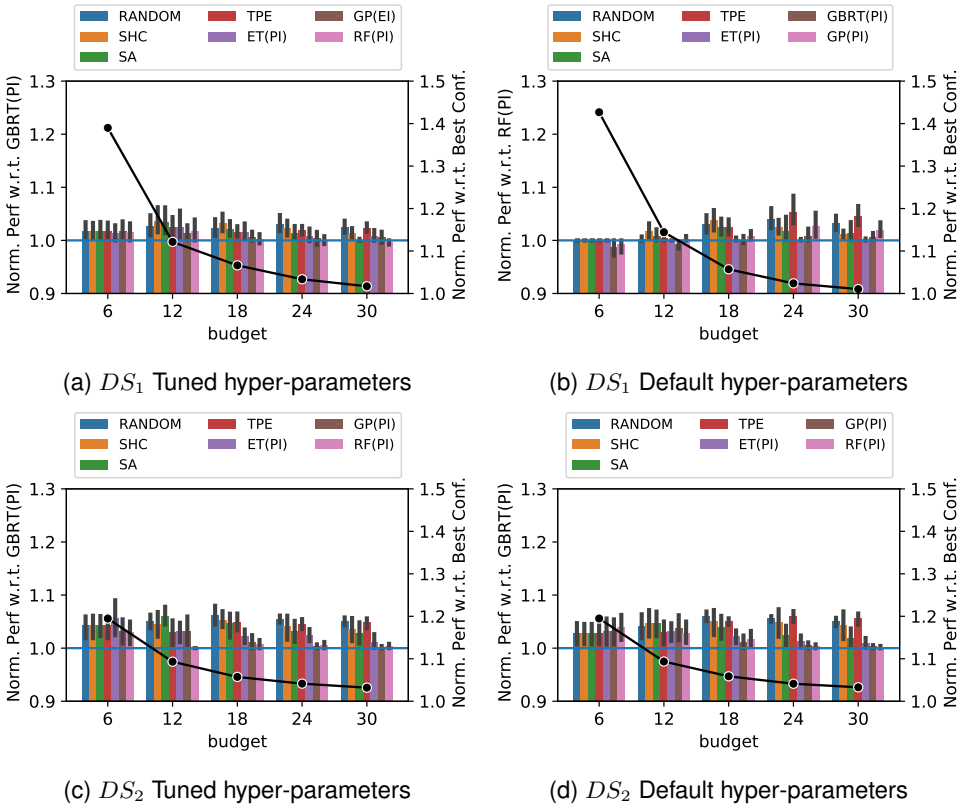
(a) BO(GBRT) (b) BO(GP)

Figure 3.8: Configurations tested by GBRT and GP when optimizing for execution cost km_B.

similar in the search space. However, the key difference is the fact that when optimizing for execution cost there is a large local minima when using the m4 family. Figure 3.8 shows a full run of GBRT and GP (using LCB acquisition function) when optimizing for execution cost for km_B. The annotated configurations represent the configurations that have been explored by the optimization algorithm (for optimization budget of 18). The configurations tested by GBRT are clustered together as GBRT is doing more exploitation based on the 3 initial samples (labeled 1-3). On the other hand, GP tests a configuration in the r4 family on the fourth run and then proceeds to explore more configurations in that family later on. This allows GP to explore the good configurations, which are all in the r4 family in this case.

Henceforth, we show the results with the best performing variants based on each surrogate model for BO, for each optimization scenario.

Takeaways: Based on our evaluation, GBRT(PI) performs well when optimizing for execution time on both of the search spaces. RF(PI) outperforms GBRT(PI) in some cases with high optimization budget for DS_1. On the other hand, GP is the better BO variant when optimizing for execution cost, with EI and LCB as the best performing acquisition functions for DS_1 and DS_2, respectively.



(a) DS_1 Tuned hyper-parameters

(b) DS_1 Default hyper-parameters

(c) DS_2 Tuned hyper-parameters

(d) DS_2 Default hyper-parameters

Figure 3.9: Comparison of optimization algorithms when optimizing for execution time using tuned or default algorithm-specific hyper-parameters.

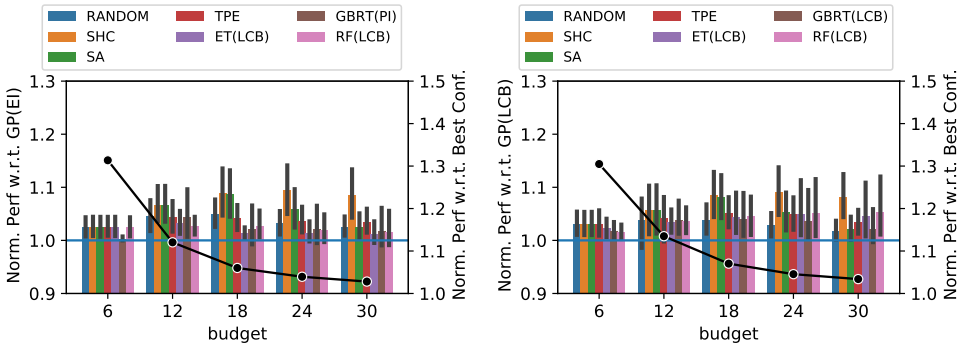
3.5.3 Which optimization algorithm is better?

Now that we selected the best candidates from BO variants, we can compare them against the other optimization algorithms we discussed in Section 3.3. Just like the previous section, we compare both the execution time and the execution cost objective functions.

3.5.3.1 Optimizing for execution time

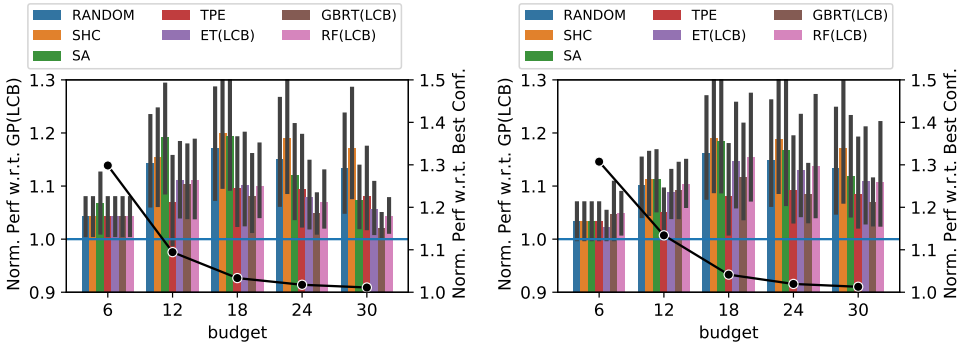
Figure 3.9 shows the comparison of different optimization algorithms when optimizing for execution time. We present the results for the scenario where we use the best algorithm-specific hyper-parameter values mentioned in Section 3.5.1.2 and a scenario where default hyper-parameter values are used. For DS_1 (shown in Figures 3.9a and 3.9b), BO with GBRT(PI) is the best candidate with tuned hyper-parameter values and BO with RF(PI) is only slightly better with the default. Using the best algorithms for execution time optimization of DS_1 provides up to 20% lower execution time. But in most cases, the benefit is less than 10%.

GBRT(PI) remains the best optimization algorithm for DS_2 (as shown in Figures 3.9c and 3.9d), with or without hyper-parameter tuning. With tuned hyper-parameters for all optimization algorithms, GBRT(PI) provides up to 10% better execution time compared to Random and LHS, but only up to 5% compared to the rest of the algorithms. Similar performance differences exist when using the default hyper-parameter values.



(a) DS_1 Tuned hyper-parameters

(b) DS_1 Default hyper-parameters



(c) DS_2 Tuned hyper-parameters

(d) DS_2 Default hyper-parameters

Figure 3.10: Comparison of optimization algorithms when optimizing for execution cost using tuned or default algorithm-specific hyper-parameters.

Takeaways: In most cases, BO with GBRT and PI acquisition function is able to find configurations with lower execution time than other methods. GBRT(PI) outperforms other algorithms with default as well as tuned values for hyper-parameters. However, RF(PI) performs slightly better than GBRT(PI) for an optimization budget of ≥ 24 runs.

3.5.3.2 Optimizing for execution cost

Using execution cost as the objective function to optimize also changes the characteristics of the search space, as previously shown in Figure 3.6. Therefore, in addition to using execution time as the objective function, we also want to compare different optimization algorithms using execution cost as the objective function. We have previously determined that the best optimization algorithm among the BO variants is GP instead of GBRT when optimizing for execution cost in our optimization scenarios.

As shown in Figure 3.10, when optimizing for execution cost, the performance differences between the best optimization algorithm and the rest are much more pronounced. When optimizing for execution cost for workloads in DS_1, using GP(EI) provides up to 29% and 40% lower execution cost than other algorithms with tuned and default hyper-parameters, respectively.

As shown in Figures 3.10c and 3.10d, the benefits of using GP(LCB) are more apparent when optimizing for workloads in the larger search space (DS_2). The performance benefit of using GP(LCB) is up to 40%.

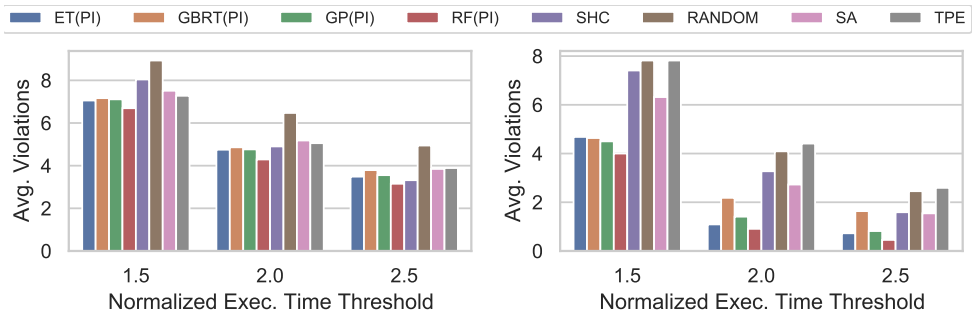
Takeaways: The best optimization algorithm based on our results does not change with using either the default or tuned values for algorithm-specific hyper-parameters. The difference in the performance of optimization algorithms is much more pronounced when optimizing for execution cost and particularly when optimizing for execution cost on the large search space of DS_2.

3.5.4 What is the best method for online optimization?

Prior work in choosing cloud configurations concentrates on *offline* optimization: the user waits until the completion of the optimization runs before deploying the workload in a production setting. However, that might require a lengthy offline benchmarking phase, which may not be feasible in some deployments. In this section, we evaluate an alternative *online* scenario, in which a user tests different configurations using production runs. In this case, the problem is that testing configurations online may lead to excessively long execution times for the production runs.

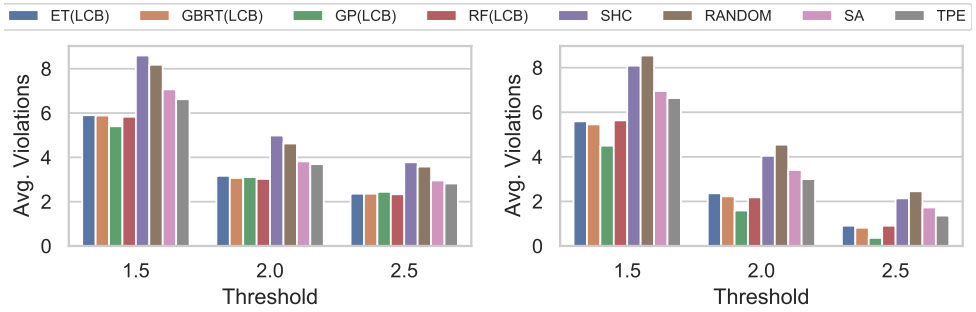
To evaluate this scenario, we assume the existence of a threshold beyond which the execution time obtained with a configuration is considered a *violation* to a service level objective (SLO). For configuring the various optimization algorithms, we use the respective default acquisition functions, as mentioned in the prior sections. For the values of algorithm-specific hyper-parameters, we use $\xi = 0.01$, $\kappa = 1.0$, and $\gamma = 0.1$, since, among the hyper-parameter values we have tested, these values provide the highest emphasis on exploitation rather than exploration. Emphasis on exploitation would mean that optimization algorithms would perform more conservatively, searching for better configurations in the vicinity of already known good configurations, and thus leading to fewer objective function value violations.

Figure 3.11 shows the average number of violations during optimization across all workloads in DS_1 and DS_2. The threshold is marked in the figure as a normalized runtime threshold, i.e., a threshold of 1.5 means that any configuration that leads to an objective function value that is more than 1.5$\times$ the minimum in the search space is considered a violation. Optimization algorithms that test configurations with widely different execution times result in a larger number of violations and are therefore less suitable



(a) DS_1: Execution time

(b) DS_2: Execution time



(c) DS_1: Execution cost

(d) DS_2: Execution cost

Figure 3.11: Aggregate number of SLO violations for varying thresholds (as a factor of the objective function value of the best configuration in the search space). Each optimization algorithm is using the default algorithm-specific hyper-parameter values.

for online optimization.

We run the optimization algorithm for 30 iterations. We will assume that the three initial samples are executed offline even with online optimization, so they are not counted as violations. We can see in Figure 3.11 that BO methods have fewer violations than other optimization algorithms. BO variants have up to 2× fewer violations compared to Random search, SHC, SA, and TPE. We also analyzed the performance of the optimization algorithms with these configurations values and the findings from previous sections still hold true. In particular, BO(GP) and BO(GBRT) provide the best optimization performance when optimizing for execution cost and execution time, respectively. Similarly, in the latter case, BO(GBRT) is closely matched by BO(RF) when the optimization budget is high.

Takeaways: When doing online optimization using the hyper-parameter values that emphasize exploitation, BO variants cause up to 2× fewer performance constraint violations, on average, compared to other algorithms, especially Random Search and SHC. The reason is that Random Search and SHC inherently lack an objective function model.

3.5.5 When are more optimization runs worth the extra cost?

Cloud configuration optimization is generally targeted towards periodic workloads. Therefore, it is important to understand how many periodic runs are required to break-even with the extra cost and time required for performing the optimization runs. In this section, we will present these break-even points for our workloads. We generate the break-even points using a ratio between the cost or time it takes to

Table 3.4: Number of extra periodic repetitions of the workload required in production to break-even with the time and cost of optimization when optimizing for execution time and execution cost, respectively.

Algorithm	Execution time (min–mean–max)		Execution cost (min–mean–max)	
	DS_1	DS_2	DS_1	DS_2
Random	71–361–1261	88–329–578	69–275–858	80–178–364
SHC	72–257–703	55–312–717	66–396–1521	98–195–365
SA	66–276–815	54–214–432	66–326–1016	89–167–262
TPE	50–263–1065	56–326–1008	66–265–946	45–110–234
BO(ET)	51–234–647	59–151–282	35–240–1997	55–152–249
BO(RF)	34–207–706	42–136–270	11–305–4436	38–148–242
BO(GBRT)	27–291–812	54–235–616	23–243–2003	37–99–154
BO(GP)	56–267–877	61–157–302	12–275–2112	44–109–217

do more optimization runs and the improvement in the objective function value as a result of those extra runs:

$$\frac{Cost(x_{bl+s}) - Cost(x_{bl})}{f(x_{bl}) - f(x_{bl+s})}$$

where $f(x_{bl})$ and $f(x_{bl+s})$ are the best objective function value after running the optimization for baseline optimization budget bl ($bl = 6$ runs) and for budget bl plus extra s steps, respectively. In turn, the cost function $Cost(x)$ is the monetary and time cost of doing optimization runs when optimizing for the execution cost and execution time objective functions, respectively. This ratio gives us the number of additional production runs that are necessary to fully amortize the increment in the optimization costs.

Table 3.4 shows the min, mean and max values for the number of extra production runs of workloads required to break even. The table presents summary statistics that provide an overall picture of the extra runs that are required across all workloads in a dataset for a given objective function. The more detailed statistics for individual workloads can be found in the accompanying code and data repository [65]. The results show that a job may have to be repeatedly executed anywhere between a dozen times to a few thousand times to justify further optimization. Therefore, depending on the frequency with which a workload is executed, it might or might not be worth spending more time optimizing the cloud configuration. On average, BO variants require fewer extra runs of the workload to break-even, thus providing more improvement for a given increase in the optimization budget.

Takeaways: BO variants provide lower mean break-even points than other algorithms for both the execution time and execution cost objective functions. Hence, they provide more improvement given a fixed extra optimization budget compared to other optimization algorithms such as SA, SHC, TPE, and Random Search.

3.5.6 Summary

Overall, one of the main conclusions is that BO outperforms other optimization algorithms that we have evaluated. Algorithm-specific hyper-parameter tuning is less important than the choice of the initial samples and optimization budget. We found no clear best setting of hyper-parameter values (especially the number of initial samples) for any of the algorithms: the best setting varies across workloads and objective functions. For Bayesian optimization variants, different acquisition functions perform similarly in most cases, but overall PI works well when optimizing for execution time and LCB is better when optimizing for execution cost. BO with GBRT and PI acquisition function is able to outperform other

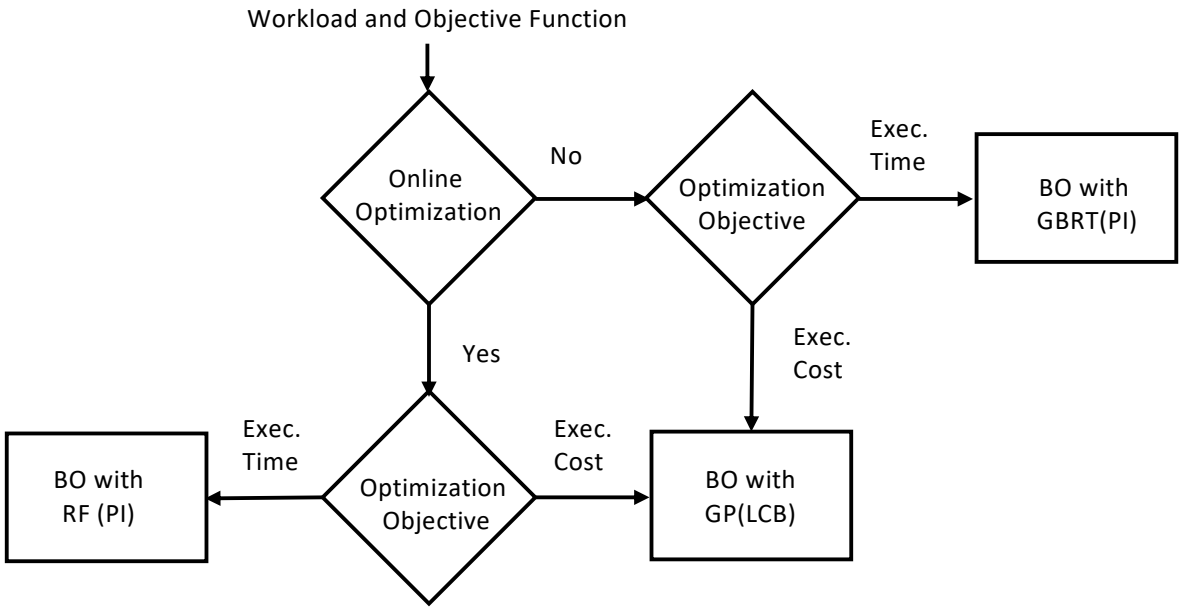


Figure 3.12: Decision tree for selecting an optimization algorithm.

algorithms by up to 20% and 10% when optimizing for execution time for DS_1 and DS_2, respectively. BO with GP and LCB provides up to 40% better performance when optimizing for execution cost in both DS_1 and DS_2. According to our evaluation, BO variants also provide fewer constraint violations for online optimization and more improvement in the objective function value as a result of a marginal increase in the optimization budget.

3.6 Choosing a Black-Box Optimization algorithm

Figure 3.12 shows the decision tree based on the evaluation in this work. For offline optimization, if the objective function is execution time, then BO with GBRT(PI) is the best choice. Whereas for execution cost, the preferred choice is BO with GP(LCB). For online optimization, we found that BO variants in our experiments had fewer constraint violations, which can be a critical aspect of an online optimization algorithm. Specifically, RF(PI) and GP(LCB) have a lower average number of violations when optimizing for execution time and execution cost. Therefore, BO with RF(PI) is a better choice for online optimization, not only because of having a lower average number of constraint violations but also because it can perform comparably to the best algorithm for execution time optimization, i.e., BO with GBRT.

3.7 Discussion

In this section, we will discuss orthogonal techniques that can be used in addition to the methods we have discussed in this chapter.

Early termination techniques to timeout likely bad cloud configurations can be used to decrease the

time and monetary cost of optimization. These enhancements have not been included in our evaluation because they can be incorporated into many of the optimization algorithms and thus are not a factor that can be used to differentiate one optimization algorithm from another.

Workload fingerprinting, along the lines of [29, 14], can be used by all BO algorithms as a prior, but it has an extra cost associated with it. We did not consider fingerprinting in our evaluation to provide a level playing field of comparison across BO and non-BO algorithms. The evaluation already shows that BO is often superior, even without fingerprinting.

In a cloud environment, we can execute multiple optimization runs at the same time, thus parallelizing the optimization process. Random sampling is easier to parallelize than *sequential* optimization algorithms such as the BO variants described in this chapter. With random sampling, virtually all configurations can be tested independently of each other. It allows users to deploy and run a set of clusters and stop the rest of the optimization runs when the best configuration has been found. Bayesian Optimization also has parallel variants [80, 81, 82]. The evaluation of the effectiveness and efficiency of these parallel optimization algorithms is an interesting avenue for future work.

3.8 Summary

In this chapter, we evaluated the performance of many popular black-box optimization algorithms in the context of choosing cloud configurations. We evaluated 8 algorithms and 23 workloads using 2 objective functions. Our evaluation showed that algorithm-specific hyper-parameter tuning is less important than the choice of the initial samples and the optimization budget. We found no clear set of optimal hyper-parameter values for any of the algorithms. While some existing works have used Bayesian optimization with Gaussian processes and Bayesian optimization with extra trees, our evaluation suggests that there is no silver bullet. Bayesian optimization with GBRT and GP perform better when optimizing for execution time and execution cost, respectively. We also examine the problem of finding optimal cloud configurations in an online setting and perform a break-even analysis to evaluate when performing more optimization runs is worth it.

Chapter 4

Finding the Right Cloud Configuration for Analytics Clusters

This chapter extends the automatic cloud configuration optimization problems to multi-framework cloud workloads. We present Vanir, a cloud configuration optimization algorithm that splits the optimization process into offline and online phases. Vanir finds good configurations with the offline phase and performs further optimization using online production runs.

4.1 Introduction

In Chapter 3, we have shown that we can use black-box optimization methods to optimize the deployment of the distributed data analytics systems running in the cloud. However, a wide variety of use cases *do not* execute on a standalone framework — instead, as depicted in Figure 4.1, they execute in an ecosystem of multiple frameworks that connect in a graph-like structure to form an analytics cluster. The pipeline depicted in this figure is very similar to the real-world deployments for Periodic ETL [83], or Lambda architectures for aggregating clickstream events [84].

A common way to deploy these analytics clusters is to *acquire resources* in a cloud environment and do so *on demand*, whenever (typically recurring) jobs need to execute. The choice of these resources dictates the cost efficiency and SLOs of the jobs, as we show in §4.2.2. Deploying analytics clusters in the cloud requires solving a *cloud configuration problem*: for each framework, determine (1) how many instances to use, and (2) which type of instances to use?

Prior methods for selecting cloud configurations [12, 13, 25, 14, 23, 26] do not consider the existence of multiple frameworks in an analytics cluster, but instead focus on configuring a single framework, such as Spark or Hadoop. Furthermore, all possible ways of applying these prior methods in the multi-framework analytics cluster setting have significant limitations. The *first way* is to use these methods to optimize each framework individually. However, as our results show, the coupling between different frameworks and incorrect distribution of budget causes this approach to miss opportunities to improve performance. The *second way* is to consider the entire analytics cluster as if it were a "single-large-

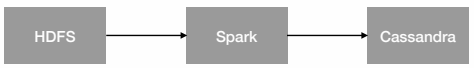


Figure 4.1: Example data analytics cluster.

system," simultaneously modifying the configuration of all frameworks. This approach enables joint optimization but increases the size of the configuration search space exponentially with the number of frameworks. Exploring a larger space naturally takes longer, causing a costly delay before users can deploy their clusters in production.

This chapter presents **Vanir**, a system for finding the right cloud configuration for multi-framework data analytics clusters. Vanir is designed for a setting where a user needs to provision and set up an on-demand analytics cluster for each run of a batch processing job. In this scenario, it is often the case that a large fraction of these deployments are recurring, as supported by reports that more than 40% of the jobs in production clusters are recurring computations [3, 4, 5]. After the job executes, the cluster is terminated or scaled down to avoid any further costs [85]. The main principle that Vanir adopts to cope with a large configuration search space is to find a *good enough* configuration via a fast benchmarking phase, and optimize that configuration during production runs, as the job recurs. A *good enough* configuration is one that satisfies user-defined SLO constraints on *both* cost and execution time. The design of Vanir combines in a novel way several techniques needed to jointly optimize the resource requirements for multi-framework analytics jobs. In particular, during a preliminary benchmarking phase, Vanir uses a metrics-based optimizer to quickly determine an initial configuration. In this stage, Vanir also attempts, whenever applicable, to use transfer learning and similarity between jobs to reuse knowledge from previous jobs. Then, to tackle the inherent limitations of a quick benchmarking search within a large search space, Vanir fine-tunes the configuration with the help of a performance model that is updated continuously with each successive run of a recurrent job. To select new configurations, Vanir employs Mondrian forests [86], an online random forest approach, aimed at incrementally improving the configuration.

This chapter describes the implementation of a Vanir prototype and use it to guide the deployment of nine different jobs on two different pipelines (including the one in Figure 4.1) using AWS EC2. We evaluate Vanir against two baselines corresponding to the above-mentioned ways of using existing optimization methods to handle multi-framework analytics clusters. Our results show that Vanir finds configurations that are comparable to completely offline methods. More importantly, this is achieved while requiring only half of the benchmarking runs, and with a total cost of optimization that is 1.3-24$\times$ lower than state-of-the-art methods.

4.2 Analytics clusters config

Analytics clusters are widely prevalent in modern data-driven organizations. A rich ecosystem of software systems gives organizations the freedom to create sophisticated analytics clusters composed of a multitude of frameworks such as Hadoop, Spark, Cassandra, or Flink. We begin by defining the cloud

configuration problem and then motivate our work by demonstrating the importance of selecting good cloud configurations. We further outline the challenges and discuss baseline solutions.

4.2.1 The cloud configuration problem

In this section we will refine the formulation defined in Section 2.1.2 to this specific research problem. We formalize the cloud configuration problem for analytics clusters as the problem of finding a configuration of resources that minimizes job execution time. A job is a combination of application (for example a random forest application in Spark) and the input data. Since a cluster comprises of multiple frameworks, we want to *jointly identify both the type and number of instances for each framework* within the cluster. Formally, a cloud configuration is denoted as a vector $x = \{\langle N_1, I_1\rangle, \ldots, \langle N_n, I_n\rangle\}$ where N_F is the number of instances for framework $F \in \{1, \ldots, n\}$ and I_F is the corresponding instance type.

As a validity condition, we require that any *valid cloud configuration satisfies user-specified constraints on the maximum execution time (ET_{max}) and maximum execution cost (EC_{max})*. In addition, we consider that the number of instances of each framework is chosen from a finite set of possible values. The minimum and maximum values across all frameworks yield the *search space bounds*. Lastly, similar to Chapter 3 the instance type defines the CPU, memory, and storage capabilities of an instance.

Thus, we can modify Equation 2.1 under these conditions to represent the cloud configuration problem in this case as follows:

$$\begin{aligned} x^* = & \arg\min_{x \in X} f(x) \\ & \text{s.t.} \quad EC(x) < EC_{max} \\ & \text{s.t.} \quad ET(x) < ET_{max} \end{aligned} \tag{4.1}$$

Where $EC(x)$ an $ET(x)$ are execution cost and execution time of the configuration x, respectively and $f(x)$ is $ET(x)$.

4.2.2 Selecting good configurations matters

We illustrate the value of picking a good cloud configuration by reporting the ratio of maximum to minimum execution time and execution cost for all the configurations that were used in our evaluation of several optimization methods (§ 4.7) and for the 9 jobs mentioned in § 4.7.1. Table 4.1 shows that selecting a good cloud configuration can drastically decrease job execution time. In particular, the max to min ratio for execution time and execution cost is roughly 2-8$\times$ and 2.4-77$\times$, respectively. Thus, optimizing cloud configuration can be crucial to satisfying time and/or monetary SLOs. Note that we do not show configurations where jobs take longer than our maximum time constraint (2,400s). Additionally, we omit results from several configurations that lead to job failures (mainly due to insufficient memory resources). As such, the ratios in Table 4.1 are a conservative report.

Job	Max-Min Ratio	
	Time	Cost
Logistic Regression (lr)	$2.7\times$	$2.4\times$
Random Forests (rf)	$5.7\times$	$8.3\times$
PageRank (pr)	$3.4\times$	$3.6\times$
Gradient Boosted Trees (gbt)	$8.0\times$	$77.1\times$
Nweight (nw)	$3.5\times$	$5.0\times$
Shortest Paths (sp)	$5.2\times$	$4.3\times$
Connected Components (cc)	$2.3\times$	$9.8\times$
Label Prop. Algorithm (lpa)	$2.0\times$	$2.8\times$
Price Predictor (pp)	$2.9\times$	$2.7\times$

Table 4.1: Max to Min ratio for execution time and execution cost of the tested configurations.

4.2.3 Challenges

While there are a few recent solutions for finding an appropriate configuration for deploying a single framework in the cloud [12, 13, 25, 14, 23], moving to multi-framework analytics clusters raises several additional challenges.

Intrinsic performance coupling. The cloud configuration problem does not decompose into several isolated, per-framework optimizations. This is because the *end-to-end* system performance depends on the performance of each framework, which is coupled with the performance of other interacting frameworks. Moreover, performance coupling means that one cannot simply optimize the cluster configuration one framework at a time following a natural, pre-established sequential order. The baseline method (Baseline 1) described below highlights that not accounting for performance coupling leads to sub-optimal configurations (§4.7.2).

Huge configuration space. Jointly optimizing multiple frameworks inevitably leads to a large configuration space, which increases exponentially with every additional framework. Thus the search for possible configurations has inherent scalability problems in this new setting. As such, traditional solutions that use exclusively offline optimization are not practical with a large configuration space. Our results in §4.7.2 show that the search time and search cost of an exclusively offline method can be high, and it may not be practical to use them.

Uninformative offline profiling. Gathering profiling information offline is a typical strategy to enhance the configuration search speed [30, 31, 29, 28]. However, in these methods, it is necessary to profile information for *all possible* configurations, which becomes untenable in our setting, since the number of configurations grows exponentially with the number of frameworks. We show that, in our solution, a short benchmarking phase (combined with transfer learning where possible) eliminates the need for comprehensive offline profiling.

4.2.4 Baseline solutions

As we outlined, there are two ways of adapting the state-of-the-art solutions to work with multi-framework analytics clusters. We will use the following two solutions as baselines to compare against our method.

Baseline 1. Baseline 1 consists of applying a traditional black-box algorithm like Bayesian optimization with Gaussian processes (used in Cherrypick [12]), to optimize one framework at a time. The main issue

is that the order in which each framework is optimized influences the optimization outcome. Since the best order depends on the application and the cluster composition under consideration, we follow the DAG of the analytics cluster as the default order. Note that, even though the baseline optimizes one framework at-a-time, they are not optimized in isolation. In other words, to optimize a given framework, we execute the job in a multi-framework setting, so that the optimization of that single framework is done in its actual execution environment. At a high level, the main limitation of Baseline 1 is that it has a limited view of the configuration search space since it determines the best configuration for only one framework at a time in the specified sequence. Thus, it only observes a smaller subset of the full configuration space and might be unable to explore better configurations.

Baseline 2. For Baseline 2, we also use Bayesian optimization with Gaussian processes, except that this baseline treats the whole cluster as a single black-box system. Thus it configures all of the frameworks simultaneously. Unlike Baseline 1, Baseline 2 is capable of exploring the entire configuration search space. However, Baseline 2 has to deal with a bigger search space than Baseline 1.

Baseline 1 and 2 are *offline* methods that require all optimization runs to be executed on a representative input dataset before the job can be deployed in production.

4.3 Overview of Vanir

Vanir solves the cloud configuration problem by splitting the optimization process into two phases. **Phase 1:** A quick heuristics-based benchmarking phase that determines a *good enough* initial configuration. **Phase 2:** A production optimization phase that uses machine learning to progressively find a better configuration. To further improve the performance and scalability of the solution, Vanir adopts a similarity scoring mechanism to bypass the benchmarking phase, if possible.

The two phases use separate optimizer components. The benchmarking phase uses the offline optimizer (whose runs are utilized for benchmarking purposes). The production optimization phase uses the online optimizer (wherein each run is an actual production run that is used to incrementally update the performance model for improving the configuration for the next job iteration). The online optimization process concludes after a user-specified number of runs.

We first discuss Vanir's usage for different high-level workflow scenarios. Then the subsequent sections present each component in detail and describe our design rationale. Figure 4.2 overviews Vanir's optimization process. Whenever a job request arrives, Vanir distinguishes between three scenarios:
Known: The job is the same as a previously executed job.
Similar: The job is new (i.e., not seen before), yet it is similar to a previously executed job.
Unknown: The job is new and not sufficiently similar to any previously executed job.

Vanir handles each of these scenarios as follows:

Known. The request for another run of a recurring job is passed directly to the online optimizer. The online optimizer uses the performance model for this job to select a new cloud configuration to test. Once a production run is concluded, Vanir updates the performance model for future optimization.

Similar. When a new job arrives, Vanir executes it on an initial profiling configuration and scores it

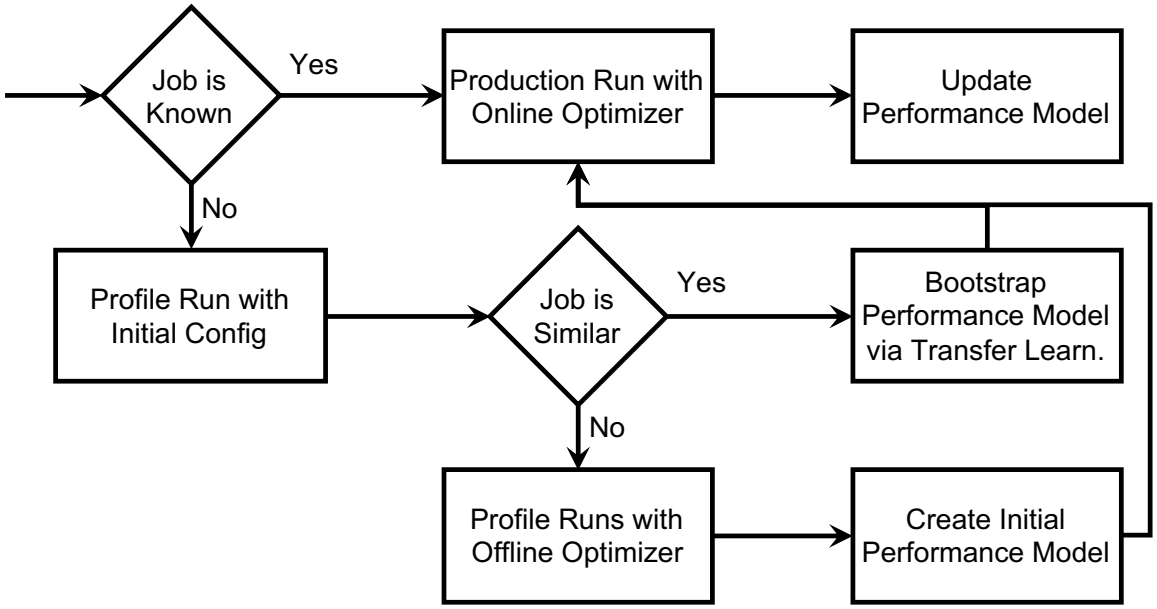


Figure 4.2: Configuration optimization workflow in Vanir.

based on similarity with previously executed jobs. If the job has a sufficient degree of similarity to one of the existing jobs, the job is passed to the online optimizer. The online optimizer loads the performance model for the most similar job and uses transfer learning to bootstrap a performance model for the new job. This performance model is then used as an initial state for the online optimizer to search for an appropriate configuration of the new job.

Unknown. When a new job does not satisfy the similarity filter, the job request is passed to the offline optimizer. The optimizer finds a good enough configuration that meets the user constraints and then launches a production run using that configuration. Subsequent submissions of the same job are optimized using the online optimizer.

4.4 Offline optimizer

The goal of the offline optimizer is to quickly find a *good enough* configuration that meets user-provided performance constraints, even if this comes at the expense of the quality of the chosen configuration. Owing to incremental improvements that the online optimizer will apply for subsequent job submissions, this is a reasonable trade-off to scale to large configuration search spaces.

Our design uses a metrics-based algorithm as the offline optimizer, which uses CPU and memory resource utilization metrics (monitored during profiling runs) to determine the configuration of each framework. Note that our aim in this part of the design is to illustrate the advantages of decomposing the configuration problem into an offline and online phase; one could replace our offline optimizer with an alternate one or even a static configuration based on domain knowledge.

The algorithm proceeds iteratively and consists of three phases: *Init*, *Resource Increase* and *Resource Adjustment*. Before we dive into the details of these phases, let us first develop the intuition behind our offline algorithm. Our first goal is to find a good enough configuration that meets the user-

specified constraints. To achieve this, we start by rapidly increasing the resource allocation so that we swiftly reach a reasonable set of valid configurations (i.e., that satisfy both execution time and cost constraints), allowing the algorithm to choose the one with the best execution time. However, if this phase only finds one valid configuration or none,[1] then, at a finer granularity, the algorithm reduces the execution cost by decreasing the resources allocated to frameworks with low resource utilization.

This simple algorithm does not deal with different instance families; it sticks with a general-purpose instance and modifies instance size and number of instances. This is intended to make the benchmarking phase short. The online optimizer handles different instance types.

4.4.1 Metrics-based algorithm

Init Phase: The optimizer first performs a profiling run on a pre-specified configuration to obtain an initial performance sample. We use an initial configuration with 4 instances of instance type $m5.large$ for each framework.

Resource Increase Phase: Then, in a sequence of profiling runs, the number of instances N_F allocated to each framework F increases according to monitored CPU and memory utilization as follows:

$$N_F = \begin{cases} 2 \cdot N_F & cpu_F + mem_F > \mu_1 \\ s + N_F & \text{otherwise} \end{cases}$$

That is, N_F doubles as long as the mean utilization of CPU (cpu_F) and memory (mem_F) cumulatively remain above a threshold μ_1. Otherwise, a lower sum of CPU and memory utilization indicates that doubling resources would be an over-allocation. In this case, N_F increases by a constant s with each profiling run. We use $s = 2$ and we discuss how to set the thresholds in §4.4.2. Since the search space bounds impose a limit on the number of instances of a particular size, if N_F reaches this limit, the algorithm shifts to the next larger instance size.

The resource increase phase applies to all the frameworks at once until one of two conditions occurs: (1) the execution time of the current configuration worsens as compared to the execution time of the previous configuration, or (2) the previous configuration was valid but the current configuration is not.

When either of these two conditions occurs, the optimizer checks whether it found more than one valid configuration. If that is the case, it terminates by returning the configuration with the best execution time. Otherwise, the optimizer enters a fine-grained resource adjustment phase to find a better configuration.

Resource Adjustment Phase: The optimizer seeks to find valid configurations by decreasing resources to decrease the overall execution cost. During another series of profiling runs, any framework whose CPU and memory utilization are not above certain thresholds sees its resources decreased. This terminates when the current configuration is not valid while the previous one was. The adjustment in the

[1] A user could specify execution cost or execution time constraints that are too stringent for the given search space. If the offline optimizer fails to find any valid configuration, then the constraints should be revised.

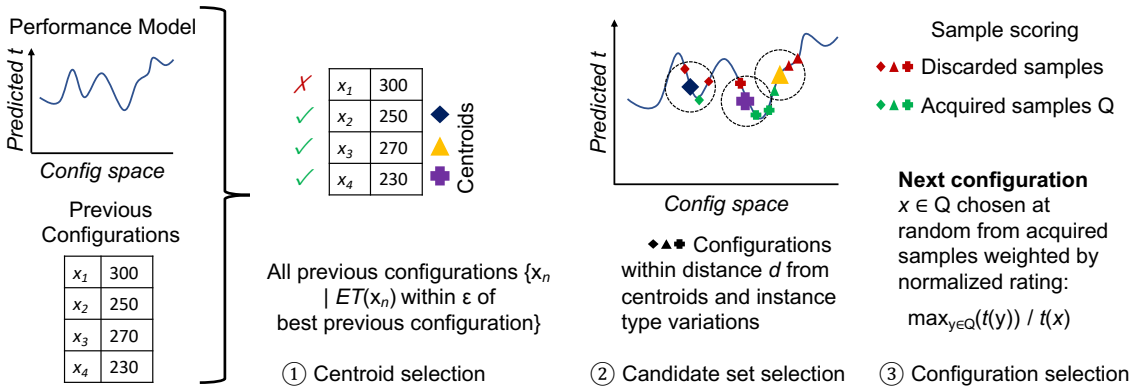


Figure 4.3: Acquisition method used by the online optimizer.

number of instances is done as:

$$N_F = \begin{cases} N_F/2 & (cpu_F < \mu_2) \wedge (mem_F < \mu_3) \\ N_F & (cpu_F \geq \mu_2) \wedge (mem_F \geq \mu_3) \\ N_F - s & \text{otherwise} \end{cases}$$

The rationale behind the way we change N_F is that we want to have the ability to make significant adjustments (halving or doubling N_F) when the utilization metrics are either too high or too low. Otherwise, we enable fine-grained adjustments (increments or decrements to N_F) to avoid over- or under-allocating the resources by a large margin.

4.4.2 Tuning the offline optimizer

The resource increase and adjustment phases depend on 3 thresholds that we set empirically. μ_1 controls the rate at which resources are increased w.r.t. resource utilization. A value of 100 means that the combined utilization of CPU and memory is 100%. The maximum value of this parameter is 200. Setting the value close to 200 makes the offline optimizer more conservative in increasing resources and likely slower. Conversely, setting it closer to 0 makes the optimizer more aggressive, which is likely to double the resources at every benchmarking run. μ_2 and μ_3 control the aggressiveness with which the resource adjustment phase decreases resources. The possible values range from 0 to 100. A low value for these parameters means fewer resources taken back from the frameworks and vice versa. In our experiments, we use the following threshold values: $\mu_1 = 100$, $\mu_2 = 50$ and $\mu_3 = 50$. These values strike a balance between aggressiveness and over-allocation of the resources in the different phases of the offline optimizer. A full analysis of these thresholds is outside the scope of this thesis.

4.5 Online optimizer

The online optimizer aims to further refine the configuration during recurring production runs of the same job. There are three key capabilities that an online optimizer should have: (1) a method to model the

job performance on different configurations, (2) the ability to update the performance model at the end of each production run, and (3) a way to select the next configuration (i.e., an *acquisition method*) that potentially improves performance.

In Vanir, the online optimizer maintains and uses a job-specific performance model $M\colon x \rightarrow t$ to predict the job execution time t of any given valid configuration x. Due to the challenges described in §4.2.3, generally, M is non-convex and its derivative is not available. This precludes standard optimization methods for finding the best cloud configuration. Moreover, M is not necessarily accurate, which requires exploratory production runs to make the model more accurate. To tackle these challenges, we will use Mondrian forests as a fundamental building block.

4.5.1 Mondrian forests

Mondrian forests use Mondrian processes [87] to construct ensembles of random decision trees. They can be trained in batch or online mode. We use MFs because of three important features [86]. First, MFs gracefully handle predictions on data points outside the space seen in the training data. In the absence of an extensive training dataset, as in our case, this property is of particular importance. Second, online training of MFs has higher accuracy than online RFs for the same amount of training data. Third, online MFs can provide prediction accuracy comparable to batch-trained RFs. These features make MFs more suitable for our case than the online versions of RFs, such as the one presented in [88, 89].

In Vanir, we use MFs as regressors. Configuration C is the vector of input features, and the output prediction is the job execution time t. Each production run of a configuration is used to update the model. We train the initial MF model M for a job using the data available from the offline optimization phase (except when a new unknown job has high similarity to other jobs, where we use transfer learning for bootstrapping the initial model of similar jobs, cf. §4.6).

4.5.2 Acquisition method

The acquisition method picks a new test configuration that potentially improves job performance and helps refine the model's accuracy. However, as discussed, the performance model is built incrementally and may be inaccurate (especially in the early stages of a recurring job). Also, due to the large search space, a brute force evaluation of the performance model is inefficient. Therefore, our acquisition method evaluates the performance model on a narrowed down candidate set of configurations before selecting the next configuration for a production run.

We start by providing an intuitive idea of how the algorithm works, as illustrated in Figure 4.3. At a high level, the acquisition method uses the previously tested configurations as centroids for a neighborhood search. In particular, the neighboring configurations around the centroids are used as potential candidates from which a better configuration might emerge. This potential candidate set is then filtered using constraints on the predicted execution time, and a single configuration is selected randomly using a weighted probability. The job is then executed with this selected configuration, and its execution time is used to update the performance model.

4.5.2.1 Centroid selection

The algorithm selects previously executed configurations as *centroids* – all valid configurations with execution time within an ϵ factor of the current lowest execution time – whose neighboring configurations can provide better configurations.

The algorithm attempts to select at least l centroids. Given that fewer than l configurations may satisfy the ϵ-factor condition (especially early in the online optimization phase), the algorithm uses a scoring function to extend the selection to previously executed configurations that meet the execution time constraint and represent good choices in terms of their execution cost. Let ET_{min} be the lowest execution time, and EC_{max} be the highest cost across all executed configurations. A configuration x is scored as follows (higher is better):

$$score(x) = \begin{cases} \frac{(1+\epsilon)ET_{min}}{ET(x)} & EC(x) \leq EC_{max} \\ \frac{(1+\epsilon)ET_{min}}{ET(x)} + \frac{EC_{max}}{EC(x)} & EC(x) > EC_{max} \end{cases}$$

Intuitively, this mechanism assigns higher scores to configurations that satisfy cost constraints and lower scores to configurations that do not, proportionally to the amount by which they exceed the constraints. This ensures that, within the configurations that meet the time constraint, preference is given to searching the neighborhood of valid configurations (as far as their cost is concerned) with lower execution times, followed by configurations that violate the cost constraint with the lowest margin.

4.5.2.2 Candidate set selection

Based on the set of centroids $\mathcal{O}$, the algorithm forms a *candidate set* of promising but untested configurations. This set is the union of configurations drawn from the neighborhood of each centroid and through instance type variations of the currently best configuration. Figure 4.4 shows an example of this process.

Neighborhood selection. For each centroid, the algorithm selects every configuration within distance d from the centroid. We define a distance metric based on the framework-wise difference of CPU and memory capacity. We say that the capacity $cap_F(x)$ of a configuration x w.r.t. framework F is the vector ⟨total CPU count, total memory⟩, where total refers to the sum of all N_F instances of F. The distance between two configurations x_n, x_m for framework F is $dist_F(x_n, x_m) = ||cap_F(x_n) - cap_F(x_m)||_1$. Therefore, so far the candidate set is $\bigcup_{o \in \mathcal{O}} \{x \mid dist_F(x, o) \leq d, \forall F\}$.

The threshold d is set as $max(d_s, d_{max})$, where d_s denotes the distance of a single step away from centroid. This only applies when the hyperparameter d_{max}, which is specified in relative terms, overly restricts the candidate set. Note that because of our definition of distance, in theory, there could be configurations that differ from a centroid but their distance from it is zero. These configurations are also included.

Instance type variations. We observed that the above candidate set would rarely include an instance family different from those of the centroids because changing the instance family (for the same number of instances) can double or halve CPU and memory resources (we expect d_{max} to be smaller than

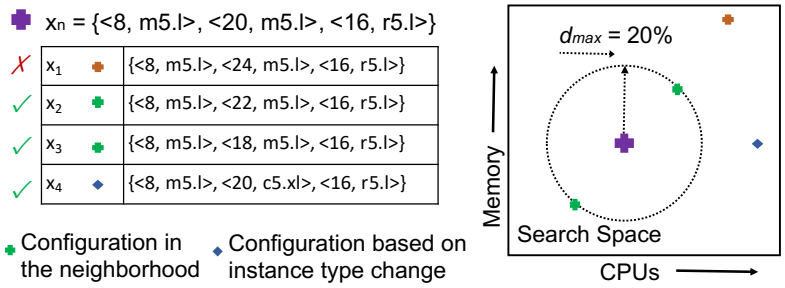


Figure 4.4: Example candidate set based on 20% maximum distance from a centroid (purple cross) for the middle framework ($\langle$20, m5.l$\rangle$). The valid configurations in the neighborhood are those with a variation of $\pm$10% memory and $\pm$10% number of CPUs (combined $\pm$20%). A selected configuration based on instance type variation doubles the number of CPUs and maintains the same memory.

100%, see §4.5.3). To allow for instance family variations, we include in the candidate set configurations more distant than d_{max} from a centroid. Namely, the algorithm enumerates all possible instance type variations for the centroid configuration and includes those that do not reduce the total available memory. The intuition behind this is that changes in CPU resources can affect the job execution time, but drastic reductions to memory capacity can cause job failures. As reference centroid, the algorithm only considers the current best configuration, and all framework-wise instance type variations are considered. Since only the best configuration is used for this step, the possible decrease in the execution time is not severe.

4.5.2.3 Configuration selection

Using the candidate set $\mathcal{Q}$, the algorithm selects a configuration in $\mathcal{Q}$ for use in the next production run. First, the algorithm filters the candidate set to discard any configuration whose predicted execution time is too high compared to the currently best configuration and the best-predicted execution time of the configurations in $\mathcal{Q}$. Namely, $x \in \mathcal{Q}$ is discarded if $t(x) > \theta \min_{y \in \mathcal{Q}}(ET(y))$ or if $t(x) > \phi ET_{min}$. Here, θ and ϕ are hyperparameters whose tuning we describe in §4.5.3.

Next, the algorithm could pick from the remaining candidates the best configuration as predicted by the model. However, given the small number of executed configurations in the beginning, the model predictions may not be very accurate. Instead, the algorithm rates each configuration x according to its predicted execution time $t(x)$: $rate(x) = \max_{y \in \mathcal{Q}}(t(y))/t(x)$; then the algorithm randomly picks the *next configuration* with a probability p following its normalized rating: $p(x) = rate(x)/\sum_{y \in \mathcal{Q}} rate(y)$. This allows for some diversity while also biasing the choice to configurations with better predicted execution time.

4.5.3 Tuning the acquisition method

ϵ controls the closeness of the centroids to the best valid configuration in terms of the execution time. For instance, a value of 0.2 means that only points that have 20% higher execution time than the best-known execution time are chosen as centroids. l determines the desired number of centroids. d_{max} controls neighborhood size around the centroids. θ and ϕ limit the configurations that can be selected based on

their predicted execution time w.r.t the best-predicted execution time in the candidate set and the best-known execution time. θ should be set slightly higher than ϵ to take into account the model prediction error. ϕ controls how conservatively the algorithm behaves; e.g., a value of 2 means that the algorithm will not select a configuration whose predicted execution time is twice the best-known execution time.

Higher values for all these hyper-parameters will allow for more exploration but might also lead to more execution time constraint violations. For our evaluation, we use the following values: $\epsilon = 0.2$, $\theta = 1.5$, $\phi = 2$, $l = 5$ and $d_{max} = 20\%$. These values worked well for our scenarios since, with these settings, the algorithm maintains a conservative approach in terms of the execution time constraint, while still being able to explore. A complete hyper-parameter exploration is left for future work.

4.6 Learning from other jobs

To try to reduce the number of benchmarking runs even further, Vanir uses the similarity between jobs and transfer learning to quickly bootstrap a performance model for the online optimizer, using knowledge from other jobs.

4.6.1 Determining if jobs are similar

Before bootstrapping the performance model for a new job from a previous one, we need to first determine whether jobs are similar. Inspired by collaborative filtering algorithms [90], Vanir uses cosine similarity as a similarity metric applied to job *signatures*. We explored several types of signatures and we settled on a simple one: a job signature is the pair of CPU utilization and memory utilization histograms. The histogram is created from a list of observations of the respective utilization metric. Observations from each instance of a framework are taken continuously and appended to create that list. In particular, these observations are collected every 5 seconds during a job run, and the histograms use bins at 10% granularity. Cosine similarity uses the concatenation of these two histograms in vectorized form.

We say that the pair of a new job and a prior job is similar when their similarity score is above a threshold (0.85 in our case). Vanir uses the model of the prior job with the highest similarity to bootstrap a performance model.

Unlike in previous work [30, 31, 29, 28], we do not use collaborative filtering to get recommendations regarding the configurations based on prior jobs. This is because collaborative filtering requires certain sparsity conditions [30, 31, 91, 28], which might be impractical to achieve. Furthermore, these prior works rely on extensive offline profiling, in which a set of benchmark applications are profiled on all configurations. This is also impractical in our case, wherein the total number of possible configurations (Table 4.2) is roughly 900k. However, collaborative filtering or machine learning methods for workload characterization [30, 31, 34, 29, 14, 28] can be useful once there is a substantial history of a large number of jobs and job runs and we leave the exploration of this possibility as future work.

While cosine similarity between jobs allows bootstrapping models for unknown jobs, it is not a perfect method and may lead to false-positive matches. We add a simple test to avoid the effects of false

Parameters	Possible Values
Instance Type (in AWS EC2)	m5.large, m5.xlarge, c5.xlarge, c5.2xlarge, r5.large, r5.xlarge
No. of Instances (per framework)	[2, 32] - step size of 2

Table 4.2: Possible values of configuration parameters.

positives: we measure the performance of the new job on the best and worst configuration of the most similar job, and, if the new job performs better on the best configuration from the known job compared to the worst configuration, then we bootstrap the model; otherwise, we fall back to the benchmarking phase. Thus, a successful application of similarity limits the number of benchmarking runs to a total of 3 (1 for similarity calculation and 2 to avoid false positives).

4.6.2 Bootstrapping models

Vanir uses transfer learning to adapt an MF model trained on a previously seen job and provide better results with fewer training samples for a new job. However, several challenges arise when putting this idea into practice. First, to the best of our knowledge, there is no existing transfer learning method for Mondrian Forests. Additionally, we want to minimize the number of samples needed from the new job to perform transfer learning. Therefore, we propose a simple technique that uses the difference in execution time between the two jobs (new job and a previously seen job) for the same configuration to offset the rest of the predictions of the model.

Assume that job J has the highest similarity with a new job H. In this case, we want to transfer J's performance model M_J to create a performance model M_H for H. To this end, we run H with the best configuration from model M_J and obtain the execution time ET_H. The difference $ET_H - ET_J$ between this execution time and the execution time obtained for the same configuration on J is then used as an offset to adjust the execution time predictions ($t_H(x)$) for M_H as: $t_H(x) = t_J(x) + ET_H - ET_J$ for all training configurations x used to build M_J.

4.7 Evaluation

We evaluate Vanir against Baseline 1 and Baseline 2 (§4.2.4), which we implement atop the Spearmint [92] library for Bayesian optimization. All experiments run in AWS.

Table 4.2 lists the configuration search space, which yields 96 configurations for a single framework while the total number of possible configurations is 884,736. As such, it is impractical to search the entire space to determine the best configuration of each job as a ground truth.

We limit the optimization budget for each job to 18 runs in total. Baseline 1 uses 3 initial samples for each framework, while Baseline 2 uses a total of 6 initial samples (generated through Spearmint's default Sobol sampling).

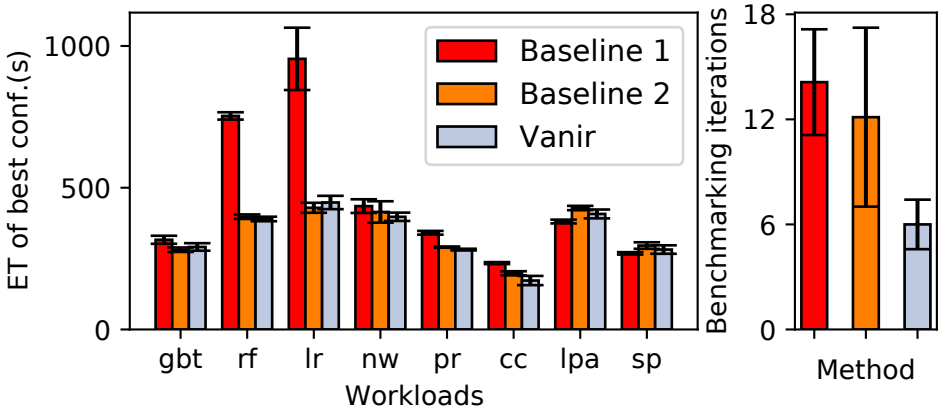


Figure 4.5: (a) Best execution time (left). (b) Number of benchmarking iterations (right).

4.7.1 Applications

We run a total of 9 realistic benchmarks (Table 4.1), 8 of which run on the analytics cluster shown in Figure 4.1. These benchmarks are adapted from applications included in the HiBench suite [75] and a few additional GraphX [93] jobs (sp, cc, lpa). To evaluate Vanir with *different pipelines*, we further run the price predictor (pp) job on a second cluster comprising a DAG of batch jobs. Our benchmarks, configs, and datasets are public at [94].

As for the dimension of input datasets, following the naming in HiBench, we use "gigantic" for pr, nw and lr; "huge" for gbt and rf; and "small" for sp, lpa, and cc. These sizes are chosen so that all jobs run in a reasonable amount of time (to reduce our costs) on at least a subset of the possible configurations. The datasets are generated synthetically using HiBench for each run.

We use the default values for all system parameters except for parallelism level and memory configurations for Spark. The parallelism level is set to $2\times$ the number of cores available in the cluster allocated for Spark, based on the recommendation from Apache Spark documentation [95]. To enable Spark to use all available memory, the amount of executor memory and driver memory is set to the amount of available memory in the instances used for Spark.

4.7.2 Results

We compare Vanir against both baseline methods across: quality and cost of optimization; the speed of search; and the contribution of different components of Vanir. We also discuss the best configurations selected by Vanir and both baselines. We discuss the effects of the bounded search limitations of the baseline methods. Lastly, we generalize our results by looking at a different pipeline and variations in input dataset size.

4.7.2.1 Quality of optimization

We analyze the best configuration found by an optimization algorithm within a given search budget. In our case, this represents the best execution time found while satisfying the constraints.

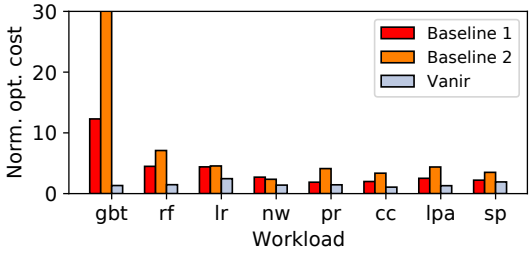


Figure 4.6: Cost of optimization. Baseline 1 and 2 are generally 1.5-6 times more expensive than Vanir.

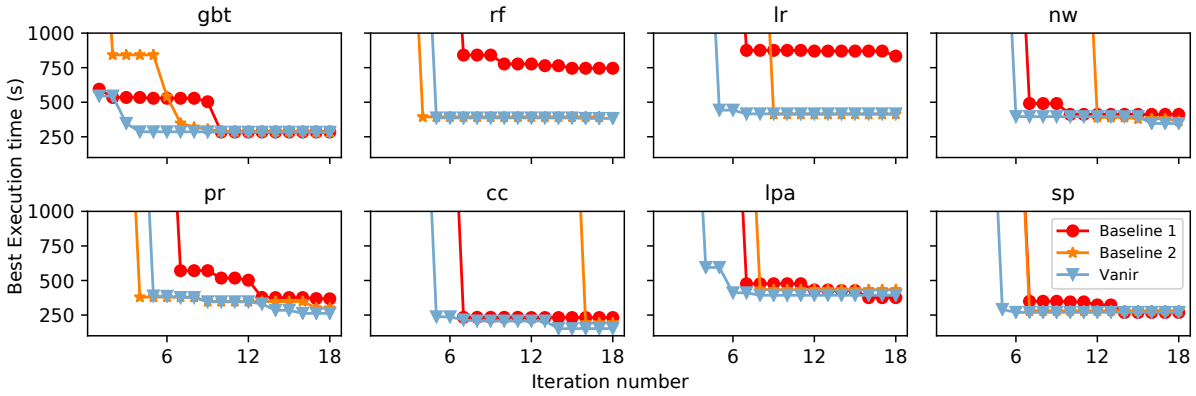


Figure 4.7: Progression of optimization for target jobs.

Figure 4.5a shows the execution time of the best configuration found by each optimization method. We repeat the job with the best configuration for each method 5 times, and the error bars indicate the standard deviation. For all jobs, Vanir finds configurations that perform comparably to those by Baseline 2. Baseline 1, in turn, performs comparably to Vanir and Baseline 2 for 6 of the 8 jobs. The two jobs where Baseline 1 is outperformed are rf and lr. For these jobs, the best execution time found by Baseline 1 is 2-2.5$\times$ higher than Baseline 2 and Vanir. This is due to the limited view of the configuration space that Baseline 1 explores, since it optimizes one framework at a time, and decides on the best configuration for a given framework before advancing to optimizing the next one.

Figure 4.5b shows the number of benchmarking iterations required by each algorithm. For Vanir, that is simply the number of iterations of the offline optimizer. For Baseline 1 and 2, this is the number of runs to reach the best found configuration within the optimization budget. In practice, there is no way to know if an algorithm has yet reached the best configuration it will find. We, therefore, assume an oracle that predicts if the baselines have reached their best configuration. Therefore, this result shows the best-case scenario for the baselines. The results show that Vanir uses an average of 6 benchmarking iterations. In turn, Baseline 1 and 2 use 14 and 12 benchmarking iterations, respectively – thus, on average, requiring to wait at least 2 times as long before starting production runs. Using only $1/2$ benchmarking iterations, Vanir finds configurations comparable to Baseline 2, thanks to online optimization in production runs.

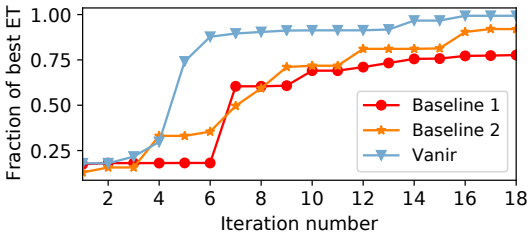


Figure 4.8: Speed of optimization. On average, Vanir takes only half the number of iterations compared to Baseline 2 to reach within 15% of the best execution time found.

4.7.2.2 Cost of optimization

It is important to consider the monetary cost of the (benchmarking) optimization phase since this consists of running an offline optimization algorithm, which costs money (and time) without doing any useful work. Ideally, this should be kept to a minimum.

Figure 4.6 shows the cost of optimization, normalized by the cost of the benchmarking phase of Vanir. The bars for Baseline 1 and 2 represent the total cost of the optimization over the budget of 18 iterations compared to the benchmarking phase of Vanir. Vanir (Full) includes both benchmarking and production phases. The cost of the production phase is only the extra cost for each iteration of the production phase compared to what would cost if one were to simply run Vanir's best found configuration.[2] The results show that Vanir incurs in substantially lower optimization costs, due to its virtuous combination of techniques, namely a quick estimate of a good configuration in the benchmarking phase followed by gradually improving the configuration at production time.

Using only the benchmarking phase of Vanir, the cost is up to 60% lower than running both the benchmarking and production optimization phases. However, that reduction in optimization cost comes at the expense of decreased quality of optimization, as shown later in this section. On the other hand, Baseline 1 and 2 are roughly 1.3-4.6× and 1.6-5.9× more expensive than Vanir (Full), respectively, for 7 out of 8 jobs. The exception is gbt, where Baseline 1 costs 5.9×, and Baseline 2 costs 24× more than Vanir. This is primarily because gbt does not require a lot of resources; in fact, bigger cluster sizes worsen its performance. Vanir's systematic approach with its benchmarking phase quickly determines that characteristic. Therefore, it avoids selecting larger configurations while the baselines often explore those regions of search space, thus incurring high optimization costs.

4.7.2.3 Speed of search

Another way to compare optimization algorithms is to evaluate which method is the fastest to reach a good configuration for a given search budget.

Figure 4.8 shows, on average across the 8 jobs, how quickly each algorithm can get close to the best execution time found by any optimization method. The results show that Vanir is the fastest. On average, Baseline 2 takes 2× longer to get to a configuration within 15% of the best one, compared to Vanir. Because Vanir uses a metrics-based method as an offline optimizer, it keeps the benchmarking

[2] As if an oracle revealed the best configuration that Vanir will find.

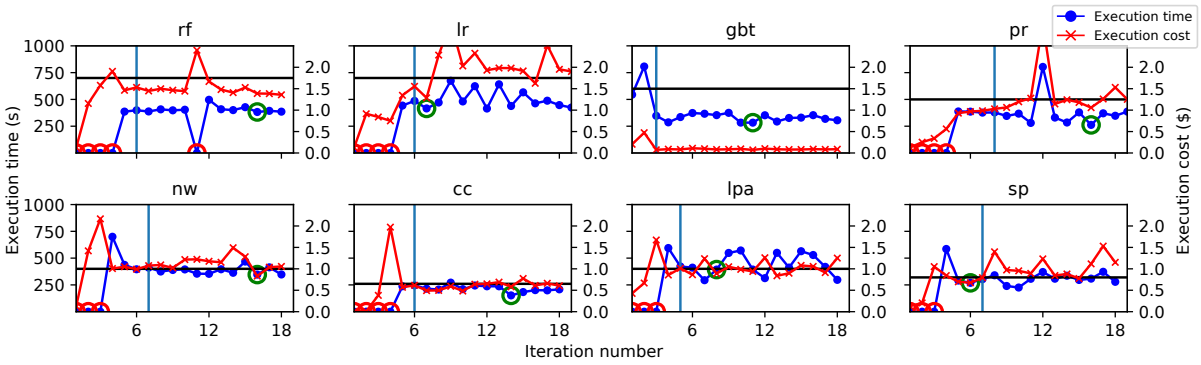


Figure 4.9: Progression of Vanir optimization across eight jobs. Only one ET constraint violation occurred in online phase.

phase short by systematically changing the resources assigned to different frameworks instead of blindly generating initial samples, as it is the case for completely black-box methods. Vanir quickly finds a good configuration to start the production runs of the recurring job. Subsequent runs of the job can benefit from the production optimization phase.

Figure 4.7 shows the progression of the best-found execution time for each job separately. Vanir is the fastest for 6 of the 8 jobs; the exceptions are for rf and pr, where Baseline 2 by chance finds a valid configuration during initial sampling.

4.7.2.4 Constraint violations

Since Vanir performs optimization during production runs, it is worth evaluating the number of violations experienced (for time or cost constraints). Vanir assumes that cost constraint violations are tolerable during the production optimization phase. However, execution time constraint violations are more severe in production environments, and therefore an online optimizer should limit them. Any unsuccessful run of a job due to failure (e.g., memory exhaustion) is a violation.

Figure 4.9 shows the progression of execution time, and execution cost as Vanir optimizes the configuration for each job. The green circle indicates the best valid configuration found. The blue vertical line is the iteration number at which the benchmarking phase ends, and the production runs start. The black horizontal line is the cost constraint (associated with the right y-axis). Optimization runs that fail to execute or take longer than the execution time constraint are shown as iterations with zero execution time.

In our experiments, the online phase of Vanir leads to only one failure due to a lack of resources (for the iteration 11 of rf). However, if such an error occurs, the online optimizer quickly learns to avoid that region of the search space. It is clear that during the production phase, Vanir tests configurations that lead to cost constraint violations. This design is by choice, since making Vanir too conservative would hamper the quality of configurations it finds.

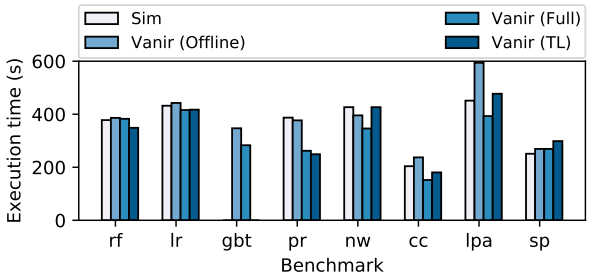


Figure 4.10: Contribution of different components of Vanir. Compared to offline-only optimization, the online phase lowers execution time by up to 36%.

4.7.2.5 Contribution of different optimizers

To understand the contribution of Vanir's components to the overall optimization process, we break down the best execution time achieved by using different components. Figure 4.10 shows that the offline optimizer reaches performance close to the best configuration found by Vanir (Full), in 3 of the 8 jobs (sp, rf, and lr). In these cases, the online optimizer improves execution time by 0%, 1%, and 6%, respectively. In contrast, the online phase improves execution time by 12-36% for the other 5 jobs.

4.7.2.6 Contribution of similarity and transfer learning

We now use a model learned on a job to optimize the configuration of another job. For a job J, using the best-known configuration from the most similar job leads to an execution time that is close to J's best-known execution time. Figure 4.10 shows that by using similarity (Sim), Vanir finds a configuration with an execution time that is at worst only 55% higher than the best-known configuration. However, these configurations do not always satisfy the execution cost constraints, a task handled by the production optimization phase that follows model bootstrapping.

Using transfer learning, we bootstrap a model for each job. Then we run the production optimization phase on the bootstrapped model to further improve performance. This is shown as Vanir (TL) in Figure 4.10. The improvement in execution time after running a production optimization phase on the bootstrapped model is 3.4%, 7.7%, 11%, and 35% for pr, rf, cc, and lr, respectively. There is no improvement in the case of nw. For sp and lpa, the execution time is worse than the one shown for *Sim*; that is because the configuration found using similarity did not satisfy the cost constraint, and so the configuration found after the production run phase in Vanir (TL) that satisfies the cost constraint has higher execution time. Interestingly, Vanir (TL) provides a trade-off between lowering the number of benchmarking runs and the quality of optimization. In 4 of the 7 cases, given the limited budget of 18 iterations, Vanir (TL) finds a configuration that is up to 20% slower than the one found by Vanir (Full). Hence, shortening the benchmarking phase is not always the right choice when the optimization quality is more important.

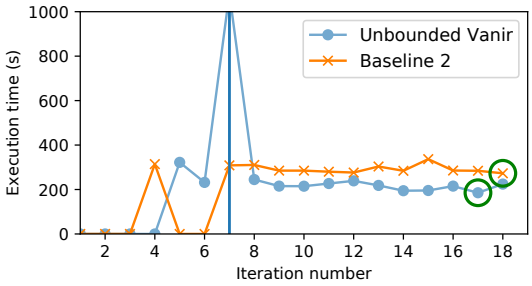


Figure 4.11: Comparing Vanir's performance on unbounded search space vs. Baseline 2 using sp. Vanir automatically determines the right upper resource limits and avoids incorrect user-defined search space bounds.

4.7.2.7 Unbounded search

Bayesian optimization, as used in Cherrypick [12] and Arrow [25], requires users to define the bounds of the optimization search space; specifically, the minimum and the maximum number of instances as well as possible instance types. While defining instance types is straightforward, defining the bounds on the number of instances is not easy without prior knowledge. In particular, when defining this bound, there is a trade-off between the size of the search space and the cost of the optimization: a larger search space would likely require a larger budget and vice versa. Given that a user would likely not know about a job's performance profile across different configurations, it is easy to make mistakes when defining bounds on the search space. Vanir does not require users to specific bounds on the number of instances which is useful property of our design.

Until now, we have been comparing both baseline methods with Vanir using a bounded search space, for a fair comparison. However, now we want to gauge the potential gains from an unbounded search; to this end, we keep the same bounds as in Table 4.2 for Baseline 2, while we set no upper limit for the number of instances of Vanir. Since the type and size of the instances would, in practice, be limited to a small number, we still keep the bounds on them. In this experiment, we use sp with the same dataset size used in previous experiments but with an execution cost constraint of $1.5 instead of $0.8 as used before.

Figure 4.11 shows the performance of the configurations that both methods find. The results show that Vanir finds a valid configuration with an execution time of 185s, while Baseline 2 only finds a configuration with an execution time of 272s. This is because Vanir explores beyond the bounds that Baseline 2 is limited to. In particular, it tests configurations with 64 instances allocated to Spark and then also with 128 instances, and automatically finds that while $\sim$64 instances still provide a speedup, $\sim$128 instances worsen the performance. Thus, unbounded search allows Vanir to find a configuration that is 32% faster than Baseline 2, for the same constraints and the same optimization budget.

4.7.2.8 In-depth look at the best configurations

We now want to reason about why one optimization algorithm performs better than another by taking a look at the best configurations found by each optimization algorithm. Answering this question can also

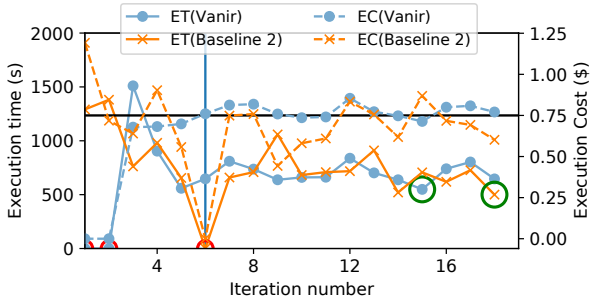


Figure 4.12: Comparing Vanir's performance on pipeline of batch jobs (pp) vs. Baseline 2. Performance is comparable; Vanir spends just 6 runs in benchmarking phase.

help us understand the inefficiencies in each optimization algorithm. We will only discuss select few cases here.

Generally, HDFS and Cassandra are allocated fewer resources than Spark due to cost constraints. We also observe different best configurations across different workloads. HDFS and Cassandra are generally allocated a similar amount of resources except for the rf workload, where HDFS is given $2\times$ resources compared to Cassandra (in the best configuration).

We find that Baseline 1 generally ends up only assigning more resources to Spark. In contrast, the best configurations found by both Vanir and Baseline 2 assign more resources to other frameworks as well. Thus, Baseline 1 misses the opportunities that the other two methods explore (since they perform joint optimization). This is particularly important for rf and lr, and that is why Baseline 1 performs poorly compared to the other two methods for those jobs.

In addition, we find that different instance types matter. Specifically, Vanir's best configurations for pr and gbt jobs use c5.2xlarge and c5.xlarge instances, respectively. These changes improve the execution time by 18% and 30% (compared to using m5 instances) for gbt and pr, respectively. Interestingly, none of the best configurations use the maximum allowed resources for every framework because: (1) the execution cost constraint means that even if higher resources decrease the execution time, they might incur a high execution cost, and (2) in some cases, assigning more resources can degrade execution time; e.g., in case of gbt.

4.7.2.9 Generalizing to a pipeline of batch jobs

To validate that Vanir's approach applies to other types of data analytics pipelines, we use a pipeline of batch processing jobs. This kind of pipelines is common in industry use case [85, 96, 97], enabled by tools such as Apache Airflow [98] and Luigi [99], which have been designed to facilitate data pipelines in the form of a DAG of batch jobs. In our case, the pipeline consists of a linear DAG of three Spark jobs [94] wherein the results of one job serve as input to the next.

Figure 4.12 shows the comparison between Vanir and Baseline 2 for this data pipeline. Both methods eventually find the best configurations (highlighted with green circles) that perform comparably. However, Vanir does not have significant performance variations during the production runs (beyond the vertical blue line) and thus it is more suitable for optimization using production runs. In contrast, Baseline 2

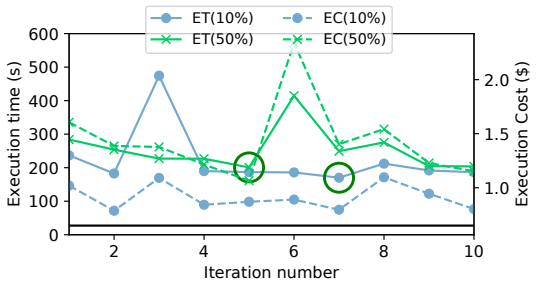


Figure 4.13: Handling changes in input data size for cc. Vanir improves the execution time by 30% within 10 optimization runs for input data size increase of 10% and 50%.

only reaches its best configuration at the end of its offline optimization budget (18 iterations). For this scenario, Vanir determines that the Update stage requires more resources than the other two stages.

4.7.2.10 Handling input data size changes

Since Vanir performs part of the optimization using production runs, one could expect that variations in the input data size for different job invocations affect performance. Vanir handles these changes (whenever input data size changes significantly, say at least a 10% change) by adapting the job's performance model to different input data sizes, treating the latter as an extra model feature. The initial configuration used is a linearly and proportionally-scaled version of the best configuration for the original input size.

Figure 4.13 shows the progression of optimization of cc over 10 iterations when the data size is 10% and 50% higher than the data size on which the original cc model was created. Vanir finds configurations (highlighted with green circles) that provide 170s and 200s execution time for input data with 10% and 50% higher data size, respectively. This represents an improvement of 30% over the initial (proportionally larger configuration).

4.8 Related Work

In Chapter 2 (§2.2) we provided a summary of prior work in the field of automatic cloud configuration and resource allocation in data center environments. In this section, we will discuss how Vanir differs from these prior works and, in some cases, how they can complement Vanir.

Cloud configuration optimization.

Prior works like Ernest [23] and Elastisizer [24] use analytical models to optimize cloud configurations. These models are often limited to the system under consideration and are not applicable for the performance prediction of multi-framework analytics clusters.

Cherrypick [12], Arrow [25], Micky [13] and Lynceus [26] targeted a single framework and therefore only had to consider a configuration search space on the order of tens of configurations. We build on that line of research but consider a much larger search space, where a completely offline method becomes impractical. Instead, Vanir splits the optimization into offline and online phases.

Scout [14], Selecta [28] and PARIS [29] use historical data to quickly choose cloud configurations

for new jobs. Selecta incorporates different storage options into the cloud configuration search space. Lessons learned from Selecta can be incorporated into Vanir's offline optimizer. The increased size of the search space resulting from including storage options would make Vanir even more attractive. However, these were designed for and evaluated in single system deployments, and the amount of historical data required for handling clusters with multiple systems might be impractical. Additionally, given their different target deployment, Scout and PARIS did not need to improve the configuration suggested by historical data at deployment time, which Vanir does through online optimization.

Resource allocation in data centers. Paragon [30] and Quasar [31] use collaborative filtering to classify an unknown incoming job to assign resources to it. Despite this being a different problem scenario, we note that a downside of collaborative filtering is that it requires an offline training set that is impractical to obtain when the configuration space is large. DejaVu's [34] method of creating workload signatures might be a complementary alternative to the simple similarity metric used by Vanir when there is a large set of jobs and performance data. Similar to Ernest and Elastisizer, Perforator [33] uses analytical models to perform resource optimization for Hive, which are not applicable in our setting with multi-frameworks.

4.9 Discussion

We now discuss the limitations of this work as well as other techniques that can be investigated to potentially improve this work.

Vanir's offline optimizer algorithm makes decisions based on CPU and memory consumption. But we can easily incorporate other resources such as network and disk utilization into the decision-making process. However, we note that the thresholds for disk and network utilization might vary across different cloud providers, since instances across cloud providers have different disk and network speeds.

More complex workload fingerprinting techniques such as the ones detailed in [29, 14, 30, 31, 34], can be used by Vanir in order potentially achieve better results. However, that is only feasible after a large number of offline benchmarking runs or once enough historical data has been collected on many jobs that Vanir is observing.

Lastly, we applied Vanir as is to optimizing a pipeline of batch processing jobs (in Section 4.7.2.9). However, further improvements can be made when using Vanir in such a setting. The batch jobs are decoupled in this case, and thus they can be independently configured. We can reduce the optimization cost of the offline process by only running the batch job whose configuration has changed. Additionally, it is easier to determine which batch job provides the most significant improvement in the performance objective as the configurations are changed. Thus, the optimization efforts can be concentrated on specific pipeline stages using a multi-arm bandit approach [100].

4.10 Summary

Vanir performs automatic cloud configuration optimization by splitting the optimization task into benchmarking and production phases. This allows Vanir to handle large configuration search spaces without requiring long offline benchmarking or optimization. Vanir finds configurations comparable to benchmarking based offline methods despite a 3× shorter benchmarking phase. Vanir has 1.3-24× lower optimization search cost, and it is 2× faster to reach within 15% of the execution time of the best configuration found across all optimization algorithms.

Chapter 5

Rethinking Resource Allocation for Serverless Functions

In this chapter, we discuss the limitations of the resource allocation options that current serverless offerings provide. We show that decoupling memory and CPU allocations as well as deploying serverless functions on different instance types can provide better execution time and execution cost. To cope with increased configuration options, we show how black-box optimization algorithms can be used to reduce the operational load on the serverless end-user.

5.1 Introduction

The serverless programming model has flourished in the last few years, mainly because it allows developers to concentrate on the application logic and not worry about scalability and resource management. Developers only have to write the code for one or more cloud functions, and have little to no control on the amount of resources the cloud provider uses to run them. Cloud providers take care of provisioning, deployment, scalability, and maintenance of the resources required for each function invocation.

Current serverless offerings in the cloud typically *couple* memory and CPU resource allocations together. Both AWS [84] and GCP [101] provide preset resource allocation configurations. In particular, AWS assigns a CPU share proportional to the amount of memory requested by the user (in a fine-grained way, but up to 10 GB); GCP provides users with seven preset resource allocation options to choose from. Azure Functions [102], in turn, guarantees at least 1 vCPU core to each function instance and allows up to 1.5GB of memory per function instance (but the user is charged based on the actual memory consumption). Additionally, none of the cloud providers provide the option to select the VM type that runs the function, even though the VM type used to run serverless functions is not always the same [103, 104].

This simple interface is one of the defining characteristics of serverless computing, with the advantage of removing the configuration burden from the user. This is in stark contrast with the complexity of selecting and provisioning machines from the *hundreds* of configuration options and VM types in regular

IaaS offerings [29, 12]. However, it also comes with significant disadvantages.

First, as we show later, in most cases, there are configurations that achieve better performance, better cost, or both, than the ones from current offerings. Second, when there are knobs, they are low-level: it is difficult for most users to translate memory and CPU to performance and cost. Third, the lack of full transparency hurts predictability, and even raises the question of whether the cloud provider is making the right choices to optimize resource usage, translating into lower costs for the users.

In this chapter, we take a step back and rethink, from a clean slate, the allocation of resources for the execution of serverless functions. In particular, we try to determine what can be gained by taking fine-grained control over the individual allocation of CPU, memory, and VM type to each serverless invocation. This requires us to understand how these allocation decisions influence the trade-off between performance and monetary cost, the predictability of these metrics, and ability to meet target execution times. Furthermore, we want to understand what is the minimal resource allocation that is required to meet such targets, if possible leveraging idle resources, whose type and availability may vary with time.

Despite the benefits of flexibility, simply offering a much larger configuration space to users negates the advantages of simplicity. Therefore, we also provide a thorough study of the effectiveness of *black-box* optimization algorithms to automatically determine the right resource configuration to remove the need for the user to deal with the complexity of choice, or to profile their functions to determine the right resources. The output of these algorithms can then be used directly in two possible ways: either by a cloud provider to *automatically* allocate resources for a user's functions, or by the serverless end-user if the cloud provider provides the *right configuration knobs*. In addition, we discuss different interfaces that an auto-tuning framework built using black-box optimization algorithms can provide to the end-user, achieving the best of both worlds: maintaining simplicity, while giving the user control of both cost and execution time, through a very simple dial-like mechanism. Alternatively, if automatic resource configuration is provided by the cloud vendor, we explore whether cloud providers can allocate the functions on different VM types to maximize the utilization of spare resources while providing predictable performance.

Our evaluation shows that the more flexible resource allocation option can provide up to 40% better execution time and 50% better execution cost than the restrictive resource allocation strategy that cloud providers currently use. We show that black-box optimization algorithms can reach within 10% of the performance of the best configuration in our resource allocation search space within 20 optimization trials (for a search space of 288 configurations). Finally, we show that cloud providers can reduce their costs (or conversely increase resource utilization) by utilizing different instance types for the same function, while providing performance within 10% of the best configuration.

Contributions. In this chapter, we make the following contributions:

• We determine the ground truth about the execution time and cost of 6 serverless applications across 288 resource configurations and multiple inputs (§5.2). Our benchmarks and data will be released as open source.

• We analyze the potential benefits of enabling a more flexible resource allocation for serverless functions (§5.4).

Resource Configuration	Values
CPU share	[0.25, 0.5, 0.75, 1, 1.25, 1.5, 1.75, 2]
Memory limit (MB)	[128, 256, 512, 768, 1024, 2048]
Instance families	[c6g, m6g, c5, m5, c5a, m5a]

Table 5.1: Resource allocation search space. A configuration is a value for CPU share, memory limit and instance family. For instance family names, a prefix 'c' means compute-optimized, and 'm' is a general-purpose instance; a suffix 'g' means Graviton2 ARM-based, 'a' AMD-based, and no suffix corresponds to Intel-based instances. The search space has 288 configurations.

• We analyze the accuracy of 4 Bayesian Optimization algorithms for determining the best resource allocation in terms of execution time and execution cost (§5.5).

• We determine whether the serverless functions of our study have data-dependent performance characteristics (§5.5.3).

• We propose a set of possible interfaces for enabling the user to benefit from multi-objective optimization (§5.6.1).

• We evaluate the cost reduction opportunities for the cloud providers by using different instance types while providing predictable performance (§5.6.2).

5.2 Motivation

Resource allocation decisions for functions can significantly impact the performance and cost of serverless functions. To characterize this performance and execution cost variation, we exhaustively test six benchmark functions on the IaaS resource configuration space defined in Table 5.1. (We defer to §5.3 further details of the setup for these experiments, including a description of the functions.) We run these benchmarks on AWS and adopt their nomenclature of instance families. As AWS does not provide information about the per-core or per-GB price, we calculate the execution cost based on assumptions and methodology defined in §5.3.2.

A resource allocation choice specifies a CPU share, a memory limit, and the instance family. The CPU share is the timeshare of a vCPU allocated to the function. So, a share of 0.25 means that a quarter of vCPU time is allocated to the function whereas a share of 1 or 2 implies allocations of 1 and 2 entire vCPUs, respectively. The memory limit, expressed in MB, is simply the amount of memory allocated to the function. In addition, we run functions on a diverse range of instance families, which influence the CPU type used by the function. The caption of Table 5.1 lists their nomenclature.

Figure 5.1 shows a box plot summary[1] of the normalized execution time and execution cost for each function across the entire configuration search space of Table 5.1. For each function, the normalization is done w.r.t. the best (minimum) execution time and execution cost for that function in the search space.

We observe that in the worst case, selecting the wrong configuration can lead up to 14.9× worse execution time and 5.6× worse execution cost compared to the best configuration. From this result, we conclude that an incorrect resource allocation can lead to significant performance and cost penalties.

[1] All boxplots in this chapter show median, 1st and 3rd quartiles, with whiskers showing the distribution 1.5× IQR past the high and low quartiles, and anything beyond is shown as outliers.

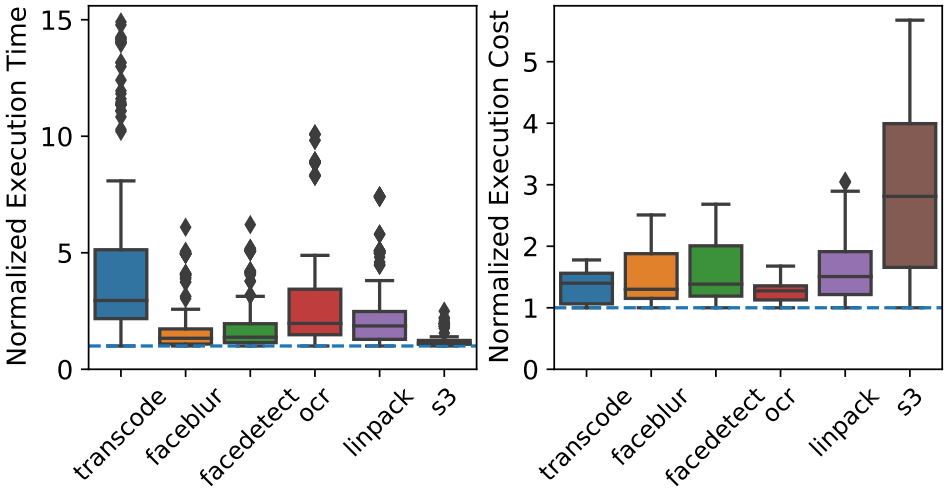


Figure 5.1: Execution time and cost of each function across the entire configuration space defined in Table 5.1, normalized w.r.t. the best configuration of each function.

Now that we have established the contrast between the best and worst-case performance of the serverless functions in the search space from Table 5.1. In the rest of the chapter, first, we will explain our experimental setup. Then we discuss the potential benefits of decoupling CPU and memory resources and using different instance types for serverless function. These motivational results are followed by a discussion of techniques and design choices involved in realizing this vision of more flexible resource allocation for serverless functions.

5.3 Experimental setup

To understand the impact of different resource allocation decisions on the cost and performance of running serverless functions, we use the OpenFaaS [105] serverless framework to execute and measure the performance of a diverse range of benchmark functions across the resource allocation search space. We deploy OpenFaaS atop the k3s distribution [106] of Kubernetes as a cluster running on AWS EC2 instances. We prepare multi-architecture Docker containers (amd64 and arm64) for all benchmark functions using *docker buildx* without using any platform-specific optimized libraries.

We collect the ground truth performance and cost data as follows. We execute each function using multiple input data samples on every resource allocation in the configuration search space of Table 5.1. To minimize the impact of performance outliers, we execute each function at least 5 times on a given configuration. We use a function execution timeout of 600s, which is comparable to the timeouts in current serverless offerings [107]. The input samples – while arbitrary – were chosen from publicly available datasets.[2] One input sample is used as the default but we analyze how performance depends on input data (a modest 20% at most, §5.5.3). Overall, we run over 5,000 combinations of resource configurations, benchmarks and input samples. From this ground truth data, we identify the overall best configuration for each function with regard to both execution time and execution cost.

[2]Input data samples for each benchmark will be made public.

Name	Purpose	Language	Important Resources			
			C	P	M	N
facedetect	Image face detection	Go	•			
faceblur	Image face blurring	Go	•			
transcode	Video transcoding	Python & C	•	•	•	
ocr	Optical character recognition	Python & C++	•	•		
linpack	Solves linear equations using matrices	Python	•		•	
s3	Download and then uploads an object from one S3 bucket to another	Python				•

Table 5.2: Benchmark serverless functions. Each benchmark has certain resources that are more important than others, either for performance or to avoid function failure. Resources: C: CPU, P: parallelism, M: memory, N: network.

5.3.1 Benchmark functions

Table 5.2 shows the benchmark functions used in this work. These benchmarks are a mix of applications taken from prior works and applications from OpenFaaS function store [108]. transcode's and ocr's request handlers are in Python but use internal bindings to invoke C and C++ applications, respectively. facedetect and faceblur use Go libraries [109, 110]. linpack uses an application from FunctionBench [111]. Both transcode and ocr are able to effectively utilize > 1 vCPU.

5.3.2 Cost calculation

To perform cost analysis in our resource allocation setting, we have to determine the pricing per-vCPU and per-GB memory. The current pricing model of AWS Lambda charges users primarily based on GB per ms consumed by a function execution. Users only select the amount of memory allocated to each function instance. Thus, the precise per-vCPU and per-GB memory pricing necessary for execution cost calculation for this work are not available. Therefore, we have to create a reasonable pricing model.

We use the AWS instance pricing to calculate the per-vCPU and per-GB memory costs for different instance types. We adopt a simple cost model that assumes that CPU and memory are the two components contributing to the total cost of an instance. We calculate per-vCPU and per-GB memory costs by solving a system of linear equations for instances with the same CPU architecture type assuming that those instances would have the same per-GB pricing. Each equation is of the form below that defines the instance price $P_{instance}$ (given as input) as a sum of per-vCPU cost X and per-GB of memory cost Y:

$$\alpha X_{vCPU} + \beta Y_{mem} = P_{instance} \tag{5.1}$$

To create a well-defined system of equations, we use publicly available information from AWS to determine the number of vCPUs α and the amount of memory in GB β for a sufficient number of equations. For example, x86 instances m5, c5, and r5 are assumed to have the same per-GB memory cost Y_{mem}.

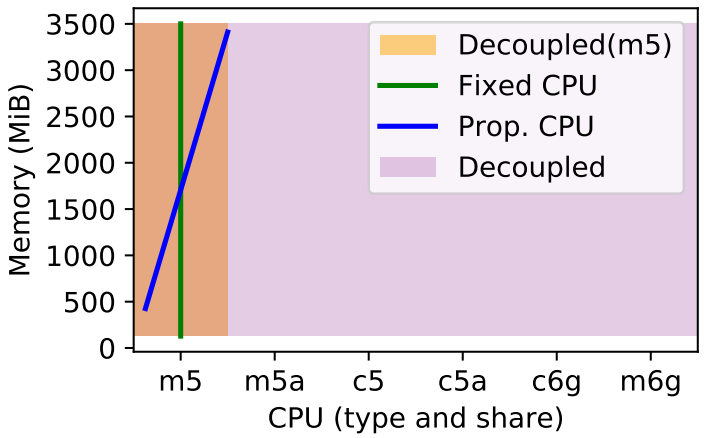


Figure 5.2: Four strategies for configuring serverless functions. Each strategy defines a search space, corresponding to a subset of the possible resource allocation configurations.

Moreover, m5 and r5 instances have same CPU type. Thus, we have X^1_{vCPU} for c5 and X^2_{vCPU} for m5 and r5. This yields a system of equations with 3 unknowns and 3 equations, using $\alpha = 2$ and $\beta = 4, 8, 16$ for c5, m5, and r5 instances, respectively. While we do not use r5 instances, its pricing information is used to solve the system.

We use the same approach to calculate per-vCPU and per-GB costs for ARM-based and AMD-based instances. The values for α and β are the same as mentioned above, for these instance families.

5.4 How beneficial is flexible resource allocation?

To understand the opportunities that current serverless offerings are missing, our study starts by characterizing the best execution time and cost for different resource allocation options and the potential for using different instance types for the functions while providing predictable performance.

5.4.1 Larger search spaces yield advantages...

First, we want to illustrate the best execution time and cost available within the search spaces provided under different resource allocation options.

Setup. We consider four strategies for resource allocation, with an increasing level of flexibility. Each strategy corresponds to a subset of the configuration search space, as depicted in Figure 5.2. The first three strategies assume a fixed instance type, which is the m5 type in our experiments.

Fixed CPU allocates a single-vCPU for each function instance, whereas the memory is charged based on the average actual consumption. This strategy is inspired by Azure Functions.

Prop. CPU allocates a share of CPU proportional to the amount of memory selected. This strategy is inspired by AWS Lambda and Google Cloud Functions.

Decoupled (m5) decouples CPU and memory allocations. In this case, the search space includes all CPU and memory values in Table 5.1, but it uses only the default m5 type.

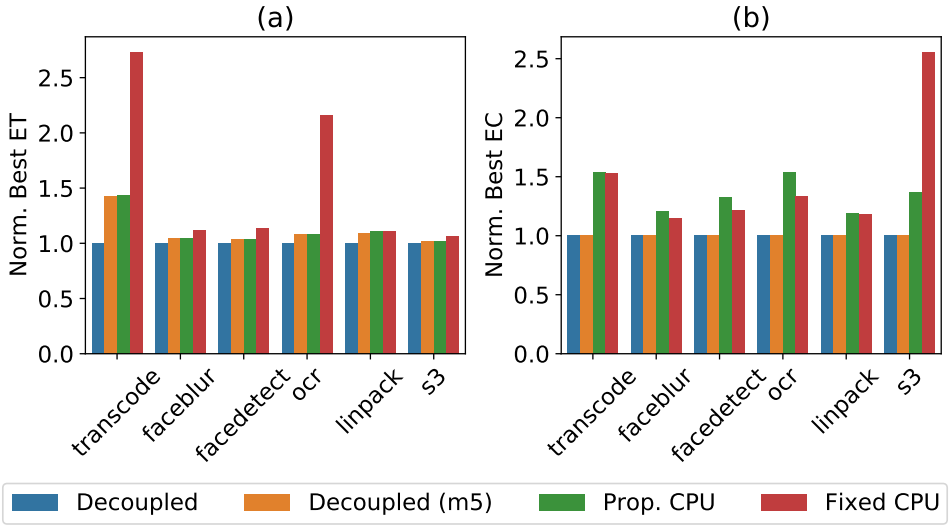


Figure 5.3: Potential gains within each search space. The graphs show the best (a) Execution Time (ET) and (b) Execution Cost (EC) of each function across different search spaces, normalized to the overall best configuration.

Decoupled has the largest search space, encompassing all other strategies and covers the search space in Table 5.1.

Results. We measure the best execution time and execution cost for each benchmark function in each of the different resource allocation search spaces. Figure 5.3 shows best execution time (ET) and execution cost (EC) of each strategy's search space normalized w.r.t. the best possible ones in *Decoupled*, since its search space includes all others.

Observations. Figure 5.3a shows that by using different instance types, *Decoupled* can provide 5%-40% better execution time than *Decoupled (m5)* and *Prop. CPU*. In addition, decoupling memory and CPU in *Decoupled (m5)* is sufficient to provide 10%-50% better execution cost compared to *Prop. CPU*, as shown in Figure 5.3b.

Fixed CPU leads to 2.7× and 2.1× higher execution time for transcode and ocr, respectively and 2.6× higher execution cost for s3. This is because, for transcode and ocr, having 1 fixed vCPU per function invocation does not exploit available parallelism opportunities. For s3, *Fixed CPU* has higher execution cost because the function is not compute intensive and its execution time already plateaus with CPU share < 1.

Takeaways: Using different instance types allows for improving the execution time, while decoupling CPU and memory enables potential improvements in execution cost compared to currently deployed resource allocation strategies.

5.4.2 ... and potential to reduce idleness

We now turn to another potential advantage of flexible resource allocation: there may be cloud resources that are idle, but of the "wrong" instance type for a given function. *Decoupled* affords us the opportunity to use different instance types when doing so provides sufficient performance (when compared to the best configuration) according to different objectives. We quantify these opportunities below.

Benchmark	Execution Time			W_t = 0.25 & W_c = 0.75			W_t = 0.5 & W_c = 0.5			W_t = 0.75 & W_c = 0.25			Execution Cost		
	Threshold (θ)			Threshold (θ)			Threshold (θ)			Threshold (θ)			Threshold (θ)		
	5%	10%	20%	5%	10%	20%	5%	10%	20%	5%	10%	20%	5%	10%	20%
ocr	2	4	5	1	1	2	1	1	4	1	3	4	1	1	2
transcode	0	0	2	2	2	2	2	2	2	0	2	2	1	2	2
faceblur	1	2	2	0	1	3	1	1	3	1	1	2	0	3	4
facedetect	1	4	4	1	1	3	2	3	3	2	3	3	1	1	3
linpack	2	2	4	0	2	3	0	2	3	1	1	4	1	1	3
s3	3	5	5	0	1	5	1	4	5	0	4	5	0	0	3

Table 5.3: Number of alternative instance types with one or more resource allocation configurations with the performance metric within threshold (θ) of the best configuration in the *Decoupled* search space. The cells in red indicate cases where there is no alternative instance type able to reach within θ% of the best configuration. The cells in blue denote cases where all instance types have at least one configuration that provides performance within θ% of the best configuration.

Setup. To find the number of different instance types (from our search space) that provide satisfactory performance, we use multiple performance thresholds (θ) w.r.t the best configuration and multiple weighted combinations of execution time and cost performance objectives.

Results. Table 5.3 shows the number of instance types that have at least one configuration that is within θ% of the best *Decoupled* configuration. This table demonstrates the potential of using alternate instance types while providing comparable performance for different performance objectives for each benchmark function. The objectives are execution time (left), execution cost (right), and three weighted combinations of the two (denoted with W_t and W_c for execution time and cost, respectively).

Observations. We highlight two types of special cases in the table. The cells in red indicate cases where there is no alternate instance type able to reach within θ% of the best configuration. In other words, choosing a different instance type degrades performance by more than θ%. The cells in blue denote cases where all instance types have at least one configuration that provides performance within θ% of the best configuration.

The likelihood of being able to use idle resources clearly increases with a higher count of alternate instance types and depends on the function as well as the parameter θ.

Takeaways: We found that except for two scenarios, there are opportunities to use idle instances of different types while providing performance within 10% of the best configuration.

5.5 Is automatically discovering good configurations possible?

As seen, more flexibility in the choice of resource allocation is beneficial. However, in the presence of a large search space, the task of determining the right configuration can be daunting, requiring performance modeling and/or profiling, and particularly, it contrasts with the overarching goal of serverless — to be a hassle-free computing service. Thus, we now turn to exploring the effectiveness of using *black-box* optimization algorithms – briefly reviewed below – to automatically determine the right allocation of resources.

5.5.1 Background on optimization techniques

As previously surveyed in § 2.2, several recent works have developed techniques for automatic cloud configuration [12, 14, 112, 113]. At the core of many of these approaches there lie black-box optimization techniques ranging from model-based optimization algorithms to sampling-based search techniques. This is in contrast to other works, such as Ernest [23], that rely on analytical modeling to create a mathematical performance model that is dependent on the characteristics of the application and thus varies from one application to another.

We believe that black-box optimization methods are a good fit for serverless functions, since the search space is large, function invocations can provide performance indicators, but the underlying objective function remains a black-box. Even though the resulting configurations may be, in some cases, suboptimal compared to more precise analytical models, we believe that the advantage of being able to quickly reuse existing algorithms outweighs those limitations.

We next briefly review a range of optimization techniques that we use to automatically determine resource allocations for serverless functions. Finally, we point out a relevant change when using these methods for the serverless scenario.

Model-based algorithms build a model of the underlying black-box objective function. Following a comprehensive study [112] of black-box optimization algorithms, we adopt the best-performing method, which is Bayesian Optimization (BO). We consider four variants of the surrogate model (required for approximating the objective function): (1) Gaussian Processes (GP), (2) Gradient Boosted Regression Trees (GBRT), (3) Random Forests (RF), and (4) Extra Trees (ET). In all cases, we use the popular Expected Improvement (EI) as the acquisition function.

We use the Scikit-Optimize Python library [71] as a readily-available implementation of these algorithms and use its default parameters unless otherwise stated. By default, based on previous findings [12], we use three random initial samples to bootstrap model-based algorithms.

Sampling-based search techniques sample the search space to find good configurations. These methods are simple to implement and easy to parallelize. We use both Random Sampling as well as Latin Hypercube Sampling (LHS) [114]. LHS samples the search space using a space filling design to generate near-random samples. We use pyDOE [115] to generate LHS samples.

Adapting to the serverless scenario. An issue that arises when using the above techniques out-of-the-box is that they can produce configurations on which the function fails because not enough memory is allocated. At first, we attempted to address this by assigning a large value to represent the performance objective (e.g., execution time) of a failed function invocation. However, that created a non-smooth underlying function, which affects the quality of the optimization.

Problem formulation. Therefore, we instead deal with this issue by slicing the search space to remove from it the resource configurations with memory less than or equal to every memory configuration for which we determine that the function failed. This is based on the simple property that if a function fails for a certain memory limit, it is very likely to continue to fail with a lower memory limit. Thus, every time we record a function failure, the search space is dynamically reduced, removing all configurations that would lead to failure due to a lower memory limit.

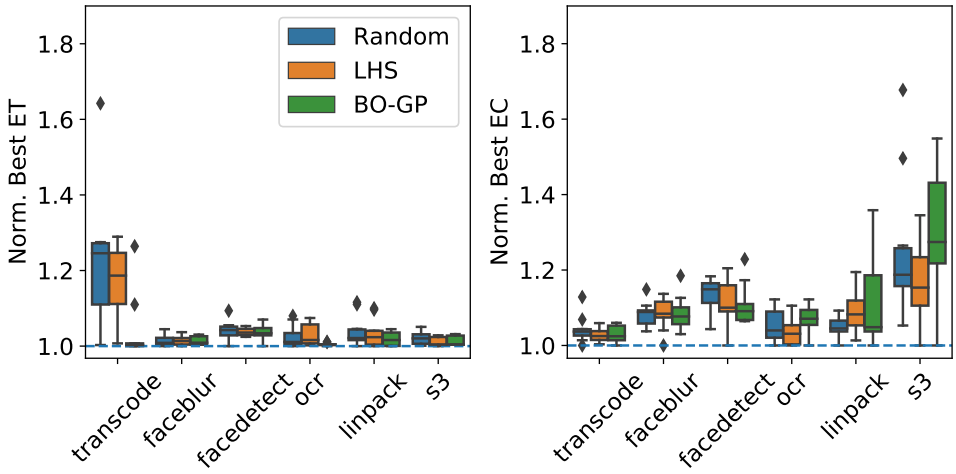


Figure 5.4: Performance of the best-found configurations for sampling-based and model-based algorithms.

In this chapter, the configuration x is a serverless resource configuration that is represented as a tuple $x = (S_{cpu}, S_{mem}, I_F)$ where S_{cpu} and S_{mem} represent the CPU and memory share, respectively, and I_F is the instance family, as shown in Table 5.1. Thus, for this chapter, Equation 2.1 can be simplified into the following, where f is either execution time or execution cost:

$$x^* = \arg\min_{x \in X} f(x) \tag{5.2}$$

Next, we present the methods and results of our study, structured by its main findings.

5.5.2 Optimization techniques are effective

Sampling-based vs. model-based algorithms. We first study how well black-box algorithms optimize the resource allocation as compared to the ground-truth best configuration in the *Decoupled* search space. We fix a budget of 20 trials for each method and report on the best execution time and cost found within those trials. For the sampling-based methods (Random Sampling and LHS), this entails generating 20 samples. For the black-box optimization method (BO), we start with 3 initial samples and then use the acquisition function to repeatedly sample from the search space a series of configurations to test until the budget is matched. We repeat the optimization process 10 times using different random seeds (for sampling) and initial samples (for BO).

Figure 5.4 shows the best-found execution time and execution cost normalized w.r.t. the best configuration in the search space. The boxplot captures the variation in the best-found configuration across different repetitions. For these results, we only show the values for BO with GP, because this method outperforms other BO variants.

We observe that both sampling methods and BO with GP perform comparably in most cases, although GO with GP finds a better execution time for transcode, whereas sampling methods reach a better execution cost for s3. We obtained similar results for weighted combinations of execution time and execution cost.

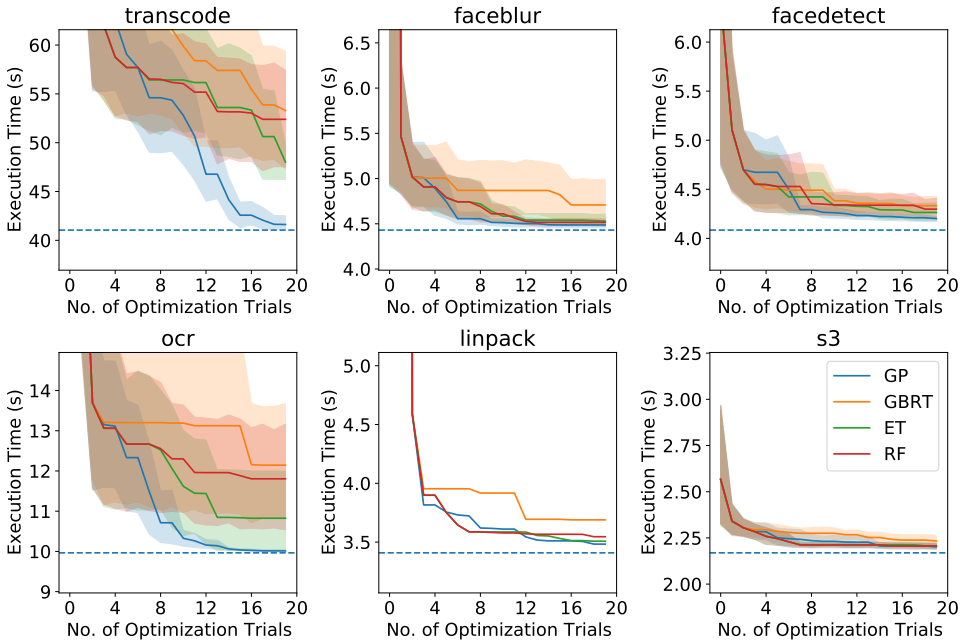


Figure 5.5: Execution time performance of the best found configuration as optimization by different BO variants progresses.

We additionally note that, while sampling-based methods are simpler to use, model-based methods generate models that can provide predictions for yet-unseen configurations. Therefore, given the overall good performance of model-based methods and the importance of predicting configurations with larger search spaces, we mainly use model-based methods in the rest of this chapter.

Convergence speed of model-based algorithms. We now analyze how fast model-based methods converge towards the best configuration in the search space for each benchmark function. This analysis offers an early indication of how many optimization trials would be necessary as a baseline, as we later turn to online optimization in §5.5.4.

We run the four BO variants for 20 steps (including the 3 initial samples), using execution time and execution cost as the performance objective. Figures 5.5 and 5.6 show the execution time and execution cost, respectively, of the best-found configuration as the optimization process progresses. The dashed lines denote the overall best execution time (resp. execution cost) in the search space. We repeat the experiment 10 times for each optimization method using different random initial samples. The shaded area is the 95^{th} percentile confidence interval.

With regard to execution time, while almost all optimization algorithms perform comparably, BO with GP overall tends to outperform other methods since it reaches configurations with comparable or better execution time compared to the second-best optimization method. In particular, BO with GP reaches within 5% of the best execution time in 20 optimization trials, in all cases.

When optimizing for execution cost, the best configurations are harder to find and thus there is a larger gap between the best-found configurations and the overall best ones in the search space. This is particularly true for s3 and facedetect. In 5 out of 6 benchmarks, BO with GP finds configurations that are within 20% of the best execution cost in the search space. But for s3, it only finds configurations that

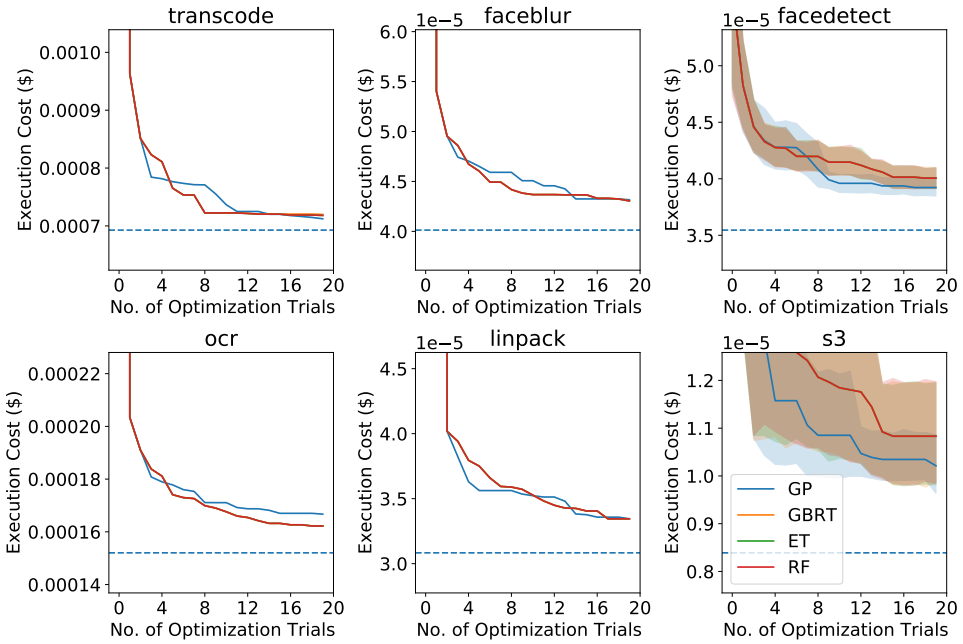


Figure 5.6: Execution cost performance of the best found configuration as optimization by different BO variants progresses.

are within 30%. However, similarly with optimizing for execution time, BO with GP either outperforms or performs comparably to other BO variants.

Takeaways: While both sampling-based search and model-based optimization methods find good resource allocations, we favor model-based methods since they can provide predictions for untested configurations as well. Different BO variants perform comparably when optimizing for execution time and execution cost, with an advantage for BO with GP.

5.5.3 Input data variation has modest influence

The performance of a function in general depends on its input data. For instance, transcode significantly depends on the input video dimension whereas facedetect and faceblur depend on the input image size. Indeed, all of our benchmark functions have input data-dependent execution time.

Dealing with data dependence is challenging. One approach would be to create data-specific performance models that account for the input data characteristics. However, this adds significant complexity to the optimization framework. First, the model needs to be sophisticated enough to encapsulate these performance-determining characteristics, which is especially difficult if the relation between input data and execution time is complex (e.g., if it has many modalities that require advanced profiling [116]). Second, even utilizing such a performance model is difficult because, at invocation time, the serverless framework would need to evaluate with minimal overhead the input data and route accordingly.

A simpler approach instead is to use a generic optimization process to create a model using representative input data samples. This is based on the intuition that, even though the absolute performance may vary, a good configuration for one input data sample might also be a good configuration for other input data samples.

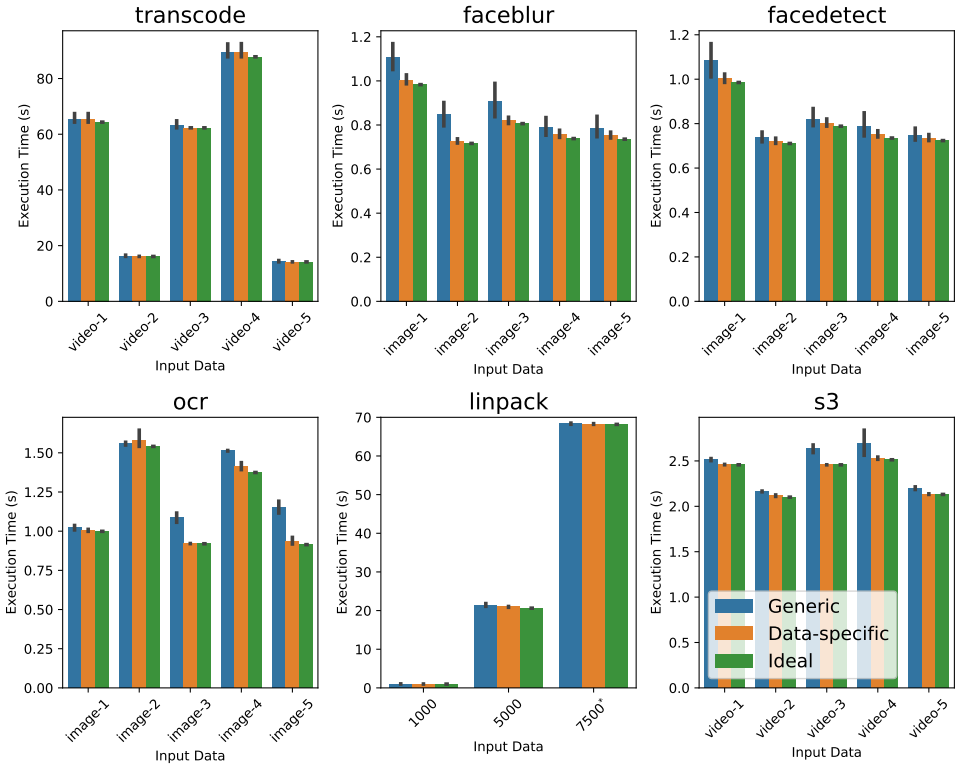


Figure 5.7: Execution time of the best configurations found by a generic optimization process (blue), a data-specific optimization process (orange), and the overall ideal configuration (green).

We therefore study to what extent the resource configuration depends on input data and the effectiveness of this second approach. To this end, we use the methods in §5.5.2 so that, for each function, we use the default input to create a single generic model as well as 10 input-specific models, one for each input sample.

Figure 5.7 contrasts the performance of the best-found configuration for the two model types – generic (blue) vs. input-specific (orange) – for each input, together with the overall best configuration (green) in the search space for that same input. For clarity, we only show the results of 5 out of 10 input samples (except for linpack). We show the optimization scenario for execution time and note that optimizing for execution cost obtains similar results.

In our experiments, using input-specific models provides up to 20% better execution time. This modest improvement in performance comes with significantly increased complexity. In addition, in a real setting, each input cannot have its own model. (We leave as future work to create more sophisticated performance models.)

linpack with a $N = 7500$ matrix is a special case since it requires a large memory and the optimization process using a default input leads to lack-of-memory failure in 3 out of 10 repetitions of the optimization process (which we exclude). Similarly, for larger video sizes (for transcode application), more memory would be required for the function. However, this can be dealt with by splitting large videos into smaller chunks and processing them in parallel to ensure predictable performance.

Thus, we conclude that our experiments expose a trade-off between framework complexity and potential gains (up to 20%) due to accounting for input-specific characteristics. Consequently, in the rest

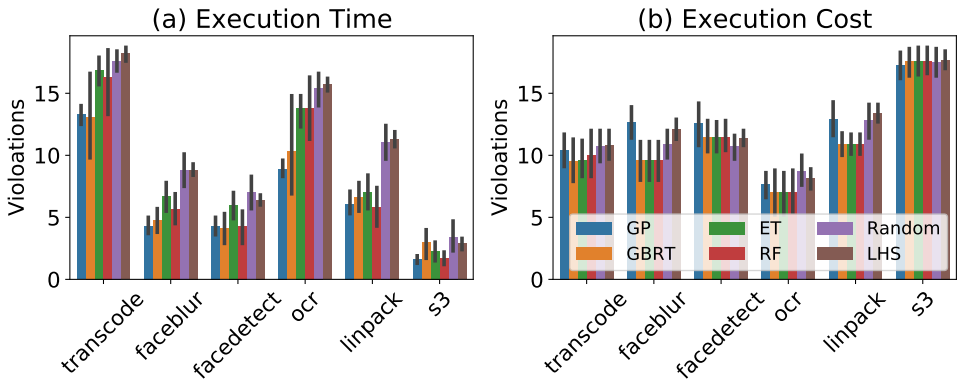


Figure 5.8: Average number violations during online optimization for (a) execution time (ET) and (b) execution cost (EC).

of the chapter, we adopt the generic models.

Takeaways: While the performance of a serverless function in general depends on its input data, our experiments indicate that configurations that are good for one input sample are also good for others. Input-specific optimization leads up to 20% improvement in execution time, but because input-specific performance models are more complex to create and maintain, this trade-off needs to be considered carefully.

5.5.4 Online optimization is feasible

The optimization of resource configurations can occur in two ways: (1) offline or (2) online. Offline optimization requires running the optimization process described in §5.5.2 upon function deployment with representative input samples. Conversely, online optimization exploits function invocations in production as trials of the optimization process. However, in the online scenario, it is important to reduce the possibility of performance degradation due to trials with bad configurations. Thus, we now analyze which optimization methods lead to fewer degraded runs during optimization.

Figure 5.8 shows the average number of violations during 10 repetitions of the optimization process. Here we consider a violation when the performance objective is at or above 1.5$\times$ the objective value for the best configuration in the search space. On average, BO with GP has a number of violations comparable to sampling-based search techniques for execution cost, but it has a lower number of violations for execution time. Overall, sampling-based search techniques have a slightly higher number of violations compared to model-based methods.

Takeaways: BO with GP has slightly lower average number of violations for execution time compared to other methods. For execution cost, it has slightly higher number of violations compared to other BO variants, but overall it leads to fewer violations than sampling-based search methods.

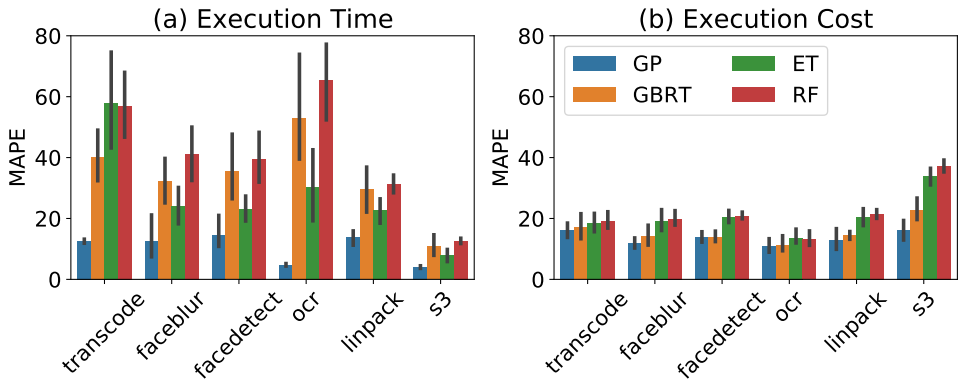


Figure 5.9: MAPE for different benchmarks and optimization methods when optimizing for (a) execution time and (b) execution cost.

5.5.5 Resource allocation models can predict performance of untested configurations

One of the key advantages of model-based methods over sampling-based methods is that model-based optimization methods can predict the performance of an untested configuration. However, if we find that model-based methods have a low prediction accuracy, then sampling-based search methods, which are simpler, maybe sufficient after all. Therefore, we now analyze how well model-based methods can predict the performance objective across configurations in two different scenarios: (1) over the entire *Decoupled* search space (except the failed runs), and (2) when the configuration prediction is restricted to match a particular instance type. This second scenario is relevant in the context of helping the cloud provider utilize idle resources of different instance types while providing predictable performance (as we elaborate in §5.6.2). We repeat every measurement 10 times and report average and 95^{th} percentile confidence interval of the error metric.

Scenario 1. Figure 5.9 shows, for each BO variant, the Mean Absolute Percentage Error (MAPE) across all configurations between the actual execution time/cost and the predicted value. We observe that BO with GP has lower error than other optimization algorithms. Compared to other variants, BO with GP, on average, has up to 16× and 2.3× lower MAPE for execution time and execution cost objective, respectively.

Scenario 2. Figure 5.10 shows the MAPE across the best predicted configuration for each instance type. Similar to the previous case, BO with GP generally outperforms other BO variants in most cases, except for transcode and ocr when optimizing for execution cost. BO with GP, on average, has up to 7× and 3.5× lower MAPE than other variants for the execution time and execution cost objective, respectively.

Takeaways: BO with GP not only works well when it comes to convergence towards the best configuration (as shown in §5.5.2) but overall has lower prediction error compared to models created using other BO variants. In our experiments, models built with BO with GP provide up to 16× lower MAPE than models build with other BO variants. This means that performance models built with BO with GP are better at predicting the performance of untested configurations.

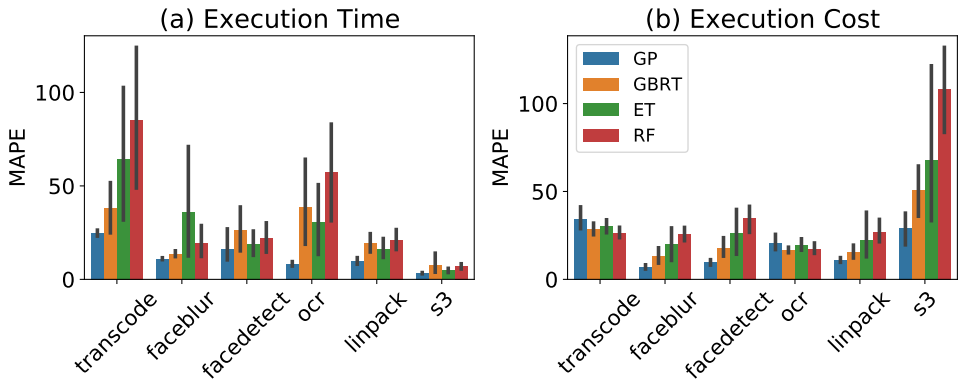


Figure 5.10: MAPE for different benchmarks and optimization methods when comparing the best configuration for each instance type, when optimizing for (a) execution time, and (b) execution cost.

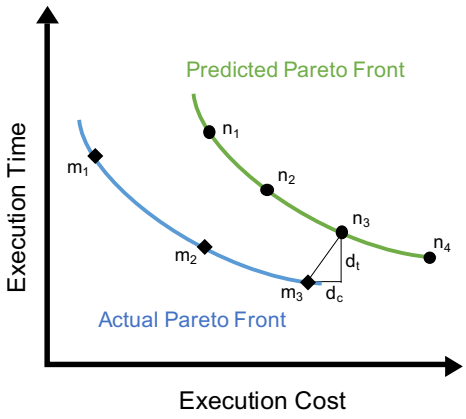


Figure 5.11: Method for measuring the distance between the configurations in the predicted and actual Pareto front. The distance is split into execution time and execution cost components and measured between each configuration in the predicted Pareto front and the nearest configuration in actual Pareto front.

5.6 Automatic resource allocation from the provider's perspective

Having looked at the potential gains from flexible resource allocation, and the effectiveness with which we can exploit these gains using black-box optimization algorithms, in this section we discuss how the cloud provider can not only expose this to users without unduly complicating the 'serverless' interface, but also leverage model predictions to opportunistically select available instance types that can reduce costs without significantly sacrificing performance.

5.6.1 On the interface between user and provider

Given the performance and cost benefits of flexible resource allocations, the provider could expose the knobs for fine-grained resource configuration (selecting instance type, memory and CPU allocation separately). While possible, this would, however, shift the complexity of configuration selection back to the user, and negate one of the big advantages of serverless: its simplicity. Another alternative would be for the provider to abstract away that interface and allocate resources automatically and transparently, but this would prevent the user from choosing different points in interesting and relevant trade-off spaces. As

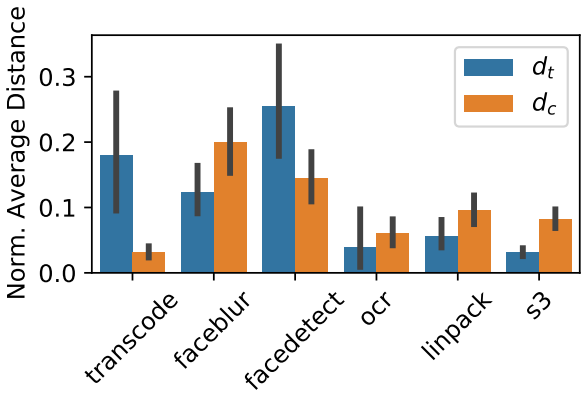


Figure 5.12: Normalized absolute average distance between the points in the predicted Pareto front and the actual front for the default input. We show normalized execution cost (d_c) and execution time (d_t) components of the distance separately.

we will show next, our automatic exploration of the configuration space allows the best of both worlds: a simple, high-level interface that exposes to the user the cost/performance benefits of decoupled resource allocation.

We describe three ways to allow users to select a trade-off between execution time and cost: 1) Providing configurations from the predicted Pareto front, 2) Weighted multi-objective optimization [117] to provide best configurations for different weights, and 3) Hierarchical multi-objective optimization [117] to satisfy a user-provided trade-off constraint.

Pareto front: We can use the predictions from the black-box model to create a Pareto front and expose to users the configurations that have different trade-offs between execution time and execution cost. The predicted Pareto front is created by normalizing the execution time and execution cost so that the values of the two objectives are on a similar scale. Since we do not know the actual minimum value of the objectives, we use the minimum values observed while optimizing execution cost and execution time to perform normalization. Therefore, two models have to be trained to create a Pareto front.

To assess the effectiveness of using the configurations from the predicted Pareto front, we measure the distance of those configurations from the predicted Pareto front to the nearest configuration in the actual Pareto front, as shown in Figure 5.11. We measure the distances in terms of normalized execution time (d_t) and cost (d_c) separately. d_t and d_c for each configuration are normalized using the corresponding objective value for the nearest configuration in the actual Pareto front.

Figure 5.12 shows, for each function, the average distance between the points in the predicted Pareto front and the actual Pareto front for the default input. Note that the prediction error in the model leads to $d_t \geq 0, d_c \geq 0$. In our experiments, the average difference between the configurations in predicted Pareto front and actual Pareto fronts is up to 20% (cost) and 25% (time).

The interface to the user exposes the small set of configurations (between 2 and 10) in the Pareto front, with the corresponding predicted cost and execution times.

Weighted multi-objective optimization: With weighted multi-objective optimization, the cloud provider can select relative weights for execution time (W_t) and execution cost (W_c), where $W_c = 1 - W_t$. Using these weights, the cloud provider can form a weighted objective function $F_w(X)$ for a configuration X,

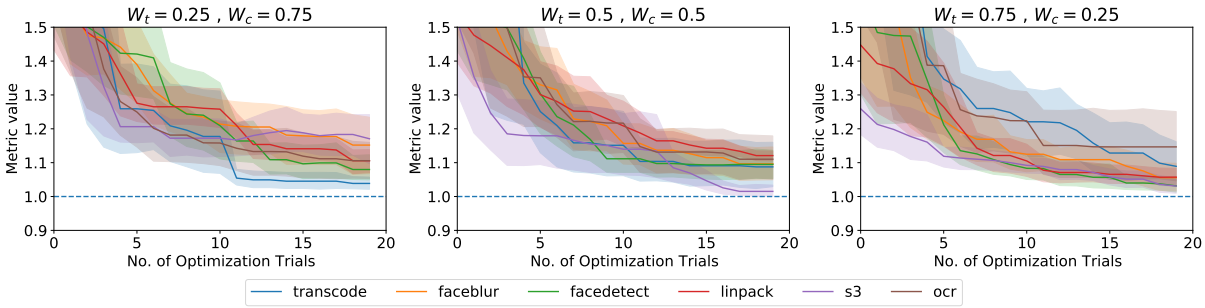


Figure 5.13: Convergence of the optimization process (BO with GP) for all the benchmarks and different weights for execution time and execution cost.

from normalized objective functions for execution time ($F_t(X)$) and cost ($F_c(X)$):

$$F_w(X) = W_t \times \frac{F_t(X)}{B_t} + W_c \times \frac{F_c(X)}{B_c} \tag{5.3}$$

where B_t and B_c are the minimum values for execution cost and execution time objectives found during the optimization process for $F_t(X)$ and $F_c(X)$. To simplify the process for the user, we train three models with $W_t \in \{0.25, 0.5, 0.75\}$. The two models trained first for execution time and execution cost (to obtain B_t and B_c) correspond to $W_t = 1$ and $W_t = 0$, giving a total of five models and five best configurations (one configuration suggested by each optimization process) for the user to choose from.

Figure 5.13 shows the best configurations found by the weighted multi-objective optimization (using BO with GP) as we perform more optimization trials. We can see that even in the weighted multi-objective setting, for most cases, the optimization process is able to find configurations that are within 20% of the best configurations in the search space after 20 optimization trials. While not shown here, we increased the number of trials and observed that within 40 trials, BO with GP is able to find configurations with performance within 5-10% of the best configuration, for all cases.

The resulting interface is very similar to the Pareto front one: in this case, the user can choose between at most 5 configurations, based solely on their predicted cost and performance.

Hierarchical multi-objective optimization: In hierarchical multi-objective optimization, we first optimize one of the objective functions (primary objective). Then, the optimized model can be used to find configurations that minimize the value for the second objective function (secondary objective) while degrading the primary objective value by at most a user-defined amount (θ). This optimization process may be easier for users to reason about. Once they are shown the best value for the primary objective function, they can decide if they are willing to degrade that value by a certain percentage to improve the secondary objective function's value. For example, users can choose to increase the execution time by 20% compared to the best execution time found as long as the corresponding execution cost decreases.

Figure 5.14 shows the normalized value for execution time and execution cost metrics after the hierarchical optimization (satisfying user's constraints) for the two combinations of primary and secondary objective functions. We used a threshold of $\theta = 20\%$ for this experiment. ET/EC and ideal-ET/ideal-EC represent the best configuration using the prediction model and oracle-like knowledge. The normalization is done w.r.t. the best configuration found after optimizing the primary objective only. The dashed line

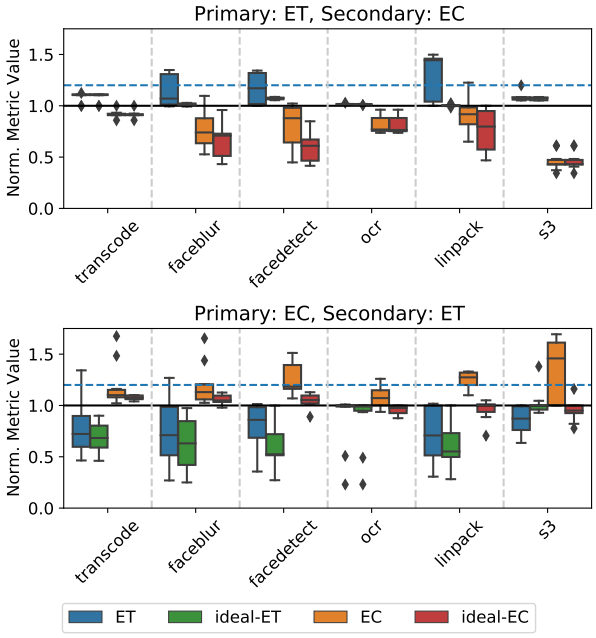


Figure 5.14: Normalized values for execution time and execution cost after a hierarchical optimization. ET/EC and ideal-ET/ideal-EC represent the best configuration using the prediction model and with oracle-like knowledge, respectively. The normalization is done w.r.t the best configuration observed during optimization process

shows the user-specified degradation threshold for the primary objective. In some cases, we can see that prediction error leads to a higher degradation of the primary objective than the threshold. But in other cases, hierarchical optimization using the performance models performs comparably to the ideal case. Unlike weighted multi-objective optimization, only one model is trained for hierarchical multi-objective optimization.

Takeaways: While not the final answer, these three options give the user access to a much broader space of configurations than current offerings, without requiring the user to deal with complex resource allocation choices. In fact, they shift the language from *resources* to *outcomes*: performance and cost. They also represent different trade-offs in terms of simplicity and effectiveness. With Pareto front and weighted multi-objective optimization users would simply get a small set of cost-performance tuples that they can select from. In our experiments, both methods found configurations that were up to 30% worse than the best in both cost and performance. The hierarchical optimization offers a more explicit prioritization, but users have to choose the threshold themselves. It also has good results: for an increase of roughly 20% in the primary metric, it leads to a reduction of up to 50% in the other. They also present different costs for the provider. For the hierarchical multi-objective optimization and the Pareto front, we need to train one and two models, respectively. For weighted multi-objective optimization, the number of models we need to train depends on the number of weighted combinations of ET and EC. A full evaluation of these interfaces would require user studies and a more thorough operating cost analysis of each interface (from the perspective of cloud provider), and we leave it as future work.

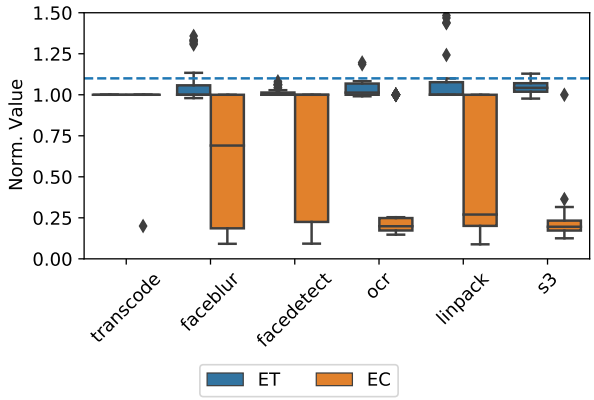


Figure 5.15: Reduction in cloud provider costs while keeping the objective function value within 10% of the best found configuration in the search space. Cost reduction is based on model predictions for the best configurations of each instance type, assuming an 80% discount in pricing for idle resources.

5.6.2 Cost-performance trade-off while using different instance types

We now turn our attention to evaluate how effectively we can utilize the potential for using idle resources to make serverless more cost-effective while maintaining good performance, as shown in Table 5.3. In particular, we measure this benefit by translating utilization of idle resources into a cost decrease. Similarly to spot instances, we assume that a serverless instance type with many idle instances is assigned a lower cost, to incentivize the utilization of the idle resources. We assume that the spot pricing for the serverless instance will decrease the per-CPU and per-GB cost to a fraction of the original price.

Figure 5.15 shows the decrease in deployment cost that the cloud provider can observe by utilizing the best configurations for each instance type predicted by the model. For this figure, we assume that spot pricing is 20% of the normal pricing [3]. Figure 5.15 shows the decrease in execution cost while the performance model is predicting configurations that are within 10% of the execution time (marked by the dashed line) of the best found configuration. The figure shows the normalized value for execution time and execution cost w.r.t. to the best configuration found by the optimization process.

We can see from Figure 5.15 that, by using the predicted best configurations of other instance types (which might have more idle instances), we can achieve between 25-75% reduction in execution cost, on average, for different benchmarks. This execution cost reduction comes at a $<$10% increase in execution time, on average. There are, however, outliers where the execution time penalty is up to 50% because of prediction error. The main exception is the transcode application, for which there are very few execution cost reduction options available, as shown in Table 5.3.

Takeaways: In our experiments, we found that some configurations utilize different instance types but provide performance similar to the best configuration in the search space. A cloud provider can exploit this behavior and use prediction models to achieve lower execution costs (by using idle resources) while providing comparable execution time to the best found configuration. Even with prediction error, we show that it is possible to significantly reduce costs while delivering performance within 10% (on average) of the execution time of the best found configuration.

[3] Spot instances prices are usually around 10% to 30% of on-demand instance prices

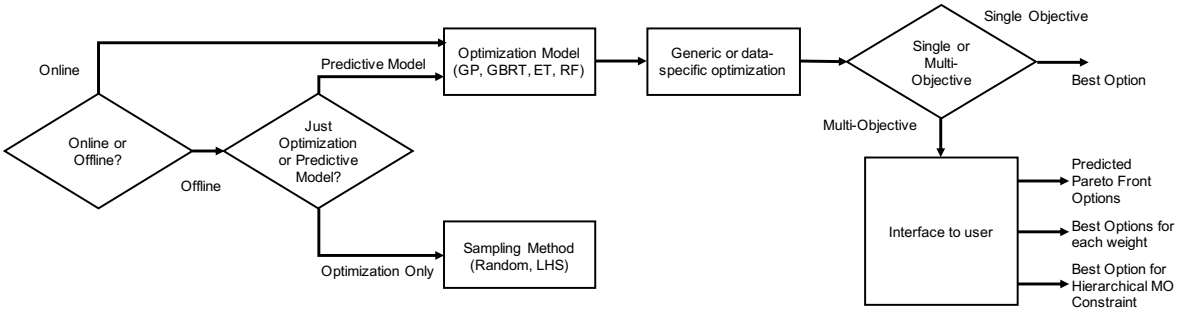


Figure 5.16: Systematic set of design choices of a system providing automatic resource allocation for serverless functions.

5.7 Design space

Now we take a step back and attempt to convey the big picture, covering all the axes that may be at the disposal of a user or a cloud provider when deploying serverless functions. Figure 5.16 shows the design choices that should be considered when developing an automatic resource allocation system for serverless. First, the designer has to decide between providing offline or online optimization. If offline optimization is desired, then both search-based (sampling) and model-based optimization are valid choices. But with online optimization, we recommend a model-based approach. We found that Bayesian Optimization with Gaussian Processes performs better than other optimization algorithms that we tested, as it converges to the best configuration faster and has lower prediction errors than other BO variants.

After deciding on the optimization algorithm, a designer would need to decide whether to create a data-specific or a generic optimization model. A data-specific optimization process might provide better performance but with added complexity. Irrespectively of whether the model is data-specific or not, one has to decide whether to provide single objective or multi-objective optimization options. A single objective optimization would provide the user with the option of either finding the configuration with the lowest execution time or cost. For multi-objective optimization, in turn, we discussed three options: 1) Pareto front, 2) weighted multi-objective optimization, and 3) hierarchical multi-objective optimization. Hierarchical optimization may be the most intuitive for the end-user to reason about, since it allows the user to select a given trade-off the user is willing to accept.

5.8 Related work

Serverless computing. Several recent papers demonstrated the advantages of serverless computing by applying this paradigm to a series of applications, ranging from data analytics [118, 119], DAG processing [120], video transcoding [121], compilation [122], machine learning [123, 124, 125] and more [126, 127]. Within this general area, there are a few recent proposals that aim to optimize the cost of serverless functions [128, 129, 130, 131]. Sizeless [131] and [129] use a regression model to minimize execution cost of AWS lambda functions. [130] uses memory tracing information to collect memory utilization metrics to find the right memory allocation for functions. Costless [128] uses function

fusion, splitting the function between edge and cloud, and allocating memory resources for a sequence of functions to optimize cost. However, these works are limited to the resource allocation strategy that AWS Lambda or other existing serverless offerings expose at the moment. In contrast, we take a step back and analyze the broad space of possible fine-grained configurations, and rethink the interface of services like AWS Lambda.

Several works gain insights into the characteristics of public serverless offerings by creating experiments and benchmarks to test the behavior of these platforms [103, 107, 104, 132]. However, they do not analyze the space of possible configurations beyond the current offerings and their effects.

HarvestVMs [133] create flexible VMs that can grow and shrink based on the available unallocated resources in an underlying server, allowing cloud providers to utilize their resources more efficiently. Zhang et al. [134] show how to use HarvestVMs for serverless. Our work is complementary since it can inform scheduling functions on the variable resources of HarvestVMs, and can enable the use of different instance types, minimizing costs while providing predictable execution times.

Cloud configuration optimization. Unlike the cloud configuration optimization works discussed in Chapter 2 (§2.2), the study in this chapter is not aimed at finding the best optimization algorithm for automatic resource allocation for serverless functions; instead, we deal with design space questions to show the potential opportunities and how they can be utilized. We used the BO variants that were part of the discussion in some of these works, but this is orthogonal to our main contribution, since other black-box optimization methods in prior works can replace the BO variants. In particular, if an analytical performance model of a serverless function can be created, such a model can be used instead of black-box performance modeling techniques to potentially lower the prediction error and speed up convergence.

Resource allocation in data centers. Similarly, collaborative filtering and clustering used in prior works on resource allocation in data centers (as discussed in §2.2) could also replace the black-box optimization methods we have discussed, provided that data on the performance of a representative set of benchmarking applications is available.

Multi-objective optimization. Our work builds on the techniques from research area of multi-objective optimization. While we reuse a set of specific methods from this area, there are several other multi-objective optimization schemes [117, 135] that can be potentially used to provide a cost-performance trade-off for serverless users.

5.9 Discussion

What about tail latency? In this chapter, we have used median latency as a performance metric. Using tail latency as a performance metric should not change the algorithms or the techniques discussed; however, it could change the best configuration that is found. In addition, it might require more measurements and could potentially increase the time it takes to find a good configuration. More importantly, the main sources of tail latency are due to queuing and interference, which should be addressed by scheduling, more so than by our resource selection. The models in this work can inform scheduling

decisions, but this is beyond the scope of this work.

Inference and Resource contention In this work, our central concern is with resource allocation for the function. Interference and resource contention would influence this decision, but we see interference as an orthogonal problem that can be dealt with through interference-aware scheduling techniques such as CloudScope [136], rather than resource allocation decisions.

Limitations First, while the limited influence of input data variation over the best configuration is true for the serverless applications we have tested in this chapter, other applications might have a significant change in the choice of appropriate configuration. In that case, a data-dependent approach might be required. Second, we note that BO with GP will not be a good optimization algorithm if the underlying performance model is not smooth. Other optimization algorithms might perform better in that case. Finally, if an analytical model is known for a function, e.g., if a function is embarrassingly parallel with ideal scalability, then using a black-box optimization method is inefficient, and a simple mathematical model would be better. Therefore, while we use black-box methods in this work so that it is independent of the function's actual behavior, that might be inefficient in some cases.

5.10 Summary

In this chapter, we demonstrated the benefits of decoupling memory and CPU resource allocations and using different instance types for serverless functions. Using the performance data we have collected, we established a potential to improve execution cost and execution time by up to 50% and 40%, respectively, by decoupling CPU and memory resource allocations and utilizing different VM types for serverless functions.

One way to deal with the increased number of resource allocation choices is by using black-box optimization methods. Our results showed that sampling-based and model-based black-box methods could find configurations that are within 10-20% of the performance of the best configuration within 20 optimization trials. However, model-based methods are the obvious choice if predictions for untested configurations are needed. We found BO with GP to be the best model-based method in terms of reaching the best configuration within a limited number of optimization trials and providing a much lower prediction error than other BO variants we tested. Additionally, our evaluation showed that good configurations for one input sample are generally good for others: performing an input-specific optimization leads to less than 20% improvement in execution and execution cost.

We outlined three types of interfaces to the user and their underlying optimization methodology to allow the user to select different cost-performance trade-off points. Lastly, we showed that, even in the presence of prediction error, we could utilize our performance models to enable the use of different instance types while providing predictable performance. This is key to allowing a cloud provider to use the idle resources of non-optimal instance types while minimizing the performance variation for the end-user.

Chapter 6

Automatic Parameter Tuning of Streaming Processing Systems

As the final main contribution of this thesis, in this chapter, we will discuss a complementary problem to cloud configuration optimization. As discussed in Chapter 1, in addition to defining cloud configurations, users will also need to set the values of the system parameters for the distributed systems running their jobs. In this chapter, we present and compare three solutions for automatic parameter tuning of stream processing systems.

6.1 Introduction

The use of stream processing is rapidly increasing in the industry because real-time streaming analytics allows companies to react to changes in data quickly [137]. For example, an e-commerce company can use streaming analytics to create targeted ads and real-time promotional offers for its customers. A study showed that more than 90% of surveyed organizations plan to increase investment in stream processing [137]. Dozens of companies are already using Apache Storm [138], and several others are likely using other stream-processing frameworks.

Popular stream-processing systems such as Apache Storm [10], Heron [11], Apache Flink [9] and Spark Streaming [8] have dozens of available configuration parameters. The performance of big-data applications that utilize these frameworks for real-time processing crucially depends on parameter tuning. For example, Apache Storm has about 201 parameters that can be set by the user. Among these, there are 49 parameters that can be controlled on a per-application basis. Not all configurable parameters affect performance; however, several of these parameters should be tuned to achieve performance goals, such as meeting Service Level Objectives (SLOs) for (tail) latency and throughput in production deployments. Moreover, suitable parameter configurations vary depending upon the available resources, workloads and applications.

Currently, these parameters are manually tuned, possibly requiring several hours of investigation and testing by performance engineers [139, 140, 141, 142, 143]. In addition, performance engineers may

require detailed knowledge of the stream-processing system itself to find a configuration that meets the performance requirements of an application because a best-practices approach is not always sufficient—a case that we make later in our evaluation. This makes configuration optimization a daunting and time-consuming task; we believe that automating configuration optimization is a viable alternative to manual tuning.

The configuration parameters in big-data frameworks can be broadly classified into two categories: (i) resource allocation parameters and (ii) application- or system-specific parameters. Resource allocation has been the subject of prior work, especially in conjunction with MapReduce jobs [144, 24]. On the other hand, there have been fewer studies on determining the optimal application-specific parameters for big-data frameworks. Most existing works present solutions specific to MapReduce jobs [145, 46, 45]. Only a few recent studies have considered performance optimization for stream-processing systems like Storm [146, 42]. Although BO4CO [42] deals directly with parameter tuning, the focus in this work is on latency optimization and not on throughput. To the best of our knowledge, a broader consideration of complete performance optimization in stream-processing systems has yet not been attempted.

Stream-processing systems are unique and present different challenges than those related to tuning databases and batch-processing systems. First, stream-processing applications are long-running (potentially infinite) programs. Tuning such applications requires determining an experiment's duration such that it is long enough to provide accurate performance measurements of a configuration yet short enough to quickly converge. Second, there are generally two metrics of interest that are jointly optimized: throughput and latency (typically at a high latency distribution percentile). Tuning is thus a case of multi-objective optimization. Most previous works considered single metrics [42, 145, 46, 45, 35] rather than multiple metrics. Third, measuring latency with no overhead is challenging when throughput is high, such as at hundreds of thousands of processed data items per second. Without sacrificing accuracy to measure latency, we turn to a probabilistic data structure called T-Digest (§6.5), which is able to handle millions of latency numbers per second. Fourth, unlike MapReduce-based frameworks, stream-processing applications are generally multi-stage data processing pipelines. Thus, we need to configure the parameters and the level of parallelism of each stage, yet each stage is also dependent on its interconnected stages and only a suitable balance in the levels of parallelism leads to the best achievable performance.

In this chapter, we introduce a framework for automated off-line parameter tuning of stream-processing systems. We use iterative optimization algorithms on measurements from actual runs of streaming applications, from which the framework collects feedback. We focus on Apache Storm as a stream-processing system for our experiments. However, several of the concepts can be directly translated to other stream-processing systems because component-level parallelism is common across all stream-processing systems and the black-box approaches presented here are not bound to specific parameters or systems.

In summary, we make the following main contributions:

- A framework for automated parameter tuning of stream-processing systems along with an implementation of five optimization algorithms. Our framework is released as open-source code [147].

- A novel gray-box optimization algorithm based on a heuristics-based approach.
- A new sampling method for the hill-climbing algorithm that accounts for desirable initial configurations of stream-processing applications.
- An evaluation of the performance of different algorithms using three benchmark applications for Apache Storm.

6.2 Preliminaries

6.2.1 Stream-Processing Systems

Stream-processing systems are software systems that are designed to analyze and act on (potentially unbounded) streaming data, generally in real time or in near real time (considered as having latencies in the range of sub-second up to a few minutes). Stream-processing systems have the ability to perform streaming analytics, i.e., to continuously calculate mathematical or statistical analytics on the data stream. Modern stream-processing systems are constructed on distributed, scalable, and fault-tolerant architectures that handle high volumes of data in real time.

There are several open-source stream-processing systems currently available and the number of systems is growing steadily. Apache Storm [10], Heron [11], Apache Flink [9] and Spark Streaming [8] are a few examples of production-grade stream-processing systems. In this chapter, we use Apache Storm as a case study; however, our concepts and approach are not specific to Storm and can be generalized to other systems.

Apache Storm is a distributed, real-time stream-processing system written in Java. An application in Storm is called a topology. A Storm topology is a Directed Acyclic Graph (DAG) composed of nodes that perform computation and edges that represent the flow of data being passed between nodes. The nodes send data between one another in the form of tuples. A tuple is an ordered list of values. The data stream originates at the source(s) node of the DAG, which is called a "spout". A spout generates a stream of tuples that are consumed and processed by "bolt" components according to a user-defined logic that defines the processing performed on the tuples at the node.

Apache Storm provides an application programming interface (API) to access metrics on the cluster, topology and component levels. The metrics that are provided include: average latency, number of tuples processed and per-component capacity. Capacity provided by Storm's metrics API is the utilization of each component/executor. We use the number of tuples processed by the topology as well as the per-component capacity for our framework.

6.2.2 Problem Formulation

Consider a system for which we want to perform parameter tuning for a set of N parameters. The goal of the configuration optimization process is to find the configuration, X, i.e., the vector of assigned parameter values, $X = x_1, \ldots, x_N$, that provides the desired performance according to a specific

metric of interest (e.g., throughput or latency). The possible values of these parameters come from range $R_{x_i}\ \forall\ x\ \in\ X.\ \ DOM\ =\ \prod_{i=1}^{N} R_{x_i}$ represents the space of all possible configurations that the system can have. More concretely, let y denote the performance metric of interest as a function of the configuration, i.e., $y = G(X)$. Then, the configuration optimization problem can be defined as finding the parameter values, X^*, such that:

$$X^* = \underset{X \in DOM}{\arg\min}\, G(X) \tag{6.1}$$

In our case, since we consider throughput in addition to the minimization of latency, we define $G(X)$ as a discontinuous objective function consisting of two underlying black-box functions, $T(X)$ and $L(X)$, representing throughput and latency, respectively, such that:

$$G(X) = \begin{cases} L(X) & T(X) \geq t \\ \infty & T(X) < t \end{cases} \tag{6.2}$$

where t is the throughput objective. Thus, a configuration needs to attain a certain minimum throughput objective to be considered a candidate solution. In practice, $T(X)$ and $L(X)$ are often unknown or do not have a closed form.

A possible approach to solving this optimization problem would be to resort to analytical modeling to predict the throughput and latency of a given stream-processing application. Although performance models are useful because they provide insight into the system's behavior, can be solved quickly, and allow for large parameter space explorations, such models are unfortunately time- and labor-intensive to obtain, typically rely on simplifying assumptions that hinder accuracy, and need to be recalibrated and revalidated as the conditions change.

Instead, we seek to automate parameter tuning to achieve specific performance goals with minimal human effort. Thus, we run the actual system in a simpler and more accurate way to determine its performance.

6.2.3 Our Approach

We use actual experiments that run the real system in a profiling environment using a small number of machines to evaluate the throughput and latency functions and taking measurements of these metrics. Such an experimental approach has been shown to produce system configurations that are as good as those resulting from idealized, perfectly accurate models [148, 35, 49].

Figure 6.1 presents an overview of our optimization framework. The user provides the application, a sample workload, SLO/performance goals and the possible ranges of parameters. The initial configuration is generated by the optimization algorithm and is used to submit the application/topology to the Storm cluster. After a specific duration, the application is terminated and the metrics exposed via Storm API (i.e., throughput, utilization, processing latency) as well as custom metrics (tail latency in our case) are collected to evaluate the current configuration. Subsequently, a new configuration is generated according to the optimization algorithm and the same application is submitted with this new configuration.

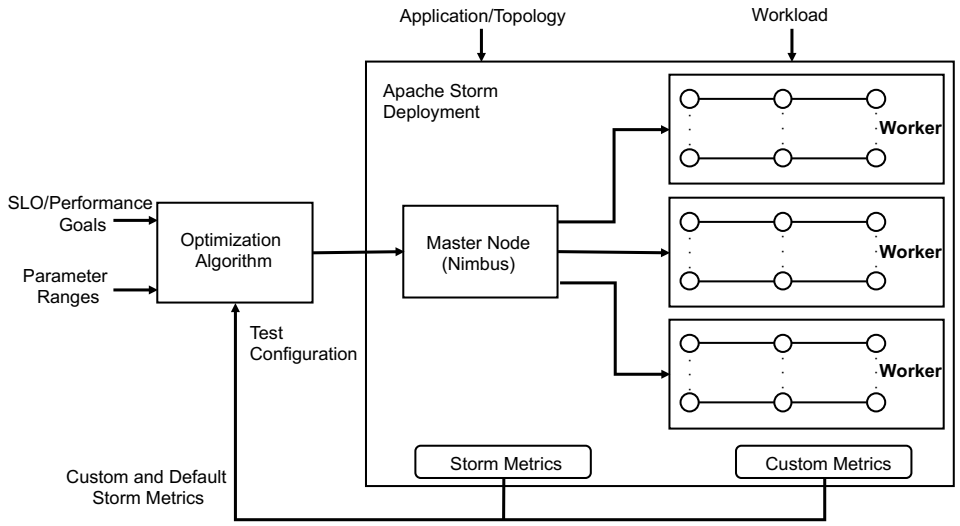


Figure 6.1: Configuration optimization framework.

This process continues until the optimization algorithm terminates the search based on some allocated resource budget (e.g., the amount of time we allow for configuration optimization) and provides the user with the best configuration according to the performance goals.

Our framework supports multiple optimization algorithms (§6.4). To demonstrate this generality, we later implement and evaluate several algorithms.

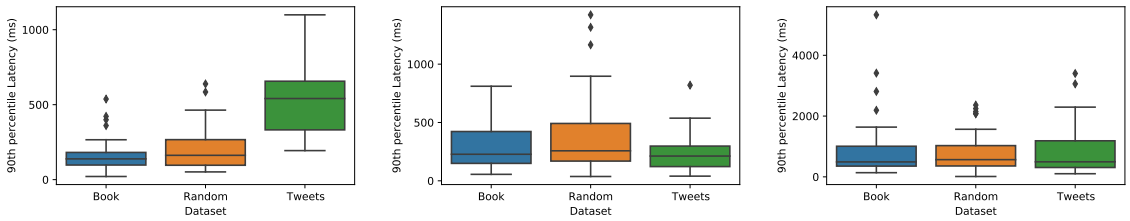
6.2.4 Background on Latin Hypercube Sampling

Exploring the complete space of all configurations is intractable when the number of parameters is large. We rely on an experimental design approach to obtain an initial sample from the configuration space using the Latin Hypercube Sampling (LHS) method. Here, we provide a short description of this method and refer the reader to McKay et al. [16] for a detailed introduction to LHS.

LHS is a stratified sampling formulation that is used to generate a near-random sample of parameter values from a multidimensional distribution. Since it is a stratified sampling approach, LHS can provide better coverage of the sample space than random sampling can. The LHS algorithm for generating K random configurations of dimension N (the number of parameters) can be briefly described as follows:

- Generate N random permutations, $P^1, \ldots, P^N$, of set $S = 1, \ldots, K$, where $P^i = (P^i_1, \ldots, P^i_K)$.
- For the i-th dimension with $i = 1, \ldots, N$, divide the parameter range, R_i, into K non-overlapping intervals of equal probabilities.
- The k-th sampled point is an N-dimensional vector, with the value for dimension i uniformly drawn from the P^i_k-th interval of R_i

Unfortunately, LHS by itself does not rule out bad spreads. For example, all samples can be spread along the diagonal. This problem can be addressed by generating multiple sets of LHS samples and choosing the one that maximizes the minimum distance between any pair of samples. That is, suppose



(a) Rolling Count Topology (b) Rolling Top Words Topology (c) Sentiment Topology

Figure 6.2: Variation of the 90^{th}-ile latency for different configurations and workloads in three benchmark topologies.

l different sets of LHS samples, $L1, \ldots, Ll$, were generated. Then, the approach selects the set L^* such that:

$$L^* = \arg\max_{1 \leq i \leq l} \min_{X^{(j)}, X^{(k)} \in L_i, j \neq k} \text{dist}(X^{(j)}, X^{(k)}), \tag{6.3}$$

where dist is a distance metric like Euclidean distance. This method is called Maximin Latin Hypercube Sampling [149].

However, we cannot directly use LHS for our case because different parameters have different numbers of values. For example, the number of workers might only four discrete values while the number of bolt executors might have 15 different values. If we were to use LHS, then we would not generate more than four samples because a specific value of a parameter cannot appear more than once in an LHS design. Thus, we use a modified version of LHS that we call mLHS. mLHS generates samples using LHS in a [0-1] N-dimensional design space. These samples are then mapped to the discrete bounded ranges in the N-dimensional space, which means that the end design is not a Latin Hypercube since an input parameter can have the same discrete value in two or more configurations, depending on the number of samples generated.

6.3 Impact of Parameter Tuning

Actual applications of stream processing on the industrial scale typically adhere to strict SLOs. The most common performance metrics used to specify SLOs for stream processing consist of (i) latency: how much time the system should take to process each input so that outputs reflect the latest inputs, and (ii) throughput: how much data the system should process within a time interval. An SLO specifies a desired level for throughput and/or latency that the stream-processing system must attain so that data will not be queued and eventually dropped or so that data will not cause cascading errors.

Performance tuning is therefore a crucial task that occurs when an application is deployed or when the application or workload changes. Changing the configuration of the parameters is one of the main ways to address performance tuning and to ensure that an SLO is met. Although improper tuning of parameters can have dire consequences on system performance, making sure that an application is running as efficiently as possible is a daunting task given the very large space of possible configura-

Parameter	Range	Step Size
Number of workers	3–12	3
Number of Acker executors	3–12	3
Number of Spout executors	3–12	3
Number of Bolt executors	3–45	3
Max Spout pending	2000–10000	2000

Table 6.1: Selected parameters and their value ranges.

tions. To understand the variation in performance achievable through parameter tuning, we conducted experiments on a multi-node Apache Storm cluster.

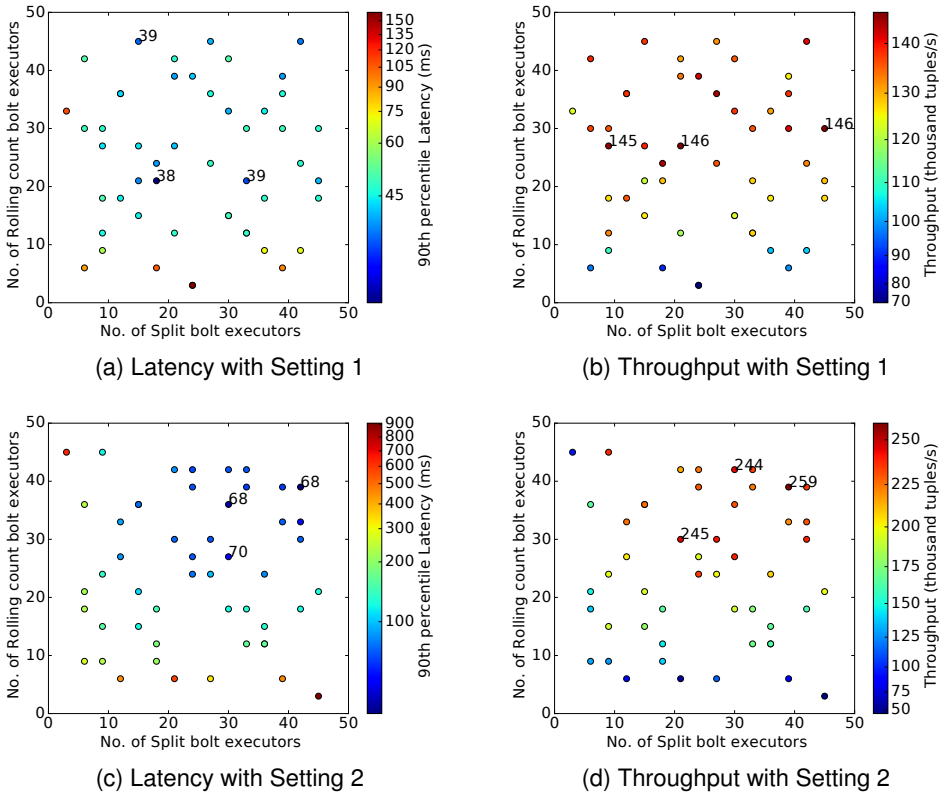
The setup included three worker nodes and a master node, each with a dual CPU with eight cores with enabled hyper-threading. This meant that a total of 32 threads could simultaneously run on each node. We configured Storm to use the Netty transport layer, as it achieves better performance than the default transport layer, ZeroMQ [150]. We left Acking, which is the process by which bolts inform Storm when they have finished processing an input tuple, enabled to access performance metrics from Storm UI for our measurements. We used three topologies as benchmarks [151, 152, 153]: (i) rolling word count (RC), (ii) rolling top words (RTW) and (iii) sentiment analysis (SA). Details about these topologies are presented in Sec. 6.5.1 below.

Table 6.1 shows the parameters that we selected to tune and the ranges and increment step sizes considered in our experiments, unless otherwise noted. We empirically chose this subset of parameters from all possible parameters of Apache Storm based on their importance according to documented parameter-tuning best practices [139, 140, 141, 142, 143]. The number of workers indicates the number of worker Java VMs that Storm generates for the topology. “Number of Spout executors” and “Acker executors” defines the number of threads that generates tuples for the topology to process and the number of threads responsible for handling tuple acknowledgments, respectively. Similarly, users can also specify the “Number of Bolt executors” for each bolt in the topology. Lastly, “Max Spout pending” specifies the maximum number of tuples (per spout) that can be simultaneously on-the-fly in the topology, i.e., the max number of tuples per spout that have not been acknowledged yet.

The step size used for all parameters except “Max Spout pending” is a direct result of the number of worker nodes that we use. We selected the step size to be equal to the number of nodes to avoid any kind of imbalance in the DAG placement, which is chosen by the default Storm scheduler. This strategy might be unnecessary if a different scheduling algorithm is used.

Note that the “Number of Bolt executors” parameter is a per-bolt parameter. Thus, the number of parameters to be tuned depends on the topology size, which is between 2 to 5 bolts in our topologies. The total number of parameters that we use for tuning purposes therefore ranges from 6 to 9 parameters.

Parameters significantly affect latency. We used mLHS to generate 30 different configurations for each topology. The purpose of this sample set was to illustrate the variation in performance metrics that can occur due to different configurations. For each configuration, the topologies were executed for 200 s while samples from the 90^{th} percentile of latencies and throughput were recorded for the last 100 s of the run to exclude noisy measurements, which are typical in the initial phase during execution. We execute each configuration with three different data streams based on a text corpus from a book, a



(a) Latency with Setting 1

(b) Throughput with Setting 1

(c) Latency with Setting 2

(d) Throughput with Setting 2

Figure 6.3: Latency and throughput for different values of executors for Rolling Count and Split bolt for the RC topology.

random process and a dataset of real Twitter tweets.

Figure 6.2 shows a boxplot of the variation in the 90^{th} percentile of latency for the 30 configurations tested for different workloads. We observe that latency depends on all three factors, configuration, workload and application. Importantly, the configuration of parameters has a significant impact on latency, influencing the measured 90^{th}percentile of latency by even two orders of magnitude in some cases (e.g., the sentiment topology with the book dataset).

Parameters have complex second-order (or higher) effects. The relationship between different configuration parameters can be complex and can include second-order or higher effects. Thus, simply increasing the value of one parameter does not correspond to a monotonic increase or decrease in the performance metric. To illustrate the second-order effects, we generate 50 configurations of the number of executors using mLHS.

The results for the RC topology are presented in Figure 6.3. There are two separate experiments shown there. Each experiment varies the number of executors for split bolt (x-axis) and rolling count bolt (y-axis) while keeping the remaining parameters fixed. Figure 6.3a and 6.3b show the different values of latency and throughput obtained when varying the number of executors while keeping the number of spouts, workers and ackers set at 12 (Setting 1). Figure 6.3c and 6.3d show similar results with the fixed parameters set at 6 instead (Setting 2). Each figure also highlights the top three candidate configurations according to its respective metric. While using more executors for the bolts can be a good strategy when the throughput objective is high, it does not necessarily yield the best configuration. These figures show

that often comparable performance is achievable by multiple different configurations in the configuration space. For example, in Figure 6.3a, the best configurations in terms of the 90^{th} percentile of latency are with $x = 42$ and $y = 39$ as well as $x = 33$ and $y = 36$. In Figure 6.3c, the best 90^{th}-ile latency achieved is when $x = 18$, $y = 21$; or $x = 15$, $y = 45$; or $x = 33$, $y = 21$. In the case of Setting 2, since the throughput is limited by the number of spout executors and the number of workers, increasing the number of executors to either bolt beyond 18 does not significantly impact the 90^{th} percentile in latency. Depending on the throughput objective, the suitable configuration for minimizing latency varies. The value selected for one parameter dictates the value that needs to be selected for another parameter to achieve the desired performance.

Summary. Based on a sample of 30 configurations, we observed 8 to 25 times variation in 90th percentile of latency (excluding the outliers). The same configurations provide different performance metrics for different workloads. In addition, we found that there is an inter-dependence between parameters. The desired throughput objectives and values set for other parameters dictates the best achievable latency and the values that need to be set in terms of the number of executors to achieve that latency.

6.4 Optimization Algorithms

In this section, we describe the algorithms that we have used to automatically tune the parameters for Apache Storm. These algorithms constitute the "Optimization Algorithm" block of our framework shown in Figure 6.1. We implemented the optimization algorithms listed in Table 6.2. Two of these five algorithms are introduced in this work. The remaining algorithms were previously documented in the literature to achieve good results, though they were not previously considered for parameter tuning in the stream-processing context.

Algorithm	Abbr.	Introduced in	Target setting
Hill-climbing algorithm	HC	[49]	Application server
Modified hill-climbing algorithm	mHC	This work	Stream processing
Heuristics-based approach	HB	This work	Stream processing
Genetic algorithm	GA	[44]	Batch processing
Random recursive search algorithm	RRS	[50]	Network protocols

Table 6.2: Algorithms used in our framework for parameter tuning.

The random recursive search and genetic algorithms did not achieve as good results as the other three approaches in most of our experiments that adhered to our optimization duration budget (which we set at 50 experiments or equivalently 250 minutes). We therefore do not discuss them in subsequent sections.

6.4.1 Hill-Climbing Algorithm (HC)

Our implementation is based on a hill-climbing algorithm previously introduced in the literature [49]. However, unlike the previous approach, our version of the algorithm does not use weighted LHS since this assumes that a priori information is available regarding the correlation between parameters, whereas

this might not be the case for us in general. The rest of the algorithm is implemented as it is described by Xi et al. [49].

The hill-climbing algorithm used in this work consists of the following phases:

• Initial Sampling: In this phase, n initial samples are generated using mLHS from the design space and the application is executed using these configurations.

• Local Search: In this phase, the algorithm uses mLHS sampling to generate m samples from the neighborhood of the best configuration found in the Initial sampling phase. It also checks whether a configuration in these m samples is better than the best configuration previously found.

• Potential Best Configurations: After the local search, the algorithm uses a polynomial fitting on the parameter values to create a configuration that might provide better performance. If a better configuration is found, it goes back to the local search phase. If not, it proceeds with including this new result and performing polynomial fitting again. If a better configuration is still not found, it proceeds to the shrink phase; otherwise, it goes back to local search.

• Shrink: The shrink phase decreases the size of the neighborhood and restarts the local search phase. If the neighborhood size is below a threshold, then the algorithm enters a restart phase.

• Restart: The restart phase generates l samples from the design space using mLHS and checks if the best configuration found in these l samples is better than the k-th percent of the configurations tested until now; if so, then it enters the local search phase again. Otherwise, it initiates the restart phase again.

The hill-climbing algorithm terminates after a fixed optimization budget is reached. Note that we avoid repeating experiments of configurations already explored during the local search phase. Thus, unique samples from the neighborhood are tested every time. If no new samples are generated by mLHS, then we enter the restart phase.

In all our tests, we set $n = 12$, $m = 5$ and $l = 5$ to achieve a balance between local search and global search. The neighborhood threshold value, t, is set to 0.4 times the size of the design space and k is set to 80%. The values for these settings are chosen according to the recommendations provided in [50] and through empirical testing to obtain the best performance in our settings; however, we did not perform an exhaustive analysis and these settings might need to be reconfigured according to need.

6.4.2 Modified Hill-Climbing Algorithm (mHC)

The difference between the hill-climbing algorithm (HC) and the modified hill-climbing algorithm (mHC) is the way in which the initial sample set is generated. In HC, we use mLHS to generate the initial sample set. In the case of mHC, we resort to a custom heuristic. In particular, the values of the number of workers and “Max Spout pending” are generated through mLHS, while the numbers of spout and

acker executors are set equal to the number of workers. The number of executors for different bolts are generated randomly using a Dirichlet distribution to distribute the remaining number of cores among different bolts, according to Equation 6.4. The Dirichlet distribution is a family of continuous multivariate probability distributions parameterized by a vector, α, of positive real numbers. It can be used to generate random numbers such that their sum is equal to 1.

$$t_{bolts} = t_{cores} - t_{spout} - t_{ackers} \tag{6.4}$$

where t_{cores} is the total number of cores in the cluster, t_{spout} and t_{ackers} is the number of spout and acker executors, respectively, and t_{bolts} is the number of executors that will be divided equally among the bolts in a topology.

6.4.3 The Heuristics-based Approach (HB)

The heuristics-based approach (HB) is a gray-box heuristic approach. It takes hints from the user who provides a ranking of parameters according to a priority level and specifies for each parameter whether an increase in parameter value has an overall positive or negative impact on latency and throughput. This is a greedy approach. It favors finding a suitable configuration quickly at the expense of optimality. Also, it attempts to minimize the number of resources (executors) that are required to achieve the required throughput objective.

Algorithm 1 shows the heuristics-based approach. The algorithm has three distinct phases: (i) throughput objective satisfaction phase, (ii) latency minimization phase, and (iii) executor adjustment phase.

(i) The throughput objective satisfaction phase changes the value of the parameters to achieve the throughput objective specified by the user. To make this phase aggressive, we update all the parameters at the same time according to their expected effect on the throughput (i.e., if increasing the parameter values increases the throughput, then the parameter values are increased or vice versa).

(ii) Once the throughput objective is satisfied, HB enters the latency minimization phase during which it does one-at-a-time tuning of parameters to minimize the latency. This phase iteratively goes through each parameter according to its user-provided ranking and tunes the value of the parameter to achieve a better performance than previous performances. If change in a parameter's value does not improve performance, the change is reverted and the algorithm moves to the next parameter. Once this iterative phase has taken all the parameters into account, it selects the best configuration it has found and enters the executor adjustment phase.

(iii) The executor adjustment phase involves two sub-phases. The first sub-phase reduces the number of executors assigned to each of the bolts as long as their capacity reported by Storm metrics is less than 80%. The second sub-phase reassigns the executors from other bolts to the bolt that is running at the highest capacity. Both of the phases are executed repeatedly for as long as the throughput objective is met. This last phase might also lead us to a configuration that can satisfy the throughput objective but with higher latency, while using a smaller number of executors.

```
Data: Ranked parameters, Parameter ranges, application and tputObjective
Result: Best configuration
Divide executors equally among all bolts;
Set values for all other configurations to their minimum;
while throughput objective unmet do
    change values of all parameters simultaneously to increase throughput;
    if throughput objective met then
        bestConfig = currentConfig;
        bestLatency = currentLatency;
        break;
    else
        continue;
    end
end
foreach parameter by ranked priority do
    while true do
        change value of the parameter by one step;
        get throughput and latency ;
        if throughput ≥ tputObjective and latency <bestLatency then
            bestConfig = currentConfig;
            bestLatency = latency;
        else if throughput < tputObjective or latency ≥ bestLatency then
            break ;
    end
end
Decrease the number of executors per bolt depending on the capacity;
if latency ≤ BestLatency then
    bestLatency = latency;
    bestConfig = currentConfig;
while throughput ≥ tputObjective do
    Reallocate executors from bolts with low capacity to bolt with max capacity;
    if latency ≤ BestLatency then
        bestLatency = latency;
        bestConfig = currentConfig;
end
```

Algorithm 1: Heuristics-based approach.

The basic parameter ranking that we use for all topologies ranks bolt executors at the top (based on their utilization; a higher utilization gets higher priority) followed by executors for spouts, ackers, max spout pending and number of workers.

Discussion. We note that all algorithms described in this section are not specific to Apache Storm and could be used with other stream-processing systems. For instance, both mHC and HB algorithms mainly provide heuristics for selecting the parallelism levels for the number of executors for bolt stages. These heuristics are directly applicable to systems like Apache Flink and Twitter Heron, which also have parallelism-level configuration parameters.

6.5 Evaluation

We now demonstrate that automated parameter tuning is both practical and beneficial for stream-processing systems by conducting an evaluation of the algorithms on a range of topologies and workloads to answer the following questions:

- How well do the considered optimization algorithms perform?
- What impact does the ranking of parameters have on the performance of the heuristics-based approach?

- Is it possible to scale via extrapolation a small cluster's best configuration to a larger cluster?
- Given a resource budget, are the benefits of using automated parameter tuning greater than employing such resources to scale the cluster?

Results Highlights. Based on our evaluation described below, we observe that:

- Our results demonstrate the feasibility and benefits of automated parameter tuning across multiple benchmark Storm applications and workloads, suggesting that our framework is a viable solution to the problem.
- Overall, mHC and HB perform comparably while outperforming HC in most cases. However, HC performs better when we are dealing with low throughput objective cases because of the uniformly sampled design space.
- HB is two to five times faster than the hill-climbing algorithms in terms of convergence time, thanks to the initial sample and throughput objective satisfaction phases.

6.5.1 Applications

We selected three topologies from publicly available Storm benchmarks [151, 152, 153]. The topologies are: Rolling count (RC), Rolling top words (RTW) and Sentiment analysis (SA). These topologies consist of three, five and six stages, respectively. These topologies can be found at [147] In our experiments, we use HDFS as the source of input data to the topologies. The data is streamed from HDFS to bolts using Storm spouts. To provide the impression of infinite data stream, since the data in HDFS is limited, the spouts simply restart from the beginning of the data file once they reach the end of the file.

6.5.2 Experimental Setup

Our main setup consists of a four-node testbed, with each node having 16 physical cores and 32 virtual cores. Three of the nodes are used as workers for Apache Storm and one node is used as the master.

Latency measurements are done on a per-tuple basis. Latency is measured based on the time it takes for the tuple to progress from a spout to the last bolt. Spouts tag each tuple with a timestamp and the total latency is calculated by subtracting the time at the last bolt from the timestamp tagged by the spout. To make sure that the timestamps are consistent across multiple nodes, we use Network Time Protocol (NTP) to synchronize the time.

The Storm Metrics API provides only average latency numbers, which are not sufficient for our case. Measuring per-tuple latencies is essential to measure tail latencies. Since throughput levels can reach hundreds of thousands of tuples per second, we cannot use a conventional method to measure tail latencies based on per-tuple latencies. To handle this challenge, we use a probabilistic data structure called T-Digest [154]. We have implemented a client-server application based on T-Digest that is able to handle several millions of tuple latencies/s [155].

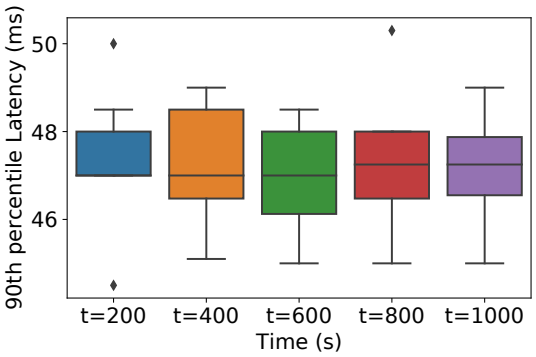


Figure 6.4: Latency results for different duration of experiments

Since we have a stream processing system with long-running applications, we needed to limit the duration of a single experiment. We tested different time durations for our experiments and selected 200 s as the duration of a single run. This time limit provides a good trade-off between the stability of the results and the time it takes for a single experiment to run. The results are stable because the tail latency does not change significantly if the experiments are executed for longer durations, as shown in Figure 6.4.

The latency measurements were collected for the specified time interval, in each experiment, for a total of 10 runs. We observed that the median 90^{th} and 99^{th} percentiles of latency across all runs differed by less than 2%. Therefore, we executed each experiment for 200 s. To allow for the setup phase to finish, we measured the latency only after the first 100 s. All subsequent latency readings were of the last 100 s of each experiment.

6.5.3 Performance of Parameter Tuning

We evaluated each optimization algorithm based on two criteria: (i) *Convergence*: how quickly the algorithm is able to search through the configuration space and find the best configuration, and (ii) *Best latency achieved*: what performance is attained by the best configuration that the algorithm finds within a certain amount of time.

In our evaluation, we focused on the minimization of latency (particularly at its tail) while sustaining a certain average throughput objective as the primary goal of the parameter tuning. This is a typical scenario envisioned for parameter tuning.

We present results obtained after five repetitions of each algorithm for each scenario. The convergence graphs (Figures 6.5a, 6.5c, 6.6a, 6.7a) show the progress of the algorithms in terms of the best latency found at each point during execution of the experiments. The y-axis reports the median best latency found at each point across five repetitions of the algorithm. Note that the convergence graph for HB shows the runtime with the longest execution.

The best latency graphs (Figures 6.5b, 6.5d, 6.6b, 6.7b) show a box-plot of the best latencies found by each algorithm across multiple repetitions. Overall, mHC performed the best in our experiments although HB outperformed it in some cases.

Figure 6.5b shows the best 90^{th} percentile of latency achieved by the three algorithms on the RC

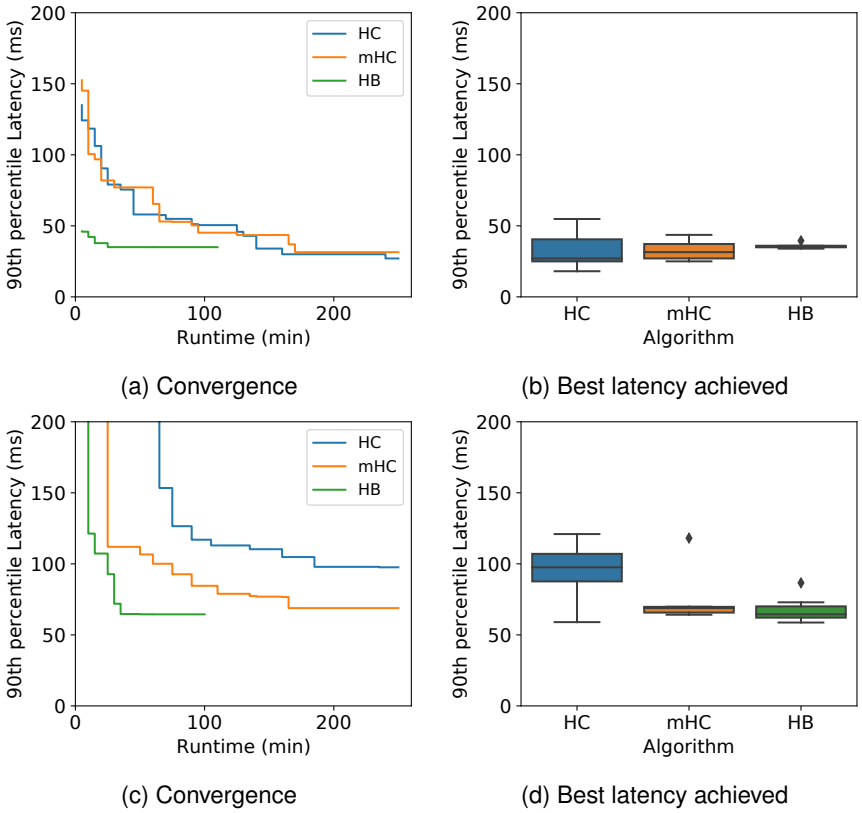
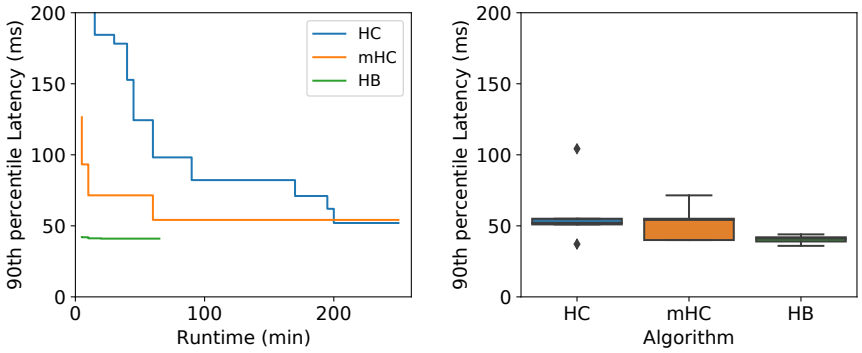


Figure 6.5: Performance of the three algorithms on the RC topology with book workload and throughput objective of 150k (a,b) and 300k (c,d) tuple/s.

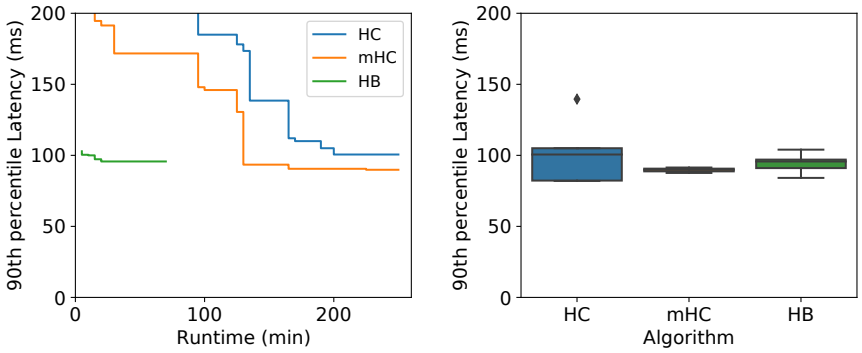
topology with a throughput objective of 150k tuples/s. We observe that HB has the least variance across multiple runs but the best configuration it finds can have higher latency as compared to the other two algorithms. This is because at a relatively low throughput objective, at 150k tuples/s, the configurations that provide the best latency are the ones with a low number of executors. Since HB starts from a configuration that utilizes all resources to begin with, it never arrives at these configurations. The mHC algorithm can suffer from a similar issue. While HB is able to find a good configuration within 10 iterations, it might miss some of the configurations that could improve the performance even further. On the other hand, HC generates a more uniformly distributed initial sample set, which leads it to configurations that provide on average 30% and 20% better latency as compared to HB and mHC, respectively.

Figure 6.5d illustrates the best 90^{th} percentile of latency when we increase the throughput objective to 300k tuples/s. The importance of the initial sampling technique for hill-climbing algorithms becomes clear at this throughput objective. HC uses LHS to sample the configuration space in a stratified manner, which means that it includes plenty of samples from the configuration space that are not suitable in high-throughput objective scenarios. On the other hand, mHC uses a sampling strategy that includes only configurations that have the total number of executors equal to the total number of cores, which enables it to find configurations that are easily able to achieve the throughput objective. In this case, on average only 2 out of 12 initial samples for HC were able to achieve 300k tuples/s throughput as compared to 9 out 12 initial samples for mHC. In terms of best latency achieved, HB and mHC perform comparably



(a) Convergence (b) Best latency achieved

Figure 6.6: Performance of the three algorithms on RTW topology with book workload and 150k throughput/s objective.



(a) Convergence (b) Best latency achieved

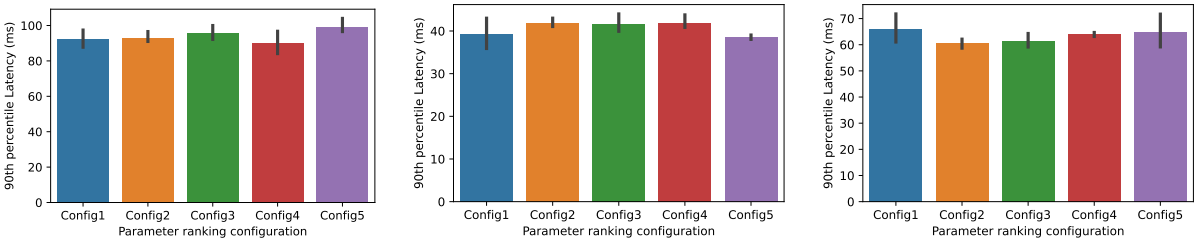
Figure 6.7: Performance of the three algorithms on SA topology with book workload and 100k throughput/s objective.

while HC provides 40% higher latency, on average.

The results for the RTW topology with the 150k tuples/s throughput objective are shown in Figure 6.6. HC and mHC perform comparably on average, while HB outperforms both. This is the case primarily because the initial configuration used by HB is very close to the configuration that provides the best latency. Thus, after a few iterations, HB can reach a configuration with the 90^{th} percentile of latency that is 25% lower compared with results from both HC and mHC.

For the sentiment analysis topology, several bolts have similar per-tuple execution latencies, which means that having a configuration where the number of executors is equally divided among multiple bolts is actually one of the best configurations. The Logging bolt is the bottleneck bolt in this case; however, within our cluster setup, we could not increase the parallelism level to that bolt enough to improve the performance further. The latency provided by mHC, on average, is only 5% and 10% lower than that provided by HB and HC, respectively.

While not shown in the figures, best practices suggested in various blog posts [139, 140, 141, 142, 143] and one-shot configurations were not able to satisfy the throughput objectives that we used in our tests. Their results are therefore not comparable. The best-practice approach that we adopted suggested that there should be one worker per node and the number of executors should be equal to



(a) SA topology (100k tuple/s) (b) RTW topology (150k tuple/s) (c) RC topology (300k tuple/s)

Figure 6.8: Performance variation due to different parameter rankings for mentioned throughput limit.

the number of cores. We set the “Max Spout pending” configuration to 2000, with one spout and acker thread per worker. These approaches were able to achieve 70k tuples/s throughput for the sentiment analysis topology and 100k tuples/s for the RC and RTW topologies. Following the best practices did not therefore provide us with even the throughput objective that we wished to satisfy . The main reason for this behavior is the fact that our setup required more than one worker per node to achieve higher throughput.

6.5.4 Impact of Parameter Ranking on HB

We now turn to an evaluation of the impact of parameter ranking on the performance of HB. Since HB does not backtrack on the configurations in most of its steps, the role of parameter ranking might be important to the achievable performance. To evaluate this aspect, we took four random parameter rankings for three test cases along with the ranking that we used for our previous experiments. Note that our previous ranking was based on our intuition and it might not always be the best ranking. Thus, we compare five parameter rankings to observe their influence on the best achievable performance. We execute HB three times using each ranking.

The results are shown in Figure 6.8. Surprisingly, we observe that parameter ranking does not significantly impact the best latency achieved. Even with different rankings, HB is able to achieve comparable performance. This observation can be explained based on the fact that since there are multiple configurations that are able to perform comparably, HB is able to arrive at one of those configurations from different starting points. More importantly, we observe that the throughput objective satisfaction phase in our test cases gets us very close to one of the best configurations in terms of latency. Because the changes in parameter ranking only affect the latency minimization phase, the affect of parameter ranking on best latency achieved is limited. While this may be the case for our tests, we believe that there will be other scenarios in which parameter ranking might be important and an approach to determine the ranking might be necessary, especially since HB does not backtrack to previous parameters.

6.5.5 Comparison with Other Approaches

To the best of our knowledge, the state of the art in parameter optimization for stream-processing systems is the work by Jamshidi et al. [42]. Their work follows a similar approach to iTuned [35] and uses

Gaussian processes to do Bayesian optimization. The methodology and techniques presented in their paper are *suitable when latency is the only metric under consideration*. We argue that only optimizing for latency can limit the applicability of such a technique because often a throughput requirement is present as well. We ran their system on our RC topology and observed that because the system only optimizes for latency, it misses configurations for which latency values with up to 30% higher throughput were achievable. In addition, such a method does not allow for any trade-off between throughput or latency. For example, there were cases in which up to 45% better throughput was achievable while sacrificing 1 ms of latency (increasing the 90th percentile of latency from 5 ms to 6 ms). Their techniques are clearly unable to navigate the trade-off between latency and throughput.

Bayesian optimization, as it has been discussed in [42, 35], is a single objective optimization technique. We have transformed our multi-objective optimization problem into a single objective one, using Equation 6.2. The function produced as a result has discontinuities, violating the smoothness assumptions of the generic covariance functions used in Gaussian processes [156]. Hence, Bayesian optimization with Gaussian processes used in these prior works is not applicable here. We leave as future work a study of multi-objective Bayesian optimization or Bayesian optimization with inequality constraints.

6.5.6 Extrapolation to Large Clusters

We now explore the following question: is it possible to use the results obtained on a small cluster of machines to tune the parameters of larger production clusters? This would be useful in several scenarios where it would be impractical or too costly to run the algorithms on the production cluster; however, parameter tuning over a few machines is possible if the best configuration found can inform the best configuration on the larger cluster. To evaluate the applicability of configurations found on smaller test clusters to larger production clusters, we use a simple extrapolation process.

We find that a combined throughput and latency objective is difficult to predict because the throughput does not scale linearly. However, we observe that in terms of best latency, the configurations found using a small cluster can be scaled to larger clusters while preserving the small cluster's latency optimization characteristic.

To confirm these observations, we ran the HB algorithm to find the configuration that provides the lowest 90^{th} percentile of latency in a four-node cluster (1 master and 3 worker nodes) and, afterwards, we ran the configuration with executors/number of workers scaled by five on a 16-node cluster (1 master and 15 workers). We compared the results of this scaled configuration with the best configuration provided by a run of all three algorithms on the 16-node cluster. We found that the scaled configuration provides 22 ms latency at the 90^{th} percentile while the best configurations provided by the runs of mHC, HC and HB algorithm were 27 ms, 33 ms and 21 ms, respectively. Hence, the scaled configuration is able to perform comparably to the best configurations found by running the optimization algorithm with the optimization budget of 50 runs. This intuitively makes sense because all the configurations except for "Max Spout pending" are scaled by five times. Thus, with five times greater load and five times more executors to handle that load, the configuration should perform just as well. The major condition here is

Throughput (k tuples/s)	W/o opt.		With opt.		Break-even point
	Nodes	Cost(in $)	Nodes	Cost(in $)	
200	4	0.75/h	3	0.6 + 0.6/h	6h
250	5	0.90/h	3	0.6 + 0.6/h	3h
300	6	1.05/h	3	0.6 + 0.6/h	2h

Table 6.3: Cost-benefit analysis contrasting the cost of running parameter tuning versus the cost of scaling out the stream processing cluster. The benefits of automated parameter tuning always occur within a few hours.

that other resources, such as the network, do not become the bottleneck.

6.5.7 Cost-Benefit Analysis

To understand whether automated parameter tuning is viable in practice, we performed a cost-benefit analysis. We considered a cloud environment where machines could be used for configuration optimization or could simply be added to the cluster to scale out the stream-processing systems configured according to documented best practices. We calculate the break-even point where the cost of running the cluster with configuration optimization is better than simply putting in more resources and using the best practices. For this experiment, we used three-worker nodes in AWS EC2 and a throughput objective of 200k, 250k and 300k tuples/s.

The total cost is the sum of the costs for the configuration optimization (fixed cost) and per hour based on the number of instances and their types. We used m4.4xlarge instances at a price of $0.15 per instance per hour. In our experiments, we observed that HB takes an hour to find a configuration that satisfies each of the throughput objectives. When using the best-practices approaches, we had to add more worker nodes to be able to achieve the desired throughput objective. Thus, the additional cost of parameter tuning is amortized over time.

Table 6.3 shows the break-even point, i.e., the duration of the application run after which the cost of using the automatic configuration is amortized. In the worst case, the break-even point is after six hours. If the application is supposed to be executed for periods longer than this point, then using automated parameter tuning provides cost savings.

6.6 Related Work

Parameter Tuning. There exists a large body of work on automatic parameter tuning, primarily in the fields of application servers, batch-processing systems and databases. We briefly discuss works closely related to ours.

iTuned [35] is a system for online parameter tuning of databases. It uses Bayesian optimization with Gaussian processes to find the best values for a set of parameters. One limitation with Gaussian processes is that they require a large amount of training data to be able to accurately model a complex multi-dimensional black-box function. iTuned's goal is to optimize for completion time, which makes it a single objective Bayesian optimization problem. On the other hand, our goals are to improve latency

while achieving a certain average throughput objective, making our problem a multi-objective Bayesian optimization. Moreover, as the workload changes, the model will need to be trained again. OtterTune [36] also uses Gaussian processes with workload characterization to tune DBMS configurations. Similarly, BO4CO [42] uses Gaussian processes for parameter tuning but in the context of stream-processing systems. As detailed in 6.5.5, their work focuses on a single performance metric and the techniques presented are not directly applicable to multi-objective optimization. Dhalion [40] proposes a policy-based self-regulation system for Twitter Heron. Users can define policies comprising a set of symptom, diagnosis and resolution actions. Symptoms provide information about observed anomalies in the behavior of the stream-processing system, which leads to the generation of a single diagnosis. As a result, a corresponding resolution action is carried out by the framework to bring the stream-processing system back to the normal state. Thus, policies are essentially a flexible form of rule-based automation. Their work is not limited to parameter tuning and can be used for parameter tuning. When the correct policies are written for each parameter and performance metric under consideration, their work can be used as an online tuning framework complementing the initial parameters selected using offline tuning by our approach.

For optimizing batch processing jobs, MROnline [43] uses a hill-climbing algorithm along with custom rules to tune certain MapReduce parameters to optimize the average execution time of MapReduce jobs. Gunther [44] is a search method based on genetic algorithms to optimize parameters for MapReduce. However, their primary goal is to decrease the execution time of the job. They focus on optimizing only for one metric. Xi et al. [49] also used a hill-climbing algorithm to find appropriate configurations for application servers. However, as we show in our evaluation, LHS is not necessarily the best sampling strategy for finding the best configuration in stream-processing systems. Tao et al. [50] and Heinze et al. [51] used recursive random search algorithm for parameter tuning in network protocols and an elastic scaling algorithm, respectively. However, in our experience, recursive random search is out-performed by the hill-climbing algorithm in the context of stream-processing systems.

Lin et el. [41] determined the degree of parallelism for each stage of the processing DAG based on input data and the computational cost. Thus, they were able to adjust the parallelism level to accommodate required throughput but they did not consider latency requirements. Zhu et al. [**?**] proposed online tuning of parameters for perception applications by learning a cost-based model using a parameter dependency graph. Cost-based models have also been used in parameter tuning for database systems as in [53]. However, cost-based models can have difficulty in accurately predicting performance across different types of hardware and application versions. For example, changes to the architecture of a database might render its cost-based model inapplicable. A more in-depth survey of parameter tuning approaches is provided in [157]

Several works have proposed sophisticated approaches for designing experiments to run when benchmarking servers [158] or optimizing their configuration parameters [159, 160], though they do not directly apply to the stream-processing context. In addition to search-based algorithms, there have been several efforts to use machine learning for the purpose of automatic parameter tuning [161, 162]. However, approaches using machine learning generally require a large amount of training data to be able

to build a classifier with good accuracy because they can only learn about scenarios and configurations that have been seen in the past.

Cloud configuration optimization works such as Ernest [23], Cherrypick [62], Scout [14], Micky [25], PARIS [29], Selecta [28] and Lynceus [26] among others are complementary to the proposal in this chapter, since they can be used to tune the cloud configurations, which is an orthogonal problem since because our approach tunes the parameters assuming that the deployment size is known.

Online Adaptation in Stream Processing. Several works proposed adaptive scaling and load balancing of stream-processing systems [163, 164, 165, 166, 167]. Lohrmann et al. [168] adaptively adjust buffer sizes and perform task chaining according to QoS constraints. Das et al. [169] use dynamic batch sizing for stream processing in Apache Spark to avoid queuing delays as the input data rate changes. Venkataraman et al. [170] dynamically adjust the number of batches that are grouped together for scheduling. We view these works as complementary to ours because we first need to find an efficient configuration to have an efficient scaling strategy.

6.7 Discussion

A limitation of the heuristics-based approach is that it moves in a unidirectional fashion to change the parameters and improve performance. It does not backtrack to change the value of a previous parameter once a change leads to desired performance improvement. This means that there can be configurations that might lead to improved performance but that are never explored by the heuristics-based approach. This is a trade-off between the speed of the optimization algorithm and its effectiveness.

According to our initial experiments, the throughput of our Storm topologies does not scale linearly with the cluster size. Thus, we are unable to predict the throughput of the topology as the cluster is scaled out. In addition, the optimization algorithms need to be run again in case of a significant workload change as well as a change in the cluster hardware.

The framework introduced in this chapter is for offline parameter tuning. Online parameter tuning requires the stream-processing system to be dynamically configurable without incurring long reconfiguration times. An online parameter-tuning system is also limited by the constraint that it should not use configurations that can possibly degrade the performance of the production system. Developing online parameter tuning to work alongside the offline framework is part of our future work.

Our framework assumes that the scheduling is static. Dynamic scheduling can change placements of different executors of Storm (e.g., [171, 172]), which can impact performance and might make the same configuration behave differently from one run to another.

Lastly, interesting challenges arise in a multi-tenant Storm cluster. In a shared environment, the parameters of a topology can be tuned automatically using the approaches discussed in this chapter as long as enough resources are available to avoid overloading and other topologies running with a fixed configuration. On the other hand, if multiple topologies need to be tuned simultaneously, we see two possible avenues, which we leave for future work. One possible approach would be to optimize the topologies separately, while measuring and taking into account the interference from other topologies.

A second approach would be to optimize the parameters of multiple topologies jointly so that each achieves its respective performance goal, in essence achieving Pareto optimality where each topology has its performance constraints met.

6.8 Summary

We have presented a framework to automate parameter tuning in stream-processing frameworks and we have applied it to Apache Storm as a case in point. The concepts and approaches presented in the chapter are not specific to Storm and could be extended to other stream-processing systems. We have shown that our modified hill-climbing algorithm and heuristics-based approach outperform the basic hill-climbing algorithm in three of the four test scenarios. The heuristics-based approach might provide up to 40% higher latency as compared to hill-climbing algorithms, in the worst case. However, HB can converge two to five times faster to that point as compared with the other two approaches and thus might be suitable for online parameter tuning.

Chapter 7

Conclusions

With the proliferation of cloud computing and the deployment of distributed data processing systems in the cloud, selecting the right cloud configuration is extremely important. The cloud configuration chosen for a deployment dictates the performance as well as the cost of the deployment. Manually figuring out the right cloud configuration is a tedious and time-consuming process requiring significant expertise and knowledge about cloud hardware and the deployed workload. This motivates the need for automatic mechanisms for configuration optimization for distributed systems deployed in cloud environments.

In this thesis, we thus present and evaluate several techniques for automatic configuration optimization. We deal mainly with two types of configurations: Cloud configurations and system parameters. More concretely, the summary and insights from each work discussed in this thesis are outlined in the next section.

7.1 Summary and Insights

The most important insights from each of the chapters in this thesis can be summarized as follows:

Chapter 3: In this chapter, we evaluated the performance of many popular black-box optimization algorithms in the context of choosing cloud configurations. We evaluated 8 algorithms and 23 workloads using 2 objective functions. Our evaluation showed that algorithm-specific hyper-parameter tuning is less important than the choice of the initial samples and the optimization budget. We found no clear set of optimal hyper-parameter values for any of the algorithms. While some existing works have used Bayesian optimization with Gaussian processes and Bayesian optimization with extra trees, our evaluation suggests that there is no silver bullet: the best optimization algorithm depending on the optimization scenario. In our experiments, Bayesian optimization with GBRT and GP perform better when optimizing for execution time and execution cost, respectively.

Chapter 4: In this chapter we presented Vanir, which performs automatic cloud configuration optimization by splitting the optimization task into benchmarking and production phases. This allows Vanir to handle large configuration search spaces without requiring long offline benchmarking or optimization. Vanir finds configurations comparable to benchmarking-based offline methods despite a $3\times$ shorter

benchmarking phase. Vanir has 1.3-24× lower optimization search cost, and it is 2× faster to reach within 15% of the execution time of the best configuration found. Thus, Vanir enables finding good configurations with fewer benchmarking runs, lower optimization search cost and with faster convergence to the best configuration.

Chapter 5: In this chapter, we demonstrated the benefits of decoupling memory and CPU resource allocations and using different instance types for serverless functions. Using the performance data we have collected, we established a potential to improve execution cost and execution time by up to 50% and 40% by decoupling CPU and memory resource allocations and utilizing different VM types for serverless functions. One way to deal with the increased number of resource allocation choices is by using black-box optimization methods. We found BO with GP to be the best model-based method for reaching the best configuration within a limited number of optimization trials and providing a much lower prediction error than other BO variants we tested. We outlined three types of interfaces to the user and their underlying optimization methodology to allow the user to select different cost-performance trade-off points. Lastly, we showed that, even in the presence of prediction error, we could utilize our performance models to enable the use of different instance types while providing predictable performance. This is key to allowing a cloud provider to use the idle resources of non-optimal instance types while minimizing the performance variation for end-user.

Chapter 6: Lastly, this chapter presented a framework to automate parameter tuning in stream-processing frameworks, using Apache Storm as an example. We have shown that our modified hill-climbing algorithm and heuristics-based approach outperform the basic hill-climbing algorithm in three of the four test scenarios. In the worst case, the heuristics-based approach might provide up to 40% higher latency than hill-climbing algorithms. However, HB can converge two to five times faster than the other two approaches and thus might be suitable for online parameter tuning.

7.2 Future directions

We wrap up this thesis by highlighting some interesting directions for future work that build up and improve on the work presented in this thesis and prior research work. The end goal is to eliminate the burden of resource allocation decisions from the end-user.

Reducing optimization cost Most of the current research on cloud configuration optimization focuses on achieving the performance and cost goals for the workload/job under consideration. But another aspect to consider is the time and cost of the optimization process. This becomes crucial when considering automatic resource configuration as a part of a service from the cloud provider. At cloud scale, a reduction in optimization time and cost can have a significant impact.

Research into ways to use a smaller subset of input data to test and predict performance for the full-scale workload is one possible way to reduce optimization time and costs. TrimTuner [173], which uses sub-sampling of the training dataset to perform hyperparameter and cloud configuration optimization for ML workloads, is an interesting research work in this direction. However, it is not as simple to sub-sample datasets used in analytics jobs because the data points are not treated independently as in

ML training. Reduction in optimization time and costs of the optimization process can also open other opportunities such as joint optimization of cloud configuration and system parameter values for data analytics workloads.

Joint optimization In this thesis, we have dealt with cloud configuration and hyperparameter optimization separately. However, ideally, we should jointly optimize them. The challenge with jointly tuning hyperparameters and cloud configuration is the drastic increase in the search space size. Generic black-box optimization techniques do not scale well with the search space size. As mentioned before, one way to handle this search space increase is to reduce time and cost for the optimization process. Another way to handle this challenge is by splitting the optimization process into offline and online phases as we did in Vanir [113]. Lastly, grey-box techniques that add hints and expert knowledge to the black-box approaches could also be an interesting venue to explore. Such methods could help intelligently prune the search space to reduce the number of promising configurations and thus make the problem more tractable.

Systems built for dynamic scalability Generally, a batch processing job is comprised of a DAG of connected individual processing tasks. The current generation of batch processing systems like Apache Spark and Apache Flink do not scale up or down automatically during different tasks of the job. Apache Flink supports a reactive mode [174] in which it can adjust to worker nodes joining and leaving the system, if an external service is scaling the cluster up or down. For batch processing jobs, the users have to determine and assign resources based on the task in the job that consumes the most resources. During the rest of the tasks, these resources might be underutilized. Since these systems have a well-defined programming API, it might be feasible for the system itself to determine the amount of resources needed for the next task in the job based on the task and the input data to the task. However, the challenge might be to capture the data-dependent performance characteristics of the task.

Representative benchmarking workloads Research works like PARIS [29], and Paragon [30] rely on offline benchmarking of several applications. The performance of these applications is often tested on all available configurations. New workloads are then classified in terms of similarity to one of the benchmark workloads to determine the best configuration for the new unknown workloads. The choice of these benchmark applications can be critical, and an interesting future work direction is to develop techniques to create the minimal set of benchmarking workloads from a large set of production workloads to satisfy desired coverage and prediction accuracy requirements.

An optimization framework for all occasions In this thesis, we have mainly used techniques and algorithms that do not assume the availability of historical performance data. These algorithms can start from scratch and find good configurations based on user-specific performance metrics and constraints. With Vanir, we use existing historical data if it is present, but further research is needed to create optimization frameworks that work well in the absence of performance data for a new deployment and then transition to make good use of historical data. This removes the need to collect extensive benchmarking data in the beginning. Still, the framework should utilize an increasing amount of performance data to improve the optimization efficiency as time goes on. Combining search-based methods, model-based methods, and a machine learning system could potentially achieve this goal by utilizing different

algorithms at different stages of the optimization process. It would be an interesting future direction to explore meta-learning algorithms such as one-shot or few-shot algorithms, for example, using Deep Kernel Surrogates [175] when sufficient historical performance data has been accumulated.

A true managed service Current managed data analytics service offerings by the cloud providers, e.g., Amazon EMR, still rely on users to decide the resource allocation, i.e., the type and number of instances. Thus, part of the operational burden is still on the end-users. A fully managed big data analytics service would eliminate this burden and accelerate cloud adoption even more. It would enable users to think in terms of performance requirements rather than resource requirements. Such a resource allocation framework would need to have several features, including online optimization, use of data subsets for performance prediction, and transfer learning/meta-learning. However, one of the significant challenges would be the interface of such a service for the end-user. End-users might want predictable, steady-state performance and costs across multiple runs of a job. The users might also require a general trend of decreasing costs and increasing performance. The cloud provider could also either hide the costs of optimization from the user or choose to be transparent about them. Each of these decisions and features would add a set of constraints over the desired behavior of the optimization algorithm.

Resource allocation and scheduling In Chapter 5 we make a case for decoupling serverless resource allocation and enabling the use of different instance types for serverless functions. An interesting future direction is to evaluate the impact of this design choice on the scheduling complexity for the cloud provider since there is a trade-off between the scheduling complexity and the number of possible resource configurations for serverless functions. There is a need to determine the granularity of decoupling beyond which inefficiencies of scheduling overshadow the benefits of decoupling. Additionally, scheduling decisions can impact resource allocation decisions, especially when it comes to tail-latency. For example, a scheduling decision that leads to interference between two co-located serverless functions might require a change in resource allocation for them to maintain a tail latency SLO. Thus, it can be important to combine these decisions to enable scheduling decisions to inform resource allocation decisions and vice versa.

Optimizing pipeline of Serverless functions Serverless functions are often chained in the form of pipelines with latency SLOs. Optimizing each serverless function individually, as we have done in Chapter 5 is not efficient when the goal is to optimize the whole pipeline. A potential approach to address this problem could be Multi-arm bandit [100]. Multi-arm bandit approaches can direct the optimization efforts on specific serverless functions to get the most benefit out of the time and cost spent on optimization. Similar techniques can also apply to a pipeline of batch jobs or microservices to achieve desired overall SLOs.

Bibliography

[1] Enterprise-analytics. 2020 Global State of Enterprise Analytics. https://www3.microstrategy.com/getmedia/db67a6c7-0bc5-41fa-82a9-bb14ec6868d6/2020-Global-State-of-Enterprise-Analytics.pdf.

[2] Gartner-forecast. Gartner Forecasts Worldwide Public Cloud End-User Spending. https://www.gartner.com/en/newsroom/press-releases/2020-11-17-gartner-forecasts-worldwide-public-cloud-end-user-spending-to\protect\discretionary{\char\hyphenchar\font}{}{}grow-18-percent-in-2021.

[3] S. Agarwal, S. Kandula, N. Bruno, M.-C. Wu, I. Stoica, and J. Zhou. Re-optimizing data-parallel computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pages 21–21. USENIX Association, 2012.

[4] A. D. Ferguson, P. Bodik, S. Kandula, E. Boutin, and R. Fonseca. Jockey: guaranteed job latency in data parallel clusters. In *Proceedings of the 7th ACM european conference on Computer Systems*, pages 99–112. ACM, 2012.

[5] L. Shao, Y. Zhu, S. Liu, A. Eswaran, K. Lieber, J. Mahajan, M. Thigpen, S. Darbha, S. Krishnan, S. Srinivasan, C. Curino, and K. Karanasos. Griffon: Reasoning about Job Anomalies with Unlabeled Data in Cloud-Based Platforms. In *SoCC*, pages 441—-452. ACM, 2019.

[6] previous-gen. EC2 Previous Generation Instances. https://aws.amazon.com/ec2/previous-generation/.

[7] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. In *Proceedings of the 6th conference on Symposium on Operating Systems Design & Implementation-Volume 6*, pages 10–10, 2004.

[8] Apache Spark. http://spark.apache.org/.

[9] Apache Flink. http://flink.apache.org/.

[10] Apache Storm. http://storm.apache.org/, .

[11] S. Kulkarni, N. Bhagat, M. Fu, V. Kedigehalli, C. Kellogg, S. Mittal, J. M. Patel, K. Ramasamy, and S. Taneja. Twitter Heron: Stream Processing at Scale. In *SIGMOD*, 2015.

[12] O. Alipourfard, H. H. Liu, J. Chen, S. Venkataraman, M. Yu, and M. Zhang. CherryPick: Adaptively Unearthing the Best Cloud Configurations for Big Data Analytics. In *NSDI*, 2017.

[13] C. Hsu, V. Nair, T. Menzies, and V. Freeh. Micky: A cheaper alternative for selecting cloud instances. In *CLOUD*, 2018.

[14] C.-J. Hsu, V. Nair, T. Menzies, and V. W. Freeh. Scout: An experienced guide to find the best cloud configuration. *arXiv 1803.01296*, 2018.

[15] M. Casimiro, D. Didona, P. Romano, L. Rodrigues, W. Zwaenepoel, and D. Garlan. Lynceus: Cost-efficient tuning and provisioning of data analytic jobs. In *ICDCS*, 2020.

[16] M. McKay, R. Beckman, and W. Conover. A Comparison of Three Methods for Selecting Values of Input Variables in the Analysis of Output from a Computer Code. *Technometrics*, 21(2), 1979.

[17] C. K. Williams and C. E. Rasmussen. Gaussian Processes for Regression. In *Advances in Neural Information Processing Systems (NIPS)*, 1996.

[18] T. Hastie, R. Tibshirani, and J. Friedman. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. Springer Science & Business Media, 2009.

[19] L. Breiman. Random Forests. *Machine learning*, 45(1), 2001.

[20] P. Geurts, D. Ernst, and L. Wehenkel. Extremely randomized trees. *Machine learning*, 63(1), 2006.

[21] M. Hoffman, E. Brochu, and N. de Freitas. Portfolio Allocation for Bayesian Optimization. In *Conference on Uncertainty in Artificial Intelligence (UAI)*, 2011.

[22] J. S. Bergstra, R. Bardenet, Y. Bengio, and B. Kégl. Algorithms for Hyper-Parameter Optimization. In *Advances in Neural Information Processing Systems (NIPS)*, 2011.

[23] S. Venkataraman, Z. Yang, M. J. Franklin, B. Recht, and I. Stoica. Ernest: Efficient Performance Prediction for Large-Scale Advanced Analytics. In *NSDI*, 2016.

[24] H. Herodotou, F. Dong, and S. Babu. No One (Cluster) Size Fits All: Automatic Cluster Sizing for Data-intensive Analytics. In *SoCC*, 2011.

[25] C.-J. Hsu, V. Nair, V. W. Freeh, and T. Menzies. Arrow: Low-Level Augmented Bayesian Optimization for Finding the Best Cloud VM. In *ICDCS*, 2018.

[26] M. Casimiro, D. Didona, P. Romano, L. Rodrigues, W. Zwanepoel, and D. Garlan. Lynceus: Cost-efficient tuning and provisioning of data analytic jobs. *arXiv preprint arXiv:1905.02119*, 2019.

[27] Y. Liu, H. Xu, and W. C. Lau. Accordia: Adaptive cloud configuration optimization for recurring data-intensive applications. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 479–479, 2019.

[28] A. Klimovic, H. Litz, and C. Kozyrakis. Selecta: Heterogeneous cloud storage configuration for data analytics. In *USENIX ATC*, pages 759–773, 2018.

[29] N. J. Yadwadkar, B. Hariharan, J. E. Gonzalez, B. Smith, and R. H. Katz. Selecting the Best VM Across Multiple Public Clouds: A Data-driven Performance Modeling Approach. In *SoCC*, 2017.

[30] C. Delimitrou and C. Kozyrakis. Paragon: QoS-aware Scheduling for Heterogeneous Datacenters. In *ASPLOS*, 2013.

[31] C. Delimitrou and C. Kozyrakis. Quasar: Resource-efficient and QoS-aware Cluster Management. In *ASPLOS*, 2014.

[32] S. A. Jyothi, C. Curino, I. Menache, S. M. Narayanamurthy, A. Tumanov, J. Yaniv, R. Mavlyutov, I. Goiri, S. Krishnan, J. Kulkarni, et al. Morpheus: Towards automated slos for enterprise clusters. In *OSDI*, 2016.

[33] K. Rajan, D. Kakadia, C. Curino, and S. Krishnan. PerfOrator: Eloquent Performance Models for Resource Optimization. In *SoCC*, 2016.

[34] N. Vasić, D. Novaković, S. Miučin, D. Kostić, and R. Bianchini. Dejavu: Accelerating Resource Allocation in Virtualized Environments. In *ASPLOS*, 2012.

[35] S. Duan, V. Thummala, and S. Babu. Tuning database configuration parameters with ituned. *Proceedings of the VLDB Endowment*, 2(1):1246–1257, 2009.

[36] D. Van Aken, A. Pavlo, G. J. Gordon, and B. Zhang. Automatic database management system tuning through large-scale machine learning. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 1009–1024. ACM, 2017.

[37] A. Mahgoub, P. Wood, S. Ganesh, S. Mitra, W. Gerlach, T. Harrison, F. Meyer, A. Grama, S. Bagchi, and S. Chaterji. Rafiki: a middleware for parameter tuning of NoSQL datastores for dynamic metagenomics workloads. In *Middleware*, pages 28–40, 2017.

[38] Z. Cao, V. Tarasov, S. Tiwari, and E. Zadok. Towards better understanding of black-box auto-tuning: A comparative analysis for storage systems. In *ATC'18*, pages 893–907, 2018.

[39] Z. L. Li, C.-J. M. Liang, W. He, L. Zhu, W. Dai, J. Jiang, and G. Sun. Metis: Robustly tuning tail latencies of cloud systems. In *ATC'18*, pages 981–992, 2018.

[40] A. Floratou, A. Agrawal, B. Graham, S. Rao, and K. Ramasamy. Dhalion: Self-Regulating Stream Processing in Heron. In *VLDB*, 2017.

[41] W. Lin, Z. Qian, J. Xu, S. Yang, J. Zhou, and L. Zhou. Streamscope: continuous reliable distributed processing of big data streams. In *Proc. of NSDI*, pages 439–454, 2016.

[42] P. Jamshidi and G. Casale. An uncertainty-aware approach to optimal configuration of stream processing systems. In *Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS), 2016 IEEE 24th International Symposium on*, pages 39–48. IEEE, 2016.

[43] M. Li, L. Zeng, S. Meng, J. Tan, L. Zhang, A. R. Butt, and N. Fuller. MRONLINE: MapReduce Online Performance Tuning. In *HPDC*, 2014.

[44] G. Liao, K. Datta, and T. L. Willke. Gunther: Search-Based Auto-Tuning of MapReduce. In *European Conference on Parallel Processing*, 2013.

[45] S. Babu. Towards automatic optimization of mapreduce programs. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 137–142. ACM, 2010.

[46] H. Herodotou and S. Babu. Profiling, what-if analysis, and cost-based optimization of mapreduce programs. *Proceedings of the VLDB Endowment*, 4(11):1111–1122, 2011.

[47] A. Fekry, L. Carata, T. Pasquier, A. Rice, and A. Hopper. Tuneful: An online significance-aware configuration tuner for big data analytics. *arXiv preprint arXiv:2001.08002*, 2020.

[48] H. Du, P. Han, W. Chen, Y. Wang, and C. Zhang. Otterman: A novel approach of spark auto-tuning by a hybrid strategy. In *ICSAI*, pages 478–483, 2018.

[49] B. Xi, Z. Liu, M. Raghavachari, C. H. Xia, and L. Zhang. A smart hill-climbing algorithm for application server configuration. In *WWW*, 2004.

[50] T. Ye and S. Kalyanaraman. A recursive random search algorithm for large-scale network parameter configuration. *ACM SIGMETRICS Performance Evaluation Review*, 31(1), 2003.

[51] T. Heinze, L. Roediger, A. Meister, Y. Ji, Z. Jerzak, and C. Fetzer. Online parameter optimization for elastic data stream processing. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*, pages 276–287. ACM, 2015.

[52] Q. Zhu, B. Kveton, L. Mummert, and P. Pillai. Automatic tuning of interactive perception applications. In *Proceedings of the Twenty-Sixth Conference on Uncertainty in Artificial Intelligence*, pages 743–751, 2010.

[53] E. Kwan, S. Lightstone, A. Storm, and L. Wu. Automatic configuration for IBM DB2 universal database. *IBM Perf. Technical Report*, 2002.

[54] F. Hutter, L. Kotthoff, and J. Vanschoren. *Automated Machine Learning*. Springer, 2019.

[55] S. Shan and G. G. Wang. Survey of modeling and optimization strategies to solve high-dimensional design problems with computationally-expensive black-box functions. *Structural and Multidisciplinary Optimization*, 41(2), 2010.

[56] M. A. Muñoz, Y. Sun, M. Kirley, and S. K. Halgamuge. Algorithm selection for black-box continuous optimization problems: A survey on methods and challenges. *Information Sciences*, 317, 2015.

[57] D. Golovin, B. Solnik, S. Moitra, G. Kochanski, J. Karro, and D. Sculley. Google Vizier: A Service for Black-Box Optimization. In *International Conference on Knowledge Discovery and Data Mining (SIGKDD)*, 2017.

[58] Y. Zhu, J. Liu, M. Guo, Y. Bao, W. Ma, Z. Liu, K. Song, and Y. Yang. BestConfig: Tapping the Performance Potential of Systems via Automatic Configuration Tuning. In *Symposium on Cloud Computing (SoCC)*, 2017.

[59] C. Wu, T. Summer, Z. Li, A. Woodard, R. Chard, M. Baughman, Y. Babuji, K. Chard, J. Pitt, and I. Foster. Paraopt: Automated application parameterization and optimization for the cloud. In *2019 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 255–262. IEEE, 2019.

[60] T. Chiba, R. Nakazawa, H. Horii, S. Suneja, and S. Seelam. Confadvisor: A performance-centric configuration tuning framework for containers on kubernetes. In *2019 IEEE International Conference on Cloud Engineering (IC2E)*, pages 168–178. IEEE, 2019.

[61] C. Li, S. Wang, H. Hoffmann, and S. Lu. Statically inferring performance properties of software configurations. In *EuroSys*, pages 1–16, 2020.

[62] O. Alipourfard, H. H. Liu, J. Chen, S. Venkataraman, M. Yu, and M. Zhang. Cherrypick: Adaptively unearthing the best cloud configurations for big data analytics. In *NSDI*, volume 2, pages 4–2, 2017.

[63] C.-J. Hsu, V. Nair, T. Menzies, and V. Freeh. Micky: A Cheaper Alternative for Selecting Cloud Instances. In *International Conference on Cloud Computing (CLOUD)*, 2018.

[64] M. Feurer and F. Hutter. Hyperparameter Optimization. In *Automated Machine Learning*. Springer, 2019.

[65] BBO Arena. `https://github.com/MBtech/bbo-arena`, . [Online; accessed 10/07/2020].

[66] N. Hansen. The CMA Evolution Strategy: A Comparing Review. In *Towards a New Evolutionary Computation*. Springer, 2006.

[67] C. Ansótegui, M. Sellmann, and K. Tierney. A Gender-Based Genetic Algorithm for the Automatic Configuration of Algorithms. In *International Conference on Principles and Practice of Constraint Programming (CP)*, 2009.

[68] J. Kennedy and R. Eberhart. Particle swarm optimization. In *International Conference on Neural Networks (ICNN)*, 1995.

[69] L. Li, K. Jamieson, G. DeSalvo, A. Rostamizadeh, and A. Talwalkar. Hyperband: A Novel Bandit-Based Approach to Hyperparameter Optimization. *The Journal of Machine Learning Research*, 18(1), 2017.

[70] A. C. Elliott and W. A. Woodward. *Statistical Analysis Quick Reference Guidebook: With SPSS Examples*. Sage, 2007.

[71] Scikit Optimize Website. `https://scikit-optimize.github.io/stable/`. Accessed on May 27th, 2021.

[72] J. Bergstra, D. Yamins, and D. Cox. Making a Science of Model Search: Hyperparameter Optimization in Hundreds of Dimensions for Vision Architectures. In *International Conference on Machine Learning (ICML)*, 2013.

[73] HyperOpt. `https://pythonhosted.org/pyDOE/`. [Online; accessed 10/07/2020].

[74] Solid. `https://github.com/100/Solid`. [Online; accessed 10/07/2020].

[75] Intel's HiBench benchmark. `https://github.com/intel-hadoop/HiBench`, .

[76] Scout Dataset. `https://github.com/oxhead/scout`. [Online; accessed 10/07/2020].

[77] H. Hoos, K. Leyton-Brown, and F. Hutter. An Efficient Approach for Assessing Hyperparameter Importance. In *International Conference on Machine Learning (ICML)*, 2014.

[78] BBO Arena: Best hyper-parameters. `https://github.com/MBtech/bbo-arena/blob/master/docs/best-hyperparams.md`, . [Online; accessed 10/07/2020].

[79] skopt-gp. GP Minimize Scikit Opt. `https://scikit-optimize.github.io/stable/modules/generated/skopt.gp_minimize.html`.

[80] J. Wang, S. C. Clark, E. Liu, and P. I. Frazier. Parallel Bayesian Global Optimization of Expensive Functions. *Operations Research*, 2020.

[81] J. Očenášek and J. Schwarz. The Parallel Bayesian Optimization Algorithm. In *The State of the Art in Computational Intelligence*. Springer, 2000.

[82] J. González, Z. Dai, P. Hennig, and N. Lawrence. Batch Bayesian Optimization via Local Penalization. In *Artificial Intelligence and Statistics (AISTATS)*, 2016.

[83] Flink Use cases. `https://flink.apache.org/usecases.html`. [Online; accessed 01/03/2020].

[84] amazonlambda. AWS Lambda. `https://aws.amazon.com/lambda/`. Accessed on May 27th, 2021.

[85] How Verizon Media Group migrated from on-premises Apache Hadoop and Spark to Amazon EMR. `https://aws.amazon.com/blogs/big-data/how-verizon-media-group-migrated-from-on-premises-apache-hadoop-and-spark-to-amazon-emr/`. [Online; accessed 25/05/2020].

[86] B. Lakshminarayanan, D. M. Roy, and Y. W. Teh. Mondrian forests: Efficient online random forests. In *Advances in neural information processing systems*, pages 3140–3148, 2014.

[87] D. M. Roy, Y. W. Teh, et al. The mondrian process. In *NIPS*, pages 1377–1384, 2008.

[88] A. Saffari, C. Leistner, J. Santner, M. Godec, and H. Bischof. On-line random forests. In *Computer Vision Workshops (ICCV Workshops), 2009 IEEE 12th International Conference on*, pages 1393–1400. IEEE, 2009.

[89] M. Denil, D. Matheson, and N. Freitas. Consistency of online random forests. In *International Conference on Machine Learning*, pages 1256–1264, 2013.

[90] B. M. Sarwar, G. Karypis, J. A. Konstan, J. Riedl, et al. Item-based collaborative filtering recommendation algorithms. In *WWW*, volume 1, pages 285–295, 2001.

[91] F. Cacheda, V. Carneiro, D. Fernández, and V. Formoso. Comparison of collaborative filtering algorithms: Limitations of current techniques and proposals for scalable, high-performance recommender systems. *ACM Transactions on the Web (TWEB)*, 5(1):2, 2011.

[92] Spearmint Github repo. `https://github.com/HIPS/Spearmint`.

[93] GraphX lib github. `https://github.com/apache/spark/tree/master/graphx/src/main/scala/org/apache/spark/graphx/lib`.

[94] Modified version of Intel's HiBench benchmark. `https://github.com/MBtech/HiBench`, .

[95] spark-docs. Tuning Spark. `https://spark.apache.org/docs/latest/tuning.html#level-of-parallelism`.

[96] Quoble Pipeline. `https://www.qubole.com/developers/spark-getting-started-guide/workflow/`. [Online; accessed 25/05/2020].

[97] Build a Concurrent Data Orchestration Pipeline Using Amazon EMR and Apache Livy. `https://aws.amazon.com/blogs/big-data/build-a-concurrent-data-orchestration-pipeline-using-amazon-emr-and-apache-livy/`. [Online; accessed 25/05/2020].

[98] Apache Airflow. `https://airflow.apache.org`. [Online; accessed 25/05/2020].

[99] Luigi. `https://github.com/spotify/luigi`. [Online; accessed 25/05/2020].

[100] H. Robbins. Some aspects of the sequential design of experiments. *Bulletin of the American Mathematical Society*, 58(5):527 – 535, 1952.

[101] googlecloudfunctions. Google Cloud Functions. `https://cloud.google.com/functions/`. Accessed on May 27th, 2021.

[102] azurefunctions. Azure Functions. `https://azure.microsoft.com/en-us/services/functions/`. Accessed on May 27th, 2021.

[103] L. Wang, M. Li, Y. Zhang, T. Ristenpart, and M. Swift. Peeking behind the curtains of serverless platforms. In *ATC*, 2018.

[104] P. Maissen, P. Felber, P. Kropf, and V. Schiavoni. Faasdom: A benchmark suite for serverless computing. In *DEBS*, 2020.

[105] OpenFaaS Wesite. `https://www.openfaas.com`, . Accessed on May 27th, 2021.

[106] K3S Website. https://k3s.io. Accessed on May 27th, 2021.

[107] J. Wen, Y. Liu, Z. Chen, Y. Ma, H. Wang, and X. Liu. Understanding characteristics of commodity serverless computing platforms. *arXiv 2012.00992*, 2020.

[108] OpenFaaS Function Store. https://github.com/openfaas/store, . Accessed on May 27th, 2021.

[109] pigo. Pure Go Face Detection Library. https://github.com/esimov/pigo.

[110] stackblur. Go Implementation of Stackblur Library. https://github.com/esimov/stackblur-go.

[111] functionbench. FunctionBench Github. https://github.com/kmu-bigdata/serverless-faas-workbench. Accessed on May 27th, 2021.

[112] M. Bilal, M. Serafini, M. Canini, and R. Rodrigues. Do the best cloud configurations grow on trees? an experimental evaluation of black box algorithms for optimizing cloud workloads. In *VLDB*, 2020.

[113] M. Bilal, M. Canini, and R. Rodrigues. Finding the right cloud configuration for analytics clusters. In *SoCC*, 2020.

[114] M. McKaya, R. Beckmana, and W. Conoverb. Comparison of three methods for selecting values of input variables in the analysis of output from a computer code. In *Technometrics*, 1979.

[115] pyDOE Website. https://pythonhosted.org/pyDOE/. Accessed on May 27th, 2021.

[116] D. Rogora, A. Carzaniga, A. Diwan, M. Hauswirth, and R. Soulé. Analyzing System Performance with Probabilistic Performance Annotations. In *EuroSys*, 2020.

[117] J. S. Arora. *Introduction to Optimum Design (Third Edition)*. Elsevier, 2012.

[118] I. Müller, R. Marroquín, and G. Alonso. Lambada: Interactive data analytics on cold data using serverless cloud infrastructure. In *SIGMOD*, 2020.

[119] Q. Pu, S. Venkataraman, and I. Stoica. Shuffling, fast and slow: Scalable analytics on serverless infrastructure. In *NSDI*, 2019.

[120] B. Carver, J. Zhang, A. Wang, A. Anwar, P. Wu, and Y. Cheng. Wukong: A scalable and locality-enhanced framework for serverless parallel computing. In *SoCC*, 2020.

[121] S. Fouladi, R. S. Wahby, B. Shacklett, K. V. Balasubramaniam, W. Zeng, R. Bhalerao, A. Sivaraman, G. Porter, and K. Winstein. Encoding, fast and slow: Low-latency video processing using thousands of tiny threads. In *NSDI*, 2017.

[122] S. Fouladi, F. Romero, D. Iter, Q. Li, S. Chatterjee, C. Kozyrakis, M. Zaharia, and K. Winstein. From laptop to lambda: Outsourcing everyday jobs to thousands of transient functional containers. In *ATC*, 2019.

[123] J. Carreira, P. Fonseca, A. Tumanov, A. Zhang, and R. Katz. Cirrus: A serverless framework for end-to-end ml workflows. In *SoCC*, 2019.

[124] L. Feng, P. Kudva, D. Da Silva, and J. Hu. Exploring serverless computing for neural network training. In *CLOUD*, 2018.

[125] H. Wang, D. Niu, and B. Li. Distributed machine learning with a serverless architecture. In *INFOCOM*, 2019.

[126] L. Ao, L. Izhikevich, G. M. Voelker, and G. Porter. Sprocket: A serverless video processing framework. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 263–274, 2018.

[127] A. Singhvi, J. Khalid, A. Akella, and S. Banerjee. Snf: Serverless network functions. In *SoCC*, 2020.

[128] T. Elgamal. Costless: Optimizing cost of serverless computing through function fusion and placement. In *SEC*, 2018.

[129] Ö. Sedefoğlu and H. Sözer. Cost minimization for deploying serverless functions. In *SAC*, 2021.

[130] J. Spillner. Resource management for cloud functions with memory tracing, profiling and autotuning. In *WoSC*, 2020.

[131] S. Eismann, L. Bui, J. Grohmann, C. L. Abad, N. Herbst, and S. Kounev. Sizeless: Predicting the optimal size of serverless functions. *arXiv 2010.15162*, 2020.

[132] T. Yu, Q. Liu, D. Du, Y. Xia, B. Zang, Z. Lu, P. Yang, C. Qin, and H. Chen. Characterizing serverless platforms with serverlessbench. In *SoCC*, 2020.

[133] P. Ambati, Í. Goiri, F. Frujeri, A. Gun, K. Wang, B. Dolan, B. Corell, S. Pasupuleti, T. Moscibroda, S. Elnikety, et al. Providing slos for resource-harvesting vms in cloud platforms. In *OSDI*, 2020.

[134] Y. Zhang, Íñigo Goiri, G. I. Chaudhry, R. Fonseca, S. Elnikety, C. Delimitrou, and R. Bianchini. Faster and cheaper serverless computing onharvested resources. In *SOSP*. ACM, 2021.

[135] R. T. Marler and J. S. Arora. Survey of multi-objective optimization methods for engineering. In *Structural and multidisciplinary optimization*, 2004.

[136] X. Chen, L. Rupprecht, R. Osman, P. Pietzuch, F. Franciosi, and W. Knottenbelt. Cloudscope: Diagnosing and managing performance interference in multi-tenant clouds. In *2015 IEEE 23rd international symposium on modeling, analysis, and simulation of computer and telecommunication systems*, pages 164–173. IEEE, 2015.

[137] Investment in Fast Data Processing Growing: Survey. `http://www.enterpriseappstoday.com/data-management/investment-in-fast-data-processing-growing-survey.html`.

[138] Companies using Apache Storm. `http://storm.apache.org/Powered-By.html`, .

[139] S. T. Allen, M. Jankowski, and P. Pathirana. *Storm Applied: Strategies for Real-time Event Processing*. Manning Publications Co., Greenwich, CT, USA, 1st edition, 2015. ISBN 1617291897, 9781617291890.

[140] How spotify scales apache storm. https://labs.spotify.com/2015/01/05/how-spotify-scales-apache-storm.

[141] Apache storm performance tuners. https://www.ericsson.com/research-blog/data-knowledge/apache-storm-performance-tuners.

[142] Notes on tuning storm+trident. https://gist.github.com/mrflip/5958028.

[143] Performance tuning for hdinsight storm and microsoft azure eventhubs. https://blogs.msdn.microsoft.com/shanyu/2015/05/14/performance-tuning-for-hdinsight-storm-and-microsoft-azure-eventhubs.

[144] B. Palanisamy, A. Singh, L. Liu, and B. Jain. Purlieus: locality-aware resource allocation for MapReduce in a cloud. In *SC*, 2011.

[145] K. Kambatla, A. Pathak, and H. Pucha. Towards Optimizing Hadoop Provisioning in the Cloud. In *HotCloud*, 2009.

[146] M. J. Sax, M. Castellanos, Q. Chen, and M. Hsu. Performance optimization for distributed intra-node-parallel streaming systems. In *ICDEW*, 2013.

[147] Parameter tuning framework repository. https://github.com/MBtech/stormbenchmark/, .

[148] W. Zheng, R. Bianchini, G. J. Janakiraman, J. R. Santos, and Y. Turner. JustRunIt: Experiment-Based Management of Virtualized Data Centers. In *USENIX ATC*, 2009.

[149] M. D. Morris and T. J. Mitchell. Exploratory designs for computational experiments. *Journal of statistical planning and inference*, 43(3):381–402, 1995.

[150] Making storm fly with netty. https://yahooeng.tumblr.com/post/64758709722/making-storm-fly-with-netty.

[151] Intel storm benchmark. https://github.com/intel-hadoop/storm-benchmark, .

[152] Rolling top words topology from bigdatabench. http://prof.ict.ac.cn/bdb_uploads/bdb_streaming/SocialNetwork-JStorm.tar.gz.

[153] Sentiment analysis storm. https://github.com/zdata-inc/StormSampleProject.

[154] T-digest. https://github.com/tdunning/t-digest, .

[155] T-digest service. https://github.com/MBtech/TDigestService, .

[156] R. Calandra, J. Peters, C. E. Rasmussen, and M. P. Deisenroth. Manifold gaussian processes for regression. In *Neural Networks (IJCNN), 2016 International Joint Conference on*, pages 3338–3345. IEEE, 2016.

[157] H. Herodotou, Y. Chen, and J. Lu. A survey on automatic parameter tuning for big data processing systems. *ACM Computing Surveys (CSUR)*, 53(2):1–37, 2020.

[158] P. Shivam, V. Marupadi, J. Chase, and S. Babu. Cutting Corners: Workbench Automation for Server Benchmarking. In *USENIX ATC*, 2008.

[159] R. Thonangi, V. Thummala, and S. Babu. Finding Good Configurations in High-Dimensional Spaces: Doing More with Less. In *IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, 2008.

[160] W. Zheng, R. Bianchini, and T. D. Nguyen. Automatic Configuration of Internet Services. In *EuroSys*, 2007.

[161] J. Singer, G. Kovoor, G. Brown, and M. Luján. Garbage collection auto-tuning for Java MapReduce on multi-cores. *ACM SIGPLAN Notices*, 46(11):109–118, 2011.

[162] A. Ganapathi, K. Datta, A. Fox, and D. Patterson. A case for machine learning to optimize multicore performance. In *HotPar*, 2009.

[163] M. A. Shah, J. M. Hellerstein, S. Chandrasekaran, and M. J. Franklin. Flux: An adaptive partitioning operator for continuous query systems. In *ICDE*, 2003.

[164] Y. Xing, S. Zdonik, and J.-H. Hwang. Dynamic load distribution in the borealis stream processor. In *21st International Conference on Data Engineering (ICDE'05)*, pages 791–802. IEEE, 2005.

[165] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. R. Madden, F. Reiss, and M. A. Shah. TelegraphCQ: Continuous Dataflow Processing. In *SIGMOD*, 2003.

[166] M. A. U. Nasir, G. D. F. Morales, N. Kourtellis, and M. Serafini. When two choices are not enough: Balancing at scale in distributed stream processing. In *Data Engineering (ICDE), 2016 IEEE 32nd International Conference on*, pages 589–600. IEEE, 2016.

[167] M. A. U. Nasir, G. D. F. Morales, D. García-Soriano, N. Kourtellis, and M. Serafini. The power of both choices: Practical load balancing for distributed stream processing engines. In *Data Engineering (ICDE), 2015 IEEE 31st International Conference on*, pages 137–148. IEEE, 2015.

[168] B. Lohrmann, D. Warneke, and O. Kao. Massively-parallel stream processing under qos constraints with nephele. In *Proceedings of the 21st international symposium on High-Performance Parallel and Distributed Computing*, pages 271–282. ACM, 2012.

[169] T. Das, Y. Zhong, I. Stoica, and S. Shenker. Adaptive stream processing using dynamic batch sizing. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 1–13. ACM, 2014.

[170] S. Venkataraman, A. Panda, K. Ousterhout, A. Ghodsi, M. J. Franklin, B. Recht, and I. Stoica. Drizzle: Fast and adaptable stream processing at scale, 2016. `http://shivaram.org/drafts/drizzle.pdf`.

[171] L. Aniello, R. Baldoni, and L. Querzoni. Adaptive Online Scheduling in Storm. In *DEBS*, 2013.

[172] B. Peng, M. Hosseini, Z. Hong, R. Farivar, and R. Campbell. R-storm: Resource-aware scheduling in storm. In *Proceedings of the 16th Annual Middleware Conference*, pages 149–161. ACM, 2015.

[173] P. Mendes, M. Casimiro, P. Romano, and D. Garlan. Trimtuner: Efficient optimization of machine learning jobs in the cloud via sub-sampling. In *2020 28th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 1–8. IEEE, 2020.

[174] flink-scaling. Elastic Scaling in Flink. `https://ci.apache.org/projects/flink/flink-docs-release-1.13/docs/deployment/elastic_scaling/`.

[175] M. Wistuba and J. Grabocka. Few-shot bayesian optimization with deep kernel surrogates. *arXiv preprint arXiv:2101.07667*, 2021.