POLITECNICO DI MILANO
DIPARTIMENTO DI ELETTRONICA, INFORMAZIONE E BIONGEGNERIA
DOCTORAL PROGRAMME IN INFORMATION TECHNOLOGY

CONSTRAINT-AWARE PERFORMANCE AUTOTUNING IN LIVE PRODUCTION ENVIRONMENT

Doctoral Dissertation of:
Stefano Cereda

Supervisor:
Prof. Paolo Cremonesi

Tutor:
Prof. Nicola Gatti

The Chair of the Doctoral Program:
Prof. Luigi Piroddi

2022 – Cycle XXXIII

Sommario

I Moderni sistemi informatici offrono centinaia di parametri di configurazione che ne modificano comportamento e prestazioni. Cercare manualmente una configurazione ben performante è un compito estremamente difficile, specialmente considerando che la prestazione del sistema dipende sia dalla configurazione applicata che dal carico a cui il sistema è esposto. In letteratura esistono diversi approci basati su intellligenza artificiale per cercare configurazioni ottimizzate. Tuttavia, queste soluzioni spesso sono indirizzate verso applicazioni specifiche, necessitano di grandi quantità di dati (che devo essere periodicamente aggiornate) o richiedono di eseguire numerosi test di performance per poter trovare configurazioni ben performanti. Tipicamente, questi test di performance sono eseguiti in un ambiente di test, dove è possibile valutare tutte le configurazioni desiderate. In molti casi reali, tuttavia, questo non è possibile visto che non esiste un ambiente di test, e le configurazioni devono essere valutate direttamente nell'ambiente di produzione, con l'applicazione soggetta al carico reale su cui non abbiamo alcun controllo. Questa tesi indirizza queste problematiche proponendo una metodologia di ottimizzazione di tipo Bayesiano. Nello specifico, l'approccio proposto è generico ed olistico in quanto non indirizzato verso un'applicazione o tecnologia specifica, e capace di considerare i diversi componenti di un tipico stack IT. Le condizioni di lavoro (come il carico) vengono modellate attraverso dei processi Gaussiani contestuali. Un modulo di predizione modella il carico in ingresso all'applicazione, consentendo di decidere automaticamente quando è più opportuno eseguire i test di performance nell'ambiente di produzione. Viene inoltre dedicata particolare attenzione a rispettare diversi vincoli sulle prestazioni dell'aplicazione. Il modello viene valutato su diversi sistemi, come database e applicazioni Java.

Abstract

MODERN IT systems offer hundreds of tunable knobs that impact their performances. Manually finding a well-performing configuration is a daunting task, especially when considering that the performance associated with a knob configuration varies with the workload to which the system is exposed. Hence, many machine-learning-based approaches have been proposed to find optimized configurations. However, they usually target a specific application, need huge knowledge bases (which must be updated periodically) or perform many performance tests to find well-performing configurations.

Usually, these performance tests are run in a performance test environment, where we can evaluate as many configurations as we desire. In many situations, this is not feasible, as the test environment is not available, and we must evaluate the configurations directly in the production environment while exposed to the real live workload, over which we have no control. This thesis proposes a configuration tuning methodology addressing these issues based on Bayesian Optimisation. In particular, the proposed approach is generic and holistic in that it is not tailored around a specific application and considers multiple layers of the IT stack. We model the external working conditions (such as the workload) via Contextual Gaussian Processes. We use a forecasting module to decide when to schedule performance testing experiments in the production environment. Finally, we give special consideration to satisfying service level agreement (SLA) constraints. We evaluate the approach on various tunable systems, such as database management systems and java web applications.

Contents

1 Introduction 1
1.1 Motivation . . . 1
1.2 Goals . . . 3
1.3 Research Contributions . . . 4
1.4 List of Publications . . . 5
1.5 Structure . . . 5

2 Related Works 7
2.1 Configuration Autotuning . . . 7
2.1.1 Compiler Autotuning . . . 7
2.1.2 DBMS Autotuning . . . 10
2.1.3 Autoscaling . . . 11
2.1.4 Other Fields . . . 12
2.2 Constrained Optimisation . . . 13
2.2.1 Constraints Taxonomy . . . 13
2.2.2 Approaches to Constrained Bayesian Optimisation . . . 15
2.3 Context Forecasting . . . 15
2.3.1 Workload Characterisation . . . 15
2.3.2 Workload Forecasting . . . 16

3 Proposed Approach: CC-GP 19
3.1 Problem Definition . . . 19
3.1.1 Space Normalisation . . . 21
3.2 Bayesian Optimisation . . . 23
3.2.1 Acquisition Functions . . . 24
3.3 Contextual Gaussian Process Bandit Optimisation . . . 26
3.4 Constraint-Aware Optimisation . . . 27
3.4.1 QUAK Constraints - Parameters . . . 28
3.4.2 QRAK Constraints - Exploration . . . 28
3.4.3 NUSH Constraints - Startup Failures . . . 30
3.4.4 QRSK Constraints - Metric SLAs . . . 30

3.4.5 Unsolvable Constraints . . . 32
3.5 Context-based Normalisation for Relevant Priors . . . 32
3.5.1 Availability of a Control Group . . . 34
3.6 Context Clustering . . . 34
3.7 Context Forecasting . . . 35
3.7.1 Experiment Scheduling . . . 37
3.8 Safety . . . 38
3.8.1 Local Safety . . . 38
3.8.2 Global Safety . . . 38

4 Experimental Evaluation Framework **41**
4.1 Analytical Function . . . 42
4.2 Dataset-based Simulations . . . 42
4.2.1 Search spaces . . . 44
4.2.2 Models Accuracy . . . 45
4.2.3 Workload Patterns . . . 46
4.3 Evaluation Metrics . . . 48
4.3.1 Normalised Performance Improvement . . . 50
4.3.2 Online Optimality — Cumulative Reward . . . 50
4.3.3 Offline Optimality — Iterative Best . . . 51
4.3.4 Constraint Violations . . . 52
4.3.5 Workload Forecasting . . . 52
4.4 Baseline Algorithms . . . 53
4.5 Hyperparameter Tuning . . . 53

5 Offline Autotuning **55**
5.1 Single Context . . . 55
5.1.1 Analytical Functions . . . 55
5.1.2 Dataset-based Simulation . . . 59
5.2 Multiple Workloads . . . 62
5.2.1 Dataset-based Simulation . . . 62

6 Online Autotuning **65**
6.1 Workload Tracking . . . 65
6.2 Workload Forecasting and Experiment Scheduling . . . 66
6.2.1 Considered Workloads . . . 66
6.2.2 Workload Forecasting . . . 66
6.2.3 Experiment Scheduling . . . 67

7 Constraints **71**
7.1 Combined Goal Function . . . 71
7.1.1 Suggested Configurations with Combined Goal . . . 73
7.2 SLA Constraints . . . 75
7.2.1 Constraints vs Combined Goal Function . . . 75
7.2.2 Local vs Global Safety . . . 76

8 Reaction-Matching Characterisation **79**
8.1 Top Popular — TP . . . 80

8.2 Exploiting characterisation metrics 81
8.2.1 Content-Based Filtering — CBF 82
8.2.2 Collaborative Filtering — CF 82
8.3 Experimental Evaluation of Collaborative Filtering 83
8.3.1 Reaction Matching vs Performance Counters 86

9 Time Complexity 89

10 Conclusions and Future Works 93

Bibliography 95

CHAPTER 1

Introduction

1.1 Motivation

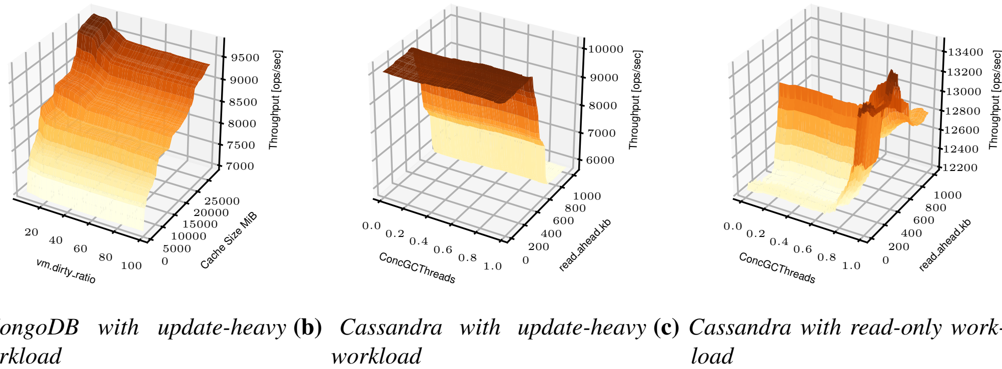
A modern IT system has hundreds of tunable configuration parameters that control its behaviour [38, 65, 5]. Selecting the proper configuration is crucial to improving performance or reducing cost. However, manually finding well-performing configurations can be a daunting task since the parameters often behave in counter-intuitive ways and have inter-dependencies. Furthermore, a modern system sits on top of a complex IT stack that comprises several layers, like the Java Virtual Machine (JVM) or the Operating System (OS). Each layer has its tunable parameters, which affect the final behaviour of the system, as we show below. To unlock the full performance potential of a system, we have to tune the entire IT stack jointly.

Unfortunately, we cannot run an extended search and find the optimal configuration which is the best one for our particular stack (i.e., the combination of application, OS, hardware and other layers). Even if we had an infinite budget to run this search, we would still find a suboptimal solution as the optimal configuration depends upon the particular workload to which the system is exposed.

We could even imagine running an extensive search to find the optimal configuration for each particular workload or at least a well-performing configuration for each workload. However, all this knowledge would become obsolete quickly, as new software versions are released, changing the effects of the parameters. Furthermore, new software releases also modify the available parameters, increasing the complexity of reusing old knowledge bases, which lack information about novel parameters.

To highlight the effects of configuration tuning, we ran a series of experiments on the MongoDB[1] and Cassandra [63][2] database management systems (DBMSs) using

[1] https://www.mongodb.com
[2] https://cassandra.apache.org



(a) *MongoDB with update-heavy workload* **(b)** *Cassandra with update-heavy workload* **(c)** *Cassandra with read-only workload*

Figure 1.1: *Throughput of a DBMS as a function of some of its tunable parameters. In (a) we use MongoDB and vary its cache size and the* `vm.dirty_ratio` *Linux parameter. In (b) we use Cassandra, varying the number of threads for the concurrent garbage collector of the JVM (expressed as a percentage of the available cores) and the* `read_ahead_kb` *parameter of the Linux kernel. In (c) we repeat the experiment of (b), but using a different YCSB workload.*

the YCSB load injector [24][3]. We modified the values of the tunable parameters while measuring the throughput of the DBMSs. The results are reported in Figure 1.1.

In Figure 1.1a, we modify the cache size of MongoDB and the `vm.dirty_ratio` parameter of the Linux kernel. Correctly setting the latter parameter gives a significant boost to the performance of the DBMS: without considering the entire IT stack, we would have lost this improvement. In Figure 1.1b, we use Cassandra, which is written in Java. The Java Virtual Machine has its tunable parameters; here, we tune the number of concurrent garbage collection threads alongside the `read_ahead_kb` Linux parameter. The read-ahead has an impressive effect on the performance of Cassandra, and selecting an improper value destroys the performance. In Figure 1.1c, we modify the two same parameters but use a different YCSB workload. More precisely, we move from an update-heavy workload to a read-only workload. The effect of the two parameters is severely different in the two workload conditions.

These examples motivate the need to consider the entire IT stack and the current workload when tuning a DBMS, and similar cases can be made for any modern application. To further increase the complexity, we could consider other tunable parameters like the compilation flags of MongoDB, the version of the JVM or the kind of cloud instance on which to run the experiments. However, increasing the number of considered layers increases exponentially the number of parameters and the problem's complexity.

Notice that if we were to tune the entire IT stack while looking only at the DBMS, we might destroy the system's performance, as some OS settings that are beneficial to the DBMS could be detrimental to other services running on the same machine. However, a proper selection of the target performance metric to be optimised is sufficient to avoid this problem. We can consider a single server running a DBMS and a Java backend application as a simple example. If we tune the OS to increase the DBMS performance as much as possible, the backend will suffer, leading to a poor user experience. On the other hand, if we tune to minimise the response time of some user-facing services,

[3] `https://ycsb.site/`

we will converge toward configurations that allow both the DBMS and the backend application to work at their best. In reality, tuning the entire system while targeting a user-facing performance metric is an advantage, as it allows to tune the system to work in the desired way. If we were to separately tune the DBMS to increase its throughput and the JVM to decrease the garbage collection time (as it's often done in the industry) we would have no guarantee that the resulting combined system would behave well.

As many IT systems are moving from a monolithic approach to a graph of thousands of microservices, the performance autotuning problem becomes even more complex. The entire system's performance is affected by all the microservices in complex ways, and even the slow-down of a single service can impact the user experience [44]. Add to this that a system could be dependent on a service offered by another system that is not under our control, and this second system can potentially misbehave, slowing down the first system and making the tuning process harder. A proper autotuning system should thus be able to model and cope with these situations.

Finally, we must consider that the final goal of the performance analyst is not just to optimise performance, but to optimise performance while satisfying some constraints. For example, we might be interested in maximising throughput under a budget constraint or in minimising cost while meeting some service level agreements on the response time. The existance of constraints makes the optimisation harder, as we need to understand how the applied configuration affects the constrained metric, while testing configurations that should satisfy the constraints as much as possible. Depending on the tuning scenario, we might be interested in satisfying the constraints only for the current workload, or we might be interested in finding a configuration that satisfies the constraints on all the possible workloads. The same problem applies to the dependency on external services we explained above.

Notice that there exists different families of constraint: we have constraints on the applied configurations (e.g., the heap size of the JVM must be smaller than the constainer memory limit), which we must satisfy in order to provide an applicable configuration, and constraints on the performance metrics resulting from evaluating an applied configuration. Among metric constraints, some are mandatory (e.g., the application has to start successfully), while other can be violated (e.g., an SLA on the response time), even if our goal is to guarantee that they are not violated. Specifically, we assume that violating a constraint is much worse that providing a sub-optimal configuration. This work hence considers different types of constraints, using different strategies to deal with them, considering constraint satisfaction the first goal of the tuning process.

In conclusion, having an autotuner that respects SLA constraints would allow to deploy the autotuner directly in a production environment without resorting to a duplicate performance testing environment. Apart from the cost-saving, this would also allow obtaining a more reliable configuration, as it is tested on the real environment with the real workload, preventing any issue arising from human errors in the environment duplication.

1.2 Goals

The main goal of this work is to develop an autotuner that is:

- *Generic*: it works on a wide variety of target applications as it uses a black-box

approach without making any assumptions about the underlying system. This avoids the burden of building and maintaining models of the various systems and allows to focus on the desired performance metrics.

- *Holistic*: in that it simultaneously targets various layers of the IT stack. This allows to explore the inter-dependencies of various parameters and achieve better performance.
- *Contextual*: the autotuner models the performance as a function of the tunable parameters and other external factors, called context. The context can be used to model the incoming workload, obtaining a workload-aware autotuner, but it can also be used to model external dependencies of the system, such as the response time of an external system that affects our target system but is not under our control.
- *Safe*: the autotuner is aware of Service Level Agreement (SLA) constraints and tries to suggest configurations that satisfy them. This capability works in conjunction with the Contextual part, as the autotuner can be configured to suggest a configuration that is safe for the current context (e.g., when we try to adapt the configuration to the incoming workload) or a configuration that is safe on all the previously observed contexts (e.g., when we want to take into account the possible slow-down of an external system).

1.3 Research Contributions

In this section, we detail the research contributions of this work, which are mainly related to the development of a production-ready, workload-aware, SLA-constrained autotuner for generic IT stacks.

1. *Introduction of evaluation framework.* To compare different tuning approaches, some clearly defined evaluation criteria and metrics are necessary. As we are dealing with noisy performance measurements, it is crucial to discern whether an observed performance improvement is due to a change in the configuration or is just some random fluctuation. To this end, we introduce a set of normalised metrics that are interpretable and easy to understand.
2. *Context-aware autotuner.* As mentioned above, the suggested configuration must be adapted to the context to which the system is exposed. We introduced an autotuner based on Contextual Gaussian Process bandits (CGP) able to deal with context variations, such as the workload.
3. *Context forecasting.* When dealing with a variable context, it is crucial to be able to forecast the value of future contexts to modify the applied configuration proactively. Furthermore, testing a configuration when the context is unstable would lead to noisy measurements. Hence, we introduce a context forecasting module working in tight conjunction with the autotuner and scheduling when to run the experiments.
4. *SLA-constraints.* In real scenarios, optimising a single performance metric is not the actual goal. Instead, it is essential to optimise a metric while meeting a set

of Service Level Agreement (SLA) constraints. Violating these constraints, especially in a production environment, leads to economic penalties, which the autotuner must avoid at any cost. We thus modify the Bayesian Optimisation framework to deal with different kinds of constraints.

5. *Reaction matching characterisation.* To speed up the tuning of an application, we can re-use the knowledge collected in the tuning of another application. To transfer knowledge in a helpful way, it is crucial to select relevant applications from an existing knowledge base of previous tuning sessions. To do so, we introduce the Reaction Matching characterisation, which can be used to characterise applications and thus find similarities, which we plan to use as a future extension to add a transfer learning module to the existing autotuner.

1.4 List of Publications

- Stefano Cereda, Gianluca Palermo, Paolo Cremonesi, and Stefano Doni. A collaborative filtering approach for the automatic tuning of compiler optimisations. In *The 21st ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*, pages 15–25, 2020
- Stefano Cereda, Stefano Valladares, Paolo Cremonesi, and Stefano Doni. Cgptuner: a contextual gaussian process bandit approach for the automatic tuning of it configurations under varying workload conditions. *Proceedings of the VLDB Endowment*, 14(8):1401–1413, 2021
- Stefano Doni, Giovanni Paolo Gibilisco, and Stefano Cereda. Method and apparatus for tuning adjustable parameters in computing environment, September 17 2020. US Patent App. 16/818,263
- Stefano Cereda, Paolo Cremonesi, Giovanni Paolo Gibilisco, and Stefano Doni. Method and apparatus for tuning a computing environment using a knowledge base, 2022. US Patent App. currently in submission

1.5 Structure

- Chapter 1 introduces this thesis and describes its motivations and research contributions, as well as list the publications directly related to the work carried out for the writing of this thesis.
- Chapter 2 covers the current state of the art of configuration autotuning and gives basic information on constrained optimisation and time-series forecasting, which are used in the developed model.
- Chapter 3 details the proposed approach, covering all the aspects of the autotuner.
- Chapter 4 details the evaluation framework used in the experimental evaluation and describes all baseline algorithms used, the optimisation procedure and the metrics that are reported in the evaluation.

- Chapter 5 reports the results obtained in offline tuning scenarios, meaning tuning sessions where the experiments are executed in duplicated performance testing environments.
- Chapter 6 reports the results obtained in online tuning scenarios, meaning tuning sessions where the experiments are executed in the production environments and hence the systems are subject to more unstable working conditions (e.g., workload).
- Chapter 7 reports the results for constrained tuning, analysing the ability of the autotuners to respect the given Service Level Agreements (SLAs).
- Chapter 8 reports the results obtained for the tuning of binary search spaces.
- Chapter 9 analyses the time complexity of the proposed autotuner.
- Chapter 10 presents the conclusions and future extensions.

CHAPTER 2

Related Works

This chapter provides a brief overview of existing autotuning approaches and, in general, the broader topic of optimisation, with close attention to constraints' handling. We also cover the topic of workload characterisation and forecasting.

2.1 Configuration Autotuning

At its core, the configuration autotuning problem is simply an optimisation problem, for which, in principle, we could apply many out-of-the-box optimisation algorithms. However, a generic optimisation approach would fail miserably in a real scenario, as it would fail to address many of the peculiarities of the targeted domain. Hence, the configuration autotuning problem has been studied in many different IT fields, especially in compiler autotuning and DBMS knobs autotuning.

2.1.1 Compiler Autotuning

When compiling a program from a high-level language to its executable binary, we can enable several compiler optimisations (e.g., loop unrolling, register allocation, function inlining). These optimisations control how the code is transformed and generated and severely impact the performance of the compiled program. Compilers usually offer some predefined optimisation levels, such as `-O1`, `-O2`, `-O3` and `-Os` for GCC[1], which contain various optimisations empirically determined to be beneficial in most cases [54]. However, the optimal set of enabled optimisations depends on the specific program to be compiled, making those predefined levels sub-optimal. Selecting which optimisations to enable is known as the flag selection problem. Another deeply studied problem is the phase-ordering of optimisations (i.e., selecting the order in which

[1] Gnu Compiler Collection `https://gcc.gnu.org/`

to perform the various code optimisations). Here we only focus on the flag selection problem since the ordering of various options is specific to the compiler domain, and we are interested in building a generic autotuner.

The most basic approach to solving this problem is repeatedly compiling the program with several different configurations and running it against a known and fixed workload, looking for well-performing configurations. However, testing all the possible configurations would require too much time as the search space grows exponentially with the number of available optimisations. GCC 9.0.1, for instance, offers 244 optimisation flags that can be turned on or off and 215 optimisation parameters for which we can set numerical values. The compiler autotuning problem's high dimensionality forced researchers toward faster exploration strategies. Some focus on using more innovative search algorithms, while others try to gain some knowledge from previously performed compilations.

Several techniques have been proposed to find good optimisation sets. As testified by two recent and extensive surveys [5, 110], automatic tuning of compiler optimisations has recently undergone a revamp. This is not only due to the advantages provided by a customised selection of optimisations, but also to the rise of new architectures like RISC-V (but also ARM), for which the standard compiler optimisations levels like `-O2, -O3` do not provide good-enough performance as done for Intel processors. Instead of manually deriving novel optimisation levels, researchers focused on finding automatic ways to find good sets of optimisations, adapting them to the particular program and target architecture.

Iterative Compilation (IC) techniques work by testing a large number of compiler optimisation sets until a sufficiently well-performing one is found [12, 23]. IC results in superior performance over predefined optimisation levels [3, 26]. However, this requires long search times, which is a significant barrier to the general adoption of these techniques. To make IC more widespread, several search strategies have been proposed to obtain faster explorations [25, 61, 60, 26, 39, 72, 79, 43, 2, 104]. We consider OpenTuner [4] as a representative of this family of approaches. Instead of focusing on a specific search technique, OpenTuner contains several algorithms (such as genetic algorithms, hill climb and multi-armed bandits) and dynamically decides which one to use. When tuning a program, OpenTuner starts by randomly trying different techniques, and, as the tuning proceeds, it allocates a larger proportion of tests to better performing techniques. In this way, OpenTuner selects the best-performing search algorithm for the specific program it is tuning. Nonetheless, many works reported that a random search performs as well as more sophisticated techniques, and is indeed an effective tool for exploring the space of the available optimisations [2, 18, 23, 58].

While search-based techniques derive their knowledge online and treat every program to be compiled as a new search problem, other solutions try to exploit some previously collected knowledge [81, 2, 18, 6]. The main idea is that the information gathered while searching for the optimal set for a program can be re-used to speed-up the search on another program. Such approaches work by extensively exploring the optimisation space for many programs, so to build a knowledge base which is then shipped with the compiler.

When compiling a new program, previous knowledge is used to guide the search. Since the knowledge base contains information about many programs, these methods

need a way to understand which are the most informative data that can be exploited for the program under compilation. Multiple techniques can be used to identify this relevant information.

Static code features have been used in [2], in conjunction with machine learning models, to guide the selection of optimisations. The approach starts by characterising the program under compilation. The extracted features are then transformed with Principal Component Analysis (PCA) [55]. Using the resulting feature vector and a Euclidean distance, makes it possible to find the closest program among the available ones. This similar program is then used to focus the search process on the new program. It is possible to use a random search (or a genetic algorithm) and speed up the search process by giving a bigger probability of being selected to the optimisation sets which lead to performance improvement on the similar program.

This approach is refined in [18], where dynamically extracted features are used instead of static ones. Dynamic features are collected while the program runs and require costly instrumentation. However, they can better describe the behaviour of a program. Afterwards, logistic regression is used to directly learn a mapping from these features to the set of good optimisations.

Cobayn [6] also builds on this idea, combining static and dynamic features and using a Bayesian Network (BN) [40] to focus the search process. The proposed framework is based on a program characterisation step performed both statically on the source code using MilePost [42] and dynamically with MICA (Microarchitecture-Independent Characterization of Applications) metrics [53]. To collect these metrics, the program is compiled with a predefined optimisation set (i.e., `-O3`) and run against the interesting workload. As the program runs, the characterisation metrics are collected. These metrics are then processed with PCA or Exploratory Factor Analysis [51] (EFA) to reduce their dimensionality. The resulting characterisation vector is fed in the BN which has been previously trained so to output the optimal set according to the input metrics.

Most of the existing approaches are based on the assumption that having similar characterisation metrics implies having similar optimal sets. However, this largely depends on the methodology used to characterise the program. Since manually deciding which metrics may be relevant is a complex task, deep learning methodologies have been proposed to extract them from source code automatically. DeepTune [28] uses a series of artificial neural networks taking as input the source code and producing as output the expected optimal value for a configuration parameter. Such approaches are well suited to analyse short kernel programs, but the characterization becomes more problematic when the dimension of the code base increases.

Features extracted statically from the code (such as MilePost or the ones implicitly used in DeepTune) do not consider the actual workload to which the program is exposed. Dynamic features (such as the MICA ones) avoid this problem and produce better results [6]. However, even dynamic features have their limitations, as they are expensive to collect and are just a noisy proxy to the measure we are interested in, which is how a program performs when compiled with specific optimisations.

A reaction-based approach can outperform approaches exploiting code-based features: in [17] programs are characterised in terms of the speed-up they receive when compiled with four *canonical* transformations. By compiling and running a program with only four sets of optimisations, they accurately predict the speed-up obtained by

the program over 88000 possible sets, outperforming feature-based models. These four canonical sets are decided a priori by maximising an information gain measure.

Reaction-based approaches can target both whole programs and specific code sections since they are independent of code structure. Moreover, by measuring performance speed-ups, they can be tailored to a specific workload, which is not possible when working with source code. The approach presented in [17] is focused on the problem of predicting the speedup obtained with a specific set, whereas we focus on an IC model, where we would like to find the optimal set as soon as possible.

FuncyTuner [109] focuses on assembling an optimised executable by separately optimising different code regions, obtaining superior performance. However, doing so requires executing a greater number of compilations, which is feasible in some domains where programs are executed repeatedly with similar inputs, but is a problem in other domains, where one can test a limited number of binaries before the input changes.

In [20] we introduced Reaction-Matching, a characterization methodology based on how different programs reacted to different optimisations. The basic idea is that if two different programs benefit from the same optimisation, we should consider them similar and optimise them similarly. As we will show in Chapter 8, this approach will be used as a future extension to speed up the proposed autotuner when a knowledge base is available.

2.1.2 DBMS Autotuning

DBMS are notoriously challenging to deploy and administer. Hence, many efforts have been devoted to developing tuning tools, working on many aspects of a DBMS [82]. As we are interested in building a generic autotuner, we ignore the tools that target the design of a database, such as indexes, partitioning or views. Instead, we focus on the problem of selecting the values of configurable parameters, sometimes called knobs.

iTuned [38] works by creating a response surface of the DBMS performance with Gaussian Processes and using this model to select the next configuration to test. However, a different response surface is built for each workload without sharing potentially useful information.

In their seminal paper [108], Van Aken et al. introduced OtterTune: a machine learning solution to the DBMS tuning problem. OtterTune leverages experience and collects new information to tune DBMS configurations: it uses a combination of supervised and unsupervised machine learning methods to (1) select the most impactful parameters, (2) map unseen database workloads to previous workloads from which it can transfer experience, and (3) recommend parameters settings. The key aspect of OtterTune is its ability to leverage experience to speed up the search process on new workloads. However, to do so, it requires an extensive collection of previous experiments. To fully leverage the OtterTune approach, all these experiments should contain all the available parameters. This might be feasible when considering only the DBMS, but it gets more and more complex as we consider more layers of the IT stack since the dimension of the search space grows exponentially. As an example, in [108] the authors had to collect "over 30k trials per DBMS" just to bootstrap OtterTune. Even with very short measurement periods of 5 minutes per trial, this results in more than three months of computation just to collect the initial knowledge base. Furthermore, OtterTune is exploring a single layer of the IT stack, whereas we want to consider as

many layers as possible so to extract all the potential performance. As the number of layers increases, the complexity of collecting a knowledge base increases too.

Finally, this abnormous knowledge base should be updated periodically to reflect changes in hardware components and software versions. Different software versions react differently to the same configurations, and new software versions introduce new tunable parameters. To take these parameters into account, one would have to periodically re-build the knowledge base from scratch.

In [21] we introduced CGPTuner, an autotuner designed to tune parameters across different layers of the IT stack and adapt the suggested configurations to the current workload without relying on a previously collected knowledge base. CGPTuner is based on contextual Bayesian optimisation, which is a technique specifically designed to handle the exploration-exploitation trade-off, with theoretical solid guarantees about its convergence and data-efficiency [97].

2.1.3 Autoscaling

In many situations, users have to specify the amount of resources required to complete a certain task. For example, in cloud systems, we have to select the instance where we want to tune our application, specifying the number of CPU cores and the size of RAM. Likewise, when starting a container, we can specify a limit for the amount of CPU and memory resources. In a Kubernetes cluster, users have to decide the number of pod replicas and the resource limits for each pod. When the specified limits are exceeded, the container is throttled or killed, which causes substantial performance issues. Therefore, it is natural to specify higher limits than needed, which results in a waste of resources [92]. A lot of effort has thus been devoted to the development of *auto-scalers*.

Autoscalers are usually categorised as *vertical* or *horizontal*:

- horizontal auto-scalers work by adding or removing replicas;
- vertical auto-scalers work by tuning the amount of resources requested by each replica.

The majority of the existing approaches work in response to a change in the workload or the resource utilisation, potentially with an additional component to provide workload forecasts and resource demand estimation to act proactively.

As a fundamental example, [91] identifies three main challenges to automatic resource provisioning:

- *Workload Forecasting*: while releasing resources is an easy task, allocating additional resources leads to performance overheads, as we need to wait for the deployment of the resources and the application startup (and potentially warmup). We thus desire to acquire resources before they are actually needed, and this can be done only if we can predict the future workload.
- *Identify Resource Requirement for Incoming Workload*: once we can predict the future workload, we still have to estimate the amount of resources that will be required to serve that workload, so we can acquire them.

- *Resource Allocation while Optimizing Multiple Cost Factors*: the optimal solution would be to vary the required resource to match the current workload continuously. In reality, however, we have some overhead when allocating resources, and a proper auto-scaler needs to consider this.

In [68], auto-scaling techniques are classified into five main categories: static threshold-based rules, control theory, reinforcement learning, queuing theory and time series analysis. The same work identified threshold-based auto-scalers as the most popular approach due to their simplicity and highligted their questionable effectiveness under bursty workloads. Control theory and queueing theory models work by creating a model of the system performance, and their performance clearly depends on the quality of the model. Moreover, queueing theory models are identified as too rigid. Finally, time series analysis is identified as the primary enabler of proactive auto-scaling techniques and reinforcement learning as a way to create a model of the system without relying on a-priori knowledge.

Nonetheless, by focusing only on auto-scaling the resources without considering the entire IT stack with its tunable properties, potential performance improvements and cost reductions are left on the table. A practical solution consists in tuning the application configuration for a fixed test workload during the development process, and then shipping it on the cloud enabling resources auto-scaling. Nonetheless, we have no guarantees that the identified configuration is the best one for all the possible workloads, hence we should exploit the flexibility offered by the cloud to continuously vary the application configuration to match the working condition, just like we do with resources auto-scaling.

Moreover, with the recent rise of microservices architecture, the applications are becoming more complex, up to the point where modelling the performance of a system as a function of its configuration and incoming workload is simplistic. In reality, the application performance depends on the performance of its underlying dependencies, and a spike in the response time of a microservice that is not under our control can increase the response time of our application, potentially even to SLAs violation. Hence, in our approach, we do not optimise for a certain workload, but consider the entire working condition (called context), which includes the workload but can be extended to include any measurable metric which has an impact on the performance of our system but is not under our control. Then, we try to provide configurations that are expected to respect the given SLAs not only on the forecasted workload, but also on all the possible contexts, hence taking into consideration the potential failure of external services.

2.1.4 Other Fields

Autotuning approaches have been successfully employed in many specific applications, such as high-performance computing [7], machine learning [9], and even cookie recipes [50].

As for generic software configuration autotuning, many approaches are also available. BestConfig is an autotuning system for automatically finding the best configuration setting within a resource limit for a deployed system under a given application workload and is based on an iterative sampling strategy [114]. BestConfig does not try to create a model of the system. Instead, it iteratively samples a new configura-

tion by mixing random exploration of the search space and restricting the search space according to observed performance.

Conversely, other approaches are explicitly based on the existence of a performance model to exploit during the tuning process. The model can be known entirely a-priori (like in utilisation-based autoscaling), or it can be derived from observed data, like in Bayesian Optimisation. Sometimes, an a-priori model can be mixed with observed data, like in BOAT [29]. Approaches like OtterTune [108, 113] try to collect a preliminary knowledge base and use it to build a starting model, which is then coupled with collected data.

Sometimes, domain-specific knowledge is exploited in the construction of the autotuner to select a subset of interesting tunable knobs and a target performance metric, and a straghtforward optimisation algorithm is used to drive the search process. As an example, Otterman [37] is an effective Spark autotuner whose optimisation module is a combination of Least Squares and Simulated Annealing. Nonetheless, as we will argue in the experimental section, the fact that a very simple search algorithm finds configurations that are better than the vendor default one simply implies that the vendor default configuration is a bad one for the evaluated workload, and should not be used to evaluate the quality of an autotuner.

2.2 Constrained Optimisation

Further complicating the autotuning problem, we must consider the existence of some performance constraints. As an example, we might be interested in minimising the cost of our deployment while still achieving an acceptable response time. Such constraints, usually referred to as Service Level Agreements (SLAs), are of paramount importance and the autotuner should try to satisfy them. Also, we usually have some constraints over the optimisation space, such as the minimum and maximum heap dimensions for the JVM. Finally, we can even define some constraints over the optimisation process: we might be interested in allowing the optimiser to only evaluate configurations that are not too uncertain.

2.2.1 Constraints Taxonomy

As we are dealing with a variety of constraints, we need a way to classify them and treat similar constraints in a similar manner. We adopt the QRAK taxonomy, introduced in [34]. The QRAK taxonomy is tailored for simulation-based optimisation, where evaluating a configuration requires running a costly simulation. This is precisely the case of configuration autotuning, where evaluating a configuration requires running an expensive performance test.

The QRAK taxonomy, represented graphically in Figure 2.1, works by classifying each constraint according to four binary categories. The combination of these four decisions leads to the classification of the constraint.

The four classification criteria are:

- Known (K) versus hidden (H): a known constraint is a constraint that is explicitly given in the problem formulation. A hidden constraint is not explicitly known by the optimiser.

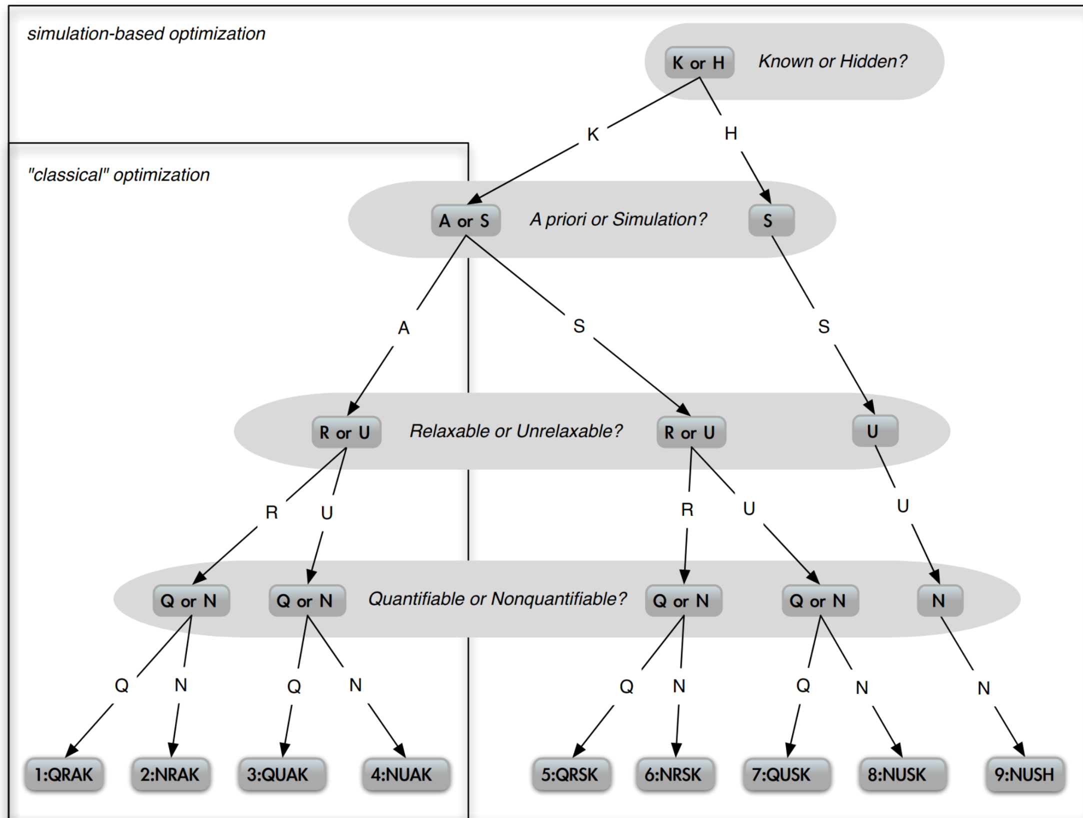


Figure 2.1: *Tree-based view of the QRAK taxonomy of constraints. Each leaf corresponds to a class of constraints [34].*

- A Priori (A) versus simulation-based (S): an a priori constraint is a constraint for which feasibility can be confirmed without running a simulation. A simulation-based constraint (or simulation constraint) requires running a simulation to verify feasibility.

- Relaxable (R) versus unrelaxable (U): a relaxable constraint is a constraint that does not need to be satisfied to obtain meaningful outputs from the simulations. An unrelaxable constraint must be satisfied for meaningful outputs to be obtained.

- Quantifiable (Q) versus nonquantifiable (N): a quantifiable constraint is a constraint for which the degree of feasibility and/or violation can be quantified. A nonquantifiable constraint is one for which the degrees of satisfying or violating the constraint are both unavailable.

Every possible constraint type is identified with a sequence of four letters, depending on how it classifies on the four properties detailed above. We will give in Chapter 3 a description of the various constraints used in configuration autotuning and their classification.

2.2.2 Approaches to Constrained Bayesian Optimisation

There exists several extensions to the Bayesian Optimisation framework to handle constraints [97, 10, 101, 49, 46]. In general, they work by modifying the acquisition function to penalise the exploration of configurations that are likely to violate some constraint. As we will show later, we are dealing with several different types of constraints, and these general approaches lead to sub-optimal performance. In this work, we propose a novel approach to Bayesian Optimisation to deal specifically with the kinds of constraints usually present in configuration autotuning.

2.3 Context Forecasting

As the autotuner has to adapt the suggested configuration to the external context, it is necessary to include a context characterisation and forecasting module, so to prepare a configuration for the next context. By context, we indicate any external factor that might impact on the system performance or our optimisation decisions, such as the incoming workload, the data center temperature or the spot cost of some AWS instances.

Essentially, this step involves selecting and observing some metrics that characterise the context, and then applying a time-series forecasting algorithm to derive the next estimate. We thus have two major decisions to take: the forecasting algorithm and the selection of characterisation metrics. We mainly focus on workload characterisation, as the selection of other external factors (e.g., datacenter temperature) is essentially unbounded and thus left to the user.

2.3.1 Workload Characterisation

To determine the performance of a system, it is necessary to know the load it is processing, and to provide a quantitative description. Workload characterisation has thus been extensively studied since the early 1970s. A comprehensive survey can be found in [16], where it is underlined that, depending on the target application, we should focus on different characterisation metrics. Hence, as our goal is to develop a generic autotuner, we take for granted the availability of a numerical description of the workload, without focusing on a specific technique. Here we focus on some of the approaches already used in configuration autotuning.

The most straightforward approach consists in simply monitoring the resource utilization, which is then used as the primary input for the tuning process. As an example, the Vertical Pod Autoscaler[2] (VPA) feature of Kubernetes observes the current resource utilization of an application and then restarts the application in a new container with new resources requirements, essentially trying to bring the utilizations near to the maximum while keeping some safety margin. The same idea is used in many different approaches to resource auto-scaling, which is essentiallly a form of workload-dependent autotuning [88, 77, 8].

However, a high resource utilisation does not necessarily imply that there is a demand for more resources (i.e., that the application would use more resources if they were allocated). Hence, some works have tried to estimate resource demands (which cannot be measured) and use them to guide decisions over deployed resources [30, 90].

[2] https://github.com/kubernetes/autoscaler/tree/master/vertical-pod-autoscaler

To better understand how the incoming workload affects resource demands, it is necessary to exploit some information regarding the target application. As an example, DBSeer [75] uses DBMS query logs to capture the running transactions, and then cluster together transactions with similar access patterns (i.e., they access the same tables in the same order and perform similar operations). QueryBot 5000 [70] forecasts the workload of a DBMS by clustering together queries with similar templates and arrival rates and using a separate forecaster for each cluster of queries.

2.3.2 Workload Forecasting

Once we have numerically characterised the workload (and the context) we might be interested in forecasting the future workload, albeit this is not mandatory. Essentially, the user has to decide whether the autotuner should continuously update the suggested configuration to reflect changes in the workload (e.g., resource auto-scaling), or converge toward a configuration that satisfies the constraints on all the observed workloads. Also, we might be interested in forecasting the workload but focusing on the worst-case scenario for the rest of the contextual information, as it might contain some components which are not predictable. We will explore these possibilities in Section 3.8, for now, we just focus on existing approaches to time series forecasting.

Time series analysis and forecasting is a widely studied topic that has been applied to many applications such as business, stock market and exchange, weather, electricity demand, cost and usage of products such as fuels, electricity, etc., and in any kind of place that has specific seasonal or trendy changes with time [71]. As already said, being able to forecast the workload of an IT system is a key factor for proactive tuning [68]. As an example, [15] uses autoregressive integrated moving average (ARIMA) models to predict the workload of cloud systems.

The first forecasting methods were proposed in the 1950s (see [47, 48]). With the rise of big-data traditional techniques such as Exponential Smoothing and ARIMA have started to show their limitations [31], and recent developments have introduced deep learning models to exploit huge datasets and outperform traditional approaches [66].

Producing reliable forecasts is, nonetheless, still a challenging problem. Prophet [105] is another popular solution, mainly developed to solve two issues:

- completely automatic forecasting techniques can be hard to tune and are often too inflexible to incorporate useful assumptions or heuristic;
- the analysts responsible for data science tasks throughout an organization typically have deep domain expertise about the specific products or services that they support, but often do not have training in time series forecasting.

Hence, Prophet provides a configurable model with interpretable parameters that can be intuitively adjusted by analysts with domain knowledge about the time series. Prophet then produces a forecast for this model and evaluates forecast performance over historical data. When there are poor performance, Prophet signals the potential problem to a human analyst. Based on this feedback, the analyst can then inspect the forecast and potentially adjust the model.

The Prophet approach is especially useful in performance autotuning, as it allows to efficiently exploit the extremely valuable domain knowledge provided by performance experts, without requiring them to be trained in data science tasks.

In the experimental section, we also evaluate DeepAR, MQCNN and DeepState. DeepAR [93] provides probabilistic forecasts by using an autoregressive recurrent neural network. DeepState [86] is another probabilistic time-series forecaster that combines state space models with deep learning. MQCNN [111] combines quantile regression with both recurrent and convolutional neural networks.

CHAPTER 3

Proposed Approach: CC-GP

In this chapter, we start by formalising the configuration autotuning problem, and then we describe Bayesian Optimisation and its contextual extension. We then describe how we handle different constraints inside the optimisation, and, finally, we cover the context characterisation and forecasting component.

3.1 Problem Definition

The goal of our optimisation process is to find the configuration vector $\vec{x}$ in the configuration space X and apply it to the IT system so to optimise a certain performance indicator $y \in \mathbb{R}$. The applied configuration vector $\vec{x}$ has to satisfy some constraints.

According to the QRAK classification of [34] (summarised in Section 2.2.1) we consider the following constraints:

- Parameter constraints: they are defined over the optimisation space and simply involve parameters (e.g., `minimum_heap` $\leq$ `maximum_heap`), they classify as QUAK:
  - Being simple disequalities, they are Quantifiable.
  - Failing to respect a constraint prevents the application from starting; hence they are Unrelaxable. Arguably, one could introduce relaxable parameter constraints to help the optimiser move toward good regions. We do not treat these kind of constraints, but they can be added to the optimisation process with minimal effort.
  - Measuring their violation does not require running a performance test; hence they are measurable A Priori.
  - We suppose they are Known by the optimiser.

- Exploration constraints: when the autotuner works in a production environment, we do not want it to test random configurations, as would typically happen in the initial iterations without no useful information. Hence, we want the suggested configuration to change "slowly" from one iteration to the other. This does not give a theoretical guarantee that the autotuner will not destroy the system or severely impact performance. However, it is empirically true that a small change in the parameter usually leads to a small change in performance. Furthermore, performance engineers often see a small change as less psycologically disturbing. Nonetheless, this feature can lead the optimiser to get stuck in local minima and should thus be used carefully. These constraints can be satisfied by limiting the distance between the proposed configuration and the previously tested ones. These constraints are QRAK, as they are similar to the parameter ones but can be relaxed.

- Startup constraints: they capture the fact that the application fails to start, preventing us from running a performance test. This can either be caused by unknown parameter constraints or by unrelaxable metric constraints (e.g., the response time violates an SLA, and our canary instance is terminated). In any case, they are NUSH:
  - We can only detect a failure in the performance test execution without quantifying the violation.
  - We cannot relax them, as they prevent us from running simulations.
  - We must start a simulation to detect their violations as we do not know them.
  - By definition these constraints are the implicit constraints that the user does not know.

- Metric constraints: they are defined over interesting performance metrics (e.g., $\texttt{response_time} \leq 10ms$. They are QRSK:
  - The violation can be Quantified.
  - Failing to meet a constraint does not prevent us to measure other constraints or the performance score, hence they are Relaxable.
  - Measuring their violation requires running a performance test.
  - Only the user can give these constraints; hence they are Known.

For all the constraints, we only consider inequalities, as parameter equalities can be handled easily by rewriting the optimisation problem and metric equalities are generally useless as they would be too sensitive to measurement noise.

The goal of the optimisation is to select $\vec{x}$ by taking into account the particular workload to which the system is exposed and any other external factor that might impact on the selected configuration (e.g., data center temperature, response time of an external service, etc.). We call all these external factors (including the workload) *context* and denote it as $\vec{c} \in C$ where C is the space of the possible contexts and $\vec{c}$ is a description of the context. In fact, the workload is not different from any other contextual information, as the key aspect of a contextual metric is that we can measure it, and it affects the performance of the SUT, but we cannot control it. As an example, we can derive $\vec{c}$ by

using any workload characterisation algorithm and then appending the measurement of the remaining contextual metrics.

The satisfiability of parameter constraints does not vary with the context, but it does for the metric ones. y can be any measurable property of the system (like throughput, response time, memory consumption or a combination thereof). Since the context evolves over time, we want to update the suggested configuration as well. At time t, the optimisation algorithm will suggest a candidate configuration $\vec{x}_t$ which is tailored to the current context $\vec{c}_t$. Applying $\vec{x}_t$ to the system under context $\vec{c}_t$ results in a performance measurement y_t and in a set of M constraints violations $v_l(\vec{x}_t, \vec{c}_t) \forall l = 1, \ldots, M$, which include all the families of constraints defined above. Formally, assuming a minimisation problem we can write the optimisation problem for the current context $\vec{c}$ as follows:

$$\min_{\vec{x}} \quad y = f(\vec{x}, \vec{c}) \tag{3.1a}$$

$$\text{subject to } v_l(\vec{x}, \vec{c}) < 0 \forall l = 1, \ldots, M \tag{3.1b}$$

where f, v indicate, respectively, the performance indicator and the constraint violations.

The tuning process works as depicted in Figure 3.1, and it is repeated iteratively as time passes. We start ① by measuring and characterising the initial context $\vec{c}_0$. Then, we feed this information to the tuner, which has access to the knowledge base (KB) of previous experiments ② (which is initially empty) and uses both the KB and the current workload to suggest a candidate configuration vector $\vec{x}_0$. The tuner takes care of respecting all the given constraints. We apply the configuration to the System Under Test (SUT) ③ and run a performance test, exposing the SUT to the externally controlled context. The monitoring platform collects some performance metrics ④, and uses them to derive the context characterisation $\vec{c}_i$, the performance score y_i and the constraints violations, aggregated in a constraint violation vector $\vec{v}_0 = (v_0(\vec{x_0}), \ldots, v_M(\vec{x_0}))$ for ease of notation.

At this point, we have concluded the first iteration of the tuning process, and we store the obtained information in the KB ⑤. Now we iterate the process: we measure the new context $\vec{c}_1$ ① and consider the previous result ② to obtain the new configuration $\vec{x}_1$ ③, which results in performance y_1 ④ and violations $\vec{v}_1(\vec{x}_1, \vec{c}_1)$. Notice that, at iteration i, the tuner can exploit all the information of previous iterations: $(\vec{x}_{0,\ldots,i-1}, \vec{c}_{0,\ldots,i-1}, y_{0,\ldots,i-1}, \vec{v}_{0,\ldots,i-1})$. However, no other knowledge is required.

We now cover in detail the various components of the tuning process.

3.1.1 Space Normalisation

The employed optimisation model works on real-valued spaces $\mathbb{R}^d$. However, this is not the common case in performance tuning, as most of the tunable parameters we have to model are either integer or categorical values. Hence, the configuration $\vec{x}$ is transformed in $\bar{x}$ by separately normalising each dimension according to the following rules:

- a float parameter x_d with an explorable domain of $[d_{min}, d_{max}]$ is scaled in a $[0, 1]$ dimension by computing: $\bar{x_d} = (x_d - d_{min})/(d_{max} - d_{min})$.

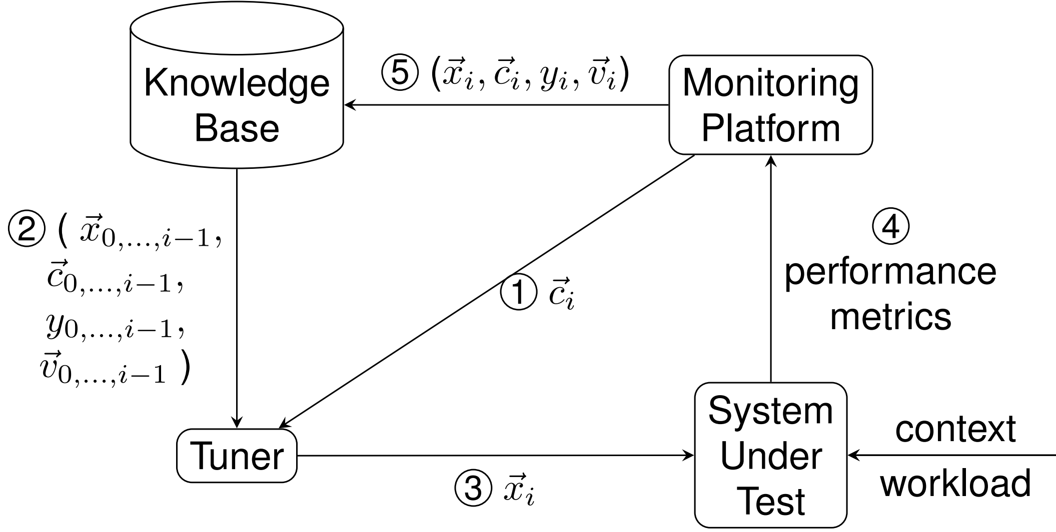


Figure 3.1: *Tuning process and architecture.*

- An integer parameter x_d with an explorable domain of $[d_{min}, d_{max}]$ is scaled in a $[0, 1]$ dimension by computing: $\bar{x_d} = (x_d - d_{min})/(d_{max} - d_{min})$.
- A categorical parameter x_d with explorable categories $(d_1, d_2, \dots, d_n)$ is transformed in n dimensions with a $[0, 1]$ domain with a one-hot encoding. That is, a normalised dimension is created for every possible category, and a point is transformed by setting to 1 the value of its category, and to 0 all the others.

Conversely, a transformed point $\bar{x}$ is de-normalised to $\vec{x}$ following the reverse process:

- a real dimension x_d is obtained by computing $x_d = \bar{x_d}(d_{max} - d_{min}) + d_{min}$ from its normalised version $\bar{x_d}$.
- An integer dimension x_d is obtained by computing $x_d = \text{round}(\bar{x_d}(d_{max} - d_{min}) + d_{min})$ from its normalised version $\bar{x_d}$.
- A categorical parameter x_d is obtained by selecting the category with the maximum value in the normalised dimensions.

Categorical parameters are the most problematic ones, as they get transformed into multiple normalised dimensions. This has two significant drawbacks: it enlarges the search space dimensions and, more importantly, it hides the real nature of the problem to the optimiser. As a simple example, consider a situation where we optimise a categorical parameter with two possible values a and b. We perform two experiments and obtain two values associated with the two categories. The optimiser, however, sees two dimensions, and the two evaluated points are normalised as $(1, 0), (0, 1)$. Hence, the optimiser might try to suggest the configuration $(0.4, 0.6)$, when, in reality, we have already completely explored the search space. This is especially relevant in the employed bayesian optimisation model, which, as we will show in the following, is biased towards the exploration of uncertain regions, exactly like $(0.4, 0.6)$. Hence, extra care will be taken in certain sections of the optimisation process to de-normalise and re-normalise

categorical dimensions so to force the optimiser to work with coherent values. Following the example above, $(0.4, 0.6)$ would be de-normalised in the category b and then re-normalised as $(0, 1)$, which is an already explored configuration.

3.2 Bayesian Optimisation

Bayesian Optimisation (BO) is a powerful tool that has gained great popularity in recent years. An extensive review of BO can be found in [97], here we give just a brief introduction, visually summarised in Figure 3.2.

Formally we want to optimise an unknown objective function f:

$$\vec{x}^* = \arg\min_{\vec{x} \in X} f(\vec{x}) \tag{3.2}$$

where f has no simple closed-form but can be evaluated at any arbitrary query point $\vec{x}$ in the domain. The evaluation produces noisy observations $y \in \mathbb{R}, y = f(\vec{x}) + \epsilon$, and usually is quite expensive to perform. Our goal is to converge toward good points to optimise f quickly. Moreover, we want to avoid evaluating points that lead to bad function values. In performance optimisation terms, we want to find a configuration that optimises a specific performance indicator, and simultaneously (1) explore the configuration space to gather knowledge, and (2) exploit the gathered knowledge to converge toward well-performing configurations quickly.

Notice that Equation (3.2) is equivalent to the autotuning problem that we exposed at the beginning of the section, except for the context-dependence and constraints, which we are not considering for now.

BO is a sequential model-based approach for solving the optimisation problem of Equation (3.2). Essentially, we create a surrogate model of f and sequentially refine it as more data are observed. Using this model, we iteratively compute the value of an acquisition function α_i, which is used to select the next point $\vec{x}_i$ to evaluate. Intuitively, the acquisition function evaluates the utility of candidate points for the next evaluation of f by trading off the exploration of uncertain regions with the exploitation of promising regions. As the acquisition function is analytically derived from the surrogate model, optimising it is straightforward.

Thus, BO has two key ingredients: the surrogate model and the acquisition function.

We focus on Gaussian Processes (GPs) as surrogate models, which is a popular choice in BO [112, 97, 87]. Notice, however, that the proposed approach can be easily adapted to use other surrogate models, such as Random Forest [56] or Neural Networks [100].

For the acquisition function we follow the GP-Hedge approach presented in [52], which, instead of focusing on a specific acquisition function, adopts a portfolio of acquisition functions governed by an online multi-armed bandit strategy, as explained in Section 3.2.1. The idea is to compute many different acquisition functions at each tuning iteration, and progressively select the best one according to previous performance. The GP-Hedge approach and the included acquisition functions are detailed in Section 3.2.1.

A GP(μ_0, k) is a nonparametric model fully characterised by its prior mean function $\mu_0 : \bar{X} \to \mathbb{R}$ and its positive-definite kernel or covariance function, $k : \bar{X} \times \bar{X} \to \mathbb{R}$. The GP works on normalised configuration vectors $\bar{x}$. Let $D_i = \{(\bar{x}_n, y_n)\}_{n=0}^{i-1}$ be

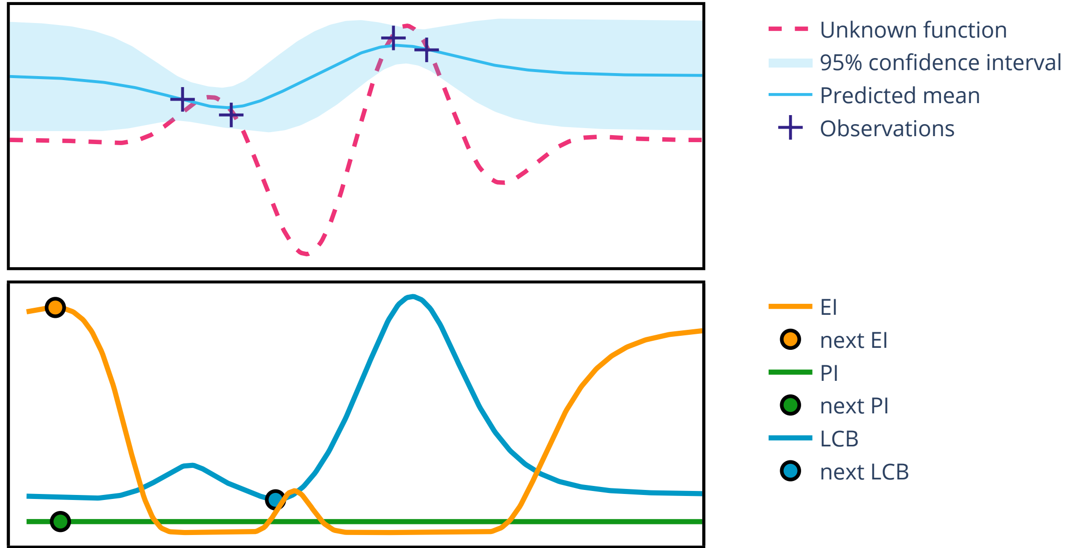


Figure 3.2: *Bayesian Optimisation. We want to minimise the unknown function and have already observed four points, which we use to compute the GP predicted mean and uncertainty. The acquisition functions combine the two values to select the next point to evaluate. Different Acquisition functions select different points.*

the set of past observations and $\bar{x}$ an arbitrary test point. The random variable $f(\bar{x})$ conditioned on observations D_i follows a normal distribution with a mean and variance functions that depend on the prior mean function and the observed data through the kernel.

Essentially, the GP is a regression model that is very good at predicting the expected value of a point and the uncertainty over that prediction. In BO, the acquisition function combines the two quantities to drive the search process. The GP prediction depends on the prior function and the observed data through the kernel. The prior function controls the GP prediction in unobserved regions and, thus, can be used to incorporate our prior beliefs about the optimisation problem. The kernel, instead, controls how much the observed points affect the prediction in nearby regions and, thus, governs the predictions in the already explored regions. In our approach, we have mostly employed a Matérn 5/2 kernel, which is the most widely used class of kernels in literature [97].

3.2.1 Acquisition Functions

The acquisition function α_i is derived from the computed model, combining the prediction with the uncertainty to select the next point to evaluate trading off exploration and exploitation. Clearly, different acquisition functions have different properties, and many different ones have been proposed in the literature, here we only report the ones which we use in our approach: Probability of Improvement (PI) [62], Expected Improvement (EI) [73], and Lower Confidence Bound (LCB). The three acquisition functions are visually represented in Figure 3.2 and described in the following. Optimising the acquisition function allows selecting the next query point $\bar{x}$, which is then denormalised to obtain the next configuration vector $\vec{x}$.

Probability of Improvement

Assuming a minimisation problem, we call τ the minimum score observed up to the current iteration i: $\tau = min_{j=0:i-1} y_j$.

Probability of Improvement (PI) measures the probability of a point $\bar{x}$ to improve upon τ. As we are measuring this probability using the posterior distribution ν of the Gaussian Process, which is a Gaussian distribution, we can easily compute this probability as:

$$\alpha_{PI}(\bar{x}, D_i) = P[\nu < \tau] = \Phi\left(\frac{\tau - \mu_i(\bar{x})}{\sigma_i(\bar{x})}\right) \tag{3.3}$$

where Φ is the standard normal cumulative distribution function. We then simply maximise α_{PI} to select the next point $\bar{x}_i$. According to PI, all improvements are treated equally, as PI simply accumulates the posterior probability mass below τ at $\bar{x}$. PI can perform really well, but in practice its performance depends heavily on the selected τ, usually leading to an overly exploitative behaviour [57, 97].

Expected Improvement

The Expected Improvement aims also to consider the entity of improvement we expect to achieve by sampling a candidate point. The Expected Improvement is defined as:

$$\alpha_{EI}(\bar{x}, D_i) = (\tau - \mu_i(\bar{x}))\Phi\left(\frac{\tau - \mu_i(\bar{x})}{\sigma_i(\bar{x})}\right) + \sigma_i(\bar{x})\phi\left(\frac{\tau - \mu_i(\bar{x})}{\sigma_i(\bar{x})}\right) \tag{3.4}$$

where ϕ is the standard normal probability density function. Differently from PI, it has been shown that using the current minimum value as τ works reasonably well [99].

Lower Confidence Bound

The Lower (Upper for maximisation) Confidence Bound has been a popular way to trade-off exploration and exploration [97]. The basic idea is to be optimistic in the face of uncertainty, and use the best case scenario predicted by the surrogate model. As the posterior distribution of a GP is Gaussian, any quantile of its distribution can be computed as:

$$\alpha_{LCB}(\bar{x}, D_i) = \mu_i(\bar{x}) - \beta_i \sigma_i(\bar{x}) \tag{3.5}$$

where β_i controls the selected quantile, and is selected according to theoretically motivated guidelines [103].

Constrained Acquisition Functions

Many optimisation problems are constrained, meaning that certain regions of the search space $\bar{X}$ are not valid, and we do not know in advance which points will result in a violation. Several approaches deal with this problem by modifying the acquisition function [97].

Clearly, depending on the actual problem, we can use different solutions, but the most basic idea is to use the weighted expected improvement [101, 49], where we simply multiply the EI by the probability of satisfying the constraints:

$$\alpha_{wEI}(\bar{x}) = \alpha_{EI}(\bar{x}, D_i) h(\bar{x}, D_i) \tag{3.6}$$

where $h(\bar{x}, D_i)$ is a GP with a Bernoulli observation process. We will use this acquisition function as a baseline to compare against in Chapter 7.

GP-Hedge

In reality, there is not a single acquisition function that is guaranteed to provide better performance for any possible optimisation problem, making it difficult to select which one to use. Furthermore, is not even guaranteed that, for a given problem, a single acquisition function will perform the best over the entire optimisation. To solve these issues, GP-Hedge [52] uses a portfolio of acquisition functions. At each iteration i each acquisition function proposes a candidate query point, and a multi-armed bandit is used to select among the candidate query points according to past performance.

3.3 Contextual Gaussian Process Bandit Optimisation

Our assumption so far has been that the performance f can be expressed as a function of the configuration $\bar{x}$ only. However, we have observed that the performance of an IT system also depends on others, uncontrolled, variables; such as the workload to which the application is exposed, or the response time of uncontrolled external services that impact the system. We called all these external factors *context* $\vec{c}$.

BO has been extended to handle such situations [59]. The key idea is that there are several correlated functions $f_{\vec{c}}(\bar{x})$ that we want to optimise. Essentially, the data from a context $\vec{c}$ can provide information about another context $\vec{c'}$. To capture this, we define a new kernel function that works over configuration and context pairs: $k((\bar{x}, \vec{c}), (\bar{x'}, \vec{c'}))$. This new kernel is formalised as the sum of two kernels defined over the configuration space X and the context space C, respectively:

$$k((\bar{x}, \vec{c}), (\bar{x'}, \vec{c'})) = k(\bar{x}, \bar{x'}) + k(\vec{c}, \vec{c'}). \tag{3.7}$$

The functions sampled from a GP with a covariance function as the one above have an additive form made of two components: $f = f_{\bar{x}} + f_{\vec{c}}$. The $f_{\vec{c}}$ component models overall trends among contexts, while the $f_{\bar{x}}$ models configuration-specific deviation from this trend. Similarly to what we did for the configuration kernel, we use a Matérn 5/2 kernel for the context kernel.

Essentially we are saying that we expect the performance of a certain configuration-context pair to correlate with the performance of nearby configurations and contexts. As the two kernels are combined using a sum operation, we consider two points similar when they have either a similar configuration or a similar context. By multiplying the kernels, we would instead consider two points as similar when they have both a similar configuration and a similar context. In other terms, by using this kernel we are enlarging the GP space with the context characterisation component, but, when optimising the AF, we only optimise the configuration subspace. What we obtain is a suggested configuration which is tailored for the current context, and this configuration is selected by leveraging all the configurations we have tested in the past, even in different contexts.

3.4 Constraint-Aware Optimisation

So far we have described the optimisation process assuming that the acquisition function can be optimised easily. When subject to constraints, however, this step becomes more complex. Existing approaches to constrained BO work by embedding the probability of satisfying the given constraints into the acquisition function [97, 10, 101, 49, 46]. The satisfaction probability is estimated in different ways starting from the data collected in previous iterations.

While valid as a general optimisation solution, this approach is suboptimal for the kind of constraints we identified in autotuning problems. As an example, quantifiable constraints (i.e., parameter and exploration) can be easily quantified, hence it makes no sense to estimate their satisfaction probability using an additional estimator. While, in principle, we could easily achieve this by constraining the search space over feasible regions, instead of simply constraining the acquisition function, this would prevent us from expressing an importance ordering over the constraints: that is, we prefer to violate a little bit the exploration constraint rather than testing a point which satisfies the exploration constraint, but we are absolutely sure will lead to a startup failure.

Likewise, QRSK constraints (i.e., quantifiable metric ones) should not be considered simply in terms of satisfaction probability, as the entity of the violation is of crucial importance. Consider the optimisation of a system with an SLA constraint on the response time, which must be below $10\,\mathrm{ms}$ and four candidate configurations with a measured response time of 5, 9.9, 10.1 and 20 ms, respectively. If we only consider constraint satisfaction, only the first two configurations are feasible. However, in reality, we have to consider the amount of the violation: it is clear that the fourth configuration is much worse than the third. Also, the second configuration is very close to the SLA; hence it is a very risky configuration. Finally, we need to allow for some noise in the measurements, and thus consider very similar configurations two and three. For this kind of constraint, a probability of violating the constraint of 50% could either mean that we have no idea whether the constraint will be satisfied or that we are sure that the configuration will yield a performance metric exactly equal to the given SLA.

Moreover, not all the constraints have the same importance, and, thus, we should not combine them into a single satisfaction probability. As an example, it is mandatory to respect parameter constraints (as they are unrelaxable), whereas we can let the tuner test a configuration that is expected to violate a metric constraint if we have no other choice. For these reasons, we propose a more refined approach to constraint handling, using different strategies for different constraints. We will show in Chapter 7 that our approach outperforms existing ones when dealing with configuration autotuning.

We now explain how we deal with each family of constraints separately. Before doing that, it is worth mentioning the two main strategies commonly used to find the point that optimises the AF. The first one, called *sampling*, consists in simply computing the AF value for a large number of randomly selected configurations and then pick the best one. The second one, called *optimisation* is built on top of the first one, and, after having selected a small nuber of good configurations, it optimises them using common optimisation algorithms like BFGS (Broyden-Fletcher-Godfarb-Shanno) or SLSQP (Sequential Least Squares Programming). Usually, they exploit the availability of the gradient on the AF to ease the optimisation. By performing some rounds of

iterative optimisation the second strategy can refine the solution without increasing the computational overhead too much. Usually, the second strategy is used after the first one, which is always present.

At a high-level view, our AF-optimisation strategy works as described in Algorithm 3.1. The basic idea is to compute or predict the parameter, exploration, startup and metric violations (as described below) and then look for candidate configurations that are predicted to be feasible. If it is not possible to find a feasible configuration, we look for configurations that minimise the violation. This is done in the sampling phase by favouring (in descending order) configurations that respect the parameter constraints (which cannot be relaxed), exploration constraints, startup constraints and, finally, metric constraints.

Notice that, before evaluating the constraints, we de-normalise and re-normalise the proposed configuration so to have proper values for the categorical parameters. Furthermore, after the AF-optimisation phase, we again de-normalise and re-normalise and check whether the optimisation actually improved the AF value while keeping the configuration feasible.

At the end of the sampling phase, we have a set of candidate configurations (either feasible with a good acquisition function or unfeasible with a minimised violation) which are given as input to the optimisation stage, which tries to optimise the acquisition function while respecting the constraints, or to minimise the constraint violation in case we could not find a feasible configuration in the sampling phase. We now cover each family of constraints in further detail, explaining how we compute the violation probability of a configuration.

3.4.1 QUAK Constraints - Parameters

Parameter constraints are Quantifiable, Unrelaxable, A priori and Known. Given a configuration it is easy to detect whether it is feasible, and we are not interested in maximising this feasibility, nor in minimising the violation as we have assumed these constraints to be unrelaxable.

Thus, during the sampling phase, we deal with these constraints by simply evaluating the constraints and discarding the configurations which are not feasible on at least one constraint. We have found this simple approach to work well with the constraints we used in our optimisation scenarios. However, when dealing with more complex constraints, which are harder to satisfy, it is straightforward to replace this simple procedure with a more refined constraint satisfaction solution [107].

Being unrelaxable and known a-priori, failing to find a feasible configuration would cause a failure in the tuning process, terminating the optimisation. We empirically avoid this by evaluating a large number of configurations in the sampling phase.

During the AF optimisation stage we use the Sequential Least Squares Programming (SLSQP) optimisation algorithm, which considers constraints. We introduce an optimisation constraint for each QUAK constraint, reformulating the disequalities to differences so to have the distance to the constraint boundary.

3.4.2 QRAK Constraints - Exploration

Exploration constraints are a psychological safety net useful when deploying the autotuner in live production environment. The basic idea is to limit the freedom of the

Algorithm 3.1: High-level view of the constraint satisfaction strategy. If we can find probably-feasible points, we look at their acquisition function, otherwise we try to minimise the constraint violation probability, by favouring (in decreasing order) exploration, startup and metric constraints. Parameter constraints cannot be relaxed, so we do not aim to minimise their violation.

input : $\bar{X}$ the normalised search space
input : vt the maximum violation threshold

Randomly select a set of candidate configurations in the search space $C \subseteq \bar{X}$;
Denormalise and re-normalise the candidate configurations $\bar{x} \rightarrow \vec{x} \rightarrow \bar{x}$;

Compute the parameter violation of each candidate configuration $PV(\bar{x})\ \forall \bar{x} \in C$;
Discard parameter-unfeasible configurations $D' = \bar{x} : PV(\bar{x}) > 0.5\ C = C \setminus D'$;

Compute the exploration violation of each candidate configuration $EV(\bar{x})\ \forall \bar{x} \in C$;
Discard exploration-unfeasible configurations $D'' = \bar{x} : EV(\bar{x}) > 0.5\ C = C \setminus D''$;

Compute the startup violation probability of each candidate configuration $SV(\bar{x})\ \forall \bar{x} \in C$;
Discard startup-unfeasible configurations $D''' = \bar{x} : SV(\bar{x}) > vt\ C = C \setminus D'''$;

Compute the metric violation probability of each candidate configuration $MV(\bar{x})\ \forall \bar{x} \in C$;
Discard metric-unfeasible configurations $D'''' = \bar{x} : MV(\bar{x}) > vt\ C = C \setminus D''''$;

```
if C ≠ ∅ then
|  optimise the AF of the probably feasible points: x̄_i = min_{x̄∈C} α_i(x̄) ;
else
|  if D'''' ≠ ∅ then
|  |  minimise the metric violation probability: x̄_i = min_{x̄∈D''''} MV(x̄) ;
|  else
|  |  if D''' ≠ ∅ then
|  |  |  minimise the startup violation probability: x̄_i = min_{x̄∈D'''} SV(x̄) ;
|  |  else
|  |  |  if D'' ≠ ∅ then
|  |  |  |  minimise the exploration violation: x̄_i = min_{x̄∈D'''} SV(x̄) ;
|  |  |  else
|  |  |  |  Randomly select a new set of candidate configurations C and restart;
|  |  |  end
|  |  end
|  end
end
```

autotuner and let it suggest configurations that are close to already evaluated ones.

Like parameter constraints, the fact of being known a-priori makes these constraints easy to handle. In the sampling phase, we consider all the previously evaluated configurations which did not result in a metric constraint violation or a startup failure X_{ok}. Then, for any candidate configuration $\bar{x}$, we compute its exploration score as the minimum L1-distance from the evaluated configurations measured in a normalised space:

$$v_{expl}(\bar{x}) = \min_{\bar{x}' \in X_{ok}} \|\bar{x} - \bar{x}'\|_1. \tag{3.8}$$

In this way, when dealing with a D-dimensional normalised space, the maximum L1-distance between two configurations is D. We can then divide v_{expl} by D to obtain a percentage of exploration, we can then compare with an exploration threshold defined by the user, and discard points that are too distant from the successfully explored region.

During the AF optimisation phase, we simply add a constraint to the SLSQP algorithm, measuring the difference between v_{expl}/D and the user defined threshold.

3.4.3 NUSH Constraints - Startup Failures

When the application fails to start we cannot run a performance test. In this case, we cannot measure the performance score nor the metric constraints and the associated context, but we know that the configuration is a very bad one. In some situations, we could still be able to measure the score and the context (e.g. when the score is a cost and the context is measured over an entire deployment, and we only tune a canary instance). However, a startup failure means that the configuration is bad for every context, so we simply ignore the context.

Differently from parameter and exploration constraints, startup failure constraints are not known a-priori, and hence are harder to satisfy during the optimisation, as we have to build a predictive model to understand which configurations to avoid. For these constraints, we fit a ν-SVR regressor [22, 94], which takes as input a normalised configuration vector (see Section 3.1.1 for details on the space normalisation process) and predicts the probability of a configuration leading to a failure. To do this, at each tuning iteration we train this regressor using past data, giving a probability of 0 to successfully evaluated configurations and a value of 1 to failed configurations. Notice that the regressor does not consider the contextual information.

As the evaluation of a configuration involves running a performance test, which is a noisy process, depending on the considered tuning scenario we have the possibility of obtaining both a failure and a successful execution when testing the same configuration multiple times. In this case, the regressor must identify which of the two outcomes represents an outlier. Since our goal is to build a safe autotuner, we prefer to err on the side of caution, and so we fit the regressor by increasing the weight of the failed experiments by a user-tunable factor.

During the AF sampling phase, we filter out candidate configurations whose prediction is too close to a user-defined feasibility threshold (typically 0.5). Likewise, during the AF optimisation phase, we add a constraint to the SLSQP algorithm so to remain in the probably-feasible region.

3.4.4 QRSK Constraints - Metric SLAs

Metric constraints are Quantifiable, Relaxable, Simulation-based and Known. Similarly to startup constraints, they are not known a-priori, and so we do not have a cheap way to evaluate them; hence we must use the data collected during the tuning process. However, differently from startup constraints, they are quantifiable as their violation values depend on the context and we can have more than one constraint.

When evaluating a configuration, we monitor the context, the performance score and all the metric constraints, saving the violation quantity for each constraint. At each tuning iteration we use the saved contexts and violations to create a regressor able to predict the violation of a configuration under a certain context for each constraint.

As different constraints have different scales for their violations, we normalise them separately so that each regressor predicts 0 for the safest observed point, 1 for the configuration with the highest violation and 0.5 for a configuration exactly on the constraint

boundary. Notice that, albeit the values are normalised in a $[0, 1]$ range, they do not represent a violation probability, but the distance to the constraint boundary (represented by the value 0.5).

For the constraint l, characterised by the (positive) violation quantities $v_l(\vec{x}_j, \vec{c}_j)$ (defined in Equation (3.1b)), which we collected in the past iterations $j = 0, \ldots, i-1$, we derive the normalised violations using a logistic function:

$$\bar{v}_l(\vec{x}_j, \vec{c}_j) = \frac{1}{1 + e^{\frac{-1}{M} v_l(\vec{x}_j, \vec{c}_j)}} \tag{3.9}$$

where M is the median positive violation if we observed any positive violation, otherwise is the median negative slack. Figure 3.3 gives a visual representation.

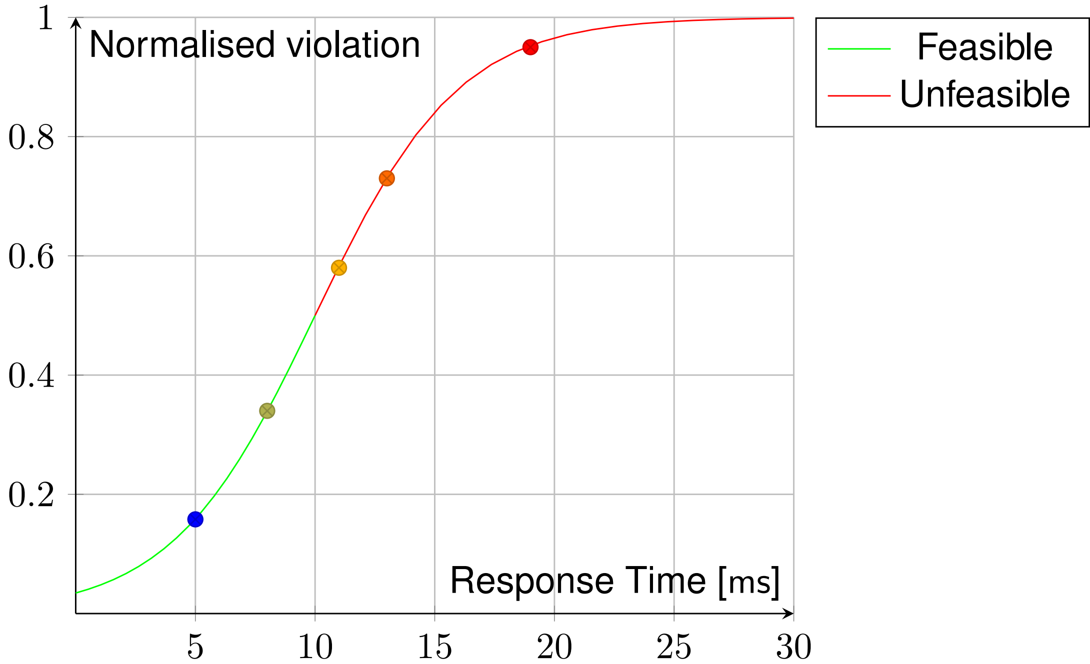


Figure 3.3: *Example of metric constraint normalisation. The constraint is response time below* 10 ms*, we have evaluated 5 configurations with response times of 5,8,11,13,19* ms*. The constraints violations are -5,-2,1,3,9 and the median positive violation M is 3.*

Similarly to the startup failure, we use a ν-SVR regressor to predict metric constraint violations $\tilde{v}$, and, again, we increase the weight of positive samples so that the model is more accurate when predicting positive violations and we obtain a safer autotuner.

We predict the violation of all the given constraints during the AF sampling phase, and select the maximum one. As the violations are normalised in a $[0, 1]$ range, one might be tempted to consider the product of the violations as the combined violation probability. However, we stress that the normalised violation is not intended to represent a probability, but a simple distance to the violation boundary. Hence, it is reasonable to consider the maximum normalised violation, as it represents the constraint that is closer to boundary while still being feasible (or the most severe violation).

We then consider a configuration probably feasible only if its maximum predicted violation is below 0.5. Notice that we can tune this threshold to favor a more exploratory or conservative behaviour of the autotuner.

During the AF optimisation phase, we introduce additional constraints to the SLSQP optimiser, formulated as differences over the prediction of each violation predictor and the feasibility threshold.

Notice that the metric constraint prediction process depends both on the configuration and the context. While the configuration part is obvious (i.e., the candidate configuration for which we are evaluating the feasibility) the context is not, and we will cover this part in Section 3.8.

3.4.5 Unsolvable Constraints

So far we focused primarily on satisfying the given constraints, as violating them would lead to severe consequences. For example, failing to respect a parameter constraint would prevent the application from starting, whereas violating a metric SLA constraint would tipically lead to economic penalties. Nontheless, it is possible that SLA constraints are set to be too restrictive and they cannot be respected.

In these situations, the proposed approach would try to minimise the violation of the constraint, without considering the score function. As visible from Algorithm 3.1, when no feasible configuration is found, we do not try to optimise the AF, but, instead, we minimise the violation of the non-satisfiable constraint. We choose to respect the constraints by favouring (in descending order) configurations that respect the parameter constraints (which cannot be relaxed), exploration constraints, startup constraints and, finally, metric constraints. This ordering can be modified depending on the specific preferences. Especially the exploration constraints can be made less relevant if the user desires.

Notice that, when respecting all the constraints is not possible, we do not consider the scoring function at all, nor do we employ any specific strategy to explore the configuration space quickly to find feasible regions, where we can then start to optimise the score. This is due to the fact that we suppose that the baseline configuration does indeed respect the given constraints, and so we can move around it to find better configurations. When this is not the case, and the baseline does violate some constraints, it might be interesting to modify the optimisation strategy to create a two-steps optimisation, where first we minimise the violation (using it as a scoring function to take advantage of the bayesian optimisation approach), and then we move to optimising the desired scoring function, starting from a feasible configuration.

3.5 Context-based Normalisation for Relevant Priors

Bayesian Optimisation is, as the name suggests, a Bayesian technique. Nonetheless, it is often used just as an optimisation technique, without paying the Bayesian components's proper attention.

As we explained above, BO works by iteratively suggesting points that optimise an acquisition function, which is computed starting from the value predicted by a regression model and from the prediction uncertainty. However, the regression model is a Gaussian Process, which derives its predictions (or posterior distribution) by combining the observed values with a prior distribution.

In other terms, when we ask the GP to predict the value of a configuration that is very different from any previously observed one, it will resort to the prior distribution.

In most implementations, the prior distribution is a zero mean, unitary variance normal distribution. If we are trying to maximise the throughput of an IT system, the GP will predict that any unknown configuration will likely destroy performance. This impacts heavily on BO, which will avoid the exploration of uncertain regions, thus remaining stuck on the initial configuration. Similarly, if we want to minimise the response time, the GP will suggest that any randomly selected configuration will probably have a response time equal to zero. This would result in an even worse optimisation experience, with the optimiser jumping randomly over configurations.

To easily obtain a relevant prior distribution, it is common to standardise the observed data. The majority of the available BO implementations start by evaluating a small set of randomly selected configurations, and use the collected information to initialise the GP and standardise future data. In this way, the GP will reasonably predict that, by picking a random configuration, we will likely obtain a performance value equal to the average value that we obtained by evaluating some randomly selected configurations.

When taking into consideration different contexts, however, it becomes crucial to standardise each point by taking into account the relevant context, and not just the initial one observed during the initialisation phase. The extent to which the context affects the performance score clearly depends on the specific situation. As a first example, consider a tuning session where we desire to reduce the response time of an application exposed to huge variations in the number of connected users. In this situation, especially if we are working near to the saturation point of the application, the context (i.e., number of users) has a great effect on the performance score (i.e., the response time). Thus, by clustering together similar contexts and normalising accordingly the response time, we allow the Gaussian Process to model only the effect of the parameters, without loosing the majority of its expressive power to model the effect of the context.

When dealing with a real-valued context characterisation vector, however, we must cluster together similar characterisation vectors, to derive a group of similar contexts that we can use to standardise values. We detail in Section 3.6 out clustering approach, for now just assume that we can map a context characterisation $\vec{c}$ to a cluster c.

To normalise performance values, we use a modified version of the Normalised Performance Improvement (NPI) [6]:

$$\tilde{NPI}(\vec{x}, \vec{c}) = \frac{\bar{f}(c) - f(\vec{x}, \vec{c})}{\bar{f}(c) - f^{+}(c)} \tag{3.10}$$

where $\vec{x}$ is the configuration we are evaluating, $\vec{c}$ is the context we measured when we tested $\vec{x}$ and $f(\vec{x}, \vec{c})$ is the corresponding performance measurement. We then call c the cluster of contexts to which $\vec{c}$ is assigned, and consequently compute $\bar{f}(c), f^{+}(c)$ as the average and best (respectively) performance measurements we observed across all the configurations we tested that were assigned to cluster c. Clearly, as we go on with the optimisation, we modify the clustering decisions and also the average and best values, hence we need to re-normalise past values at each iteration. Therefore, $\tilde{NPI}(\vec{x}, \vec{c})$ measures the optimality of the configuration $\vec{x}$ for the context $\vec{c}$. A $\tilde{NPI}$ of 0 means that, under context $\vec{c}$, the configuration $\vec{x}$ is performing like the average of the configurations we observed so far on c, whereas a $\tilde{NPI}$ of 1 means that configuration $\vec{x}$ is the best one that we have found so far for context c and a negative $\tilde{NPI}$ indicates that

the configurations is performing worse than the baseline. As we are not really interested in carefully modelling bad-performing configurations, we cap $\tilde{NPI}$ to -1, so that the Gaussian Process receives values in the $[-1, 1]$ range, retaining most of its modelling capability for the well-performing configurations.

The drawback of this approach is that, over time, the optimiser will converge toward better configurations, hence $\bar{f}(c)$ will move closer to $f^+(c)$ for any given context c. This will force the BO tuner to favour an explorative behaviour, as the prior will get closer to the value of good configurations. In practice, we have not observed this issue to cause any problem. This could be due to the fact that, when the optimiser has converged to good regions and $\bar{f}(c), f^+(c)$ begin to get closer, we have already created a good surrogate model, and the optimiser is not interested in exploring anymore. In any case, this can be easily solved by limiting the number of configurations over which we compute the average, to limit the computation to the initial iterations of the optimisation process, where we are closer to a random exploration. Otherwise, when possible, we could introduce a random exploration phase just to initialise this component.

3.5.1 Availability of a Control Group

In other situations (such as canary deployment), it could be possible to jointly evaluate $f(\vec{x}, \vec{c}), f(\vec{x}_0, \vec{c})$, where $\vec{x}_0$ is the baseline (i.e., vendor default) configuration, usually evaluated on a control group. As an example, we could be controlling a small subset of the production environment and still be able to measure the performance of the rest of the deployment, which is still running with the baseline configuration. If this is the case, we can modify the normalisation strategy to:

$$\tilde{NPI}(\vec{x}, \vec{c}) = \frac{\bar{f}(\vec{x}_0, c) - f(\vec{x}, \vec{c})}{\bar{f}(\vec{x}_0, c) - f^+(c)} \tag{3.11}$$

where $\bar{f}(\vec{x}_0, c)$ is the average performance score obtained by the control group in the iterations that were assigned to cluster c. Using this normalisation we avoid the issue of $\bar{f}(c)$ drifting towards $f^+(c)$.

3.6 Context Clustering

In Section 3.5 we have shown how we normalise the performance score of different iterations that are assigned to the same context cluster. Here, we show how we compute context cluster and how we decide whether the clustering step is actually beneficial to the optmisation process. In fact, in the previous section, we have considered the example of tuning the response time of an application subject to a varying workload. Consider now the case where we want to minimise the deployment cost of that same application. In this situation, cluster-dependant normalisation is not beneficial since our performance score (i.e., the deployment cost) is not a function of the context (i.e., the number of users). Hence, we have no benefit in separately normalising the various scores. Actually, normalising them all together allows to better estimate the average and optimum scores $\bar{f}, f^+$. When the autotuning process detects that clustering is not beneficial, we simply assign all the iterations to a single cluster and then compute $\tilde{NPI}$.

Here we have made some simple examples, where it is clear whether the context actually has a major impact on the performance score. However, this has to be detected

automatically by the algorithm, even in more complex situations. Remember that the goal of the clustering procedure is to separately normalise performance scores that are severely different due to context variations, so to take away the effect of the context and retain the expressive power of the regression model to capture the effect of the parameter. Hence, we need to understand whether the context has a major impact over the performance score. To do this, we have observed that a simple Linear Regression is sufficient.

To determine whether we should apply clustering, we thus fit two simple linear models. The first one models the non-normalised performance score starting from the configurations: $L_1 : \vec{x} \to y$. The second one maps from the contexts to the non-normalised performance score: $L_2 : \vec{c} \to y$. We then compute the coefficient of determination of the two regressors, and if the second one is higher than the first one we enable the clustering step.

For the clustering step, we fit a Gaussian Mixture Model [11] with the expectation-maximization (EM) algorithm, using the implementation provided by scikit-learn [83]. Gaussian Mixture models are very fast to compute, which helps to reduce the overhead of the autotuning algorithm, and are not biased toward any specific structure. To select the number of components to model (i.e., clusters) and the type of covariance matrix, we use the Bayesian information criteria (BIC) [95], varying the number of component from one to a tenth of the observed cluster vectors.

The pseudocode of the clustering step is reported in Algorithm 3.2.

3.7 Context Forecasting

As we said above, in order to select the next configuration to evaluate we need to evaluate its feasibility according to the constraints predictor and then the acquisition function, both the components are a function of the configuration and the context. Thus, to select the next configuration, we need to know the value of the context. In other terms, the Bayesian Optimiser needs to know the context to optimise for, and the constraint satisfaction component needs to know for which context to evaluate the feasibility. Notice, however, that in principle the two contexts can be different, and we will explore this possibility in Section 3.8. Here we focus on the simpler scenario where the context used to evaluate the feasibility of a configuration is the same one used to optimise the AF, and, more specifically, we try to predict the value that the context will have in the upcoming iteration (i.e., the context to which the system will be exposed when we will evaluate the configuration that we are selecting). Formally, at iteration i we need to consider past contexts $\vec{c}_0, \ldots, \vec{c}_{i-1}$ and predict the value of the next context $\vec{c}_i$. As each component of the context vector represents a time series, this is essentially a multivariate time series forecasting problem, which can be tackled in many different ways. As said in Section 2.3.2, we use Prophet [105], as it is easy to expose its predictions to human performance experts and adjust the model according to their feedback. Nonetheless, it is straightforward to substitute this component with other predictive models if desired [74].

The Prophet predictor, once fitted, produces the same predictions independently from new data. Thus, the forecast module is run independently from the optimiser and updated periodically to take into account new-data and feedback from human experts.

Algorithm 3.2: Pseudocode of the contextual clustering step. We first decide whether to actually cluster data by measuring the coefficient of determination of a two linear regressor modelling the score starting from the configuration or the context. Then, we fit a Gaussian Mixture Model using Bayesian Information Criteria (BIC) to select the number of components (i.e., clusters) and the type of covariance matrix.

```
input  : i: the index of the current tuning iteration
input  : X = {x⃗_j}_{j=0}^{i−1}: the set of explored configurations
input  : C = {c⃗_j}_{j=0}^{i−1}: the set of observed contexts
input  : Y = {y_j}_{j=0}^{i−1}: the set of measured scored (non-normalised)
output: l⃗ = (l_0, ..., l_{i−1}): vector of assigned clusters

L_1 = LinearRegressor().fit(X, Y);
L_2 = LinearRegressor().fit(C, Y);
if L_1.score(X, Y) > L_2.score(C, Y) then
    l⃗ = [0] ∗ i;
else
    lowest_bic = +∞;
    for cv_type in ('spherical', 'tied', 'diagonal', 'full') do
        for n_components ← 1 to len(unique(C))/10 do
            gmm = GaussianMixture(cv_type, n_components);
            gmm.fit(C);
            bic = gmm.bic(C);
            if bic < lowest_bic then
                lowest_bic = bic;
                best_gmm = gmm;
            end
        end
    end
    l⃗ = best_gmm.predict(C);
end
```

In the meanwhile, at each tuning iteration the optimisation module calls the forecasting module to receive a context prediction.

Notice that, up to this point, we have talked about the context vector $\vec{c_i}$, assuming that each of its component $c_{i,j}$ is a real number representing a certain property of the working condition to which the system is exposed during the performance test, such as the number of incoming requests. In reality, the performance test has a certain duration (e.g., half an hour), and inside this time window the working conditions vary continuously. Hence, the context vector $\vec{c}$ used in the optimisation module typically represents an aggregate value, such as the average over the entire test. Conversely, the forecast component typically works with a finer grain (e.g., 30 s), which allows achieving higher prediction accuracy. When the forecast is given to the optimisation module, the predicted values are aggregated into a single value following the same strategy used to collect $\vec{c_i}$ from the real system.

To bootstrap the forecasting module, before starting the tuning process, we keep the baseline configuration and collect context data, initialise the forecaster and manually inspect the proposed predictions. Typically, this phase lasts for one week as it is the usual time scale to observe a significant variety of contexts (especially workloads). However, this is strictly domain-dependant, and expert knowledge of the target system

is required to evaluate an appropriate initialisation duration and correct the forecast output. This initial observation period is also used to collect information regarding the baseline, allowing us to also initialise the constraint violation regressors and evaluate the noise level.

After the initial collection period, we start the online tuning process. Thus, at time t_1, the forecaster predicts (with a fine resolution) the upcoming contexts $\tilde{c}_{t_1:t_2}$, where $t_2 - t_1$ represents the duration of an entire performance test. Looking at this fine-grained predictions, the scheduling component (explained in Section 3.7.1) decides whether the predicted context is *stable* and a test can be launched. In the negative case, we wait for the context prediction to become stable. In the positive case, we take the average over time of the fine-grained prediction and give it as a context to the optimisation module, apply the suggested configuration and start a performance test. As the test runs, we monitor the actual context looking for two bad situations: an unstable context and an unexpected context.

If the actual context is unstable (whereas we predicted a stable context), we terminate the test and discard the result, as we would not be able to obtain reliable averages of the results to give to the tuning module. If instead the context is actually stable, but its average value is different from the forecasted one, it means that the applied configuration is tailored for the wrong context. In this case, we let the test continue as the collected information is still useful to the tuner.

3.7.1 Experiment Scheduling

As said above, we must run performance tests when the context is stable to obtain reliable performance evaluations. Therefore, given a fine-grained context forecast $\tilde{c}_{t_1:t_2}$ we need to tell whether it represents a stable window. As the context $\vec{c}$ (and its prediction $\tilde{c}$) is composed of multiple time series, we define a window as stable only if all the time series are stable. Clearly, the definition of a "stable" window strongly depends on the specific time series, system under test and on the actual environment where we are executing the tests. Hence, the stability function S employed to identify stable windows must be parametric and tunable by a performance expert. To this end, after the initial collection period, when the human operator adjusts the forecasts, we also propose him a visual representation of the stability function, highlighting the windows that would have been identified as stable over the startup period. The operator can then modify the stability function so that it matches its own idea of stability (i.e. when we can run a performance test and collect valid data). It is important that the proposed stability functions are easy to understand and modify to ease the operator's job.

We tested the following two stability functions, where $S_{cv}(\tilde{c}_{t_1:t_2}, \vec{c}_{t_0:t_1})$ is the stability function returning 1 in case the window $t_1 : t_2$ is considered stable, Θ contains the tunable parameters which should be selected by the operator and $\vec{c}_{t_0:t_1}$ is the past (actual) context:

- Coefficient of Variation: a window is stable if its CV does not exceed a threshold Θ. Past values are not used.

$$S_{cv}(\tilde{c}_{t_1:t_2}) = \begin{cases} 1 \text{ if } \frac{\sigma(\tilde{c}_{t_1:t_2})}{\mu(\tilde{c}_{t_1:t_2})} > \Theta \\ 0 \text{ otherwise.} \end{cases}$$

- Min-Max: a window is considered stable if all its values are contained in a range that is a fraction Θ of the range observed over the past values.

$$S_{mm}(\tilde{c}_{t_1:t_2}) = \begin{cases} 1 \text{ if } \max \tilde{c}_{t_1:t_2} - \min \tilde{c}_{t_1:t_2} \leq \Theta(\max \vec{c}_{t_0:t_1} - \min \vec{c}_{t_0:t_1}) \\ 0 \text{ otherwise.} \end{cases}$$

Finally, notice that the stability function can be used both to predict whether the upcoming window is stable and we should schedule an experiment, and after the experiment to check whether the past window was actually stable, and we can trust the obtained result.

3.8 Safety

The context forecasting module provides the optimiser module with a prediction of the upcoming context, which can then be used to select the next configuration, expected to satisfy the constraints and optimise the AF in the given context. This forecast, however, can be wrong, and an error in the context prediction can lead to an inaccurate estimation of the metric constraint violations.

As an example, consider a situation where we are minimising the deployment cost of an application with a certain SLA constraints on the response time, and the response time of our application is severely dependent on the response time of a third-party service. We can model the response time of this external service as a context for our application. However, forecasting the response time of the external service and suggesting a strictly-tailored configuration might not be our smartest choice. In fact, we could try to suggest a configuration that will respect the SLA for any possible response time of the external service. This might lead to a higher deployment cost, but we will be safer w.r.t. the SLA.

To mitigate this problem, we thus provide two working modes for the optimiser module: *local* and *global* safety.

3.8.1 Local Safety

In local safety, the forecasted context is used both to predict the constraint violations of the candidate configurations and their acquisition function. In other terms, we check that the suggested configuration does not violate the constraints only for the predicted upcoming context.

3.8.2 Global Safety

Differently from local safety, in global safety we evaluate the metric constraint violations of a configuration w.r.t. all the previously observed contexts, ignoring the forecasted one. Notice that metric constraints are the only ones that depend on context; hence global safety has an effect only on them.

At iteration i, to predict the feasibility of a candidate configuration $\vec{x}$, we use the violation predictor $\tilde{v}$ not to predict the violation over the next forecasted context $\tilde{v}(\vec{x}, \tilde{\vec{c}}_i)$, but, instead, we take the maximum violation over all the previously observed workloads:

$$\tilde{v}(\vec{x}) = \max_{j=0,\ldots,i-1} \tilde{v}(\vec{x}, \vec{c}_j). \tag{3.12}$$

This assures that, even if the context forecast is wrong, the tuner still suggests a configuration that is expected to satisfy the given constraints even in the real context.

CHAPTER 4

Experimental Evaluation Framework

This chapter describes the approach we use to evaluate the proposed approach. We start by covering the tuning scenarios we consider, in terms of the target applications, environments and workloads. Then, we describe our evaluation framework with the metrics we use to evaluate tuning performance and minimise experimental noise.

Our goal is to build a generic, holistic, contextual and goal-driven autotuner. Hence, we need to test it across a variety of IT systems, considering different stack layers, exposing the system to different workloads and tuning multiple goal metrics. Each tuning session is composed of many tuning iterations, which consist in applying a configuration and running a performance test. Unfortunately, running a performance test is quite costly and time-expensive, which limits the number of tests that we can do.

Furthermore, a performance test usually leads to noisy measurements, and this can be problematic to evaluate reliably the performance of an autotuner. Suppose, as an example, that we are comparing two autotuners that in reality are exactly the same algorithm and suggest the same configuration. However, when we go to test the suggested configuration, running two performance tests, in one test we obtain a better result due to random noise. If we were to simply compare the performance score obtained by the two tuners we would conclude that one is better than the other, which in reality is wrong. Furthermore, comparing the non-normalised performance of the best configurations found would not take into account two other important aspects: the number of tuning iterations required to find the best configurations, and how many bad configurations the tuner has tested. For example, even a random tuner finds the optimal configuration if given infinite time.

The problem still exists even if we try to devise an evaluation metric that captures the quality of the entire tuning process, and not simply of the best-performing configuration suggested. As an example, consider an autotuner that, at the first iteration suggests (by

chance) the global optimum configuration. However, we run the performance test in a noisy environment and obtain a bad performance score. This clearly impacts the entire tuning process, as the autotuner will then try to stay away from that bad-performing configuration.

To minimise as much as possible the measurement noise we thus need to repeat multiple times the entire tuning session, which increases, even more, the experimental cost. Thus, we run tests with different levels of realism. We start by testing the algorithms on analytic functions, which are very different from real IT systems but are extremely cheap to evaluate. We then move to run the test on a machine-learning simulation of a real system. This involves collecting a dataset containing the performance measurement of a real application and training a regression model, which can then be used to test the algorithms. We also empirically evaluated the proposed autotuner on real, live systems. These results, however, are not reported in this work as they are not replicable.

4.1 Analytical Function

We mainly use the Branin analytical function as a starting optimisation target to illustrate the metric computation process and some basic properties of Bayesian Optimisation. Given that computing a function value is extremely cheap, especially when compared to running a performance test, analytical functions are often used as a target for black-box optimisation. On the other hand, we cannot guarantee that the selected functions are somehow similar to real tunable systems, and, hence, they cannot fully replicate all the details of a real system.

The Branin (or Branin-Hoo) function is a common benchmark for optimisation techniques that is defined over $\mathbb{R}^2$. It is defined as:

$$f(x_1, x_2) = a(x_2 - bx_1^2 + cx_1 - r)^2 + s(1-t)\cos(x_1) + s \tag{4.1}$$

and the commonly used values are $a = 1, b = 5.1/(4\pi^2), c = 5/\pi, r = 6, s = 10, t = 1/(8\pi)$. The function is usually evaluated on the square $x_1 \in [-5, 10], x_2 \in [0, 15]$. The function has three global minima, with value $f(x^*) = 0.397887$ located at:

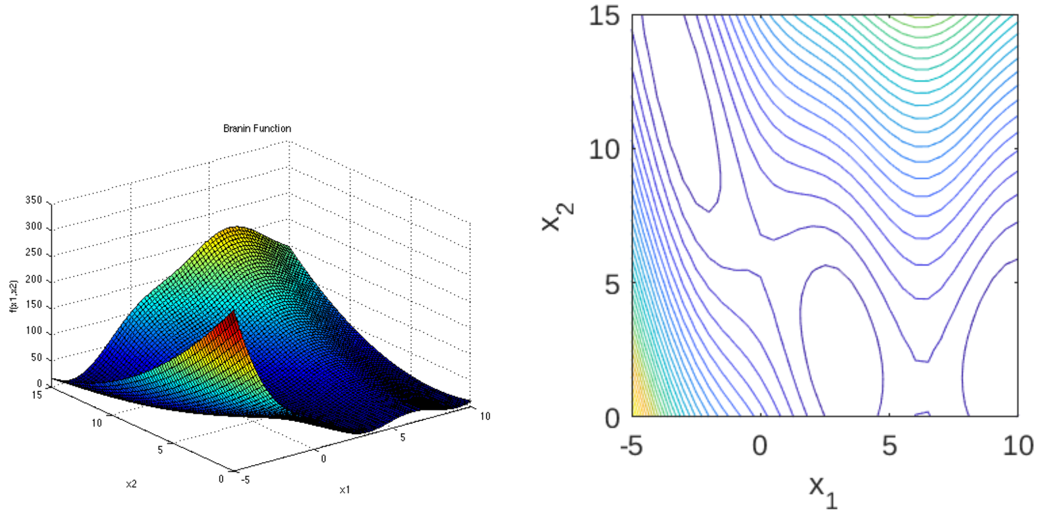
$$x^* = (-\pi, 12.275), (\pi, 2.275), (9.42478, 2.475). \tag{4.2}$$

Figure 4.1 provides visual representations of the Branin function.

4.2 Dataset-based Simulations

By dataset-based simulations, we mean creating a machine-learning regression model able to predict a certain performance indicator of a system from the applied configuration. To build these regressors, we first need to run an extensive set of experiments exploring the configuration spaces and collecting a wide variety of performance metrics. We then use the collected data to build a regressor to predict the performance of all the possible configurations in the search space.

By doing this, we build a model of the IT system performance, which is then used as the target of the tuning process for the algorithm evaluation. In this way, we make sure that our experiments are easily reproducible. Moreover, we can ensure that different tuning algorithms work precisely on the same system and exclude noise from



(a) *Branin function surface. Credit:* `https://www.sfu.ca/~ssurjano/branin.html`. **(b)** *Branin function contour. Credit:* `https://uqworld.org/t/branin-function/53`.

Figure 4.1: *Surface and contour plots of the Branin function.*

the evaluation. Furthermore, we can evaluate the variability of the tuning algorithms by repeating multiple times the tuning process without having actually to run the performance test. Notice that this step is not a requirement of the tuning algorithm, but instead is just an additional step that we perform to conduct a reproducible evaluation.

As target applications we selected the MongoDB[1] 4.0.3 and Cassandra [63][2] 3.11.4 DBMSs and JPetstore[3] 6, which is a reference Java eCommerce web application. For the two DBMSs, we ran the experiments on Amazon EC2[4] using two instances: the first one running YCSB 0.15.0 as a load generator and deployed on a c5.large EC2 instance with 2 vCPUs and 4 GiB RAM, the second one running a DBMS and deployed on an i3.xlarge instance with 4 vCPUs, 30 GiB RAM and an NVMe SSD storage. For JPetstore we ran the experiments on a c5.3xlarge EC2 instance with 8 vCPUs and 15 GiB of RAM, using Jmeter 5.3 to load the application, running on a separate c5.large instance with 4 vCPUs and 4 GiB of RAM.

The instances are summarised in Table 4.1.

Table 4.1: *Cloud instances used for dataset collection.*

Application	Node	Instance	vCPUs	RAM	NVMe SSD
Cassandra	Client	c5.large	2	4 GiB	No
	Server	i3.xlarge	4	30 GiB	Yes
MongoDB	Client	c5.large	2	4 GiB	No
	Server	i3.xlarge	4	30 GiB	Yes
JPetstore	Client	c5.large	2	4 GiB	No
	Server	c5.3xlarge	8	15 GiB	No

[1] `https://www.mongodb.com`
[2] `https://cassandra.apache.org`
[3] `https://github.com/mybatis/jpetstore-6`
[4] `https://aws.amazon.com/ec2/`

4.2.1 Search spaces

We select a set of interesting parameters to tune, including different layers of the IT stack. We use 15 parameters for MongoDB and 24 for Cassandra, reported in Tables 4.2 and 4.3. On JPetstore we focus on JVM parameters, as from Table 4.4. We manually selected the parameters, trying to collect data from different components of the stack. We started from a bigger set of parameters and then kept only those for which we observed some variation in the system performance after an initial sampling over the parameters, hence the different OS parameters for the two DBMSs. When using an autotuner in a real setting, one can automatically select the tunable parameters using available methodologies [32]. Here, we are more interested in building reliable regression models.

Table 4.2: *MongoDB parameters.*

Layer	Parameter
MongoDB	`wiredTigerCacheSizeGB`
MongoDB	`eviction_dirty_target`
MongoDB	`eviction_dirty_trigger`
MongoDB	`syncdelay`
OS	`sched_latency_ns`
OS	`sched_migration_cost_ns`
OS	`vm.dirty_background_ratio`
OS	`vm.dirty_ratio`
OS	`vm.min_free_kbytes`
OS	`vm.vfs_cache_pressure`
OS	Network `RFS`
OS	Storage `noatime`
OS	Storage `nr_requests`
OS	Storage `scheduler`
OS	Storage `read_ahead_kb`

For both Cassandra and MongoDB we select three workloads from the YCSB default ones:

(a) update heavy with a 50/50 mix in read and write operations;

(b) read mostly with a 95/5 reads/write mix;

(c) read-only.

All the workloads use a Zipfian distribution. We also vary the number of YCSB threads from 10 to 90. We use YCSB to create 30 000 000 records, roughly obtaining a 30 GiB database. For JPetstore we vary the number of virtual users using the web application, considering values from 10 to 67.

For Cassandra and MongoDB, before testing a configuration, we restart the DBMS and restore the database to its original version so to avoid any cross-contamination between the experiments of different configurations. We let the experiment run for 45 minutes, discard the initial 15 minutes and the last minute and then compute the average throughput and response time across the measurement period.

Table 4.3: *Cassandra parameters.*

Layer	Parameter
Cassandra	commitlog_compression
Cassandra	commitlog_segment_size_in_mb
Cassandra	commitlog_sync_period_in_ms
Cassandra	Compaction Strategy
Cassandra	compaction_throughput_mb_per_sec
Cassandra	concurrent_compactors
Cassandra	concurrent_reads
Cassandra	concurrent_writes
Cassandra	file_cache_size_in_mb
Cassandra	memtable_cleanup_threshold
JVM	CMSInitiatingOccupancyFraction
JVM	ConcGCThreads
JVM	GC Type
JVM	Xmx (max heap size)
JVM	MaxTenuringThreshold
JVM	NewRatio
JVM	ParallelGCThreads
JVM	SurvivorRatio
OS	CPUSchedNrMigrate
OS	MemoryTransparentHugepageEnabled
OS	MemoryVmDirtyExpire
OS	NetworkNetIpv4TcpMaxSynBacklog
OS	Storage scheduler
OS	Storage read_ahead_kb

Table 4.4: *JPetstore parameters.*

Layer	Parameter
JVM	gcType
JVM	maxHeapSize
JVM	newSize

For JPetstore, we run the performance test for 15 minutes, discard the initial 8 minutes to avoid the JVM warmup and the last minute to avoid alignment issues and finally compute the average of the performance metric over the remaining 7 minutes.

We select the test configurations using Sobol sequences [102], which are quasi-random sequences with a low discrepancy and fill the search domain quickly and evenly.

We report in Table 4.5 the number of parameters and collected points for all the datasets.

4.2.2 Models Accuracy

The performance models are useful for the evaluation of the tuning algorithms. However, if they were very different from the real systems, they would lead us to incorrect results.

To evaluate how close the prediction models are to the real systems, we use a simple

Table 4.5: *Dimensions of datasets and number of collected samples. Each sample represents a 45 minutes experiment for Cassandra and MongoDB, 8 minutes for JPetstore. The workload parameters represent the YCSB workload mix and the number of run threads for the two DBMSs, and the number of virtual users for JPetstore.*

Dataset	Tunable params	Workload params	Samples
MongoDB	15	2	4219
Cassandra	24	2	3728
JPetstore	3	1	24119

holdout validation approach and split the measurements into two sets: the first one contains 25% of the collected data and is used as a test set, the second one contains the remaining data and is used as a train set. Then, we consider progressively bigger subsets of the training set and use them for training several regressors, which we evaluate on the test set.

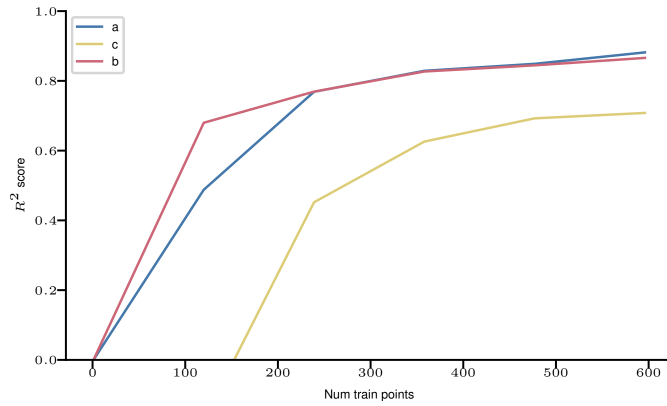
In this way, we can see how the accuracy of the regressors evolves as more data are considered. The results are in Figure 4.2, and we can see that the regressors reach convergence. This indicates that the amount of data we have is enough to create a good performance model. The remaining variability, which the models cannot explain, is probably due to the noise of the measurement, since adding more training points does not increase the score of the models. Without using these regressor models, we would have no way of discarding the measurement noise, which would severely affect the algorithm evaluation.

4.2.3 Workload Patterns

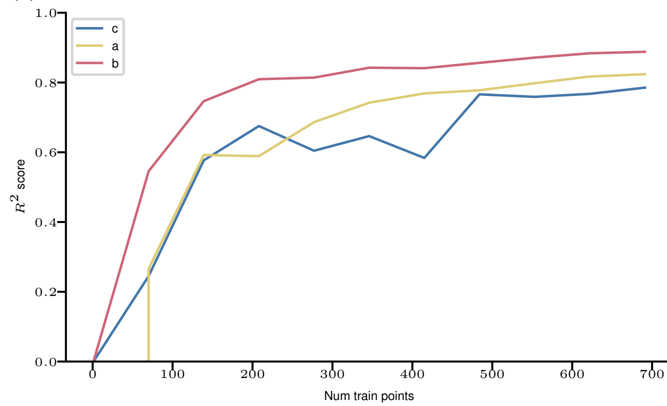
We select three basic workload patterns for the evaluation of MongoDB and Cassandra. We report the patterns in Figure 4.3, using the colours to indicate the read/write mix and the y-axes for the number of YCSB threads. The first pattern (Figure 4.3a) is the simplest one and is used only for the tuning of the hyperparameters, and we vary the number of connected threads, simulating a gradual ramp in the load. In the second one (Figure 4.3b), named Ramp, we gradually vary both the number of connected threads and the read/write mix. In the third one (Figure 4.3c), named Peaks, we try to mimic a typical day-based workload, with the majority of the load concentrated in working hours and a decrease during lunchtime.

The three patterns are repeated identically over time, but none of the considered tuners can exploit this: what matters is just the current workload. Each repetition of the pattern lasts for 30 iterations, and, since each of our measurements takes 45 minutes, each repetition roughly represents a day.

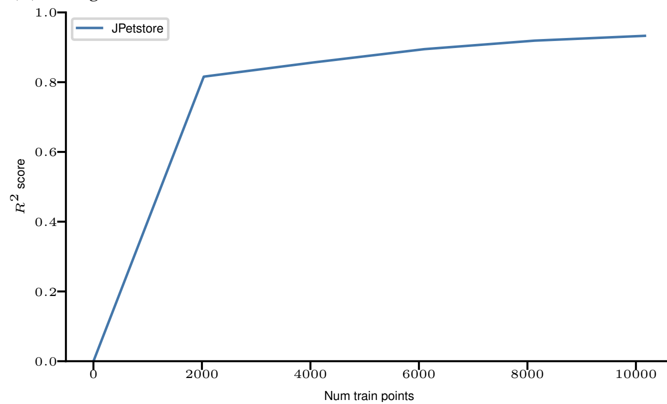
These basic patterns are used to derive the results of Chapter 5, where we evaluate how the autotuners perform when deployed to a performance testing environment where the workload is simulated; hence we use basic workload patterns to mimic a typical performance test. In Chapter 6, instead, we evaluate the autotuners deployed in production environments, where the workload varies continuously. Hence, from the two basic workload patterns used in offline tuning (Ramp and Peaks, as the first one is only used to tune the hyperparameters) we derive two more complex workload patterns by making the variations more continuous. The resulting workload patterns are



(a) *Cassandra*



(b) *MongoDB*

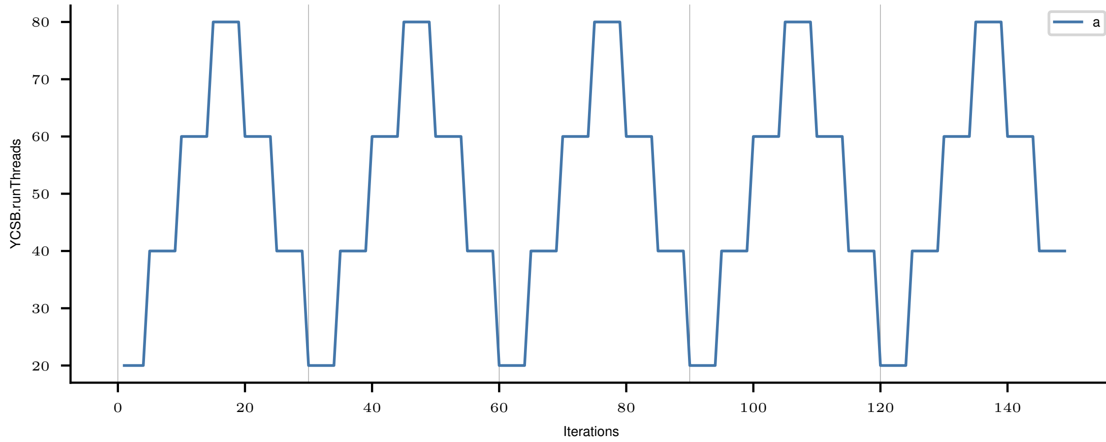


(c) *JPetstore*

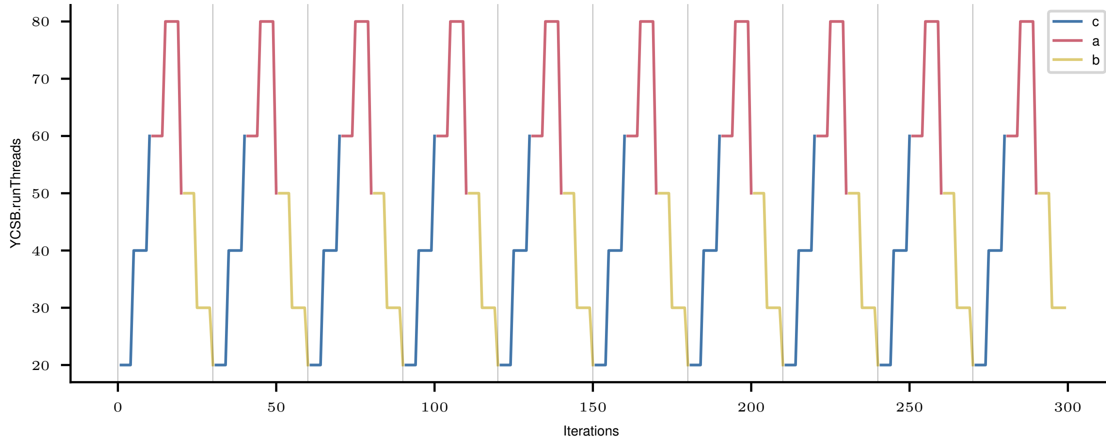
Figure 4.2: *Evolution of the R^2 score of the regressors as more training points are considered. For Cassandra and MongoDB we report a separate score for each YCSB benchmark (a, b, and c).*

reported in Figure 4.4.

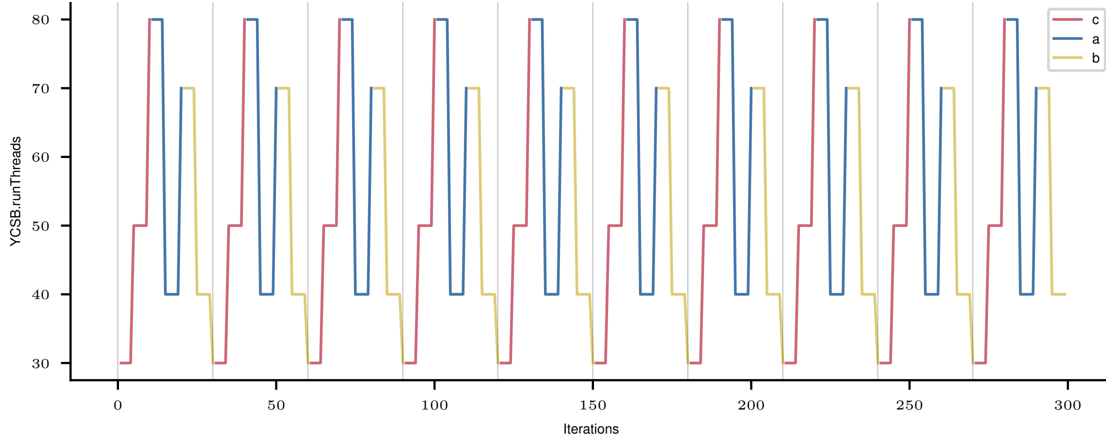
For JPetstore we use again the Ramp and Peaks workload patterns, replacing the YCSB threads variable with the number of simulated users and ignoring the YCSB workload, as we only have one workload variable for JPetstore.



(a) *Intensity pattern: fixed mix, variable intensity.*



(b) *Ramp pattern: variable mix and intensity, one peak.*

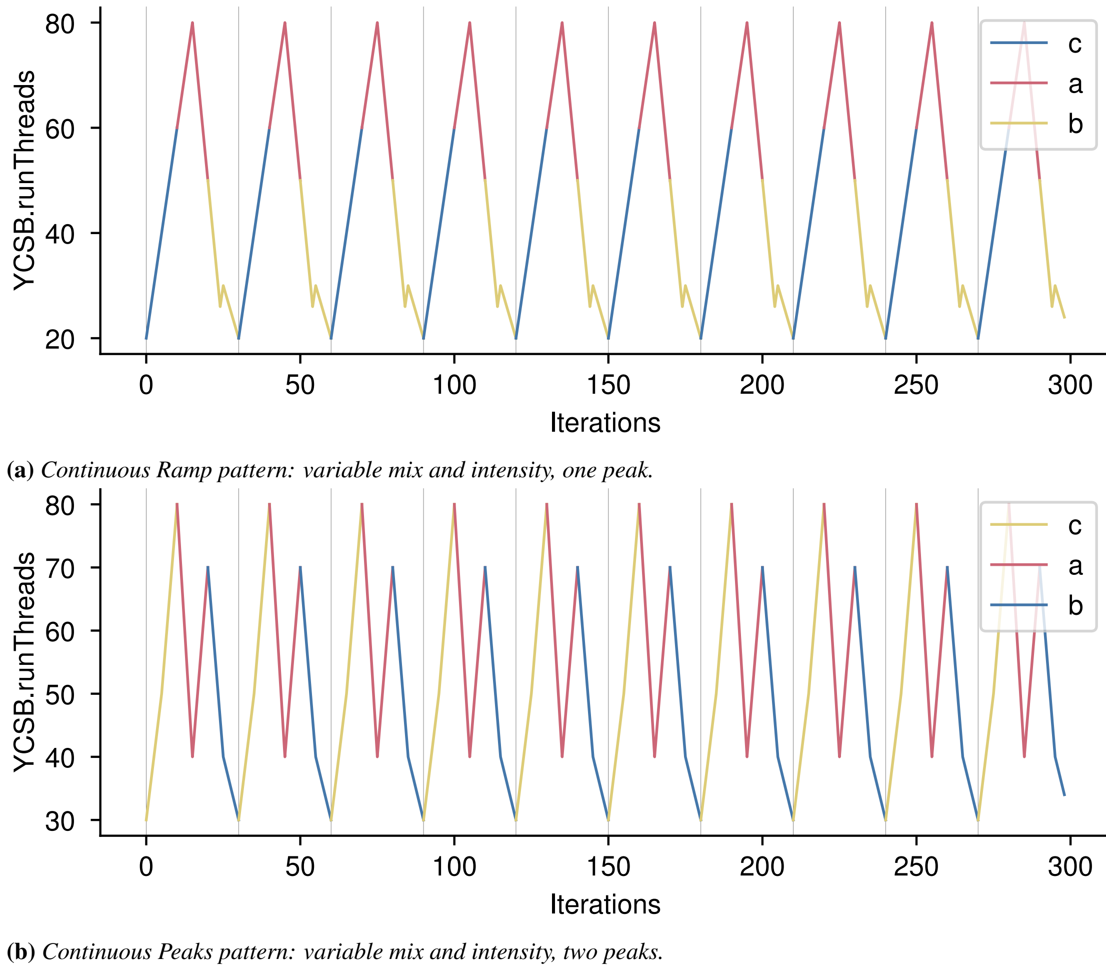


(c) *Peaks pattern: variable mix and intensity, two peaks.*

Figure 4.3: *Workload patterns. Vertical lines represent pattern repetitions and correspond to an entire day of tuning.*

4.3 Evaluation Metrics

In this section, we define the metrics we use to evaluate the performance of an autotuner. Our evaluation process is composed of two steps: the normalisation of the performance



(a) *Continuous Ramp pattern: variable mix and intensity, one peak.*

(b) *Continuous Peaks pattern: variable mix and intensity, two peaks.*

Figure 4.4: *Continuous workload patterns. Vertical lines represent pattern repetitions and corresponds to an entire day of tuning.*

score and the metric computation.

Without considering the technological constraints of the target system, in principle we have two major ways to deploy an autotuner:

- *online*: the autotuner works directly on the production environment;
- *offline*: the autotuner works in a replicated environment; when an interesting configuration is found we promote it to production.

These two approaches call for different tuning strategies. In offline autotuning, we want the autotuner to explore the search space as quickly as possible, finding good configurations that we can then move in the production environment. Conversely, in the online setup, we want the tuner to find good configurations and avoid bad ones: the trade-off between exploration and exploitation is subtler.

We use two metrics to measure the tuner quality in the two setups: Cumulative Reward for online tuning and Iterative Best for offline tuning.

4.3.1 Normalised Performance Improvement

The first step consists in normalising the performance score. The goal is to bring the scores in a coherent space across different tuning setups, to easily compare tuning quality across different scenarios. In fact, barely comparing two different performance scores might be misleading, if at all possible, considering that they could be even using different scales or units of measurement. Moreover, different applications or tuning scenarios might have different tuning difficulties. As an example, if the vendor default configuration is already the global optimum configuration for a certain system on a certain workload, no autotuner will be able to improve this configuration. Hence, not being able to improve a performance score does not mean that an autotuner is working badly: we need to take into account the performance score obtained by the vendor default configuration and, possibly, the score obtained by the global optimum to measure where the autotuner lies in the interval of achievable improvement.

To do this, we build on the Normalised Performance Improvement (NPI) [6, 20, 21], modifying it to normalise both in terms of achieved Performance Improvement (PI) and Diminishment (PD), as we also want to understand how much an autotuner is damaging performance.

Assuming a minimisation problem, the NPI at tuning iteration i is defined as:

$$NPI(i) = \begin{cases} \text{if } y_i \leq y_0 \rightarrow \dfrac{\text{achieved PI}}{\text{potential PI}} = \dfrac{y_0 - y_i}{y_0 - y^\star} = \dfrac{f(\vec{x}_0, \vec{c}_i) - f(\vec{x}_i, \vec{c}_i)}{f(\vec{x}_0, \vec{c}_i) - f(\vec{x}^\star_{\vec{c}_i}, \vec{c}_i)} \\ \text{else } \rightarrow -\dfrac{\text{achieved PD}}{\text{potential PD}} = -\dfrac{y_0 - y_i}{y_0 - y^\diamond} = -\dfrac{f(\vec{x}_0, \vec{c}_i) - f(\vec{x}_i, \vec{c}_i)}{f(\vec{x}_0, \vec{c}_i) - f(\vec{x}^\diamond_{\vec{c}_i}, \vec{c}_i)} \end{cases} \tag{4.3}$$

where $\vec{x}_0$ is the vendor default configuration, $\vec{x}_i$ is the configuration that we are evaluating at iteration i, $\vec{x}^\star_{\vec{c}_i}$ is the optimal configuration for the context observed at iteration i and $\vec{x}^\diamond_{\vec{c}_i}$ is the worst one. Likewise, $y^\star, y^\diamond$ represent the best and worst (respectively) performance scores achievable on context $\vec{c}_i$.

In practice, the NPI is a simple piecewise linear normalisation, where the worst score becomes -1, the baseline 0 and the global optimum +1, for any given context, and is represented in Figure 4.5 graphically. This metric measures the ratio of the achieved performance improvement over the potential performance improvement. For any workload, an NPI of 0 means that we have the same performance obtained by the vendor default configuration, while a positive (negative) unitary NPI means that we have found the global optimum (worst). Differently from the $\tilde{NPI}$ metric that we used in CGPTuner, here we are using the true optimal configuration (according to the collected dataset) instead of the best one observed during the tuning. This metric can thus be computed only to evaluate tuners, and not during the tuning, as it requires knowing all the $\vec{x}^\star, \vec{x}^\diamond$, which are not available at tuning time. Moreover, we can only compute the NPI score when we have access to the best and worst scores, which is not possible when dealing with a real application, but is relatively easy to obtain for analytical functions and regressor.

4.3.2 Online Optimality — Cumulative Reward

Starting from NPI, we define the two metrics that we use to compare autotuners: the Cumulative Reward and the Iterative Best. The Cumulative Reward (CR) is a standard

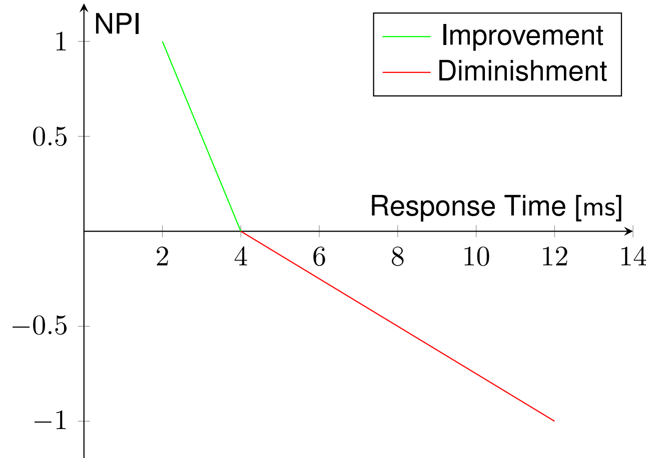


Figure 4.5: *Example of NPI normalisation. We are minimising the response time, the baseline configuration has a response time of* 4 ms*, the best achievable result is* 2 ms *and the worst one is* 12 ms*.*

metric in Reinforcement Learning and is simply defined as the sum of the obtained NPI scores:

$$CR(i) = \sum_{j=0}^{i} NPI(j). \tag{4.4}$$

As the optimal configuration has an NPI of 1, a perfect tuner has a CR with a constant unitary slope. On the other hand, keeping the baseline configuration leads to a zero reward, while testing bad configurations gives a negative reward. This metric is thus a useful indicator of the tuner's ability to understand the problem and adapt to context variations, trading off exploration with exploitation.

An ideal autotuner would always suggest the optimum configuration, thus receiving an NPI score equal to one. Therefore, its NPI curve would be the diagonal. We can thus derive an even more synthetic indicator of online optimality by computing the average NPI score over an entire tuning session, substantially obtaining the overall slope of the CR curve. We thus define online optimality after i tuning iterations as:

$$OnOpt@i = \frac{\sum_{j=0}^{i} NPI(j)}{i} = \frac{CR(i)}{i} \tag{4.5}$$

which can be easily interpreted as the percentage of online optimality of an autotuner, ranging from 0%, which corresponds to keeping the vendor default configuration to 100%, representing the perfect autotuner.

4.3.3 Offline Optimality — Iterative Best

The Iterative Best (IB) is defined as:

$$IB(i) = \max_{j: j \leq i, \vec{c}_j = \vec{c}_i} NPI(j). \tag{4.6}$$

In simpler terms, we keep a separate counter for each observed context and, at each iteration, compute an iterated maximum on the appropriate counter. The IB thus measures the ability of a tuner to quickly explore the search space and find good configurations,

without considering if it explores many bad configurations while looking for the good ones.

Similarly to online optimality, we can define offline optimality as:

$$OffOpt@i = \frac{\sum_{j=0}^{i} IB(j)}{i}. \quad (4.7)$$

Here, offline optimality of 0% means that the autotuner never suggested a better configuration than the default one, and offline optimality of 100% means that the best configuration (for each context) was suggested as soon as that context was observed.

We repeat the tuning multiple times and then compute NPI, CR and IB for each repetition. Then, we take the median over the repetitions to produce the plots.

As we normalise all the performance metrics in terms of NPI, they will be more challenging to interpret for the performance expert. However, we use them to quantitatively compare the autotuners' efficacy from an optimisation viewpoint and minimise measurement noise. If we used the non-normalised performance measurement, we could conclude that a tuner is performing well when, in reality, we just tested an easily tunable workload.

4.3.4 Constraint Violations

To evaluate the ability of an autotuner to deal with constraints, we modify the NPI definition in case we are evaluating a constrained tuning scenario. Namely, a configuration that leads to a constraint violation receives an NPI of minus one, as violating a constraint is worse than selecting a configuration with a bad score.

Modifying the NPI definition has a direct effect on the optimality scores, but we also use another metric, called normalised cumulated violations, which measures the total number of constraint violations over the entire tuning, and divide it by the number of tuning iterations. Thus, it simply measures the fraction of iterations that lead to constraint violation.

4.3.5 Workload Forecasting

To evaluate the workload forecasting module, we use the Mean Absolute Percentage Error (MAPE) and the Root Mean Squared Error (RMSE).

Formally, we want to evaluate the forecasting accuracy for a time interval $t_0 : t_n$, composed by several fine-grained forecasted values $\tilde{c}_{t_0}, \tilde{c}_{t_0+\delta}, \ldots, \tilde{c}_{t_n}$, where δ is the (fine grained) time resolution of the forecaster. Notice that the context is composed of several time-series, but we separately evaluate the forecasting accuracy on each component.

MAPE is defined as:

$$MAPE(\tilde{c}_{t_0:t_n}, c_{t_0:t_n}) = \frac{100\%}{n} \sum_{t=0}^{n} \left| \frac{c_t - \tilde{c}_t}{c_t} \right| \quad (4.8)$$

and RMSE is defined as:

$$RMSE(\tilde{c}_{t_0:t_n}, c_{t_0:t_n}) = \sqrt{\sum_{t=0}^{n} \frac{(c_t - \tilde{c}_t)^2}{n}}. \quad (4.9)$$

RMSE is more sensitive to outliers, whereas MAPE has a very intuitive interpretation in terms of relative error.

4.4 Baseline Algorithms

We compare the performance of CC-GP against Random (Rnd), BestConfig [114] (BC), OpenTuner [4] (OT) and BayesianOptimization [80] (BO). BayesianOptimization is an off-the-shelf optimiser based on Bayesian Optimisation and Gaussian Processes, hence it is similar to the core optimiser of CC-GP without the components introduced in Chapter 3.

For the context forecasting component, we evaluate Prophet [105], DeepAR, DeepState and MQCNN. We also include a naive predictor (called *daily*), which simply repeats the data of the previous day. As already said in Section 3.7, we select Prophet mainly due to its analyst-in-the-loop approach, which allows performance experts to tune forecasting parameters. Nonetheless, we will show that it is capable of producing state-of-the-art predictions.

4.5 Hyperparameter Tuning

All the considered tuners have some hyperparameters to select: in OpenTuner, we have several tuning techniques that we can use; in BestConfig, we need to decide the number of tuning rounds (i.e., partial restarts of the search algorithm). Moreover, for CC-GP we need to select whether the workload and configuration kernels should be combined with a sum or a multiplication. We use a cross-validation approach: we select a simple workload pattern (Figure 4.3a) on Cassandra and compute the final median cumulated reward of all the considered tuners. This validation pattern is then excluded from other evaluations, and it is very different from the patterns that we use to compare the tuners.

For OpenTuner, we select the “AUCBanditMetaTechniqueA”' which, according to OpenTuner description, is a Meta Technique composed by:

- a Differential Evolution technique with a crossover rate of 20%
- a Uniform Greedy Mutation technique
- a Normal Greedy Mutation technique with a mutation rate of 30%
- a Random Nelder-Mead technique.

OpenTuner uses a multi-armed bandit to decide which technique to use at each iteration.

For BestConfig, we use a number of rounds equal to the number of times we repeat the entire workload pattern. As we imagine that the entire workload pattern captures an entire day, we are basically starting a new BestConfig round each day. Finally, for CC-GP we decide to combine the kernels with a multiplication, thus considering two points similar when both the configuration and the workload are similar.

For the time-series forecasting and window stability functions, we used a continuous version of the hyperparameter tuning pattern and tuned the hyperparameters to minimise prediction RMSE and manually tuned the stability thresholds.

CHAPTER 5

Offline Autotuning

In this chapter, we report some results achieved in offline tuning scenarios, where we deploy the autotuner in a controlled performance testing environment, and expose the system under test to a controlled synthetic workload. This means that the measurement noise is relatively low. To better evaluate the various components of the proposed autotuner, we use different tuning scenarios, ranging from simple 2-dimensional analytical functions to more complex simulations with workload variations.

5.1 Single Context

To evaluate the core component of the proposed autotuner (the Bayesian optimiser), we start by considering the most simple situation where we are only interested in optimising a certain performance score in a single controlled context, without considering any constraint.

5.1.1 Analytical Functions

We start by reporting, in Figure 5.1, the results obtained for the optimisation of the Branin analytical function. The main goal of this optimisation is to visualise the result normalisation process in a simple scenario. We start by reporting in Figure 5.1a the non-normalised values obtained during the optimisation, i.e. the actual value of the Branin function. Clearly, understanding the tuner behaviour from the non-normalised results is extremely difficult. Hence, we normalise the results using the NPI normalisation, obtaining Figure 5.1b, where it is already easier to understand what is happening. As an example, we see that keeping the baseline configuration brings to a fixed NPI equal to zero, whereas the optimum is equal to one, as per NPI definition. Moreover, on average, a random configuration is similar to the baseline one. OpenTuner and BestConfig

consistently outperform the baseline, but they both fail at converging to the optimum. Conversely, both the naive BayesianOptimization and CC-GP find and converge to the optimum, but CC-GP has a faster convergence. We can derive similar conclusions by looking at Figures 5.1c and 5.1d, depicting the Iterative Best and the Cumulative Reward computed over the NPI scores. Notice that the IB score of the Random tuner reaches a nearly-optimal value, whereas its NPI scores never go above 0.2. This is because we run each tuner multiple times, and then compute the median and the standard deviation after the NPI normalisation and the IB (and CR) computation. In other words, we evaluate the performance of each tuning on its own, and then take the median as the last step. Hence, the IB curve is not visually equivalent to the superior edge of the NPI curve. This is coherent with the fact that, on a given random tuning session, we will find a nearly optimal value pretty soon (as represented by the high IB), but we will not converge toward this value (as testified by the low NPI and CR score). Furthermore, this confirms, again, that even a Random tuner is capable of finding well-performing configurations, and, thus, we should use the bare performance as a metric to compare auotuners.

We report a more synthetic representation of the same results Table 5.1. The first row contains the Best Value found, visually equivalent to the last value of the Iterative Best curve. The second row contains the Offline Optimality, visually equivalent to the area under the Iterative Best curve normalised by the number of iterations (i.e., as a fraction of the area under the Best tuner curve). The third row contains the Online Optimality, visually equivalent to the last value of the Cumulative Reward curve normalised by the number of iterations. For each metric, the best-performing tuner is reported in bold among with the statistically equivalent result using a Welch's t-test with a 5% p-value. We use Welch's t-test instead of the usual Student's t-test to take into account the fact that different tuners have different variances. When comparing n tuners we apply Bonferroni correction for $n-1$ hypothesis, as we are comparing the average of the best tuner with the other ones.

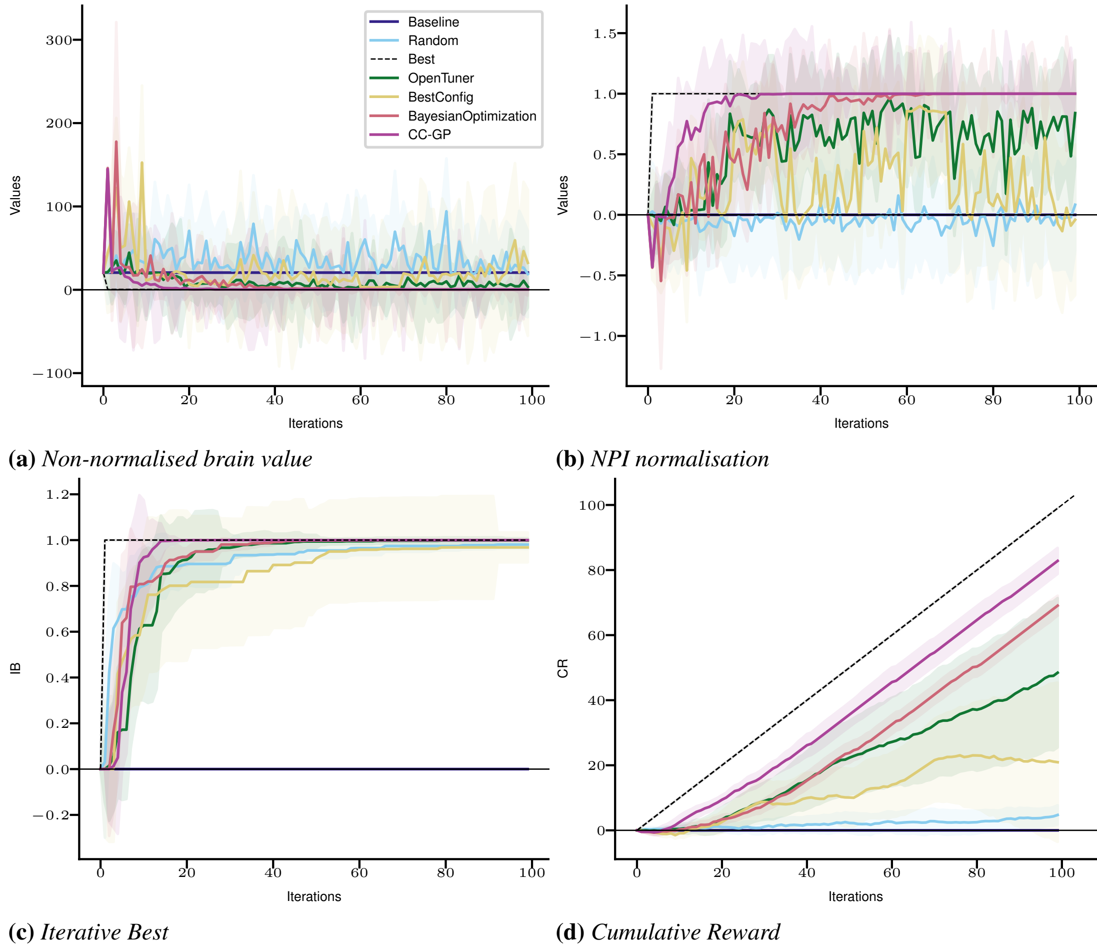
The qualitative interpretation of the results is the same one we derived from the graphs: all the tuners find a configuration close to the optimum, but BO and CC-GP find it faster, and CC-GP has the best convergence.

Table 5.1: *Compact representation of the Branin results in terms of Best Value, Offline Optimality and Online Optimality. Best tuners per scenario in bold using Welch's t-test with 5% p-value over 16 repetitions. All the tuners find a configuration close the optimum, but BO and CC-GP find it faster and CC-GP has the best convergence.*

Metric	Random	OT	BC	BO	CC-GP
BV	0.98 ± 0.02	1.00 ± 0.01	0.97 ± 0.07	$\mathbf{1.00}$	$\mathbf{1.00}$
OffOpt	0.90 ± 0.04	0.90 ± 0.06	0.86 ± 0.21	$\mathbf{0.92 \pm 0.02}$	$\mathbf{0.93 \pm 0.03}$
OnOpt	0.05 ± 0.03	0.48 ± 0.23	0.21 ± 0.24	0.69 ± 0.03	$\mathbf{0.83 \pm 0.04}$

Noise Robustness

The noise is a fundamental difference between a real problem and a simulated environment. We use simulations to take away the noise as much as possible and compare the autotuners reliably. However, we also need to consider how two autotuners behave in



(a) *Non-normalised brain value*

(b) *NPI normalisation*

(c) *Iterative Best*

(d) *Cumulative Reward*

Figure 5.1: *Optimisation of the Branin analytical function used to describe the result normalisation process. a contains the raw value obtained during the optimisation, which are normalised using NPI in b. From the NPI values we obtain the Iterative Best in c and the Cumulative Reward d. As we run each tuner multiple times, we take the average value using the median and derive the error bar using the standard deviation.*

the presence of measurement noise.

We thus use a noisy version of the Branin function, where we evaluate the original Branin function, normalise it according to $N\tilde{P}I$ and then add a normally-distributed noise with standard deviation σ. In this way, σ can be directly interpreted as a measurement noise, since $N\tilde{P}I$ rannges from -1 to 1.

We then run a set of experiments on this noisy function; and compute the optimality metrics discarding the noise. In other words, to reach the optimum, a tuner has to actually find the global optimum configuration, whereas being lucky with the noise (testing a suboptimal configuration which results in being even better than the optimum) does not impact the evaluation.

The results are available in Table 5.2. As visible from the online optimality, the convergence of CC-GP is not significantly impacted even by a really high amount of measurement noise.

Table 5.2: *Compact representation of the noisy Branin results in terms of Best Value, Offline Optimality and Online Optimality with increasing values of noise σ. Best tuners per scenario in bold using Welch's t-test with 5% p-value over 16 repetitions.*

Metric	Random	CC-GP	CC-GP 1%	CC-GP 2%	CC-GP 5%	CC-GP 10%	CC-GP 25%	CC-GP 50%
BV	$\mathbf{0.95 \pm 0.11}$	$\mathbf{1.00}$	$\mathbf{1.00}$	$\mathbf{1.00}$	$\mathbf{1.00}$	$\mathbf{1.00}$	$\mathbf{1.00}$	$\mathbf{1.00}$
OffOpt	$\mathbf{0.78 \pm 0.12}$	$\mathbf{0.87 \pm 0.05}$	$\mathbf{0.84 \pm 0.04}$	$\mathbf{0.87 \pm 0.06}$	$\mathbf{0.89 \pm 0.06}$	$\mathbf{0.87 \pm 0.07}$	$\mathbf{0.85 \pm 0.07}$	$\mathbf{0.89 \pm 0.06}$
OnOpt	-0.01 ± 0.02	$\mathbf{0.70 \pm 0.05}$	$\mathbf{0.69 \pm 0.05}$	$\mathbf{0.68 \pm 0.09}$	$\mathbf{0.65 \pm 0.07}$	0.65 ± 0.06	$\mathbf{0.65 \pm 0.11}$	0.59 ± 0.08

Dimensionality Robustness

As said in Chapter 1, an IT system has many tunable parameters. This conflicts with the nature of a Gaussian Process, which notoriously has limited scalability to high-dimensional spaces [97]. However, it is important to stress that this limitation only really applies to parameters that have a significant impact on the score function. Luckily, most of the tunable parameters do not affect the performance, even if we do not know which are the relevant parameters for the problem at hand.

To illustrare this effect, we run a series of experiments on the Branin function, modifying it to add N non-relevant dimensions:

$$branin_N(x_1, x_2, x_3, \dots, x_N) = branin(x_1, x_2). \tag{5.1}$$

The autotuner sees an $N+2$-dimensional problem, but, in reality, only 2 of the parameters really matter.

We report the results in Figure 5.2. As visible, even adding one hundred irrelevant parameters does not significantly impact the convergence of CC-GP.

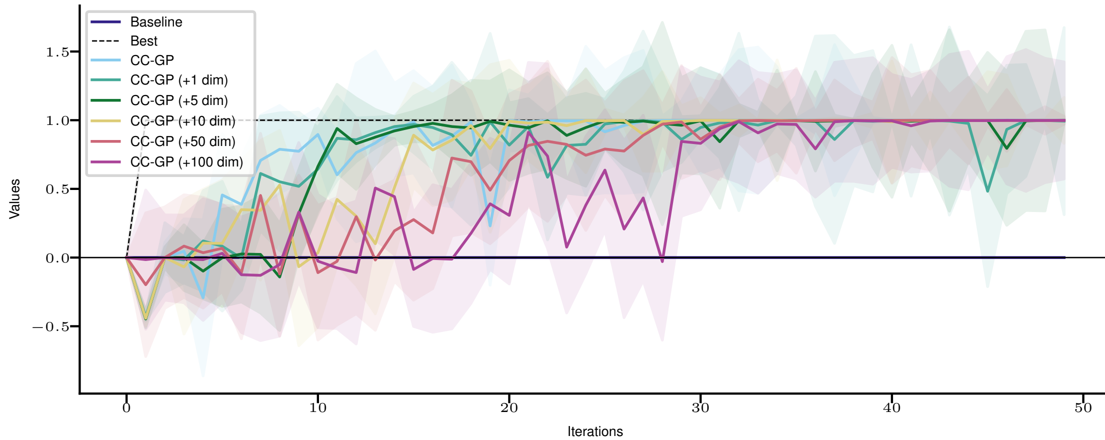


Figure 5.2: *NPI normalised values*

Table 5.3 contains synthetic results in terms of Best Value, Offline Optimalityand Online Optimality, from which we can see that adding irrelevant dimensions does not significantly degrade the performance of CC-GP.

Having observed that a GP is perfectly capable of discarding irrelevant parameters in the context of an analytical function, one could argue that in a real context, the presence of noise would pose a problem, as the GP could try to impute this noise to the irrelevant parameters.

Table 5.3: *Compact representation of the Branin results with irrelevant parameters in terms of Best Value, Offline Optimality and Online Optimality after 50 tuning iterations. Best tuners per scenario in bold using Welch's t-test with 5% p-value over 8 repetitions.*

Metric	CC-GP	CC-GP +1	CC-GP +2	CC-GP +5	CC-GP +10	CC-GP +20	CC-GP +50	CC-GP +100
BV	$\mathbf{1.00}$	$\mathbf{1.00}$	$\mathbf{1.00}$	$\mathbf{1.00}$	$\mathbf{1.00}$	$\mathbf{1.00}$	$\mathbf{1.00 \pm 0.02}$	$\mathbf{1.00}$
OffOpt	$\mathbf{0.87 \pm 0.05}$	$\mathbf{0.86 \pm 0.04}$	$\mathbf{0.86 \pm 0.05}$	$\mathbf{0.84 \pm 0.04}$	$\mathbf{0.86 \pm 0.05}$	$\mathbf{0.80 \pm 0.13}$	$\mathbf{0.84 \pm 0.09}$	$\mathbf{0.84 \pm 0.10}$
OnOpt	$\mathbf{0.70 \pm 0.05}$	0.61 ± 0.06	$\mathbf{0.68 \pm 0.07}$	0.63 ± 0.05	$\mathbf{0.67 \pm 0.05}$	$\mathbf{0.67 \pm 0.12}$	$\mathbf{0.55 \pm 0.16}$	0.44 ± 0.22

To show that this is not the case, we repeat the experiments using the noisy version of the Branin function introduced above with a noise level of 10%, and report the results in Table 5.4. In the presence of noise, irrelevant parameters impact convergence slightly more, especially in terms of online optimality, but the results are still surprisingly good, even with one hundred irrelevant dimensions.

Table 5.4: *Compact representation of the noisy Branin ($\sigma = 0.1$) results with irrelevant parameters in terms of Best Value, Offline Optimality and Online Optimality after 50 tuning iterations. Best tuners per scenario in bold using Welch's t-test with 5% p-value over 16 repetitions.*

Metric	Random	CC-GP	CC-GP +1	CC-GP +10	CC-GP +100	CC-GP +500
BV	$\mathbf{0.95 \pm 0.11}$	$\mathbf{1.00}$	$\mathbf{1.00}$	$\mathbf{1.00}$	$\mathbf{1.00 \pm 0.01}$	0.98 ± 0.03
OffOpt	$\mathbf{0.78 \pm 0.12}$	$\mathbf{0.83 \pm 0.05}$	$\mathbf{0.83 \pm 0.08}$	$\mathbf{0.75 \pm 0.13}$	$\mathbf{0.83 \pm 0.07}$	$\mathbf{0.84 \pm 0.07}$
OnOpt	-0.01 ± 0.02	$\mathbf{0.55 \pm 0.03}$	$\mathbf{0.59 \pm 0.05}$	$\mathbf{0.48 \pm 0.09}$	0.38 ± 0.18	0.24 ± 0.13

As said in Section 3.2, CC-GP is based on a Gaussian Process using a Matérn 5/2 kernel [87]. The Matérn kernel uses Automatic Relevance Determination (ARD) to set its *length scale* hyperparameters. Loosely speaking, the length scales control the correlation between a parameter and the score. We can thus use them as a sort of parameter importance as seen by the GP. In Figure 5.3, we report the (inverse of) length scales identified by the GP at the end of the tuning process with one hundred irrelevant parameters and 10% noise. ARD correctly identifies the first two parameters as the most relevant ones, and only three irrelevant parameters are identified as marginally relevant due to noise, which is coherent with the positive results observed above.

5.1.2 Dataset-based Simulation

Moving to the dataset-based simulations, we briefly report in Table 5.5 the results obtained for the tuning of the dataset-based simulations in the case of a single workload. More specifically, for Cassandra and MongoDB we selected a read-mostly workload (YCSB Workload `b`) with 50 YCSB threads, while for JPetstore we selected to tune for 35 simulated users. As on JPetstore we have less tunable parameters (hence a simple problem), we let the tuning run for 100 iterations, whereas we use 150 iterations for MongoDB and Cassandra. Essentially, the results are the same ones we observed on the Branin analytical function, with CC-GP achieving the best result on all the tuning scenarios according to all the considered metrics.

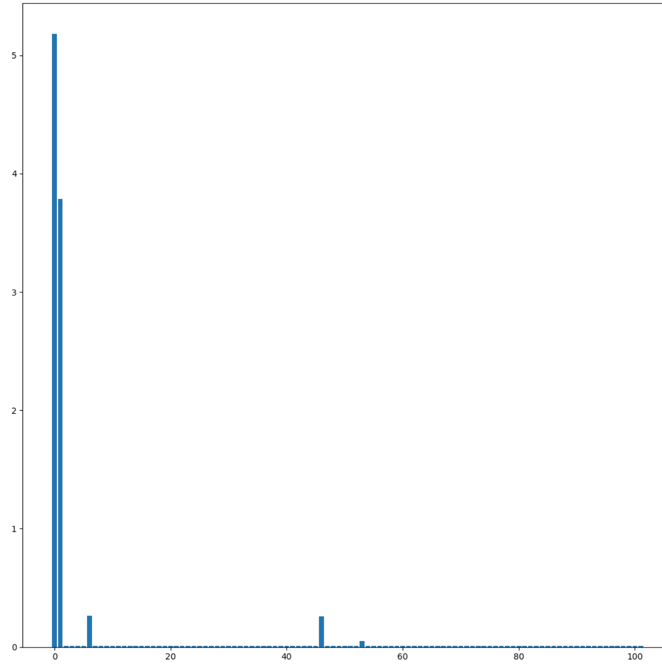


Figure 5.3: *Parameter importance identified by the GP at the end of the tuning process with 100 irrelevant dimensions and 10% noise. The first two parameters are correctly identified as the most relevant ones, and only three irrelevant parameters are identified as marginally relevant due to noise.*

Table 5.5: *Synthetic results for the offline tuning of single-workload: (1) Cassandra exposed to a read-mostly workload (B) with 50 threads for 150 tuning iterations; (2) MongoDB exposed to a read-mostly workload (B) with 50 threads for 150 tuning iterations; (3) JPetstore exposed to 35 virtual users for 100 tuning iterations. Best tuners per scenario in bold using Welch's t-test with 5% p-value over 16 repetitions and 150 tuning iterations (100 for JPetstore).*

Scenario	Metric	Random	OT	BC	BO	CC-GP
Cassandra b50	BV	0.87 ± 0.02	$\mathbf{0.96 \pm 0.02}$	0.90 ± 0.02	0.75 ± 0.24	$\mathbf{0.95 \pm 0.01}$
	OffOpt	0.83 ± 0.04	$\mathbf{0.85 \pm 0.04}$	$\mathbf{0.87 \pm 0.01}$	0.71 ± 0.25	$\mathbf{0.90 \pm 0.05}$
	OnOpt	0.15 ± 0.03	$\mathbf{0.70 \pm 0.05}$	0.64 ± 0.06	0.18 ± 0.12	$\mathbf{0.74 \pm 0.04}$
MongoDB b50	BV	0.94 ± 0.01	0.98 ± 0.01	0.93 ± 0.07	0.97 ± 0.01	$\mathbf{0.99}$
	OffOpt	0.90 ± 0.01	0.91 ± 0.04	0.89 ± 0.07	0.95 ± 0.01	$\mathbf{0.97 \pm 0.01}$
	OnOpt	0.54 ± 0.02	0.79 ± 0.04	0.60 ± 0.08	$\mathbf{0.92 \pm 0.01}$	$\mathbf{0.91 \pm 0.01}$
JPetstore 35 users	BV	0.86 ± 0.05	$\mathbf{0.92 \pm 0.03}$	0.71 ± 0.16	$\mathbf{0.92 \pm 0.25}$	$\mathbf{0.94 \pm 0.05}$
	OffOpt	0.70 ± 0.08	$\mathbf{0.84 \pm 0.08}$	0.52 ± 0.14	0.68 ± 0.26	$\mathbf{0.88 \pm 0.08}$
	OnOpt	-0.18 ± 0.04	0.18 ± 0.10	-0.04 ± 0.10	0.12 ± 0.11	$\mathbf{0.58 \pm 0.13}$

Effect of Full Stack and Holistic Approach

In Table 5.6 we show the optimum response time achievable when tuning Cassandra subject to two different workloads: an update-heavy one with 80 YCSB run threads (A80) and a read-only one with 30 YCSB run threads (C30). We consider six different situations where we tune the three different layers we have in the dataset: Cassandra, JVM and Linux OS. As visible, no single layer allows achieving the maximum performance, which motivates our full-stack approach. We also include the best result found by CC-GP after 50 tuning iterations (using all three layers). In both scenarios, it is quite

close to the global optimum. Similar results are reported in Table 5.7 for MongoDB, where the achievable gains are even more equally distributed among the two layers.

Table 5.6: *Minimum response time in milliseconds achievable in the tuning of Cassandra (exposed to an update-heavy workload with 80 run threads A80 and a read-only workload with 30 run threads C30) considering the three technological layers separately and then combining them. The last line contains the best result found by CC-GP considering all the three layers after 50 tuning iterations. Tuning all the layers is essential to achieve maximum performance.*

Layer	A80	C30
None (baseline)	7.94	2.21
Cassandra	7.83	2.20
JVM	5.42	1.92
OS	7.68	2.17
Cassandra+JVM	5.02	1.89
Cassandra+OS	7.57	2.18
JVM+OS	5.28	1.66
Cassandra+JVM+OS	4.82	1.63
CC-GP	5.14	1.87

Table 5.7: *Minimum response time in milliseconds achievable in the tuning of MongoDB (exposed to an update-heavy workload with 80 run threads A80 and a read-only workload with 30 run threads C30) considering the two technological layers separately and then combining them. The last line contains the best result found by CC-GP considering both layers after 50 tuning iterations. Tuning all the layers is essential to achieve maximum performance.*

Layer	A80	C30
None (baseline)	10.66	2.07
MongoDB	7.45	1.62
OS	9.85	1.85
MongoDB+OS	6.81	1.48
CC-GP	6.81	1.48

In Table 5.8 we report the optimal configuration in the three tunable scenarios. Albeit some parameters have the same value when tuning the single layers or combining them, there are many others whose optimal value in the combined tuning is different from the composition of the per-layer optimums.

Effect of Goal-Driven

In Table 5.9 we report the optimal configurations available in the collected dataset for the tuning of JPetstore exposed to 35 virtual users. We look for two optimal configurations: the first one minimising the 99th percentile of the response time (the main goal we use in the JPetsore studies), and the second one minimising the JVM garbage collection time. The two goals are quite different, and, reasonably, the optimal configurations are different, especially in the selected GC algorithm, which is a critical decision to take while configuring the JVM. Using a goal-driven approach we produce a tailored configuration for the requested goal.

Table 5.8: *Best configuration found while considering different layers for the tuning of MongoDB A80. The global optimum is different from the combination of the per-layer optima.*

Layer	Parameter	Baseline	MongoDB	OS	All
MongoDB	`wiredTigerCacheSizeGB`	1500	27509	-	27651
MongoDB	`eviction_dirty_target`	5	57	-	53
MongoDB	`eviction_dirty_trigger`	20	72	-	64
MongoDB	`syncdelay`	60	71	-	312
OS	`sched_latency_ns`	18000000	-	71453020	133961719
OS	`sched_migration_cost_ns`	500000	-	1717803	3655371
OS	`vm.dirty_background_ratio`	10	-	3	52
OS	`vm.dirty_ratio`	20	-	4	56
OS	`vm.min_free_kbytes`	67584	-	228803	900250
OS	`vm.vfs_cache_pressure`	100	-	11	52
OS	Network `RFS`	0	-	32044	97408
OS	Storage `noatime`	True	0	-	False
OS	Storage `nr_requests`	32	-	86	850
OS	Storage `scheduler`	none	-	kyber	kyber
OS	Storage `read_ahead_kb`	128	-	770	15

Table 5.9: *Best configuration found while tuning Jpetstore with 35 simulated users for two different goals: minimising the* 99th *percentile of the response time, and minimising the JVM GC time. The optimal configuration varies with the goal.*

Parameter	Response Time	GC Time
gcType	G1	CMS
maxHeapSize	341	340
newSize	213	263

5.2 Multiple Workloads

We now move to a more dynamic scenario where the workload varies during the tuning process. To evaluate CC-GP incrementally, we do not vary the workload continuously, but we only use some predefined workloads. This allows us to skip the clustering step presented in Section 3.6 and focus on the evaluation of the contextual component. Thus, we treat each distinct workload as a cluster and normalise them separately using $\tilde{NPI}$ as defined in Section 3.5. Furthermore, we assume to have an oracle workload forecaster, so that the autotuner knows which will be the next workload, and suggest an appropriate configuration.

5.2.1 Dataset-based Simulation

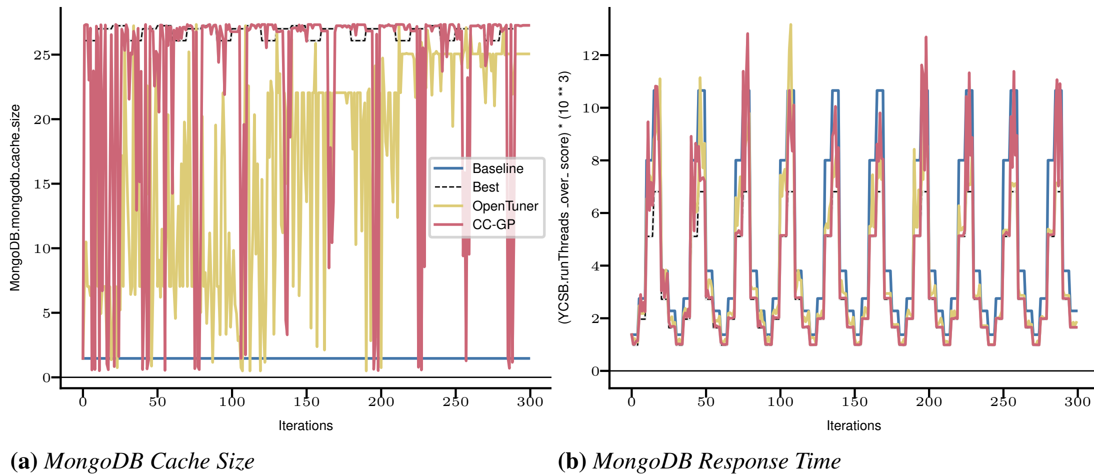
In Table 5.10 we show the results for the offline, multiple-workload tuning of Cassandra, MongoDB and JPetstore on the Ramp and Peaks workload patterns. CC-GP outperforms BO both in terms of online and offline optimality in all the scenarios.

Interestingly, all the tuners exhibit similar performance on the Cassandra and MongoDB ramp and peaks workload scenarios. To better understand these results, we report in Figure 5.4 the value of the MongoDB cache size parameter and the corresponding response time selected by OpenTuner and CC-GP during the tuning of MongoDB sub-

Table 5.10: *Synthetic results for the offline tuning of Cassandra, MongoDB and JPetstore exposed to the ramp and peaks workload scenarios. Best tuners per scenario in bold using Welch's t-test with 5% p-value over 16 repetitions and 300 tuning iterations.*

Scenario	Metric	Random	OT	BC	BO	CC-GP
Cassandra Ramp	OffOpt	0.55 ± 0.02	0.64 ± 0.04	0.61 ± 0.04	0.59 ± 0.17	$\mathbf{0.68 \pm 0.03}$
	OnOpt	0.14 ± 0.02	$\mathbf{0.47 \pm 0.13}$	$\mathbf{0.37 \pm 0.05}$	0.16 ± 0.09	$\mathbf{0.45 \pm 0.03}$
Cassandra Peaks	OffOpt	0.56 ± 0.03	0.65 ± 0.04	0.63 ± 0.03	0.57 ± 0.17	$\mathbf{0.68 \pm 0.03}$
	OnOpt	0.14 ± 0.01	$\mathbf{0.46 \pm 0.10}$	0.35 ± 0.04	0.10 ± 0.08	$\mathbf{0.45 \pm 0.03}$
MongoDB Ramp	OffOpt	0.88 ± 0.02	0.88 ± 0.04	0.77 ± 0.05	0.89 ± 0.02	$\mathbf{0.95 \pm 0.01}$
	OnOpt	0.53 ± 0.02	0.73 ± 0.07	0.55 ± 0.03	0.54 ± 0.02	$\mathbf{0.80 \pm 0.02}$
MongoDB Peaks	OffOpt	0.87 ± 0.01	0.89 ± 0.05	0.77 ± 0.03	0.89 ± 0.02	$\mathbf{0.95 \pm 0.01}$
	OnOpt	0.53 ± 0.01	0.76 ± 0.06	0.55 ± 0.04	0.52 ± 0.02	$\mathbf{0.82 \pm 0.01}$
JPetstore Ramp	OffOpt	0.40 ± 0.02	0.77 ± 0.04	0.65 ± 0.06	0.72 ± 0.05	$\mathbf{0.83 \pm 0.01}$
	OnOpt	-0.22 ± 0.02	0.23 ± 0.08	0.13 ± 0.06	0.15 ± 0.06	$\mathbf{0.52 \pm 0.05}$
JPetstore Peaks	OffOpt	0.82 ± 0.04	0.82 ± 0.05	0.68 ± 0.06	0.79 ± 0.02	$\mathbf{0.85 \pm 0.02}$
	OnOpt	-0.02 ± 0.03	0.25 ± 0.09	0.11 ± 0.07	0.20 ± 0.04	$\mathbf{0.56 \pm 0.07}$

ject to the Ramp workload scenario. We consider only one tuning repetition and the two best-performing autotuners for visualisation simplicity. As visible, to achieve a low response time, it is sufficient to select a big cache size, and the workload has nearly no effect. The situation is similar in the other tuning scenarios, which makes them rather unrealistic, as they represent a too simplistic tuning situation. A more interesting problem can be expressed by including a constraint on the cache size, which we will do in Chapter 7. Nonetheless, CC-GP still visibly outperforms OpenTuner, which is the second-best according to both online and offline optimality.

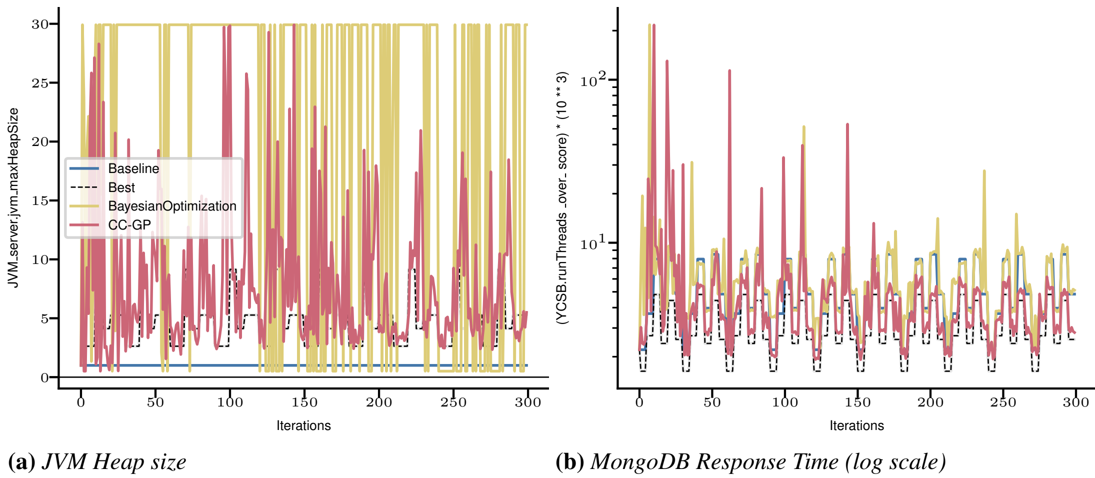


(a) *MongoDB Cache Size* **(b)** *MongoDB Response Time*

Figure 5.4: *Selected cache size and resulting response time for the tuning of MongoDB ramp. To achieve a low response time it is sufficient to configure a big cache on all the workloads.*

Furthermore, these scenarios are also valuable to assess the differences between CC-GP and BayesianOptimization. In Figure 5.5 we report the selected JVM heap size and the corresponding response time achieved during the tuning of Cassandra on the Peaks workload scenario. BayesianOptimization omits the per-context (in this case per-

workload) score normalisation introduced in Section 3.5, and the context information does not arrive at the surrogate model, which is a Gaussian Process and not a contextual Gaussian Process as in CC-GP (Section 3.3). Therefore, BayesianOptimization has no way to explain the score variability of Figure 5.5b, as similar configurations lead to different scores. As an example, looking at the first hundreds of iterations, BayesianOptimization puts the heap size nearly always to the maximum value, but the response time varies wildly (the other tuned parameters are also set to fixed values). Hence, the variability can only be explained in terms of measurement noise, which leads BayesianOptimization to exhibit a nearly random behaviour in subsequent iterations.



(a) *JVM Heap size*

(b) *MongoDB Response Time (log scale)*

Figure 5.5: *Selected JVM heap size and resulting response time for the tuning of Cassandra peaks. BayesianOptimization does not have per-workload normalization and contextual information; hence it can only impute response time variations to measurement noise, resulting in a random behaviour.*

CHAPTER 6

Online Autotuning

We now move to the evaluation of online tuning scenarios, where the autotuner works directly on a production deployment subject to continuously varying workload conditions. Here we evaluate three components of CC-GP: the contextual clustering (introduce in Section 3.6), the context forecaster (introduce in Section 3.7) and the experiment scheduler (introduce in Section 3.7.1).

We start by extending the experiments conducted in the previous chapter to the continuous versions of the workload patterns (introduce in Section 4.2.3), so to test the clustering component. We do this without including the context forecasting component, keeping the oracle forecaster we used in the previous chapter.

We then move to evaluate the forecaster and scheduling components.

6.1 Workload Tracking

To evaluate the ability of the autotuners to adapt to a continuously varying workload, we switch to the continuous versions of the Ramp and Peaks workload patterns (Figure 4.4). For now, we do not include the workload forecasting component, and use an oracle predictor allowing the tuning module to perfectly know the upcoming workload, to evaluate the clustering component.

We report the results in Table 6.1. As visible, whether enabling the clustering component is beneficial depends on the specific tuning problem. However, as explained in Section 3.6, CC-GP automatically decides whether to enable or not the clustering step and, as visible from the result, it always makes the correct decision.

Furthermore, CC-GP is the best autotuner in all the considered scenarios according to all the evaluated metrics, both with or without the clustering. Comparing these results with the ones reported in Table 5.10, we see that CC-GP is reaching similar per-

formances (even slighlty better), whereas the other autotunes have gotten significantly worse by moving on the continuous workload patterns.

Table 6.1: *Synthetic results for the online tuning of Cassandra, MongoDB and JPetsore exposed to the continuous ramp and peaks workload scenarios. Best tuners per scenario in bold using Welch's t-test with 5% p-value over 16 repetitions and 300 tuning iterations. Enabling the clustering step is beneficial only on some tuning problems, as correctly identified by CC-GP.*

Scenario	Random	OT	BC	BO	CC-GP cluster off	CC-GP cluster on	CC-GP
Cassandra Ramp	0.13 ± 0.02	0.20 ± 0.09	0.27 ± 0.03	0.14 ± 0.09	$\mathbf{0.46 \pm 0.02}$	0.45 ± 0.02	$\mathbf{0.45 \pm 0.02}$
Cassandra Peaks	0.12 ± 0.01	0.24 ± 0.13	0.29 ± 0.04	0.09 ± 0.09	$\mathbf{0.46 \pm 0.03}$	$\mathbf{0.45 \pm 0.02}$	$\mathbf{0.45 \pm 0.02}$
MongoDB Ramp	0.52 ± 0.01	0.60 ± 0.11	0.48 ± 0.07	0.53 ± 0.02	0.81 ± 0.01	$\mathbf{0.86 \pm 0.01}$	$\mathbf{0.85 \pm 0.01}$
MongoDB Peaks	0.53 ± 0.02	0.62 ± 0.06	0.52 ± 0.04	0.54 ± 0.02	0.82 ± 0.02	$\mathbf{0.85 \pm 0.01}$	$\mathbf{0.86 \pm 0.01}$
JPetstore Ramp	-0.09 ± 0.02	0.21 ± 0.10	0.03 ± 0.09	0.02 ± 0.03	$\mathbf{0.62 \pm 0.03}$	0.55 ± 0.03	$\mathbf{0.62 \pm 0.03}$
JPetstore Peaks	-0.09 ± 0.03	0.25 ± 0.10	0.04 ± 0.07	0.04 ± 0.02	$\mathbf{0.68 \pm 0.03}$	0.63 ± 0.05	$\mathbf{0.68 \pm 0.03}$

6.2 Workload Forecasting and Experiment Scheduling

The goal of the workload forecasting component is to provide the contextual tuner with a reliable workload estimate and to schedule as many experiments as possible on stable windows so to let the autotuner explore more candidate configurations. As we have already evaluated the contextual autotuner, here we focus only on the forecasting and scheduling module. As already said in Section 3.7, we select Prophet mainly due to its analyst-in-the-loop approach, which allows performance experts to tune forecasting parameters. Nonetheless, here we show that Prophet produces state-of-the-art predictions.

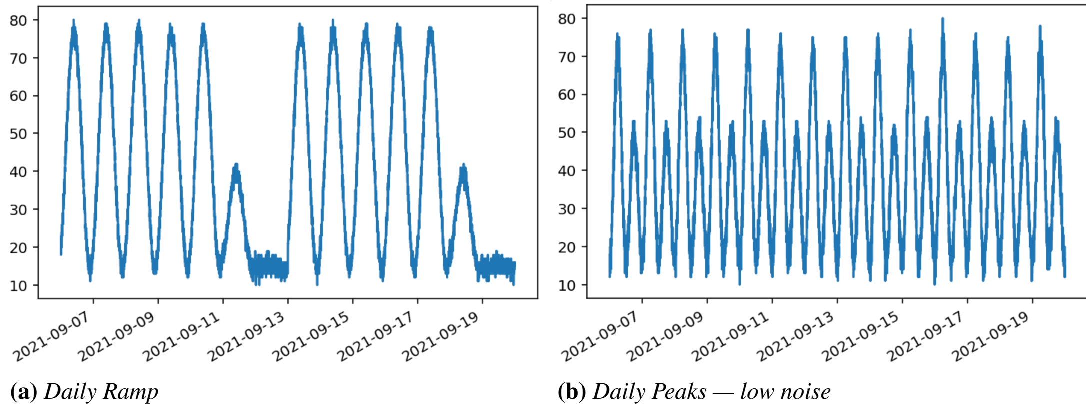
6.2.1 Considered Workloads

To evaluate the forecasters, we select two time series representing different workload patterns, similarly to the Ramp and Peaks patterns we use to evaluate the entire autotuner. However, we modify the two patterns to obtain longer and noisier time series, so to better evaluate the forecasting capabilities.

From the Ramp pattern (Figure 4.3b), we derive the Daily Ramp pattern (Figure 6.1a), which represents a two weeks period of daily ramps, with lower load levels during the weekend. From the Peaks pattern (Figure 4.3c) we derive the Daily Peaks pattern (Figure 6.1b), a two week period with daily peaks and a superimposed noisy component. To better evaluate noise resiliency, we consider three variants of the Daily Peaks pattern, with increasing levels of noise, as visible in Figure 6.2.

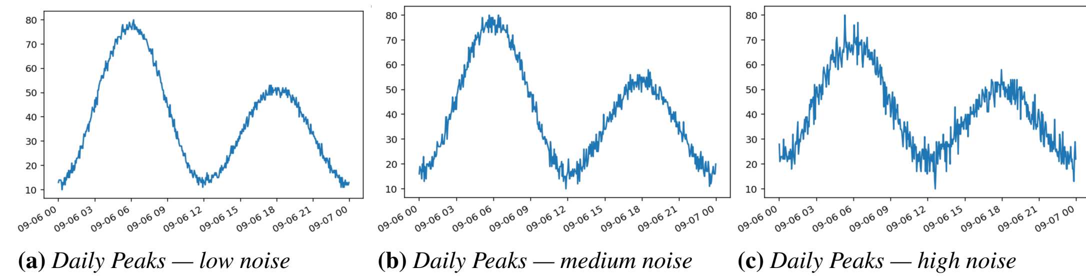
6.2.2 Workload Forecasting

Here we evaluate the considered time series forecasters (Section 4.4) in terms of MAPE and RMSE (Section 4.3.5). Every forecaster produces a prediction every 30 minutes, and is trained on a 1 week period.



(a) *Daily Ramp* **(b)** *Daily Peaks — low noise*

Figure 6.1: *Workload patterns used to evaluate time-series forecasters.*



(a) *Daily Peaks — low noise* **(b)** *Daily Peaks — medium noise* **(c)** *Daily Peaks — high noise*

Figure 6.2: *One day of the Daily Peaks workload pattern with increasing noise levels.*

As visible in Figure 6.3, Prophet has the lowest prediction error and produces good forecasts already from the first iterations.

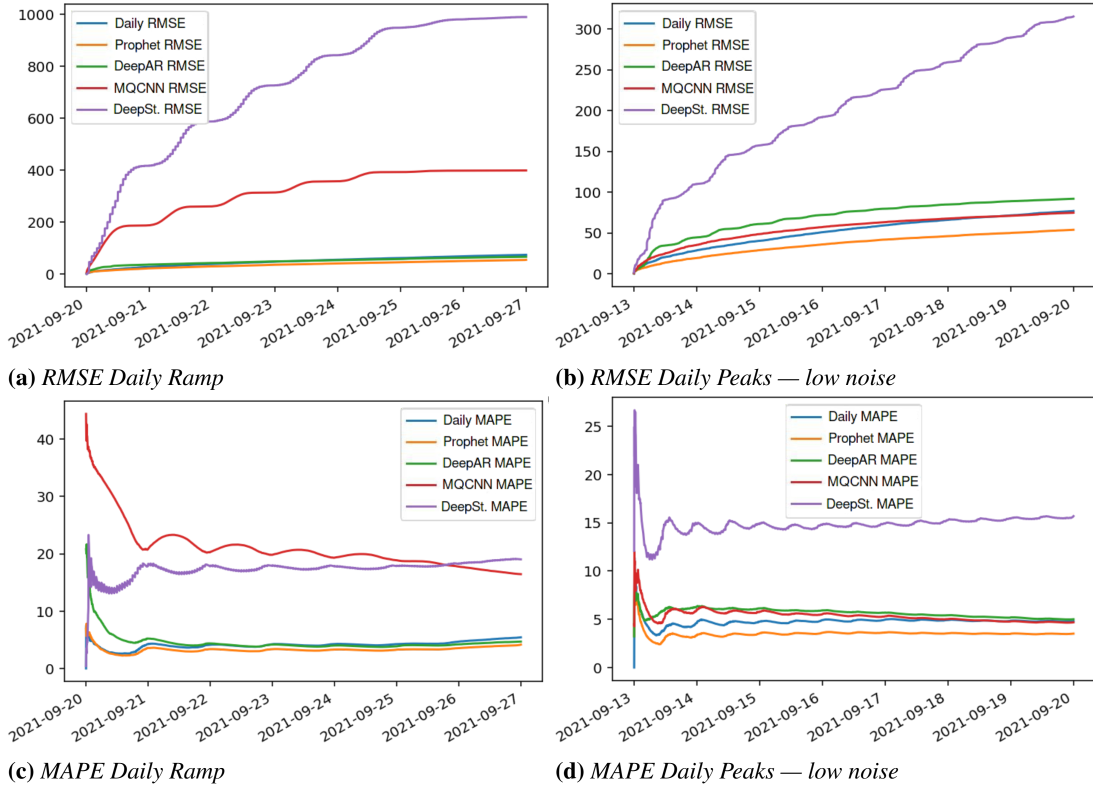
In Tables 6.2 and 6.3 we report the final RMSE and MAPE values. Again, Prophet provides the best predictions. Also, looking at the three variations of the Daily Peaks scenario, it is resilient to noise. DeepAR is the second-best, and, in more complex scenarios, it could provide better results due to its deep nature. However, this would come at the expense of the explainability provided by Prophet.

Table 6.2: *Final RMSE of timeseries forecaster. Best result in bold.*

Scenario	Daily	Prophet	DeepAR	MQCNN	DeepState
Daily Ramp	73.52	**54.32**	65.93	398.02	988.35
Daily Peaks — low	76.85	**53.76**	91.82	74.67	315.33
Daily Peaks — medium	127.10	**90.40**	138.40	183.99	270.68
Daily Peaks — high	213.15	**154.55**	165.33	212.58	1841.80

6.2.3 Experiment Scheduling

To evaluate the contribution of the forecasting module to the CC-GP autotuner, we use the Daily Ramp and Daily Peaks workload scenarios for MongoDB and Cassandra, comparing a naive version of CC-GP with the complete one using the workload forecasting and window stability modules. The naive CC-GP has an oracle forecaster, so it always knows the upcoming workload, and evaluates a new configuration at each



(a) *RMSE Daily Ramp*

(b) *RMSE Daily Peaks — low noise*

(c) *MAPE Daily Ramp*

(d) *MAPE Daily Peaks — low noise*

Figure 6.3: *RMSE and MAPE of timeseries forecaster on the Daily Ramp (low noise) and Daily Peaks workload patterns. A prediction is made every 30 minutes, and the predictors are trained with 1 week of data.*

Table 6.3: *Final MAPE of timeseries forecaster. Best result in bold.*

Scenario	Daily	Prophet	DeepAR	MQCNN	DeepState
Daily Ramp	5.43	**4.16**	4.69	16.44	19.03
Daily Peaks — low	4.81	**3.51**	5.00	4.67	15.68
Daily Peaks — medium	7.23	**5.22**	7.25	9.92	12.70
Daily Peaks — high	11.67	**8.51**	8.87	10.86	52.76

iteration. Conversely, CC-GP at each iteration produces a forecast and evaluates the stability of the upcoming window. If the window is predicted as stable, it evaluates a new configuration and observes the real workload. If the real workload is unstable (so the prediction was wrong), it stops the experiment and goes back to the best configuration observed in the past, without collecting further data so to avoid giving noisy measurements to the optimisation module. Instead, if the workload is indeed stable, it completes the experiment, measures the actual value of the workload and gives the collected values to the optimisation module to go on with the next iteration.

Furthermore, the naive tuner immediately starts to tune the system, whereas CC-GP waits one week to initialise the forecaster and the stability thresholds. We run the experiments for two weeks, so naive has twice the amount of time to tune the system.

As visible in Table 6.4, CC-GP (especially when coupled with Prophet) outperforms the naive tuner on all the four considered scenarios in terms of online optimality. This

Table 6.4: *Online optimality on MongoDB. Best result in bold.*

Scenario	Naive	CC-GP Prophet	CC-GP DeepAR
Daily Peaks — low	0.31	**0.3875**	0.25
Daily Peaks — mid	0.30	0.39	**0.41775**
Daily Peaks — high	0.31	**0.41475**	0.40
Daily Ramps	0.20	**0.24275**	0.20

is an impressive result considering that CC-GP has only half of the iteration to begin with, and this number reduces even more as it takes stability into consideration.

Nonetheless, running experiments with this many iterations is prohibitive, as the complexity of fitting a Gaussian Process scales cubically with the number of considered iterations. As we will see in the next chapter, this is probably not the best way to run an autotuner, as keeping to modifying the configuration is still a very risky process, and human supervision might be required when dealing with production systems. Hence, we could instead try to produce a configuration that is good on all the observed workloads, without trying to continuously adjust it. However, if one desires to run the autotuner so to track the evolving workload, CC-GP is indeed capable of producing good suggestions. In such a scenario, further developments are required to reduce the complexity of the algorithm. As an example, many extensions to the Bayesian Optimisation framework have already been proposed to reduce this computational burden via approximation techniques, such as Sparse Pseudoinput Gaussian Processes [96, 98, 106], Sparse Spectrum Gaussian Processes [64], Random Forests [67] and even deep neural networks [100]. Such approaches should be extended to deal with contextual information.

CHAPTER 7

Constraints

The tuning scenarios considered so far are quite simplistic, as we have mainly tried to reduce the response time. In a real scenario, however, the desiderata of the tuning process are more complex, as we usually have some SLAs to guarantee. In this chapter, we modify the online tuning scenarios to include some constraints and evaluate the ability of CC-GP to satisfy them, especially during the tuning process. As a comparison, we use OpenTuner, BestConfig and BayesianOptimization as we did so far, modifying the score to give a bad value to the experiments which violate some constraint. We also use our implementation of CC-GP, removing the constraint handling and weighting the acquisition function by a constraint satisfaction probability as predicted by a Gaussian Process Classifier, as proposed in constrained EI [46].

7.1 Combined Goal Function

An easy way to drive the autotuner to respect some given constraints is to embed them in the scoring function, trying to balance, as an example, the cost reduction with the performance degradation. In [21] we reduced the memory consumption M of Cassandra and MongoDB without increasing the response time R by combining them in the score function.

For both the DBMSs, we thus minimised the following function:

$$M[\mathrm{MiB}] + \sigma \cdot R[\mathrm{ms}] \tag{7.1}$$

where the term σ is the penalty coefficient. The choice of σ depends both on the employed measurement units and on the importance of the constraint. As we measured R in ms and M in MiB, we ran most of the experiments using $\sigma = 10^3$, essentially saying that each additional millisecond costs as much as $1\,\mathrm{GiB}$ of memory. We found

Table 7.1: *Offline tuner optimality. σ is the response time penalty coefficient. Best tuners per scenario in bold using Welch's t-test with 1% p-value over 16 repetitions.*

$\sigma[\frac{\text{MiB}}{\text{ms}}]$	Scenario	Average IB score				
		Random	BO	OT	BC	CC-GP
1×10^1	Cassandra Ramp	0.00 ± 0.02	0.45 ± 0.20	0.72 ± 0.18	0.70 ± 0.20	$\mathbf{0.89 \pm 0.03}$
	Cassandra Peaks	0.01 ± 0.03	0.33 ± 0.15	0.68 ± 0.13	0.72 ± 0.16	$\mathbf{0.81 \pm 0.09}$
	MongoDB Ramp	0.33 ± 0.10	$\mathbf{0.95 \pm 0.08}$	0.78 ± 0.14	$\mathbf{0.92 \pm 0.01}$	$\mathbf{0.94 \pm 0.05}$
	MongoDB Peaks	0.31 ± 0.06	$\mathbf{0.93 \pm 0.06}$	0.77 ± 0.06	$\mathbf{0.92 \pm 0.02}$	$\mathbf{0.94 \pm 0.02}$
1×10^3	Cassandra Ramp	0.02 ± 0.03	0.14 ± 0.09	0.36 ± 0.12	0.19 ± 0.08	$\mathbf{0.63 \pm 0.04}$
	Cassandra Peaks	0.04 ± 0.04	0.15 ± 0.09	0.36 ± 0.12	0.24 ± 0.08	$\mathbf{0.57 \pm 0.08}$
	MongoDB Ramp	0.30 ± 0.07	0.30 ± 0.17	0.58 ± 0.10	$\mathbf{0.69 \pm 0.10}$	$\mathbf{0.73 \pm 0.05}$
	MongoDB Peaks	0.27 ± 0.07	0.44 ± 0.19	0.60 ± 0.07	0.60 ± 0.08	$\mathbf{0.75 \pm 0.04}$
1×10^4	Cassandra Ramp	0.42 ± 0.03	0.41 ± 0.06	0.60 ± 0.05	0.53 ± 0.03	$\mathbf{0.66 \pm 0.03}$
	Cassandra Peaks	0.44 ± 0.04	0.41 ± 0.04	0.59 ± 0.05	0.53 ± 0.03	$\mathbf{0.65 \pm 0.04}$
	MongoDB Ramp	0.51 ± 0.04	0.57 ± 0.07	0.63 ± 0.06	0.64 ± 0.05	$\mathbf{0.71 \pm 0.05}$
	MongoDB Peaks	0.59 ± 0.04	0.63 ± 0.07	0.65 ± 0.06	0.66 ± 0.05	$\mathbf{0.75 \pm 0.03}$
1×10^5	Cassandra Ramp	0.54 ± 0.02	0.52 ± 0.03	0.63 ± 0.05	0.58 ± 0.03	$\mathbf{0.70 \pm 0.03}$
	Cassandra Peaks	0.54 ± 0.03	0.53 ± 0.03	0.65 ± 0.05	0.58 ± 0.03	$\mathbf{0.68 \pm 0.03}$
	MongoDB Ramp	0.82 ± 0.02	0.82 ± 0.02	0.82 ± 0.03	0.75 ± 0.04	$\mathbf{0.86 \pm 0.02}$
	MongoDB Peaks	0.83 ± 0.01	0.83 ± 0.02	0.83 ± 0.02	0.77 ± 0.04	$\mathbf{0.88 \pm 0.02}$

this function to be reasonably balanced, and the resulting optimisation problem is a difficult one, where the suggested configurations must be adapted to the different workloads. For completeness, we also ran the experiments using $\sigma = 10^1, \sigma = 10^4$ and $\sigma = 10^5$, where each millisecond cost $10\,\text{MiB}, 10\,\text{GiB}$ or $100\,\text{GiB}$ respectively. The resulting problems are much more unbalanced toward decreasing either the memory or the response time.

We measured the average response time in milliseconds, as reported by YCSB. For MongoDB, we used the `wiredTigerCacheSizeGB` parameter to measure M, multiplied by 1024 so to measure it in MiB. On Cassandra, we used the sum of `file_cache_size_in_mb` and JVM max heap size (`Xmx`), both measured in MiB.

Notice that the memory parameter is one of the tunable parameters, but the considered tuners have no access to the metric formula and, thus, they need to discover this link. Whichever goal function we select (even a much simpler minimise R as we did in the experiments reported above) we would still be using a function of the applied parameters, which, however, would be unknown to both us and the tuners. Conversely, by using the actual memory parameters we can implement a sanity check on the tuning results by checking their values.

Detailed results are available in [21], here we only report the synthetic ones, in terms of Offline and Online optimality in Tables 7.1 and 7.2. We report the results for different values of σ, which is the response time penalty coefficient as per Equation (7.1).

In the majority of the considered scenarios, CC-GP is the best tuner. Only BO and BestConfig reach similar offline optimality scores in the simplest scenarios. As we anticipated, in fact, $\sigma = 1 \times 10^3$ produces the most difficult problem, as testified by the lower Online Optimality reached by all the tuners. Finally, notice that even Random has a positive average Offline Optimality score on most of the scenarios, confirming that, if we can afford to run many iterations and testing bad configurations is not a problem, a simple Random approach finds interesting configurations, which confirms our previous

Table 7.2: *Online tuner optimality. σ is the response time penalty coefficient. Best tuners per scenario in bold using Welch's t-test with 1% p-value over 16 repetitions.*

		Average NPI score				
$\sigma[\frac{\text{MiB}}{\text{ms}}]$	Scenario	Random	BO	OT	BC	CC-GP
1×10^1	Cassandra Ramp	-1	-0.90 ± 0.05	-0.01 ± 0.12	0.01 ± 0.17	$\mathbf{0.70 \pm 0.07}$
	Cassandra Peaks	-0.99 ± 0.01	-0.94 ± 0.03	0.01 ± 0.09	-0.38 ± 0.16	$\mathbf{0.47 \pm 0.15}$
	MongoDB Ramp	-0.92 ± 0.02	0.13 ± 0.37	0.18 ± 0.13	0.07 ± 0.21	$\mathbf{0.74 \pm 0.11}$
	MongoDB Peaks	-0.92 ± 0.01	-0.14 ± 0.26	0.21 ± 0.10	-0.04 ± 0.18	$\mathbf{0.76 \pm 0.04}$
1×10^3	Cassandra Ramp	-0.98 ± 0.01	-0.96 ± 0.02	-0.47 ± 0.13	-0.57 ± 0.10	$\mathbf{0.14 \pm 0.10}$
	Cassandra Peaks	-0.98 ± 0.01	-0.95 ± 0.02	-0.44 ± 0.13	-0.52 ± 0.10	$\mathbf{0.08 \pm 0.11}$
	MongoDB Ramp	-0.91 ± 0.02	-0.91 ± 0.06	-0.09 ± 0.13	-0.07 ± 0.15	$\mathbf{0.16 \pm 0.06}$
	MongoDB Peaks	-0.91 ± 0.02	-0.87 ± 0.13	-0.16 ± 0.12	-0.12 ± 0.10	$\mathbf{0.15 \pm 0.07}$
1×10^4	Cassandra Ramp	-0.69 ± 0.03	-0.73 ± 0.05	-0.25 ± 0.15	-0.28 ± 0.11	$\mathbf{0.32 \pm 0.04}$
	Cassandra Peaks	-0.68 ± 0.03	-0.73 ± 0.03	-0.20 ± 0.24	-0.27 ± 0.07	$\mathbf{0.34 \pm 0.04}$
	MongoDB Ramp	-0.43 ± 0.03	-0.37 ± 0.05	0.01 ± 0.08	0.04 ± 0.08	$\mathbf{0.35 \pm 0.07}$
	MongoDB Peaks	-0.42 ± 0.02	-0.33 ± 0.07	0.05 ± 0.09	0.10 ± 0.10	$\mathbf{0.34 \pm 0.05}$
1×10^5	Cassandra Ramp	-0.53 ± 0.04	-0.57 ± 0.07	0.23 ± 0.25	0.07 ± 0.13	$\mathbf{0.48 \pm 0.06}$
	Cassandra Peaks	-0.52 ± 0.04	-0.58 ± 0.05	0.24 ± 0.20	0.18 ± 0.11	$\mathbf{0.44 \pm 0.03}$
	MongoDB Ramp	0.46 ± 0.02	0.46 ± 0.05	0.58 ± 0.03	0.58 ± 0.09	$\mathbf{0.63 \pm 0.02}$
	MongoDB Peaks	0.49 ± 0.02	0.47 ± 0.04	0.61 ± 0.03	0.61 ± 0.04	$\mathbf{0.66 \pm 0.02}$

claim that comparing the best configurations found is not the proper way to compare autotuners.

7.1.1 Suggested Configurations with Combined Goal

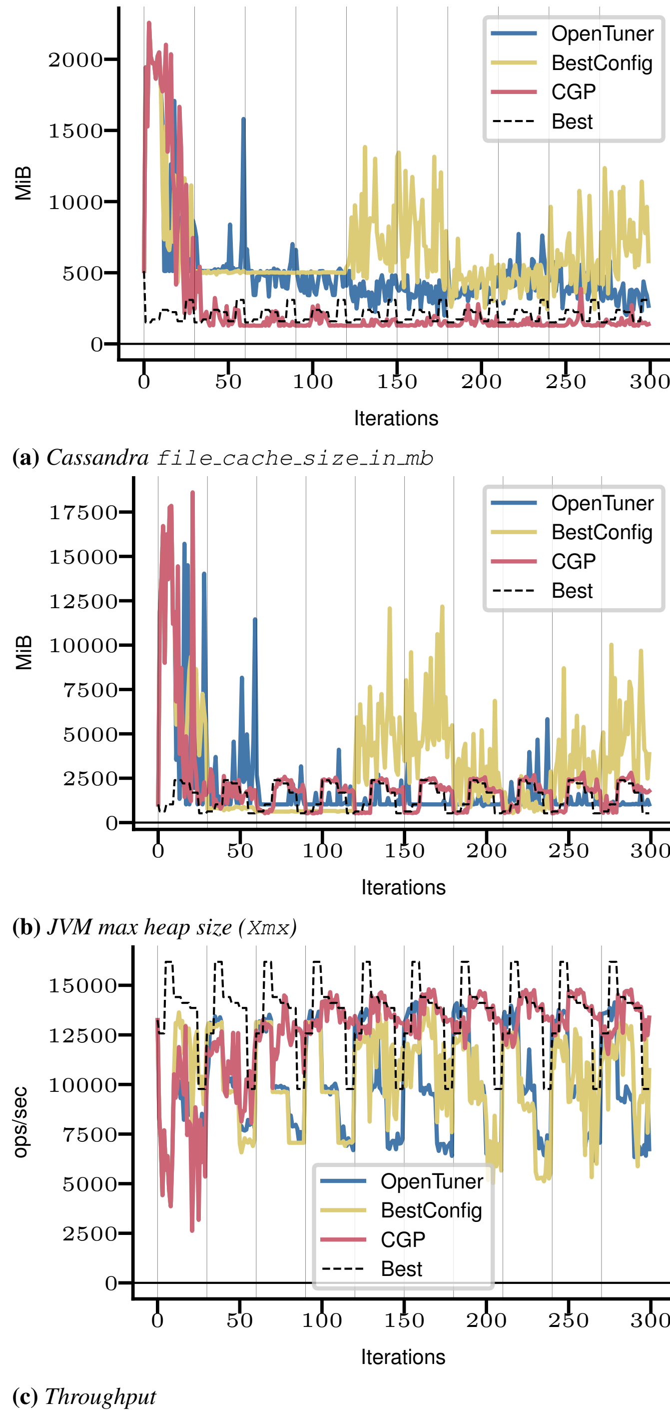
To give a useful insight, we report in Figure 7.1 the non-normalised values of the throughput and memory consumption used as a score for the optimisation. More specifically, we consider the Cassandra Ramp scenario and report the median value obtained by each tuner.

We take the median over the repetitions in order to take into account the tuner variability.

We do not use the value of the best configurations found as they would be misleading: with a proper number of iterations even a Random tuner can find good configurations (as testified by Table 7.1) and taking a single configuration over 50 tested ones would produce very noisy results.

After the two first tuning days CC-GP substantially stops exploring: it settles on a low value for the Cassandra cache and varies the JVM heap's size depending on the incoming workload, increasing it only when necessary. Both the parameters are very similar to the best configuration available in the dataset. Both OpenTuner and BestConfig are not converging, and, in general, they are exploring higher values for both the memory parameters. As for the throughput, CC-GP achieves better results on most of the workloads.

Notice that, under some workloads, the throughput of CC-GP is higher than the best one. Under the same workloads, however, CC-GP selects more memory than necessary. According to the selected goal function, the increase in throughput is not enough to justify the increased memory consumption. Furthermore, there are others, more problematic, workloads where CC-GP selects a suboptimal throughput to save more memory, which could be a problem in terms of response time.



(a) *Cassandra* `file_cache_size_in_mb`

(b) *JVM max heap size* (`Xmx`)

(c) *Throughput*

Figure 7.1: *Unnormalised Memory and Throughput for the Cassandra Ramp tuning, Best refers to the best configuration available in the collected dataset according to the goal function.*

Therefore, by combining the memory consumption with the response time in the scoring function we can find configurations which balance the two conflicting requirements. This, however, is very hard to do as it requires a careful balancing of the two components via weights which are scale-dependant, and thus should be manually tuned at each new tuning session. Furthermore, asking the autotuner to reduce the response

time is not equivalent at all to a constraint.

7.2 SLA Constraints

To evaluate the ability of the autotuners to respect the given constraints, we run a series of experiments on the Cassandra and MongoDB dataset-based simulators. We use the continuous versions of the Ramp and Peaks workload patterns already used in Section 6.1 but modify the goal of the tuning process, aiming at an efficiency goal.

For Cassandra, we aim at minimising the occupied memory, expressed as the sum of the Cassandra cache and JVM heap, subject to a constraint on the response time, which must be below $10\,\mathrm{ms}$. For MongoDB, we minimise the cache size, with the same constraint on the response time. For JPetstore, we minimise the 99th percentile of the response time, while keeping the CPU utilisation (as measured by the OS) below 60%.

Here we are mostly evaluating the constraint handling capabilities; hence we remove BayesianOptimization from the comparison, and insert wAF (weighted Acquisition Function). wAF uses the same code of CC-GP without the constraint handling section. Instead, it uses a Gaussian Process classifier to predict the probability of satisfying the constraints, and then it uses the probability as a weight for the Acquisition Function, just like wEI (Section 3.2.1). Existing approaches which weigh the acquisition function by the satisfaction probability aim at balancing exploration with constraint satisfaction. We do not call this approach wEI as we use the EI AF, and extend the weighting approach to the PI and LCB AFs, handling them with the GP-hedge approach exactly like in CC-GP.

We report the results in Table 7.3. As visible, CC-GP always has the best Online Optimality, which means that it is both converging to good values and avoiding constraint violations (Section 4.3.4). Also, apart from MongoDB, it consistently has the lowest number of violations. On the other hand, on MongoDB, the online optimality is substantially higher than the other tuners, which have a lower number of violations only because they do not tune at all.

The explicit constraint management of CC-GP allows it to outperform the weighting approach, both in terms of optimality and constraint violations.

In Table 7.4, we report the results for a similar set of experiments where the constraints are much harder to satisfy. Namely, the response time for Cassandra and MongoDB has to be lower than $8\,\mathrm{ms}$ (instead of $10\,\mathrm{ms}$), and the CPU consumption of JPetstore must be below 40% (instead of 60%). For JPetstore, the constraint is actually not satisfiable under the highest sections of the workload. CC-GP still has the best Online Optimality.

7.2.1 Constraints vs Combined Goal Function

In Figure 7.2 we report the selected value for the two memory parameters and the resulting response time achieved during the tuning of Cassandra under the Continuous Ramp workload pattern with the goal of minimising the memory consumption while keeping the response time below $8\,\mathrm{ms}$ (harder constraint).

Apart from the occasional exploratory spikes, the memory parameters remain low, and, more importantly, the response time remains below the threshold most of the time,

Table 7.3: *Online optimality and normalised cumulated number of constraint violations for the tuning of Cassandra, MongoDB and Jpetstore under the continuous Ramp and Peaks workload scenarios with 150 tuning iterations. Best tuners per scenario in bold using Welch's t-test with 5% p-value over 16 repetitions.*

SUT	Scenario	Metric	Random	OT	BC	CC-GP	wAF
Cassandra	C. Ramp	OnOpt	-0.65 ± 0.03	0.04 ± 0.16	0.18 ± 0.21	$\mathbf{0.80 \pm 0.08}$	0.41 ± 0.16
		Viol	0.34 ± 0.03	0.12 ± 0.07	$\mathbf{0.07 \pm 0.05}$	$\mathbf{0.05 \pm 0.02}$	0.19 ± 0.10
	C. Peaks	OnOpt	-0.68 ± 0.02	-0.00 ± 0.15	0.04 ± 0.14	$\mathbf{0.78 \pm 0.04}$	0.07 ± 0.23
		Viol	0.40 ± 0.04	0.18 ± 0.07	0.13 ± 0.02	$\mathbf{0.06 \pm 0.02}$	0.35 ± 0.13
MongoDB	C. Ramp	OnOpt	-0.48 ± 0.03	0.04 ± 0.06	-0.01 ± 0.04	$\mathbf{0.66 \pm 0.04}$	0.48 ± 0.05
		Viol	$\mathbf{0.00 \pm 0.01}$	0.06 ± 0.02	0.02 ± 0.01	0.13 ± 0.02	0.21 ± 0.02
	C. Peaks	OnOpt	-0.48 ± 0.02	0.09 ± 0.06	-0.00 ± 0.04	$\mathbf{0.80 \pm 0.05}$	0.71 ± 0.04
		Viol	$\mathbf{0.00}$	0.03 ± 0.01	0.01 ± 0.01	0.06 ± 0.01	0.09 ± 0.02
JPetstore	C. Ramp	OnOpt	-0.13 ± 0.04	0.27 ± 0.13	0.00 ± 0.08	$\mathbf{0.46 \pm 0.05}$	$\mathbf{0.43 \pm 0.03}$
		Viol	0.03 ± 0.01	$\mathbf{0.01 \pm 0.01}$	$\mathbf{0.00 \pm 0.01}$	$\mathbf{0.00}$	$\mathbf{0.00 \pm 0.01}$
	C. Peaks	OnOpt	-0.13 ± 0.05	0.27 ± 0.13	-0.03 ± 0.10	$\mathbf{0.55 \pm 0.06}$	$\mathbf{0.54 \pm 0.06}$
		Viol	0.02 ± 0.01	$\mathbf{0.01 \pm 0.01}$	$\mathbf{0.00 \pm 0.01}$	$\mathbf{0.00}$	$\mathbf{0.00}$

Table 7.4: *Online optimality and normalised cumulated number of constraint violations for the tuning of Cassandra, MongoDB and Jpetstore under the continuous Ramp and Peaks workload scenarios with 150 tuning iterations and hard constraints. Best tuners per scenario in bold using Welch's t-test with 5% p-value over 8 repetitions.*

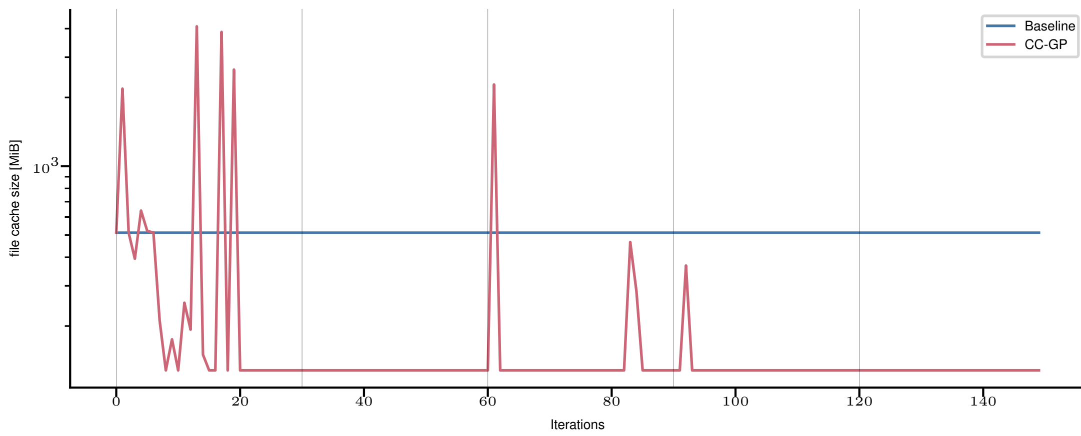
SUT	Scenario	Metric	Random	OT	BC	CC-GP	wAF
Cassandra	C. Ramp	OnOpt	-0.69 ± 0.03	-0.12 ± 0.17	0.02 ± 0.12	$\mathbf{0.67 \pm 0.06}$	0.05 ± 0.16
		Viol	0.40 ± 0.02	0.18 ± 0.07	0.17 ± 0.04	$\mathbf{0.12 \pm 0.03}$	0.41 ± 0.09
	C. Peaks	OnOpt	-0.70 ± 0.04	-0.20 ± 0.17	-0.05 ± 0.13	$\mathbf{0.64 \pm 0.07}$	-0.23 ± 0.16
		Viol	0.47 ± 0.04	0.23 ± 0.08	0.21 ± 0.02	$\mathbf{0.13 \pm 0.03}$	0.53 ± 0.09
MongoDB	C. Ramp	OnOpt	-0.38 ± 0.03	-0.10 ± 0.06	-0.06 ± 0.04	$\mathbf{0.46 \pm 0.04}$	0.28 ± 0.03
		Viol	$\mathbf{0.11 \pm 0.02}$	0.20 ± 0.02	0.15 ± 0.02	0.23 ± 0.02	0.31 ± 0.01
	C. Peaks	OnOpt	-0.46 ± 0.03	-0.02 ± 0.05	-0.03 ± 0.04	$\mathbf{0.63 \pm 0.03}$	0.51 ± 0.03
		Viol	$\mathbf{0.05 \pm 0.01}$	0.08 ± 0.02	0.06 ± 0.02	0.15 ± 0.01	0.20 ± 0.02
JPetstore	C. Ramp	OnOpt	-0.30 ± 0.03	-0.02 ± 0.17	-0.23 ± 0.04	$\mathbf{0.27 \pm 0.06}$	$\mathbf{0.24 \pm 0.05}$
		Viol	0.29 ± 0.01	0.23 ± 0.05	0.27 ± 0.02	$\mathbf{0.18 \pm 0.02}$	0.23 ± 0.02
	C. Peaks	OnOpt	-0.25 ± 0.06	0.09 ± 0.13	-0.32 ± 0.07	$\mathbf{0.31 \pm 0.04}$	0.24 ± 0.08
		Viol	0.33 ± 0.03	0.23 ± 0.03	0.34 ± 0.02	$\mathbf{0.20 \pm 0.01}$	0.24 ± 0.03

and the amount of violations decreases over time, indicating that the autotuner is learning how to respect the constraints. Notice that the amount of violations is coherent with the measurement reported in Table 7.4. Finally, compare the behaviour of the autotuner in the constrained scenario with the one achieved with the combined goal function, reported in Figure 7.1. Especially in the first fifty iterations, the constrained autotuner causes way less performance degradation.

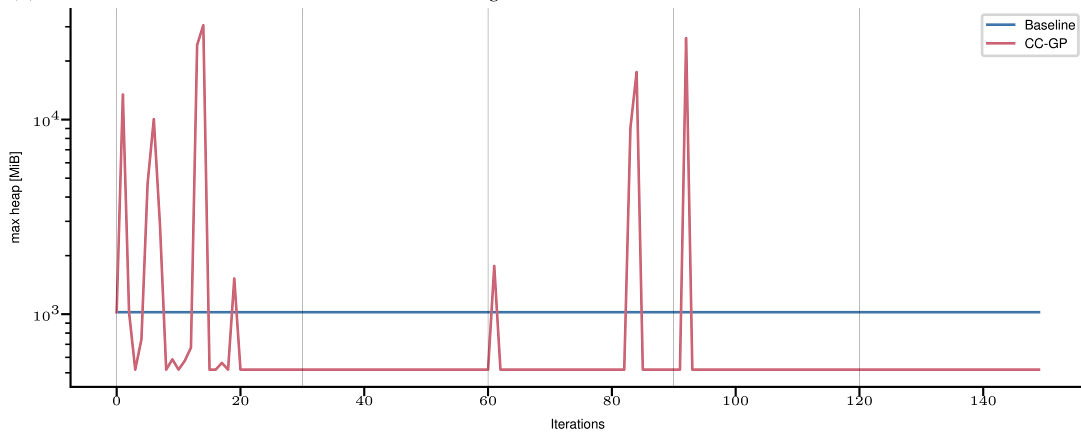
7.2.2 Local vs Global Safety

In the previous section, we have evaluated the ability of an autotuner to respect the constraints when the workload is perfectly known, as we are using an oracle forecaster. In reality, however, the forecasting could be wrong. In that case, we might desire to have an extra layer of safety, and force the autotuner to suggest configurations expected to respect constraints on all the observed workloads.

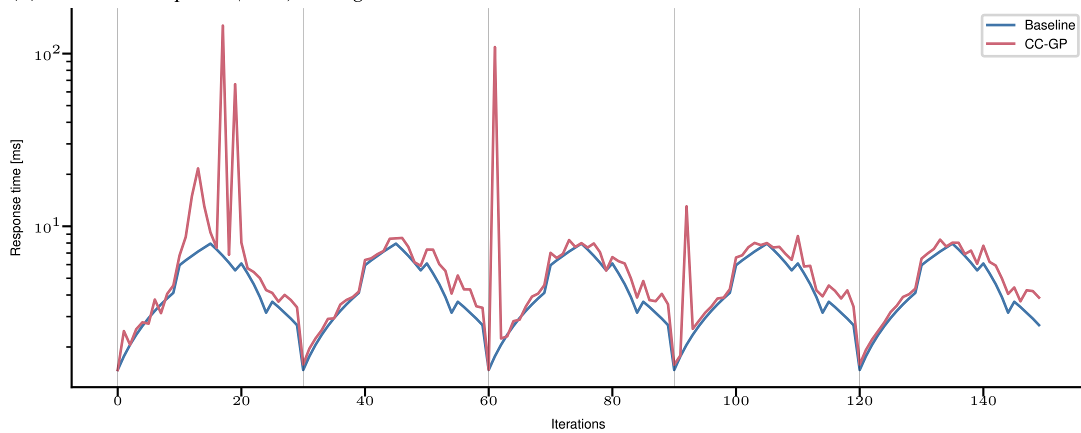
To compare local and global safety (Section 3.8.2), we modify the forecaster to al-



(a) *Cassandra* `file_cache_size_in_mb` *— Log axis*



(b) *JVM max heap size (*`Xmx`*) — Log axis*



(c) *Response time — Log axis*

Figure 7.2: *Unnormalised Memory and Response time for the Cassandra Continuous Ramp tuning with hard constraints, Best refers to the best configuration available in the collected dataset according to the goal function.*

ways predict the past (instead of next) workload, which is the simplest way to produce a forecast and repeat the constrained tunings. We report the results in Table 7.5. By forcing the autotuner to suggest a configuration that is safe for all the observed workloads, and not only on the next one, we could expect the performance of the autotuner to degrade, as it cannot tailor the configuration to the actual context. Instead, the two autotuners are statistically equivalent.

Table 7.5: *Effect of local and global safety. Online optimality and normalised cumulated number of constraint violations for the constrained tuning of Cassandra, MongoDB and Jpetstore under the continuous Ramp and Peaks workload scenarios with 100 tuning iterations. Best tuners per scenario in bold using Welch's t-test with 5% p-value over 8 repetitions.*

SUT	Scenario	Metric	Local	Global
Cassandra	C. Ramp	OnOpt	$\mathbf{0.66 \pm 0.03}$	$\mathbf{0.65 \pm 0.04}$
		Viol	$\mathbf{0.07 \pm 0.02}$	$\mathbf{0.06 \pm 0.02}$
	C. Peaks	OnOpt	$\mathbf{0.70 \pm 0.05}$	$\mathbf{0.69 \pm 0.11}$
		Viol	$\mathbf{0.08 \pm 0.02}$	$\mathbf{0.06 \pm 0.02}$
MongoDB	C. Ramp	OnOpt	$\mathbf{0.62 \pm 0.13}$	$\mathbf{0.60 \pm 0.12}$
		Viol	$\mathbf{0.12 \pm 0.02}$	$\mathbf{0.11 \pm 0.01}$
	C. Peaks	OnOpt	$\mathbf{0.75 \pm 0.05}$	$\mathbf{0.75 \pm 0.05}$
		Viol	$\mathbf{0.06 \pm 0.01}$	$\mathbf{0.05 \pm 0.01}$
JPetstore	C. Ramp	OnOpt	$\mathbf{0.50 \pm 0.06}$	$\mathbf{0.50 \pm 0.06}$
		Viol	$\mathbf{0.00}$	$\mathbf{0.00}$
	C. Peaks	OnOpt	$\mathbf{0.55 \pm 0.06}$	$\mathbf{0.56 \pm 0.05}$
		Viol	$\mathbf{0.00}$	$\mathbf{0.00}$

CHAPTER 8

Reaction-Matching Characterisation

So far we have mainly focused on real-valued search spaces, albeit we also included some categorical parameters. In some autotuning domains, however, the majority of the available parameters are categorical ones. In such scenarios, Bayesian Optimisation provides less advantages, as the information available for a certain category does not give any information on nearby categories, as we completely lack a definition of distances among different categories. Hence, in this chapter we focus on a different autotuning technique that we developed to address the problem of compiler autotuning, which focuses on boolean parameters. Furthermore, as we will discuss later, future extensions of this work will combine the Bayesian approach presented so far with the methodology exposed in this chapter.

When compiling a program from a high-level language to its executable binary, we can enable compiler optimisations which control how the code is transformed and generated, impacting the performance of the compiled binary. Selecting the proper optimisations to enable is a peculiar performance autotuning task, as the search space is mainly composed by boolean flags. In compiler autotuning, Iterative Compilation is an approach where we keep recompiling and testing a program to find an optimised configuration, which is very similar to our autotuning approach presented in previous sections. In this research field, huge attention is given to exploiting available knowledge bases: that is, we suppose to have a large collection of previously tuned applications and we try to reuse this knowledge to speed up the tuning of a new application. In [20], we proposed a technique derived from the Recommender Systems field and based on Reaction Matching: a reaction-based approach that characterises each program in terms of how different programs react to the same set of optimisation flags. Here we cover the proposed approach and the main results, and finally explain how we intend to combine the proposed approach with CC-GP to let it exploit existing knowledge bases.

The main idea of the approach is that, instead of characterising a program in terms of some code-based features or performance counters, we just look at how the various optimisation flags impact performance. When two programs benefit (i.e., improve w.r.t. the baseline) from the same optimisation, we suppose that they will have a similar reaction even to other optimisations.

RS are widely used to suggest items to users, helping them to navigate huge catalogues, like Netflix or Amazon [89]. The essential task of an RS is to predict the *relevance* $r_u(i)$ of an item i for a user u [27]. To achieve this goal, RS usually exploits similarities between items or users.

In the proposed approach, we treated the program to be tuned p as our user, and a combination of the optimisation flags x as the item within the catalogue X. Similarly to the rest of the work, $f_p(x)$ indicates the value of a certain performance indicator obtained by program p when compiled with the optimisations defined in x. We the consider the baseline set x_0 and define the relevance as:

$$r_p(x) = \frac{f_p(x)}{f_p(x_0)} - 1. \tag{8.1}$$

The relevance $r_p(x)$ reflects how much the program p benefits from the optimisations defined in x w.r.t. the baseline set x_0 (i.e., in our experiments, the predefined optimisation level `-O3` of GCC).

We suppose to have access to some previously collected information about other programs $q_0, q_1, q_2, \ldots$, where each program has been compiled and tested with a variety of optimisation sets $x_0, x_1, x_2, \ldots$. We can use RS algorithms to derive the "preferences" of p over the available x_s, and then "suggest" to the program the optimisation sets according to the preference ordering predicted by the RS algorithm.

In [20], we presented three RS-based approaches: Top Popular (TP), Content-Based Filtering (CBF) and Collaborative Filtering (CF). TP (Section 8.1) is the simplest one and simply suggests optimisations that are good for many programs in the knowledge base. CBF (Section 8.2.1) uses performance counters to characterise programs, like most of the current state-of-the-art approaches. CF (Section 8.2.2) uses a reaction-based characterisation methodology called Reaction Matching (RM).

8.1 Top Popular — TP

The simplest RS algorithm is the TP one, which just suggests popular items [27]. When optimising a program, we start by trying optimisation sets known to be effective over most the programs. The underlying assumption is the same one behind the existence of standard optimisation levels: if an optimisation set is beneficial to most programs, it is reasonable to assume that it will also be beneficial to a new program.

The TP algorithm predicts the relevance score obtained by a program p with an optimisation set x as:

$$\tilde{r}_p(x) = \tilde{r}(x) = \frac{\sum_{q \in Q} r_q(x)}{|Q|} \tag{8.2}$$

that is, the relevance does not depend on the program and is predicted as the average of the relevance scores obtained by the set x on the various programs q available in

our knowledge base Q. The knowledge base $Q = \{q_0, q_1, \dots\}$ is defined as the set of programs that we have previously explored.

	Exec time $f_q(x)$					Relevances $r_q(x)$					Average relevances $\tilde{r}(x)$
	q_0	q_1	q_2			q_0	q_1	q_2			
$x_0 : \{\neg O_0, \neg O_1\}$	3	4	2		x_0	0	0	0		x_0	0
$x_1 : \{\neg O_0, O_1\}$	1	4	1	$\overrightarrow{①}$	x_1	−0.67	0	−0.5	$\overrightarrow{②}$	x_1	−0.39
$x_2 : \{O_0, \neg O_1\}$	5	3	4		x_2	0.67	−0.25	1		x_2	0.47
$x_3 : \{O_0, O_1\}$	4	5	3		x_3	0.33	0.25	0.5		x_3	0.36

Figure 8.1: *Example of TP algorithm. We have two binary optimisation flags (O_0, O_1), for a total of 4 different optimisation sets (x_0, x_1, x_2, x_3). We evaluate all the sets on 3 different programs (q_0, q_1, q_2), compute the relevance scores and the average relevance across the programs. The TP algorithm suggests the set x_1 as the first one to try.*

A sample execution of the TP algorithm can be found in Figure 8.1. In this example, we have 2 optimisations available O_0 and O_1, which can be combined into 4 different optimisation sets x_0, x_1, x_2, x_3. We write $x_1 : \{\neg O_0, O_1\}$ to indicate that, in set x_1, optimisation O_0 is turned off and optimisation O_1 is turned on. We also have 3 programs q_0, q_1, q_2 in our knowledge base, and we know the performance value (execution time) obtained by every program with every optimisation set $f_q(x)$.

We start ① by computing the relevance scores as per Equation 8.1, starting from the measured performance values. Then ②, we apply Equation 8.2, computing the average of the relevance scores. When tuning a fourth program p, the TP algorithm suggests x_1 as the best set, and x_2 as the worst one, assuming our goal is to minimise the execution time. Indeed, x_1 is the only set that reduces the execution time of all the programs in our knowledge base and thus is a good candidate. As we are dealing with an Iterative Compilation problem, TP proceeds by suggesting x_0, x_2 and x_3.

This algorithm is an extremely simple one and, in the RS field, is often used as a baseline algorithm. TP can be viewed as the RS equivalent of standard optimisation levels, as it assumes that some sets are generally better than others.

8.2 Exploiting characterisation metrics

The TP algorithm does not require any characterisation step. However, if we have a way to characterise programs and measure their similarity, we can apply more advanced RS algorithms [78]. The formula to predict relevances in this situation is:

$$\tilde{r}_p(x) = \frac{\sum_{q \in NN_{kp}} s_{pq} r_q(x)}{\sum_{q \in NN_{kp}} s_{pq}}. \tag{8.3}$$

That is, we predict the relevance of x for p as the weighted average of the relevance scores obtained by x over the k programs q most similar to p according to a similarity measure s. We denote as NN_{kp} the set of k programs most similar to p, thus the Nearest Neighbours of p.

To measure the similarity between programs, we again take inspiration from the RS field. Generally speaking, an RS algorithm can belong to two broad categories: Content-Based Filtering (CBF) or Collaborative Filtering (CF). CBF uses items' features: if we know that a user has liked *The Lord of the Rings* we can suggest him to

watch *The Hobbit*, as the two items have a lot of common features (actors, genre, director, etc.) and are thus similar. CF, conversely, does not consider features: it finds similarities between users based on the items they liked. If we know that two users gave similar relevance scores to many items, we can conclude that they have similar taste, and thus consider them similar.

To obtain a CBF and a CF algorithm, we thus define two ways to compute similarities between programs: one based on programs' features and one based on relevance scores.

8.2.1 Content-Based Filtering — CBF

Similarly to Cobayn [6], we used MICA metrics [53] to characterise programs and then perform a PCA [55]. The similarity between two programs is computed using a distance metric between their feature vectors.

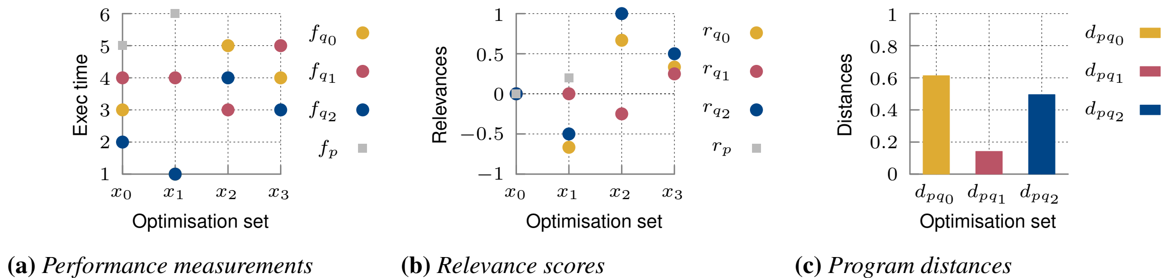
Using an Euclidean distance and calling m_p^c the c-th principal component of the MICA metrics of program p, the distance between program p and q is computed as:

$$d_{pq} = \sqrt{\frac{\sum_{c=1}^{C}\left(m_p^c - m_q^c\right)^2}{C}}. \tag{8.4}$$

Since we need a similarity measure in Equation 8.3, we just define the similarity as $s_{pq} = \frac{1}{d_{pq}}$.

8.2.2 Collaborative Filtering — CF

RM is a reaction-based characterisation methodology similar to the one proposed in [17]. We use RM to compute the distances in Equation 8.3, obtaining a CF algorithm.



(a) *Performance measurements* **(b)** *Relevance scores* **(c)** *Program distances*

Figure 8.2: *Example of RM algorithm used to compute program distances. We have three programs (q_0, q_1, q_2) and two optimisation flags (O_0, O_1) resulting in four possible combinations (x_0, x_1, x_2, x_3). We need to tune a fourth program p for which we have only evaluated x_0, x_1. Figure 8.2a represents the value of a performance metric obtained by program p when compiled with the flags specified in x, Figure 8.2b is the relevance computed with Equation 8.1 and Figure 8.2c contains the distances between program p and q_0, q_1, q_2 measured with Equation 8.5 using x_1.*

Our goal is to find a program in the knowledge base that has the same optimal set as the program we are compiling. If the two programs receive a performance boost from the same set of optimisations, we can hypothesise that they also show similar reactions to other sets. In other words, we expect the two programs to have similar code patterns so that certain optimisation sets are beneficial to both of them, while other

optimisations are detrimental, and others again have no effect at all. To measure the similarity between the two programs, we use the RM algorithm, which is represented graphically in Figure 8.2 and described in the following.

To describe how the CF method works, we use the same initial set of data we previously used for the example in Figure 8.1. Figure 8.2a plots the measured values obtained by 3 different programs when compiled with 4 different optimisation sets. As we did in Figure 8.1 for the TP algorithm, we have full information about three programs q_0, q_1, q_2. Moreover, we now have a new program p that needs to be tuned. Our CF algorithm starts by evaluating the program p with the baseline configuration x_0. As a second configuration, it evaluates the first one identified by the TP algorithm: x_1. We can now start to apply RM characterisation to find the next configuration to evaluate.

In Figure 8.2b we compute the relevance scores using Equation 8.1, which brings all the baselines to 0. After having computed the performance scores with Equation 8.1, we define the similarity between two programs as the distance of the relevance scores they received with the same sets.

Defining $\{x_i\}_{i=1}^n$ as the sequence of the optimisation sets explored during the n iterations taken so far in the iterative compilation of the new program p, and using a Euclidean distance as an example, we can compute the distance between two programs p, q as:

$$d_{pq} = \sqrt{\frac{\sum_{i=1}^{n} \left(r_p(x_i) - r_q(x_i)\right)^2}{n}}. \tag{8.5}$$

Similarly to CBF, we obtain the similarity as $s_{pq} = \frac{1}{d_{pq}}$.

In the example of Figure 8.2c, RM uses x_1 to measure the distances, and identifies q_1 as the most similar program to p. Plugging this into Equation 8.3 (using $k = 1$ for simplicity), RM suggests to evaluate x_2, which is performing well on the similar program q_1. This is in contrast with the TP algorithm, which would have suggested x_3 as a second candidate, and would have kept x_2 as the last choice. Once evaluated x_2 and having obtained $f_p(x_2)$, we need to compute its relevance score $r_p(x_2)$. Then, we recompute the distances and find the next set to evaluate.

Notice that the definition of distance depends on the optimisation sets $\{x_i\}_{i=1}^n$ which we already evaluated. In other words, RM is an online algorithm, and the computed distances vary as we proceed with IC and more sets get evaluated. This is in contrast with TP, CBF, performance counters-based methodologies and code-based approaches, which never update their beliefs. This puts our CF solution between solutions like Cobayn and DeepTune, which exploit previous knowledge but never update their beliefs, and solutions like OpenTuner, which use the results of previous IC iterations to suggest the next set to evaluate but cannot exploit knowledge previously collected on other programs. Moreover, by measuring the actual performance, the RM characterisation is workload-dependent and decoupled from the source code, which can be as huge and complex as the developer likes.

8.3 Experimental Evaluation of Collaborative Filtering

The evaluation was conducted on the dataset released with Cobayn, which consists of 24 programs taken from cBench [43], each one with 5 different workloads. In [20] we

also collected another dataset using the PolyBench suite [84].

To collect the PolyBench dataset, we used an Amazon EC2[1] a1.medium instance, which is equipped with a single ARMv8 gravitron processor and 2GB of ram, Ubuntu Server 18.04 and PolyBench 4.2.1. We collected the MICA metrics [53] using Intel PIN 3.10 [69]. The cBench dataset shipped with Cobayn has instead been collected on an ARMv7 Cortex-A9 architecture as part of a TI-OMAP 4430 processor.

We considered the 7 binary optimisations flags used in [6] and reported in Table 8.1, for a total of 128 possible different optimisation sets, with the goal of reducing execution time.

Table 8.1: *Considered optimisations.*

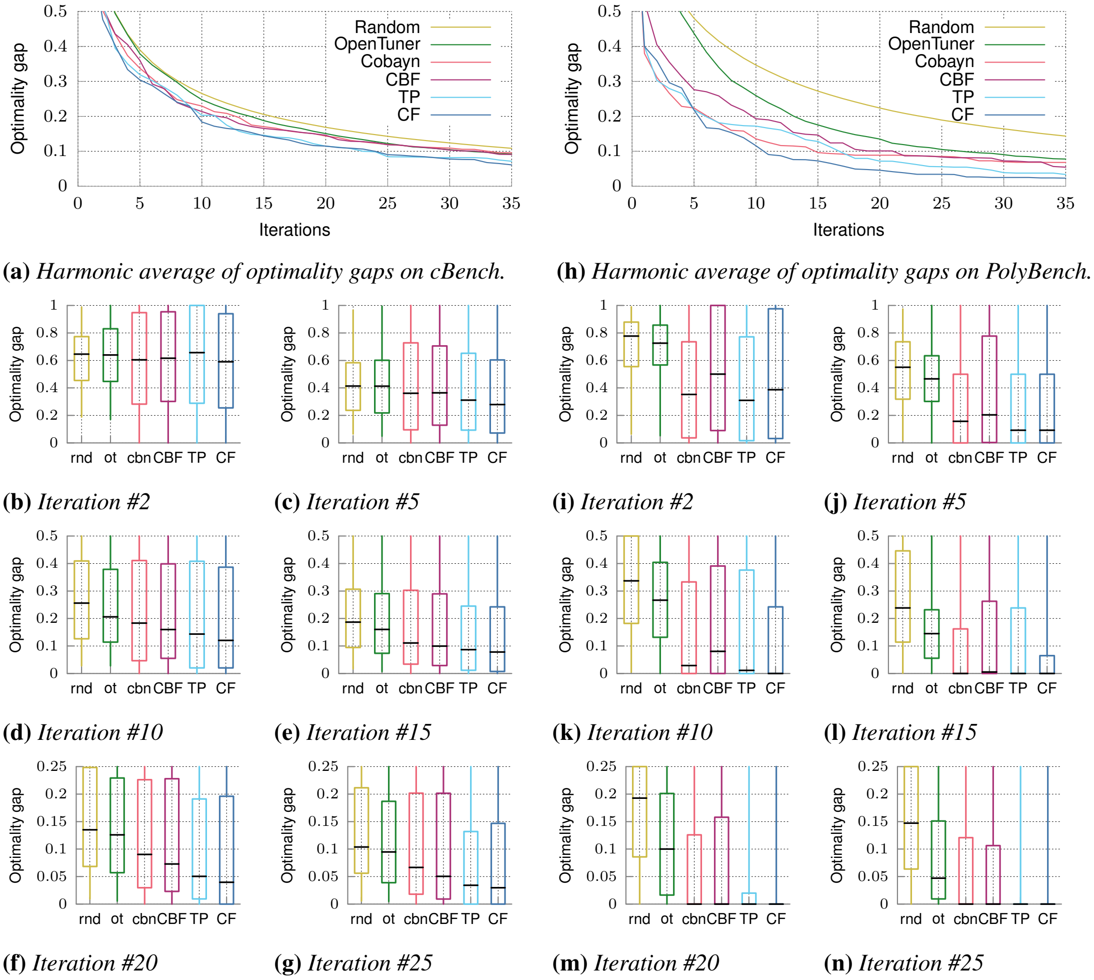
Optimisation	Short description
`-funsafe-math-optimisations`	Allow unsafe optimisations for floating-point arithmetic.
`-fno-guess-branch-probability`	Do not guess branch probabilities using heuristics.
`-fno-ivopts`	Disable induction variable optimisations on trees.
`-fno-tree-loop-optimise`	Disable loop optimisations on trees.
`-fno-inline-functions`	Disable optimisation that inline all simple functions.
`-funroll-all-loops`	Unroll all loops, even if their number of iterations is uncertain.
`-O2`	Overwrite the default -O3 optimisation level.

The tuning results (available with more details in [20]) are reported in Figure 8.3. Differently from the results reported so far, they are expressed in terms of *Optimality gap*, which is equivalent to $1-OffOpt$. In other terms, they measure the distance to the optimum considering an offline tuning scenario. Starting from cBench, in Figure 8.3a we aggregate over different programs of the suite using a harmonic average. All the solutions outperform the random approach but do so in different ways. OpenTuner is very similar to random in the first iterations, then it becomes as good as CBF and Cobayn, which, conversely, in the first iterations find better solutions than random, suggesting that the MICA characterisation helps find good solutions.

However, the TP algorithm also behaves well in the first iterations, even if it does not have a characterisation step. After 10 iterations, TP becomes significantly better than CBF and Cobayn. Most of the time, CF is slightly below TP, suggesting that RM characterisation is effective.

To better understand the algorithms behaviour, we report in Figures 8.3b to 8.3e the quartile distributions of the optimality gaps over different programs. Notice that the width of the distribution does not represent the variability of the algorithms, but is instead related to the fact that some programs are harder to tune, and thus it is slower to reach small gaps. Comparing the variabilities of Random and TP can give us insights into the dataset's nature. A low first quartile on Random implies that there are many programs for which we can find many good optimisation sets, so it becomes probable that Random quickly finds one of them. A low first quartile on TP, instead, implies that many programs like the *same* set of flags, which gets recommended by TP. Similar reasoning can be made about the third quartile, which represents programs that like a few sets (Random) of very peculiar ones (TP). Having a low first quartile is thus easy, as it is more a property of the dataset, whereas the difficulty lies in lowering the third

[1] Amazon Elastic Compute Cloud `https://aws.amazon.com/ec2/`



(a) *Harmonic average of optimality gaps on cBench.* **(h)** *Harmonic average of optimality gaps on PolyBench.*

(b) *Iteration #2* **(c)** *Iteration #5* **(i)** *Iteration #2* **(j)** *Iteration #5*

(d) *Iteration #10* **(e)** *Iteration #15* **(k)** *Iteration #10* **(l)** *Iteration #15*

(f) *Iteration #20* **(g)** *Iteration #25* **(m)** *Iteration #20* **(n)** *Iteration #25*

Figure 8.3: *Experimental results in terms of optimality gaps on cBench (figs. 8.3a to 8.3g) and PolyBench (figs. 8.3h to 8.3n) reported with harmonic average (figs. 8.3a and 8.3h) and distributions of minimum, first quartile, median, third quartile and maximum over different programs (figs. 8.3b to 8.3g and 8.3i to 8.3n).*

quartile. A proper characterisation should lead to a low third quartile, as it represents programs that are non-trivial to tune.

With this in mind, we can observe that the good performance of Cobayn, CBF, TP and CF in the first iterations is a more complex story. Looking at the second iteration, they all have a median gap slightly lower than the random one. The quartiles tough are much wider, indicating that, for some programs, the characterisation is failing. As we said earlier, the characterisation of Cobayn and CBF is fixed, so they will not be able to provide a better characterisation for these unconventional programs. Conversely, CF updates the RM characterisation at each iteration, and it reduces its third quartile already at the fifth iteration. The first quartile and the median gap of CF remain the best at all the iterations, whereas the third quartile is on par with the one achieved by TP, but still better than the ones achieved by other approaches.

Moving to PolyBench, we report the harmonic average in Figure 8.3h and the distributions in Figures 8.3i to 8.3l.

The first comment regards Random and OpenTuner. Compared to cBench, Random is now slower while OpenTuner is faster. This suggests that, on PolyBench, there are fewer good sets (which makes random slower), but they are easier to find (which makes OpenTuner faster). This is coherent with the behaviour of the other algorithms, which are performing much better. Their behaviour is similar to the one we observed on cBench: Cobayn and CBF start well but, after a while, they are reached by OpenTuner, while TP and CF perform better. On PolyBench, Cobayn is more effective than CBF and also CF performs much better.

Looking at the quartiles, we draw similar conclusions: on Polybench, there are easy programs, resulting in very low first quartiles already at the second iteration, and harder programs for which we have a high third quartile. Even here, however, CF finds the best solution on all the programs already after five iterations, and its third quartile remains significantly lower at all the subsequent iterations, indicating that, on PolyBench, the RM characterisation is particularly effective.

In short, all the considered algorithms are performing better than the random one. The second worst solution is OpenTuner, which is expected as it cannot exploit any previous knowledge. The MICA characterisation used in CBF is not improving over a simple TP, and the three best algorithms are Cobayn, TP and CF. Also, notice that Cobayn substantially outperforms the random approach on both the benchmark suites, which is coherent with the results reported in [6] and validates the fairness of our experiments. At the very first iterations, Cobayn and TP provide the best results on both the suites, suggesting that the good results of Cobayn come more from implicit exploitation of the popularity bias than from an effective characterisation given by the MICA metrics.

The main drawback of CF lies in the RM characterisation, which needs to have some points on which to base its decision before becoming effective. Nonetheless, in the first iterations, CF is on par with TP, and, already at iteration 5, it consistently finds better solutions. Moreover, the RM characterisation becomes better and better with more iterations, letting CF keep an advantage over other techniques even in later iterations. RM also allows CF to have a consistently lower variability, making it a reliable algorithm even on harder-to-tune programs. As an example, in Figure 8.3n TP and CF look almost identical. However, the harmonic gap of CF is considerably lower, as visible in Figure 8.3h).

8.3.1 Reaction Matching vs Performance Counters

The conducted experiments show that CF provides better solutions both in the initial iterations and later ones. The advantage comes from the RM characterisation, which quickly finds significant similarities among programs and becomes better and better as more sets are evaluated. Furthermore, the Top Popular algorithm, which has no characterisation, also outperforms current state-of-the-art solutions, suggesting that their good result comes more from implicit exploitation of a popularity bias than from an effective characterisation. Conversely, the RM characterisation allows CF to outperform TP even with small numbers of evaluated flags.

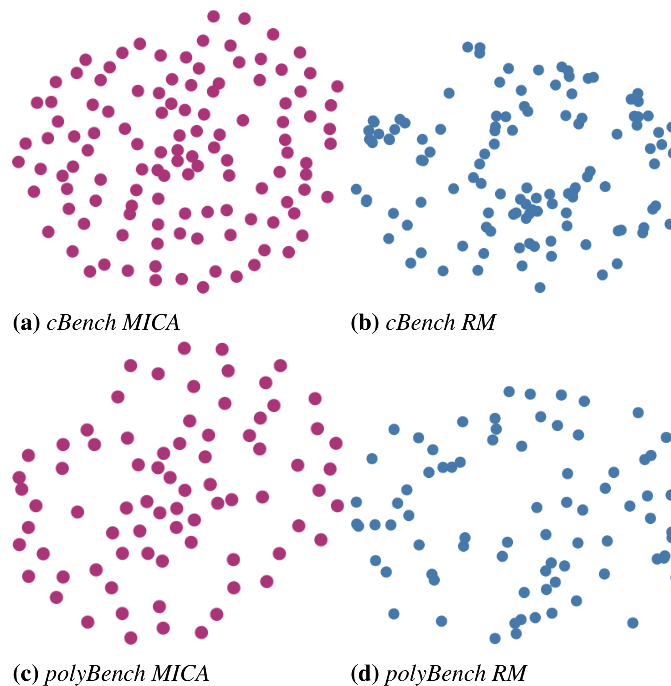
In the context of the present work, the most important results are that TP outperforms CBF, and CF outperforms TP. This suggests that the RM characterisation is more effective than the feature-based one employed in CBF and, also, that the feature-based

characterisation is actually misleading. This is a key point since program characterisation is a crucial part of the future extension of this work. In fact, exploiting a knowledge base is crucial to improve the safety and quality of the autotuner, and characterising programs is needed to navigate the knowledge base.

The advantage that CF has over CBF comes from the RM characterisation. We thus compare the characterisation provided by RM with the one coming from MICA metrics. We report in Figure 8.4 a representation of the distances between programs as identified by the two methodologies. To produce this chart, we consider every program as a node of a fully-connected graph and use the distance between two nodes as a weight of their link. By using the Fruchterman-Reingold force-directed algorithm [41], we arrange the nodes on a plane, keeping similar nodes close together. The MICA-based characterisation used in CBF leads to a more uniform distribution, whereas RM creates clusters of nodes.

Creating clusters is directly related to finding communities of similarly behaving programs. By exploiting the communities identified by RM, the CF algorithm quickly suggests good configurations when tuning a new problem.

It is worth mentioning that the distances of Figure 8.4 have been computed by using all the 128 available optimisations sets. When tuning a new program p, however, CF cannot benefit from all the sets, but only from the ones already evaluated on p. Nonetheless, as the tuning proceeds, more sets are evaluated and RM tends toward the distribution reported in the figure, whereas CBF is stuck at its initial guess.



(a) *cBench MICA* **(b)** *cBench RM*

(c) *polyBench MICA* **(d)** *polyBench RM*

Figure 8.4: *Chart representation of program/workload similarities. Every node represents a program/dataset combination and the distance reflects the similarities as computed with each methodology. The nodes are arranged using the Fruchterman-Reingold force-directed algorithm.*

CHAPTER 9

Time Complexity

At the core of CC-GP there is a Gaussian Process. The complexity of computing the posterior of a GP over N training points is usually quantified as $\mathcal{O}(n^3)$ in time [112]. Hence, many approaches have been proposed to decrease the computational cost of using GPs [14, 85, 13, 76].

Nonetheless, CC-GP also has to fit all the regressors to predict metric and startup violations. Fuerthermore, selecting the next configuration is not only a matter of optimising the acuisition function derived from the GP posterior, but it also has to evaluate all the parameter constraints and predict all the violations.

Parameter constraints are simple mathematical inequalities, hence their evaluation cost is linear in the number of candidate configurations considered. As the number of candidate configurations does not increase with the number of explored configurations N, we can conclude that the cost of evaluating parameter constraint is constant for any given optimisation problem.

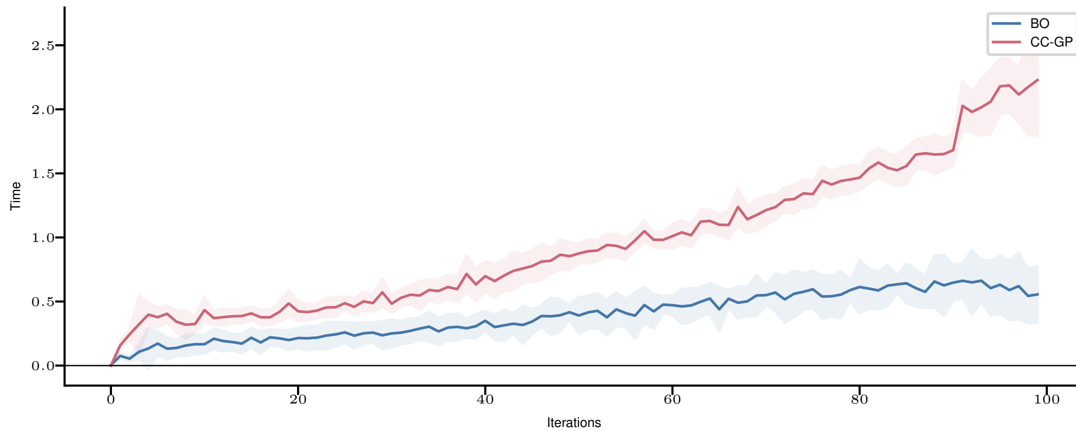
Exploration constraints involve find the minimum distance between a candidate configuration and the set of explored feasible configurations, so it scales linearly with N.

Startup constraints and metric constrasints are evaluated using ν-SVR regressors, which also scales cubically with the number of evaluated configurations [1].

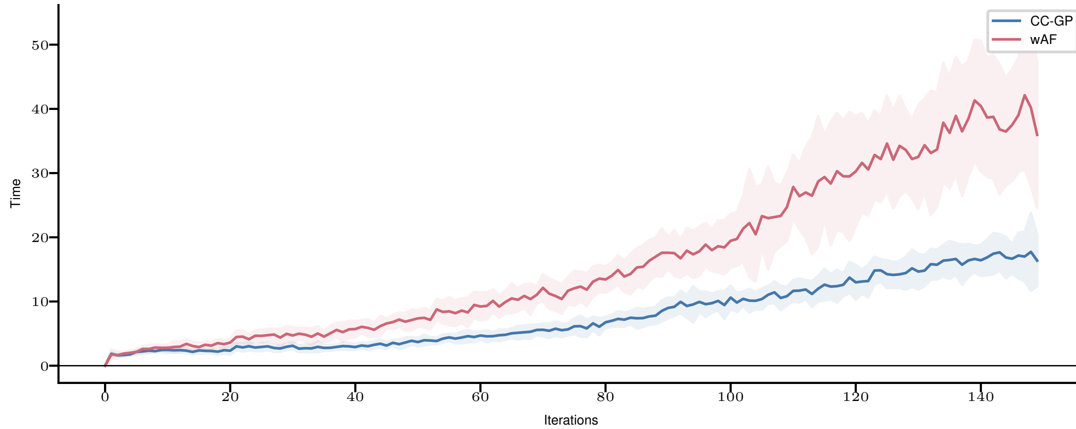
Therefore, the overall time complexity of one iteration of CC-GP is $\mathcal{O}(n^3)$, where n is the number of previously evaluated configurations.

To evaluate the time complexity from a practical viewpoint, we report in Figure 9.1 the time taken by CC-GP to suggest a configuration when run on a laptop equipped with an Intel i5-8250U CPU. We consider two of the scenarios reported above: the simple Branin function and constrained Cassandra under the Continuous Ramp workload pattern.

The Branin function is the simplest one, and allows us to evaluate the performance



(a) *Branin*



(b) *Constrained Cassandra Continuous Ramp*

Figure 9.1: *Configuration suggestion time.*

deficit of our implementation with the one used in BayesianOptimization. Technically, the only difference between the autotuners is the computation of the NPI score, which is a simple division and cannot explain the hughe performance difference. Hence, we must conclude that this gap is due to our inefficient implementation. Furthermore, we can observe that (on a good implementation) the effect of the cubical complexity is not observable in the first hundred iterations.

Conversely, in the constrained Cassandra tuning under the Continuous Ramp workload pattern we have enabled all the components of CC-GP, and thus it represents the most time-consuming experiment of the reported ones. In the case of Cassandra, a configuration is evaluated for 45 minutes, so an overhead of 30 seconds is still largely acceptable.

Comparing Figure 9.1a with Figure 9.1b we see that the time required to run CC-GP increases by an order of magnitude when considering all the types of available constraints. Analysing the time required by the various parts of CC-GP, we observe that most of the time is spent evaluating the parameter constraints. This is due to how we handle this check in our implementation, evaluating each constraint for each candidate configuration with a separate evaluation of a python function. Hence, this time can be reduced quite a lot by using a more efficient implementation.

Furthermore, it is interesting to notice that CC-GP is faster than wAF, apart from achieving better results. This is due to the fact that, when optimising the AF, wAF has to predict the constraint satisfaction probability for all the candidate configurations. Conversely, CC-GP consider one constraint at a time, sequentially removing candidate configurations which are not feasible and thus reducing the dimension of the set of configurations for which to evaluate feasibility or even compute the AF value.

CHAPTER 10

Conclusions and Future Works

In this work, we have introduced CC-GP, a configuration autotuner able to deal with generic IT systems considering multiple layers of the IT stack holistically so to optimise any user-facing performance indicator. By using Contextual Gaussian Processes and the Prophet time series forecaster, CC-GP deals with externally changing conditions, such as the workload. Apart from being goal-driven, CC-GP can also model parameter, startup, exploration and metric constraints. We have evaluated CC-GP using a set of clearly defined metrics on analytical functions and dataset-based simulations of the Cassandra and MongoDB DBMSs and the JPetstore web application, considering parameters from the Linux Operating System, the Java Virtual Machine, and the application layer. We compared CC-GP against different approaches (OpenTuner and BestConfig) and against an off-the-shelf implementation of Bayesian Optimisation, showing how its various components interact to deliver a superior tuning experience.

When dealing with a dynamic context, CC-GP can be configured to suggest configurations which are expected to satisfy the given constraints not only on the predicted context, but on all the observed ones. In this way, we expect the suggested configuration to be safe even when the forecasting is wrong. In future extensions, the same reasoning will be applied to the tuning goal, so to suggest a configuration which is expected to perform well on all the observed contexts. Differently from constraints, the rationale of this feature is not to increase safety, but to find a configuration which should not be modified continuously, as not all the software can be dinamically restarted.

Nonetheless, when executing a performance test the context will always be a specific one, which can be predicted by the forecaster. Hence, in future extensions the Acquisition Function optimiser of CC-GP will be modified to decouple the context used for the explorative component (which should be tailored to the upcoming context) and the exploitative one (which should consider all the contexts). In this way, we expect CC-

GP to provide configurations which are safe and well-performing on all the contexts, but are also more informative on the specific context where they are tested.

In the field of compiler autotuning, we have seen that the Reaction Matching characterisation is an effective tool to find similarities in the tunable properties of different systems. We are now extending CC-GP to include RM characterisation and exploit knowledge bases to speed up the tuning process.

Albeit we have succesfully validated CC-GP on other real systems, we did not include those results in this work, as a tuning session conducted in a real environment is subject to noise and thus would be misleading in evaluating different autotuners. Hence, to further validate this approach (and possibly other ones), it is foundamental to collect other datasets on which to run tuning simulations, extending as much as possible the number of modelled parameters.

Also, the approach could be extended to take advantage of system simulators. Having a simulator able to predict with a certain accuracy whether a candidate configuration will lead to some metric constraint violations would allow the optimiser to propose safer configurations. As an example, there exists queueing network models to simulate the performance of databases [35, 45], which could be used to obtain a rough estimate of the response time of a configuration, or at least a part of it. Using simulators would also allow to exploit the vast literature of simulation-based optimization [33] to furter improve the quality of the recommended configuration.

CC-GP can deal with bad-posed optimisation problems, where the constraints are too strict and cannot be solved. In this case, CC-GP tries to minimise the violation of the unsolvable constraints, without considering the objective function at all. This behaviour could be too conservative, and future extensions of the work might try to consider the optimisation goal even in these unsolvable problems, as we have empirically observed that this behaviour is too conservative when the unsolvable constraints come from noise in the system.

Finally, albeit we have shown that CC-GP easily discards irrelevant tunable dimensions, and that its overhead is acceptable even with a large number of evaluated coinfigurations, we will modify the Contextual Gaussian Process to deal with larger search spaces and datasets, which is non trivial given the presence of the context.

Bibliography

[1] Abdiansah Abdiansah and Retantyo Wardoyo. Time complexity analysis of support vector machines (svm) in libsvm. *International journal computer and application*, 128(3):28–34, 2015.

[2] F. Agakov, E. Bonilla, J. Cavazos, B. Franke, G. Fursin, M. F. P. O'Boyle, J. Thomson, M. Toussaint, and C. K. I. Williams. Using machine learning to focus iterative optimization. In *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '06, pages 295–305, Washington, DC, USA, 2006. IEEE Computer Society.

[3] Lelac Almagor, Keith D Cooper, Alexander Grosul, Timothy J Harvey, Steven W Reeves, Devika Subramanian, Linda Torczon, and Todd Waterman. Finding effective compilation sequences. *ACM SIGPLAN Notices*, 39(7):231–239, 2004.

[4] Jason Ansel, Shoaib Kamil, Kalyan Veeramachaneni, Jonathan Ragan-Kelley, Jeffrey Bosboom, Una-May O'Reilly, and Saman Amarasinghe. Opentuner: An extensible framework for program autotuning. In *Proceedings of the 23rd international conference on Parallel architectures and compilation*, pages 303–316, 2014.

[5] Amir H Ashouri, William Killian, John Cavazos, Gianluca Palermo, and Cristina Silvano. A survey on compiler autotuning using machine learning. *ACM Computing Surveys (CSUR)*, 51(5):1–42, 2018.

[6] Amir Hossein Ashouri, Giovanni Mariani, Gianluca Palermo, Eunjung Park, John Cavazos, and Cristina Silvano. Cobayn: Compiler autotuning framework using bayesian networks. *ACM Transactions on Architecture and Code Optimization (TACO)*, 13(2):1–25, 2016.

[7] Prasanna Balaprakash, Jack Dongarra, Todd Gamblin, Mary Hall, Jeffrey K Hollingsworth, Boyana Norris, and Richard Vuduc. Autotuning in high-performance computing applications. *Proceedings of the IEEE*, 106(11):2068–2083, 2018.

[8] David Balla, Csaba Simon, and Markosz Maliosz. Adaptive scaling of kubernetes pods. In *NOMS 2020-2020 IEEE/IFIP Network Operations and Management Symposium*, pages 1–5. IEEE, 2020.

[9] James Bergstra, Rémi Bardenet, Yoshua Bengio, and Balázs Kégl. Algorithms for hyper-parameter optimization. *Advances in neural information processing systems*, 24, 2011.

[10] J Bernardo, MJ Bayarri, JO Berger, AP Dawid, D Heckerman, AFM Smith, and M West. Optimization under unknown constraints. *Bayesian Statistics*, 9(9):229, 2011.

[11] Christopher M Bishop and Nasser M Nasrabadi. *Pattern recognition and machine learning*, volume 4. Springer, 2006.

[12] François Bodin, Toru Kisuki, Peter Knijnenburg, Mike O'Boyle, and Erven Rohou. Iterative compilation in a non-linear optimisation space. In *Workshop on Profile and Feedback-Directed Compilation*, 1998.

[13] Daniele Calandriello, Luigi Carratino, Alessandro Lazaric, Michal Valko, and Lorenzo Rosasco. Gaussian process optimization with adaptive sketching: Scalable and no regret. In *Conference on Learning Theory*, pages 533–557. PMLR, 2019.

[14] Daniele Calandriello, Luigi Carratino, Alessandro Lazaric, Michal Valko, and Lorenzo Rosasco. Scaling gaussian process optimization by evaluating a few unique candidates multiple times. *arXiv preprint arXiv:2201.12909*, 2022.

[15] Rodrigo N Calheiros, Enayat Masoumi, Rajiv Ranjan, and Rajkumar Buyya. Workload prediction using arima model and its impact on cloud applications' qos. *IEEE transactions on cloud computing*, 3(4):449–458, 2014.

[16] Maria Carla Calzarossa, Luisa Massari, and Daniele Tessera. Workload characterization: A survey revisited. *ACM Computing Surveys (CSUR)*, 48(3):1–43, 2016.

[17] John Cavazos, Christophe Dubach, Felix Agakov, Edwin Bonilla, Michael FP O'Boyle, Grigori Fursin, and Olivier Temam. Automatic performance model construction for the fast software exploration of new hardware designs. In *Proceedings of the 2006 international conference on Compilers, architecture and synthesis for embedded systems*, pages 24–34. ACM, 2006.

[18] John Cavazos, Grigori Fursin, Felix Agakov, Edwin Bonilla, Michael FP O'Boyle, and Olivier Temam. Rapidly selecting good compiler optimizations using performance counters. In *International Symposium on Code Generation and Optimization (CGO'07)*, pages 185–197. IEEE, 2007.

[19] Stefano Cereda, Paolo Cremonesi, Giovanni Paolo Gibilisco, and Stefano Doni. Method and apparatus for tuning a computing environment using a knowledge base, 2022. US Patent App. currently in submission.

[20] Stefano Cereda, Gianluca Palermo, Paolo Cremonesi, and Stefano Doni. A collaborative filtering approach for the automatic tuning of compiler optimisations. In *The 21st ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*, pages 15–25, 2020.

[21] Stefano Cereda, Stefano Valladares, Paolo Cremonesi, and Stefano Doni. Cgptuner: a contextual gaussian process bandit approach for the automatic tuning of it configurations under varying workload conditions. *Proceedings of the VLDB Endowment*, 14(8):1401–1413, 2021.

[22] Chih-Chung Chang and Chih-Jen Lin. Libsvm: a library for support vector machines. *ACM transactions on intelligent systems and technology (TIST)*, 2(3):1–27, 2011.

[23] Yang Chen, Shuangde Fang, Yuanjie Huang, Lieven Eeckhout, Grigori Fursin, Olivier Temam, and Chengyong Wu. Deconstructing iterative optimization. *ACM Transactions on Architecture and Code Optimization (TACO)*, 9(3):1–30, 2012.

[24] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 143–154, 2010.

[25] Keith D Cooper, Alexander Grosul, Timothy J Harvey, Steven Reeves, Devika Subramanian, Linda Torczon, and Todd Waterman. Acme: adaptive compilation made efficient. In *ACM SIGPLAN Notices*, volume 40, pages 69–77. ACM, 2005.

[26] Keith D Cooper, Philip J Schielke, and Devika Subramanian. Optimizing for reduced code space using genetic algorithms. In *ACM SIGPLAN Notices*, volume 34, pages 1–9. ACM, 1999.

[27] Paolo Cremonesi, Yehuda Koren, and Roberto Turrin. Performance of recommender algorithms on top-n recommendation tasks. In *Proceedings of the fourth ACM conference on Recommender systems*, pages 39–46. ACM, 2010.

[28] Chris Cummins, Pavlos Petoumenos, Zheng Wang, and Hugh Leather. End-to-end deep learning of optimization heuristics. In *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 219–232. IEEE, 2017.

[29] Valentin Dalibard, Michael Schaarschmidt, and Eiko Yoneki. Boat: Building auto-tuners with structured bayesian optimization. In *Proceedings of the 26th International Conference on World Wide Web*, pages 479–488, 2017.

[30] Sudipto Das, Feng Li, Vivek R Narasayya, and Arnd Christian König. Automated demand-driven resource scaling in relational database-as-a-service. In *Proceedings of the 2016 International Conference on Management of Data*, pages 1923–1934, 2016.

[31] Jan G De Gooijer and Rob J Hyndman. 25 years of time series forecasting. *International journal of forecasting*, 22(3):443–473, 2006.

[32] Biplob K Debnath, David J Lilja, and Mohamed F Mokbel. Sard: A statistical approach for ranking database tuning parameters. In *2008 IEEE 24th International Conference on Data Engineering Workshop*, pages 11–18. IEEE, 2008.

[33] AB Dieker, Soumyadip Ghosh, and Mark S Squillante. Optimal resource capacity management for stochastic networks. *Operations Research*, 65(1):221–241, 2017.

[34] Sébastien Le Digabel and Stefan M Wild. A taxonomy of constraints in simulation-based optimization. *arXiv preprint arXiv:1505.07881*, 2015.

[35] Salvatore Dipietro, Giuliano Casale, and Giuseppe Serazzi. A queueing network model for performance prediction of apache cassandra. 2016.

[36] Stefano Doni, Giovanni Paolo Gibilisco, and Stefano Cereda. Method and apparatus for tuning adjustable parameters in computing environment, September 17 2020. US Patent App. 16/818,263.

[37] Haizhou Du, Ping Han, Wei Chen, Yi Wang, and Chenlu Zhang. Otterman: A novel approach of spark autotuning by a hybrid strategy. In *2018 5th International Conference on Systems and Informatics (ICSAI)*, pages 478–483. IEEE, 2018.

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

[39] Björn Franke, Michael O'Boyle, John Thomson, and Grigori Fursin. Probabilistic source-level optimisation of embedded programs. *ACM SIGPLAN Notices*, 40(7):78–86, 2005.

[40] Nir Friedman, Dan Geiger, and Moises Goldszmidt. Bayesian network classifiers. *Machine learning*, 29(2-3):131–163, 1997.

[41] Thomas MJ Fruchterman and Edward M Reingold. Graph drawing by force-directed placement. *Software: Practice and experience*, 21(11):1129–1164, 1991.

[42] Grigori Fursin, Cupertino Miranda, Olivier Temam, Mircea Namolaru, Elad Yom-Tov, Ayal Zaks, Bilha Mendelson, Edwin Bonilla, John Thomson, Hugh Leather, et al. Milepost gcc: machine learning based research compiler. In *GCC summit*, 2008.

[43] Grigori Fursin and Olivier Temam. Collective optimization: A practical collaborative approach. *ACM Transactions on Architecture and Code Optimization (TACO)*, 7(4):20, 2010.

[44] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, et al. An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 3–18, 2019.

[45] Andrea Gandini, Marco Gribaudo, William J Knottenbelt, Rasha Osman, and Pietro Piazzolla. Performance evaluation of nosql databases. In *European Workshop on Performance Engineering*, pages 16–29. Springer, 2014.

[46] Jacob R Gardner, Matt J Kusner, Zhixiang Eddie Xu, Kilian Q Weinberger, and John P Cunningham. Bayesian optimization with inequality constraints. In *ICML*, volume 2014, pages 937–945, 2014.

[47] Everette S Gardner Jr. Exponential smoothing: The state of the art. *Journal of forecasting*, 4(1):1–28, 1985.

[48] Everette S Gardner Jr. Exponential smoothing: The state of the art—part ii. *International journal of forecasting*, 22(4):637–666, 2006.

[49] Michael A Gelbart, Jasper Snoek, and Ryan P Adams. Bayesian optimization with unknown constraints. *arXiv preprint arXiv:1403.5607*, 2014.

[50] Daniel Golovin, Benjamin Solnik, Subhodeep Moitra, Greg Kochanski, John Karro, and David Sculley. Google vizier: A service for black-box optimization. In *Proceedings of the 23rd ACM SIGKDD international conference on knowledge discovery and data mining*, pages 1487–1495, 2017.

[51] Richard L Gorsuch. Exploratory factor analysis. In *Handbook of multivariate experimental psychology*, pages 231–258. Springer, 1988.

[52] Matthew D Hoffman, Eric Brochu, and Nando de Freitas. Portfolio allocation for bayesian optimization. In *UAI*, pages 327–336. Citeseer, 2011.

[53] Kenneth Hoste and Lieven Eeckhout. Microarchitecture-independent workload characterization. *IEEE micro*, 27(3):63–72, 2007.

[54] Kenneth Hoste and Lieven Eeckhout. Cole: compiler optimization level exploration. In *Proceedings of the 6th annual IEEE/ACM international symposium on Code generation and optimization*, pages 165–174. ACM, 2008.

[55] Harold Hotelling. Analysis of a complex of statistical variables into principal components. *Journal of educational psychology*, 24(6):417, 1933.

[56] Frank Hutter, Holger H Hoos, and Kevin Leyton-Brown. Sequential model-based optimization for general algorithm configuration. In *International conference on learning and intelligent optimization*, pages 507–523. Springer, 2011.

[57] Donald R Jones. A taxonomy of global optimization methods based on response surfaces. *Journal of global optimization*, 21(4):345–383, 2001.

[58] Toru Kisuki, Peter MW Knijnenburg, and Michael FP O'Boyle. Combined selection of tile sizes and unroll factors using iterative compilation. In *Proceedings 2000 International Conference on Parallel Architectures and Compilation Techniques (Cat. No. PR00622)*, pages 237–246. IEEE, 2000.

[59] Andreas Krause and Cheng S Ong. Contextual gaussian process bandit optimization. In *Advances in neural information processing systems*, pages 2447–2455, 2011.

[60] Prasad Kulkarni, Stephen Hines, Jason Hiser, David Whalley, Jack Davidson, and Douglas Jones. Fast searches for effective optimization phase sequences. In *ACM SIGPLAN Notices*, volume 39, pages 171–182. ACM, 2004.

[61] Prasad Kulkarni, Wankang Zhao, Hwashin Moon, Kyunghwan Cho, David Whalley, Jack Davidson, Mark Bailey, Yunheung Paek, and Kyle Gallivan. Finding effective optimization phase sequences. In *ACM SIGPLAN Notices*, volume 38, pages 12–23. ACM, 2003.

[62] Harold J Kushner. A new method of locating the maximum point of an arbitrary multipeak curve in the presence of noise. 1964.

[63] Avinash Lakshman and Prashant Malik. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, 44(2):35–40, 2010.

[64] Miguel Lázaro-Gredilla, Joaquin Quinonero-Candela, Carl Edward Rasmussen, and Aníbal R Figueiras-Vidal. Sparse spectrum gaussian process regression. *The Journal of Machine Learning Research*, 11:1865–1881, 2010.

[65] Min Li, Liangzhao Zeng, Shicong Meng, Jian Tan, Li Zhang, Ali R Butt, and Nicholas Fuller. Mronline: Mapreduce online performance tuning. In *Proceedings of the 23rd international symposium on High-performance parallel and distributed computing*, pages 165–176, 2014.

[66] Bryan Lim and Stefan Zohren. Time-series forecasting with deep learning: a survey. *Philosophical Transactions of the Royal Society A*, 379(2194):20200209, 2021.

[67] Marius Lindauer, Katharina Eggensperger, Matthias Feurer, André Biedenkapp, Difan Deng, Carolin Benjamins, Tim Ruhkopf, René Sass, and Frank Hutter. Smac3: A versatile bayesian optimization package for hyperparameter optimization, 2021.

[68] Tania Lorido-Botran, Jose Miguel-Alonso, and Jose A Lozano. A review of auto-scaling techniques for elastic applications in cloud environments. *Journal of grid computing*, 12(4):559–592, 2014.

[69] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. *Acm sigplan notices*, 40(6):190–200, 2005.

[70] Lin Ma, Dana Van Aken, Ahmed Hefny, Gustavo Mezerhane, Andrew Pavlo, and Geoffrey J Gordon. Query-based workload forecasting for self-driving database management systems. In *Proceedings of the 2018 International Conference on Management of Data*, pages 631–645, 2018.

[71] Ganapathy Mahalakshmi, S Sridevi, and Shyamsundar Rajaram. A survey on forecasting of time series data. In *2016 International Conference on Computing Technologies and Intelligent Data Engineering (ICCTIDE'16)*, pages 1–8. IEEE, 2016.

[72] Luiz GA Martins, Ricardo Nobre, Joao MP Cardoso, Alexandre CB Delbem, and Eduardo Marques. Clustering-based selection for the exploration of compiler optimization sequences. *ACM Transactions on Architecture and Code Optimization (TACO)*, 13(1):8, 2016.

[73] Jonas Mockus, Vytautas Tiesis, and Antanas Zilinskas. The application of bayesian methods for seeking the extremum. *Towards global optimization*, 2(117-129):2, 1978.

[74] Luca Moroldo. Online contextual system tuning with bayesian optimization and workload forecasting. Master's thesis, Università degli studi di Padova - Department of Information Engineering, 2 2022.

[75] Barzan Mozafari, Carlo Curino, Alekh Jindal, and Samuel Madden. Performance and resource modeling in highly-concurrent oltp workloads. In *Proceedings of the 2013 acm sigmod international conference on management of data*, pages 301–312, 2013.

[76] Mojmir Mutny and Andreas Krause. Efficient high dimensional bayesian optimization with additivity and quadrature fourier features. *Advances in Neural Information Processing Systems*, 31, 2018.

[77] Thanh-Tung Nguyen, Yu-Jin Yeom, Taehong Kim, Dae-Heon Park, and Sehan Kim. Horizontal pod autoscaling in kubernetes for elastic container orchestration. *Sensors*, 20(16):4621, 2020.

[78] Xia Ning, Christian Desrosiers, and George Karypis. A comprehensive survey of neighborhood-based recommendation methods. In *Recommender systems handbook*, pages 37–76. Springer, 2015.

[79] Ricardo Nobre, Luiz G. A. Martins, and João M. P. Cardoso. A graph-based iterative compiler pass selection and phase ordering approach. In *Proceedings of the 17th ACM SIGPLAN/SIGBED Conference on Languages, Compilers, Tools, and Theory for Embedded Systems*, LCTES 2016, pages 21–30, New York, NY, USA, 2016. ACM.

[80] Fernando Nogueira. Bayesian Optimization: Open source constrained global optimization tool for Python, 2014–.

[81] David Parello, Olivier Temam, Albert Cohen, and Jean-Marie Verdun. Towards a systematic, pragmatic and architecture-aware program optimization process for complex processors. In *Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, page 15. IEEE Computer Society, 2004.

[82] Andrew Pavlo, Matthew Butrovich, Lin Ma, Prashanth Menon, Wan Shen Lim, Dana Van Aken, and William Zhang. Make your database system dream of electric sheep: towards self-driving operation. *Proceedings of the VLDB Endowment*, 14(12):3211–3221, 2021.

[83] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[84] Louis-Noël Pouchet. Polybench: The polyhedral benchmark suite. *URL: http://www. cs. ucla. edu/pouchet/software/polybench*, 2012.

[85] Joaquin Quinonero-Candela, Carl Edward Rasmussen, and Christopher KI Williams. Approximation methods for gaussian process regression. In *Large-scale kernel machines*, pages 203–223. MIT Press, 2007.

[86] Syama Sundar Rangapuram, Matthias W Seeger, Jan Gasthaus, Lorenzo Stella, Yuyang Wang, and Tim Januschowski. Deep state space models for time series forecasting. *Advances in neural information processing systems*, 31, 2018.

[87] Carl Edward Rasmussen. Gaussian processes in machine learning. In *Summer School on Machine Learning*, pages 63–71. Springer, 2003.

[88] Gourav Rattihalli, Madhusudhan Govindaraju, Hui Lu, and Devesh Tiwari. Exploring potential for non-disruptive vertical auto scaling and resource estimation in kubernetes. In *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*, pages 33–40. IEEE, 2019.

[89] Francesco Ricci, Lior Rokach, and Bracha Shapira. Recommender systems: introduction and challenges. In *Recommender systems handbook*, pages 1–34. Springer, 2015.

[90] Jennie Rogers, Olga Papaemmanouil, and Ugur Cetintemel. A generic auto-provisioning framework for cloud databases. In *2010 IEEE 26th International Conference on Data Engineering Workshops (ICDEW 2010)*, pages 63–68. IEEE, 2010.

[91] Nilabja Roy, Abhishek Dubey, and Aniruddha Gokhale. Efficient autoscaling in the cloud using predictive models for workload forecasting. In *2011 IEEE 4th International Conference on Cloud Computing*, pages 500–507. IEEE, 2011.

[92] Krzysztof Rzadca, Pawel Findeisen, Jacek Swiderski, Przemyslaw Zych, Przemyslaw Broniek, Jarek Kusmierek, Pawel Nowak, Beata Strack, Piotr Witusowski, Steven Hand, et al. Autopilot: workload autoscaling at google. In *Proceedings of the Fifteenth European Conference on Computer Systems*, pages 1–16, 2020.

[93] David Salinas, Valentin Flunkert, Jan Gasthaus, and Tim Januschowski. Deepar: Probabilistic forecasting with autoregressive recurrent networks. *International Journal of Forecasting*, 36(3):1181–1191, 2020.

[94] Bernhard Schölkopf, Alex J Smola, Robert C Williamson, and Peter L Bartlett. New support vector algorithms. *Neural computation*, 12(5):1207–1245, 2000.

[95] Gideon Schwarz. Estimating the dimension of a model. *The annals of statistics*, pages 461–464, 1978.

[96] Matthias Seeger, Christopher Williams, and Neil Lawrence. Fast forward selection to speed up sparse gaussian process regression. Technical report, 2003.

[97] Bobak Shahriari, Kevin Swersky, Ziyu Wang, Ryan P Adams, and Nando De Freitas. Taking the human out of the loop: A review of bayesian optimization. *Proceedings of the IEEE*, 104(1):148–175, 2015.

[98] Edward Snelson and Zoubin Ghahramani. Sparse gaussian processes using pseudo-inputs. *Advances in neural information processing systems*, 18, 2005.

[99] Jasper Snoek, Hugo Larochelle, and Ryan P Adams. Practical bayesian optimization of machine learning algorithms. In *Advances in neural information processing systems*, pages 2951–2959, 2012.

[100] Jasper Snoek, Oren Rippel, Kevin Swersky, Ryan Kiros, Nadathur Satish, Narayanan Sundaram, Mostofa Patwary, Mr Prabhat, and Ryan Adams. Scalable bayesian optimization using deep neural networks. In *International conference on machine learning*, pages 2171–2180. PMLR, 2015.

[101] Jasper Roland Snoek. *Bayesian optimization and semiparametric models with applications to assistive technology*. PhD thesis, Citeseer, 2013.

[102] Il'ya Meerovich Sobol'. On the distribution of points in a cube and the approximate evaluation of integrals. *Zhurnal Vychislitel'noi Matematiki i Matematicheskoi Fiziki*, 7(4):784–802, 1967.

[103] Niranjan Srinivas, Andreas Krause, Sham M Kakade, and Matthias Seeger. Gaussian process optimization in the bandit setting: No regret and experimental design. *arXiv preprint arXiv:0912.3995*, 2009.

[104] Mark Stephenson, Saman Amarasinghe, Martin Martin, and Una-May O'Reilly. Meta optimization: Improving compiler heuristics with machine learning. *ACM sigplan notices*, 38(5):77–90, 2003.

[105] Sean J Taylor and Benjamin Letham. Forecasting at scale. *The American Statistician*, 72(1):37–45, 2018.

[106] Michalis Titsias. Variational learning of inducing variables in sparse gaussian processes. In *Artificial intelligence and statistics*, pages 567–574. PMLR, 2009.

[107] Edward Tsang. *Foundations of constraint satisfaction: the classic text*. BoD–Books on Demand, 2014.

[108] Dana Van Aken, Andrew Pavlo, Geoffrey J Gordon, and Bohan 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, 2017.

[109] Tao Wang, Nikhil Jain, David Beckingsale, David Boehme, Frank Mueller, and Todd Gamblin. Funcytuner: Auto-tuning scientific applications with per-loop compilation. In *Proceedings of the 48th International Conference on Parallel Processing*, pages 1–10, 2019.

[110] Zheng Wang and Michael O'Boyle. Machine learning in compiler optimization. *Proceedings of the IEEE*, 106(11):1879–1901, 2018.

[111] Ruofeng Wen, Kari Torkkola, Balakrishnan Narayanaswamy, and Dhruv Madeka. A multi-horizon quantile recurrent forecaster. *arXiv preprint arXiv:1711.11053*, 2017.

[112] Christopher KI Williams and Carl Edward Rasmussen. *Gaussian processes for machine learning*, volume 2. MIT press Cambridge, MA, 2006.

[113] Bohan Zhang, Dana Van Aken, Justin Wang, Tao Dai, Shuli Jiang, Jacky Lao, Siyuan Sheng, Andrew Pavlo, and Geoffrey J Gordon. A demonstration of the ottertune automatic database management system tuning service. *Proceedings of the VLDB Endowment*, 11(12):1910–1913, 2018.

[114] Yuqing Zhu, Jianxun Liu, Mengying Guo, Yungang Bao, Wenlong Ma, Zhuoyue Liu, Kunpeng Song, and Yingchun Yang. Bestconfig: tapping the performance potential of systems via automatic configuration tuning. In *Proceedings of the 2017 Symposium on Cloud Computing*, pages 338–350, 2017.