ABSTRACT

NAIR, VIVEK. Frugal ways to find Good Configurations. (Under the direction of Dr. Timothy J. Menzies.)

Most software systems available today are configurable, which gives the users an option to customize the system to achieve differ- ent functional or non-functional (better performance) properties. As systems evolve, more configuration options are added to the software system. Studies report that developers find it difficult to understand the configuration spaces, which leaves considerable optimization potential untapped and induces major economic cost. To solve this problem of finding the (near) optimal configurations, engineers have proposed various techniques. Most popular among them are model-based techniques, where accurate models of the configuration space are created using as few configuration measurements as possible. We notice two major problems with the model-based techniques: 1) previous techniques are expensive to be practically viable, and 2) there are software systems whose configuration spaces cannot be accurately modeled. Consequently, there is a gap between proposed techniques and practical viability of these techniques. This dissertation will focus on proposing techniques which are easier to understand and is practically viable.

First, we present **WHAT** that exploits some lower dimensional knowledge to build performance models. Prior work on predicting the performance of software configurations suffered from either (a) requiring far too many sample configurations or (b) large variances in their predictions. Both these problems can be avoided using the **WHAT** spectral learner. WHAT's innovation is the use of the spectrum (eigenvalues) of the distance matrix between the configurations of a configurable software system, to perform dimensionality reduction. Within that reduced configuration space, many closely associated configurations can be studied by executing only a few sample configurations. For the subject systems studied here, a few dozen samples yield accurate and stable predictors?less than 10 % prediction error, with a standard deviation of less than 2%. When compared to the state of the art, WHAT (a) requires 2 to 10 times fewer samples to achieve similar prediction accuracies, and (b) its predictions are more stable (i.e., have lower standard deviation). Furthermore, we demonstrate that predictive models generated by WHAT can be used by optimizers to discover system configurations that closely approach the optimal performance.

The second contribution is a rank-based method which shows how an accurate model is not required for performance optimization, but a rank-preserving model is sufficient. We evaluate rank-based method with 21 scenarios based on nine software systems and demonstrate that our approach is beneficial in 16 scenarios; for the remaining five scenarios, an accurate model can be built by using very few samples anyway, without the need for a rank-based approach.

Additionally, in 8/21 of the scenarios, the number of measurements required by the rank-based method is an order of magnitude smaller than methods used in prior work. To further improve our second contribution, we also propose a Bayesian-based method called Flash: an alternative to model-based technique. Based on preliminary evidence, which can further reduce the cost of performance optimization.

Finally, we present **FLASH**, a sequential model-based method, which sequentially explores the configuration space by reflecting on the configurations evaluated so far to determine the next best configuration to explore. FLASH scales up to software systems that defeat the prior state of the art model-based methods in this area. FLASH runs much faster than existing methods and can solve both single-objective and multi-objective optimization problems. The central insight of this paper is to use the prior knowledge (gained from prior runs) to choose the next promising configuration. This strategy reduces the effort (ie, number of measurements) required to find the (near) optimal configuration. We evaluate FLASH using 30 scenarios based on 7 software systems to demonstrate that FLASH saves effort in 100% and 80% of cases in single-objective and multi-objective problems respectively by up to several orders of magnitude compared to the state of the art techniques.

Frugal ways to find Good Configurations

by
Vivek Nair

A dissertation submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Doctor of Philosophy

Computer Science

Raleigh, North Carolina

2018

APPROVED BY:

Dr. Kathyrn Stolee	Dr. Min Chi
Dr. Ranga Raju Vatsavai, Ph.D.	Dr. Timothy J. Menzies Chair of Advisory Committee

DEDICATION

To Ammayi—for timely advice and support

BIOGRAPHY

Vivek Nair was born and raised in the city of Kolkata, India. He graduate from West Bengal University of Technology and National Institute of Technology, Durgapur with and Bachelors in Technology and Master in Technology respectively. His primary research interest is search based software engineering with a primary goal to solve search problems to maximize the quality of the search while minimizing the cost of search. His current research investigated problems in the software engineering domain and developed techniques not only to solve software configuration problem, but also software product lines as well as cloud architecture tuning problems. Before joining the PhD program, he had two years of professional experience in the Samsung Software Engineering Labs. During his PhD program, his internship experiences include Lexisnexis Risk Solutions (2015-2017), and Microsoft Research - Redmond, USA (Summer 2018). He is a student member of IEEE.

ACKNOWLEDGEMENTS

**This acknowledgment contains plenty of subtexts and would be found as text within brackets. The subtext is meant to serve the purpose of comic relief and comic relief only.*

This thesis is no way an individual work. It is a product of a lot of love and sacrifice, tears and sweat of not just me but for various individuals who have taught lessons throughout my journey as a graduate student, here at NCSU. This text below is no way a comprehensive list of all the people who have touched my life [for better or for worse], but it has been a useful experience.

One of the primary reasons, I can write this thesis is because of Dr. George N. Rouskas and my aunt, Dr. Latha Unni. My first year at NCSU was tumultuous and eventful. As new graduate student, dealing with the stresses of graduate school and the toxic environment of my previous lab was overwhelming. This resulted in me deciding to quit graduate school—which I have been so excited to attend. Dr. Rouskas, who provided me with financial support while transitioning from my old advisor to my current advisor. Both of them counseled and provided support when I required it the most. This thesis would have never been possible without these two amazing people.

Secondly my thesis advisor [my academic father], Dr. Menzies—he is a legend [faking the Aussie accent]. He has been very [not so] patient with me and directed me towards areas which held promise. He helped me understand the value of collaboration and being a voracious reader of the literature. That said I have never had a more dramatic relationship with anyone—which in retrospect was an effective way of training graduate student. Our story has all the elements of a soap opera—there are love and respect, no so much love [hate sounds just too strong] and doubt. Either way, he played a massive part in making this thesis possible and made me a better researcher [hopefully].

Thirdly, I would then like to thank my dissertation committee members, Dr. Rangaraju Vatsavai, Dr. Min Chi and Dr. Kathryn T. Stolee, for their valuable feedback and insight on my dissertation.

I am also grateful to my research collaborators: Sven Apel (University of Passau), Norbert Siegmund (Bauhaus-University Weimar), and Pooyan Jamshidi (University of South Carolina). I would like to gratefully acknowledge researchers who generously shared their research tools and results used in my dissertation. I am very much thankful to all my peers at the RAISE Lab, for their constant support, insightful discussions and useful feedback on my research. Dr. Wei Fu, Dr. Chin-Jung Hsu, Rahul Krishna, George Mathew (also my room-mate), and Zhe Yu for the insightful and exciting conversations [interestingly I was able to write papers with all these

amazing individuals—so all our discussions were not completely useless]. I thank my internship mentors in the industry, who have immensely helped me expand my research to a broader scope: Dan Camper and Arjuna Chala (Lexisnexis Risk Solutions), and Chris Duvarney, Kim Herzig, and Hitesh Sajnani (Microsoft Research, USA).

Heartfelt thanks go to Kathy Luca, Carol Allen, and all the helpful staff at the Department of Computer Science. I thank Amruthkiran Hedge, Anand Gorthi, Sandesh Saokar, Siddartha Chauhan, Mayank Vaish, Akhilesh Tanneru [Mr. T], and George Mathew for sharing the apartment and making my stay at Raleigh eventful. Special thanks to Karen Warmbein, Blue and [Z]Simba whose love and support helped me survive the grind of Ph.D.

Finally, last but not least, I am very thankful to my family. My family was my constant source of support and encouragement throughout my journey as a doctoral student. My Ph.D. journey would not have gone so far, without you.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Introduction

Modern software systems nowadays provide configuration options to modify both functional behavior of the system, i.e. functionality of the system, and non-functional properties of the system, such as performance and memory consumption. Configuration options of a software system that are relevant to users are usually referred to as features. All the features of a system (vector of configuration options) together defines a *configuration* of a software system. The features can often taken integer, decimal or string values. One of the most important non-functional properties is performance, because it influences the how a user interacts with the system. Performance can be influenced by many factors including the environment (for example, the hardware in which the software system is current executing). A software system is required to select and set configuration options to maximize the performance of that system. For example, say we have a software system with 10 (binary) configuration options—it results in a configuration space of size 2^{10} or 1024. The user of the software system, now has to find the optimal configuration for the given task (or input) in hand. This problem can be tackled in two different ways: (1) exhaustively measuring performance of all possible configurations—which means running 1024 benchmark runs, and (2) use domain knowledge (assuming the user has tuned similar software system before) to find the best configuration. However, as the number of configuration options increase, it becomes difficult for humans to keep track of the interactions between the configuration options. This means as the configuration space grows (size of the configuration space grows exponentially) it is harder to either exhaustively measure performance for all possible configurations or find domain experts to confidently do so. Please note that the optimal configuration we are trying to find can change dramatically with different inputs (tasks) and the environment—which make domain knowledge based decision less reliable.

This exact problem has been reported by numerous researchers from different domains.

- Many software systems have poorly chosen defaults [1], [2]. Hence, it is useful to seek better configurations.

- Understanding the configuration space of software systems with large configuration spaces is challenging [3].
- Exploring more than just a handful of configurations is usually infeasible due to long benchmarking time [4].

The problem we are trying to tackle throughout this document is: "How can be find a set of configuration options which would maximize the performance of a system while minimizing the cost of search". Here were would limit our scope of study to just the configurations options or features of a particular software system (and not its environment).

Conf.	Features																Perf. (s)
$\mathbf{c}_i$	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9	x_{10}	x_{11}	x_{12}	x_{13}	x_{14}	x_{15}	x_{16}	y_i
$\mathbf{c}_1$	1	1	0	0	1	1	0	0	1	0	0	1	1	1	0	0	292
$\mathbf{c}_2$	1	0	0	0	1	1	0	1	1	1	0	0	1	0	1	0	571
$\mathbf{c}_3$	1	1	1	0	0	1	0	0	1	0	1	0	1	0	0	1	681
$\mathbf{c}_4$	1	1	0	1	1	0	0	0	1	0	1	0	1	1	0	0	263
$\mathbf{c}_5$	1	1	1	0	1	1	0	0	1	0	1	0	1	0	1	0	536
$\mathbf{c}_6$	1	0	0	1	0	1	1	1	1	0	1	0	1	1	0	0	305
$\mathbf{c}_7$	1	1	1	1	0	1	1	0	1	0	1	0	1	0	1	0	408
$\mathbf{c}_8$	1	0	0	1	1	0	1	1	1	0	1	0	1	1	0	0	278
$\mathbf{c}_9$	1	0	0	1	1	0	0	1	1	0	0	1	1	0	1	0	519
$\mathbf{c}_{10}$	1	1	0	0	1	1	0	1	1	0	1	0	1	0	0	1	781
$\mathbf{c}_{11}$	1	0	0	1	1	1	0	1	1	1	0	0	1	0	0	1	822
$\mathbf{c}_{12}$	1	1	0	0	1	0	0	0	1	0	1	0	1	0	0	1	713
$\mathbf{c}_{13}$	1	0	1	0	0	0	1	0	1	0	0	1	1	0	1	0	381
$\mathbf{c}_{14}$	1	0	1	1	1	1	0	1	1	1	0	0	1	0	1	0	564
$\mathbf{c}_{15}$	1	1	1	1	1	0	0	0	1	0	0	1	1	0	1	0	489
$\mathbf{c}_{16}$	1	1	1	1	0	1	1	1	1	1	0	0	1	1	0	0	275

Figure 1.1 A sample of 16 randomly-selected configurations of x264 and corresponding performance measurements (seconds)

1.0.1 Motivating Example

To motivate our work, we use the same example from our previous work citeGuoXXX, a configurable command-line tool x264 for encoding video streams into the H.264/MPEG-4 AVC format. In this example, we consider 16 encoder features of x264, such as encoding with multiple reference frames and parallel encoding on multiple CPUs. The user can select different features to encode a video. The encoding time is used to indicate the performance of x264 in different configurations. A configuration represents a program variant with a certain selection of features. This example with only 16 features gives rise to 1,152 configurations. Intuitively, 16 binary features should provide 216 different configurations, however, in this work we consider only valid

configurations i.e. configurations that are allowed by the system under investigation. In practice, often only a limited set of configurations can be measured, either by simulation or by monitoring in the field. For example, Figure 1.1 lists a sample of 16 randomly selected configurations and their actual performance measurements. How can we determine the performance of other configurations based on a small random sample of measured configurations? To formulate the above issue, we represent a feature as a binary decision variable x. If a feature is selected in a configuration, then the corresponding variable is set to 1, and 0 otherwise. Assume that there are N features in total, all features of a program are represented as a set X = x1, x2, ..., xN . A configuration is an N-tuple c, assigning 1 or 0 to each variable. For example, each configuration of x264 is represented by 16-tuple, e.g. c1 = (x1 = 1, x2 = 1, x3 = 0, x4 = 0, ..., x16 = 1). All valid configurations of a program are denoted by C. Each configuration c of a program has an actual measured performance value y. Per

This dissertation is organized as follows. Chapter 2 presents the background and related work. Next, Chapter 3 describes the design and implementation of *Inside-Out*, an approach to leverage low-level performance information to achieve accurate and robust performance prediction. In Chapter 4, we define cloud architecture tuning (CAT) and describe its challenges. We also discuss the state-of-the-art approaches and their pros and cons. Chapter 5 proposes using low-level performance information to enhance Bayesian Optimization, an online learning method to optimize any black-box function. In Chapter 6, we design and implement Scout, an efficient, effective and reliable CAT solution. We also consider optimizing a group of workloads in Chapter 7. In Chapter 8, we consider data placement to further optimize system performance. Finally, Chapter 9 concludes our discussion of cloud architecture tuning and describes its future directions.

Chapter 2

Background and Related Work

This chapter describes the necessary background that is related to the research problems addressed in this dissertation. We first describe data-intensive computing and its storage architecture. Next we discuss performance prediction for distributed systems. Last, we discuss related work that uses the data-driven approach to optimize system performance.

2.1 Data-Intensive Computing

Data-intensive computing is a class of parallel computing that processes large volumes of data and incurs significant processing time on I/O. Data-intensive computing is extremely important because of the desire to extract information from data [57, 74]. It is critical to design robust and efficient distributed systems for running applications that require significant computation and massive storage. Many systems such as Hadoop, Spark, and HPCC store and process large-scale data [8, 9, 59]. There is also a large body of work in optimizing data-intensive systems [5, 32, 41, 78, 102]. As more and more data-intensive computing is moving to the cloud, supporting data-intensive computing in cloud has emerged as a critical task.

This section describes two popular execution models, *MapReduce* and *Dataflow*, in data-intensive computing. We also describe *Apache Hadoop*, *Apache Spark*, and *HPCC*, which are the widely deployed systems for data-intensive computing.

MapReduce Programming Model

The MapReduce programming model is a simplified model for data processing. This model consists of two functions: *map* and *reduce*. The *map* function filters and sorts the input, and the *reduce* function summarize the output from the *map*. Although the MapReduce model is embarrassingly parallel, many data-intensive applications can be implemented using such a model [43, 85]. Examples are large-scale indexing, machine learning problems, and graph

computation [8, 36]. Applications that exploit the MapReduce model are highly scalable because it requires only loose synchronization [107]

Figure 2.1 illustrates the three phases in a MapReduce job. First, the map phase processes a portion of input data, *e.g.*, one line of a file or a XML document, and generates a list of key-value pairs. Second, the shuffle phase aggregates multiple lists of key-value pairs, and groups them by the key. The results are then redistributed to corresponding reduce tasks. Finally, the reduce phase processes the aggregated output.

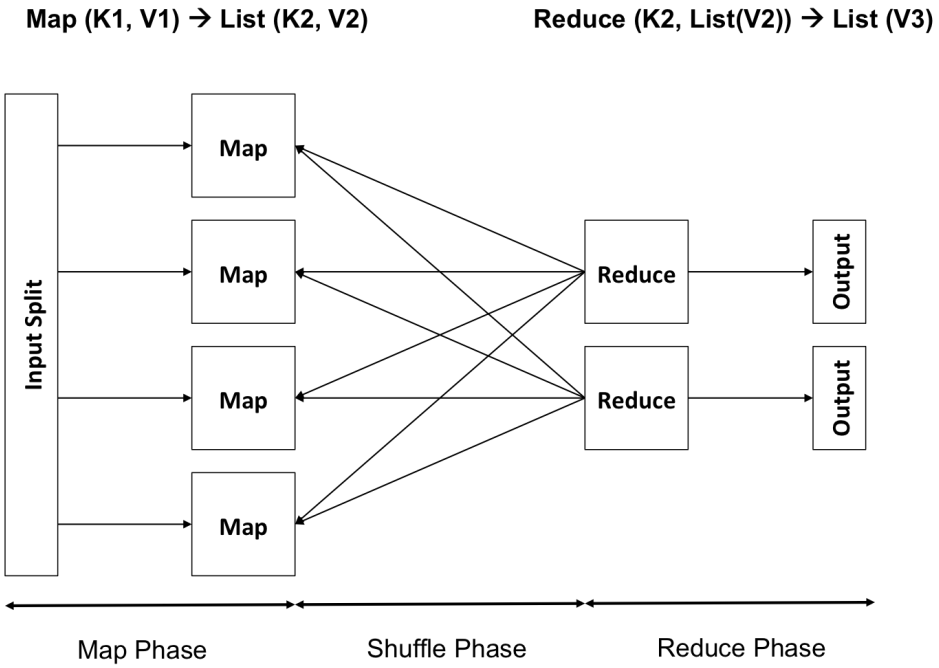


Figure 2.1 A MapReduce job consists of the map, shuffling and reduce phase. The map task processes a portion of input data and the reduce task aggregate the output from map tasks. The phase between the map and the reduce phase to dispatch intermediate result is the shuffling phase.

Dataflow Execution Model

Although the MapReduce model is well suited to many applications, it is limited as well to some such as iterative machine learning and graph processing [13]. The DataFlow execution model is a more generalized form for expressing varied data access and communication patterns. Therefore, the dataflow execution model is able to support a larger category of data-intensive applications than the MapReduce model.

There are several variants of the dataflow model. *Dryad* is a general-purpose distributed execution engine that supports of coarse-grain data parallelism [64]. Vertexes are the programs and edges represent communication channels. *Dryad* distributes computation vertexes to a set of distributed computers. Their communication are handled by either files, TCP pipes, and shared-memory FIFOs. Other attempts include bulk synchronous parallel processing in *Pregel* [79] and serving trees in *Dremel* [81].

Apache Hadoop

Apache Hadoop is an open-source implementation of the MapReduce programming model, and it is a reliable and scalable system for data processing [8]. Apache Hadoop includes four modules: 1) Hadoop YARN is a cluster resource management framework, 2) Hadoop MapReduce is the implementation to support the MapReduce programming model based on YARN, 3) Hadoop HDFS is a distributed storage system for Hadoop application data, and 4) Hadoop Common is the common utilities that are required by the above modules.

The Resource Manager (RM) is responsible for resource allocation. Once received a MapReduce job, RM allocates resource, *e.g.*, CPU and memory, to initialize a AppMaster. This AppMaster is created to negotiate computing resources with the resource scheduler, *e.g.*, FIFO Scheduler, Capacity Scheduler [25], and Fair Scheduler [42]. These schedulers allocate resources (or slots) to the AppMaster based on objectives such as data locality or fairness. Those allocated resources can be used to run map or reduce tasks.

After obtaining computation resources, the AppMaster starts multiple node containers to process input splits. Each map task handles an input split, and the size is usually 64MB or 128MB (the block size of HDFS). The map task uses *RecordReader* and *FSDataInputStream* to access input data as shown in Figure 2.2. It is possible to store data on various storage systems such as parallel file systems and object storage.

Apache Spark

Apache Spark is an in-memory data processing engine [9]. Apache Hadoop uses a linear data pipeline, which reads and writes data into disks in the map and reduce phase. This implementation does not fit well to iterative computation, which is a common requirement for applications such as in machine learning.

Resilient distributed dataset (RDD) is proposed to improve performance of application that requires iterative computation [139]. RDD is read-only and designed to be fault-tolerant. RDD serves as the working set to Spark applications in a similar form of distributed share memory [139]. Apache Spark has shown an order of magnitude performance improvement, *e.g.*, 20 times, over Apache Hadoop in many applications [9, 139].

High-Performance Computing Cluster (HPCC)

High performance computing cluster (HPCC), also known as data analytics supercomputer, was developed by LexisNexis Risk Solutions [59]. It is a parallel system that is designed for processing large-scale data. Thor is a *data refinery* that processes large datasets in parallel. It is a batch processing engine that was designed for tasks similar to those that MapReduce handles best. Roxie is a *data delivery engine* that responds to queries. It finds the answers to requests in an index that is partitioned and, if desired, replicated across the nodes. Thor generates indexes for Roxie.

The division of labor, into data refinery and delivery engine, allows each cluster to be optimized for its task. Thor is optimized for processing large datasets in parallel where the goal is end-to-end throughput. Roxie is optimized to handle massive amounts of concurrent requests with low latency.

2.2 Storage Architecture for Data-Intensive Computing

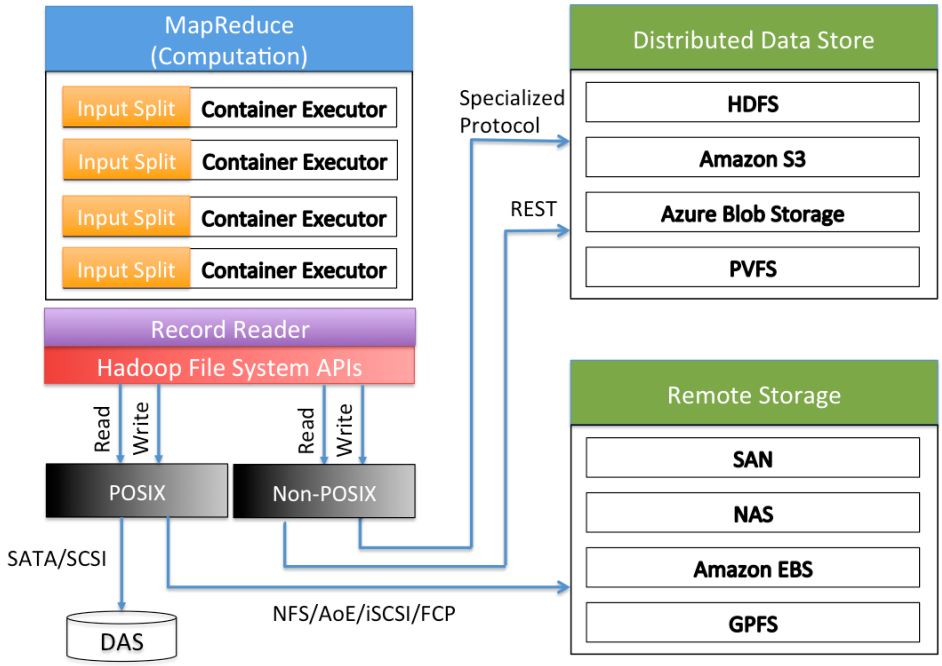


Figure 2.2 System configuration of Hadoop. Each slot handles one input split and uses the RecordReader object to read data from a POSIX compliant file system or a non-POSIX compliant file system, e.g. distributed data store.

The most common way to deploy a Hadoop system is to configure a node to run both Hadoop MapReduce and Hadoop HDFS, which can avoid bringing data to computation. However, data management, in many cases, requires separating computing and storage nodes for flexibility and efficiency. For example, enterprises prefer silos in order to manage high-value data. Amazon Elastic MapReduce primary uses Amazon S3 for its persistent data storage. Moreover, many high performance files systems are considered to replace HDFS in order to support both MapReduce application and other workload. As a result, different scenarios require different ways to deploy Hadoop. As shown in Figure 2.2, we can define three major types of Hadoop configuration as follows.

Previous work compared the performance of Hadoop integrated with different types of storage architecture [107]. The author found that in the *split architecture*, in which compute and data nodes are not co-located, Hadoop has a imbalance issue of accessing data. This can lead to poor I/O performance. In the following, we discussed the use cases that integrate Hadoop with separate storage. After that, we describe scheduling methods for Hadoop and cluster that are related to our work.

Parallel File System

Several research studies show their interests in replacing HDFS with other high performance storage systems. W. Tantisiriroj et al [119] argues that parallel file systems can support diverse workloads and provides a better tradeoff between performance and reliability. The authors proposed a PVFS shim layer to incorporate data layout of PVFS to achieve data locality. Maltzahn *et al.* considered Ceph as a scalable alternative to HDFS [80]. The authors create a mapping layer which is similar to the PVFS one. GPFS is a shared-disked file system developed by IBM, and widely adopted in supercomputers. R. Ananthanarayanan *et al.* from IBM Research modify data layout in GPFS, and expose this information to Hadoop [6]. These are the attempts to replace HDFS.

Enterprise Storage

Another research study analyzed the feasibility to use a very powerful storage node to accommodate Hadoop [99]. The study finds that Hadoop performance is dominated by the bandwidth between computing and storage facilities. Some argue that is is necessary to decouple compute and storage nodes n big data analytics because enterprise IT often deploys *silos* to manage high-value data. However, decoupled Hadoop model would incur high cost on data loading from the back-end storage system to compute nodes. MixApart includes the data-aware task scheduler, the task-aware data scheduler and a caching mechanism, to optimize Hadoop performance [84]. It mainly focused on workloads that repeat regularly.

Cloud Storage

Cloud storage typically refers to object storage hosted in cloud. Amazon S3 and Azure Blob Storage are two examples. Object storage manages data as objects. Different from POSIX-compatible file systems, RESTful API is a common way to access object storage. Object storage is the primary data repository on the cloud. Many data-intensive systems, *e.g.*, Amazon Elastic MapReduce and Azure HDInsight, support using object storage as the back-end storage [4, 132]. It is generally considered suboptimal than the Hadoop reference architecture [8].

2.3 Performance Prediction and Optimization

In this section, we describe the techniques to capture workloads and predict workload and system performance. We also discuss using data-driven approach to optimizing performance.

2.3.1 Workload Characterization

Storage performance modeling has been extensively explored in prior work. Three common modeling techniques are analytical, simulation, and data-driven approaches [10, 69, 110]. The analytical model requires domain knowledge to manually identify the factors that affect performance[69, 110]. Kelly et al. use a probability model to predict response time for an enterprise storage array. Ruemmler et al. found that a disk is too complex to model with analytical methods and designed a disk simulator to characterize storage behavior [104]. However, the simulation approach becomes inefficient when searching a large design space [69].

Mesnier et al. [82] propose a novel black-box approach that can describe the performance difference between two storage devices. With this approach, we can study the performance difference between two configurations and create a combined model with better prediction accuracy. Bodik et al. propose an exploration policy for quick collection of essential data required to train a performance model [17]. This policy can reduce the time required for offline and online model training. Chen et al. propose SLA decomposition that combines profiling and queuing model to derive resource thresholds for meeting application SLA [29].

2.3.2 Machine Learning

The data-driven or the measurement-based approach uses measurement data to derive a prediction model. Wang *et al.* adopt classification-and-regression-tree (CART) to predict the response time of a single disk and a disk array [124]. The authors propose request-level and workload-level device models for different prediction granularities. Yin et al. also use the regression tree to predict storage throughput and latency [135]. Their work mainly focuses on multiple workloads,

and proposes a scalable model, by combining related workload features. Noorshams et al. extensively analyze four different types of algorithms (including linear regression and CART models) and apply to IBM storage servers [89]. They also propose an optimization technique to search for parameters that can improve prediction accuracy. Their proposed parameter optimization complements our work for improving prediction accuracy. Machine learning has also been applied to performance modeling for virtual machines (VMs). DeepDive uses the classification technique to detect performance anomaly among VMs [90]. In [75], the authors apply regression and artificial neural network to model performance of a single VM.

2.3.3 Low-Level Insights

Low-level performance information is leveraged to identify performance bottlenecks and to predict application performance. DeepDive is designed to identify performance interference of co-existing VMs [90]. Wang *et al.* propose using the CART model to predict storage performance. Their approach requires workload information, which may not be practical for our problem setting. Inside-out provides reliable performance prediction of distributed storage service by using only low-level performance information [60]. The authors show that high-level performance can be accurately captured by only the low-level metrics. This accurate prediction model can be used to adjust resource allocation for meeting performance objectives.

2.3.4 Sequential Model-based Bayesian Optimization

Sequential model-based optimization (SMBO) is an optimization method that supports any black-box function [37]. SMBO is naturally applicable to finding the best VM. SMBO iteratively measures solutions (VM types) to optimize for an objective (execution time or deployment cost). A typical SMBO algorithm is described in Algorithm 1. An SMBO algorithm requires 4 inputs namely, a cloud set up to run a workload (f), list of VM characteristics ($vm \in VM$) or instance space, an acquisition function (S), and a choice of surrogate model (M). SMBO starts with an initial sample of VMs (chosen randomly), which are then measured (D). SMBO builds a surrogate or a machine learning model to estimate to predict workload performance. This model is constructed using VM characteristics and the measured performance. A VM is selected based on the surrogate model along with a predefined acquisition function. The selected VM (x_i) is then measured (f). The VM (x_i) along with performance (y_i) is then added to the already measured VMs ($D = \{(vm_1, y_1), (vm_2, y_2)\}$). This process terminates after a stopping criterion is reached.

Algorithm 1: Sequential Model-Based Optimization for CAT

Input: f, VM, S, M
Output: Near-optimal configurations

```
D := Initial sampling (f, VM)
for k in |VM ∉ D| do
    p(y|vm, D) := fit a surrogate model (M, D)
    vm_k := argmax_{vm∈VM} S(vm, p(y|vm, D))
    y_k := f(vm_k)
    D := D ∪ (vm_k, y_k)
    if meeting stopping criteria then
        break
    end
end
```

2.3.5 Bayesian Optimization

Bayesian Optimization (BO) follows the same formalism of sequential model-based optimization (SMBO). Like SMBO, BO has two essential components namely a (probabilistic) regression model, and an acquisition function (refer to [108] for more details.) BO has been used as a drop-in replacement to standard techniques such as random search, grid search and manual tuning in numerous domains such as hyperparameter tuning and software performance optimization [37, 50, 86, 141]. To solve the CAT problem, *CherryPick* used BO to find the best VM for a specific workload [3].

In Bayesian Optimization, Gaussian Process is the standard probabilistic model used for building the surrogate model. Gaussian Process is a distribution over objective functions specified by a mean function and covariance function. Once a surrogate model is trained, it can be used to estimate performance (of a workload) on the unmeasured VM. The surrogate model returns distribution of the estimated performance associated with the VM (mean and variance). The next VM to measure is determined by an acquisition function. Common acquisition functions are Probability of Improvement (PI), Expected Improvement (EI), and Gaussian Process Upper Confidence Bound (GP-UCB) [108]. Recently, the entropy search methods, backed by information theory, are promising alternatives [125]. In practice, EI is effective and used in *CherryPick*.

An important component in Gaussian Process is the covariance kernel function, which is crucial for model effectiveness. Covariance kernel ensures that the prior, required for GP to be effective, is met. GP assumes smoothness, or in other words, the VMs which are closer to each other in instance space have similar performance. This is particularly difficult in our problem setting, where a slight difference in the instance space can lead to significant differences in performance (cost or time). This indicates that before using BO (with GP as a surrogate model),

a practitioner needs to choose a proper kernel function to ensure smoothness in the instance space. Such a task is challenging and can affect the performance of BO. *CherryPick* chooses the *Matérn 5/2* kernel function because it does not require strong smoothness, which are the cases for many real-world applications [3, 133].

2.3.6 Hyper-Parameter Tuning

System and software performance is highly affected by configurations. StarFish is an auto-tuning system for Hadoop applications [56]. *BestConfig* proposes the Divide and Diverge Sampling strategy along with the Recursive Bound and Search method for turning software parameters [140]. Similar framework is also proposed to automate tuning system performance of stream-processing systems [16]. BOAT is a structured Bayesian Optimization-based framework for automatically tuning system performance [34] which leverages contextual information.

BO4CO uses Bayesian Optimization to continuously optimize system performance [66]. Similar to *CherryPick*, *BO4CO* leverages Bayesian Optimization but does not consider low-level metrics. *BOAT* is a structured Bayesian Optimization-based framework for automatically tuning system performance which leverages contextual information [34]. *BOAT* combines the parametric and non-parametric model for better predicting the trend in system performance. The idea behind their work and our work is very similar: leveraging domain knowledge to enhance BO. *BOAT* optimizes software configurations while *Arrow* tunes architecture (virtual machines). Parameter tuning is critical to machine learning application [37, 50, 72, 108].

Bilal *et al.* propose a framework to automate tuning system performance of stream-processing systems [16]. Their modified hill-climbing search with heuristic sampling inspired by Latin Hypercube shortens the search process by two to five times. The above methods reduce the search cost by a significant degree. However, they focus on performance tuning for the same workload (or application) on the same type of machine. It is not clear how to leverage their approaches to support different machine configurations in cloud computing. We, instead, find the best machine configuration for a given workload.

Sampling techniques focus on reducing sampling cost while building accurate models to optimize software systems [86, 91]. The above methods reduce the search cost by a significant degree. However, they focus on performance tuning for the same workload (or application) on the same type of machine. It is not clear how to leverage their approaches to support different machine configurations in cloud computing. We, instead, find the best machine configuration for a given workload.

In the literature, software configuration optimization [16, 34, 56, 140], program parameter tuning [50, 72] and sampling techniques [86, 91] are active research directions. They all focus on the same machine configuration. It is not clear how to apply their approach directly to cloud

environments, where workloads perform very differently on distinct cloud configurations, *e.g.*,VM types.

Chapter 3

End-to-End Performance Prediction for Cloud Storage

This chapter presents an approach to estimating end-to-end performance of distributed storage systems. We explain why low-level performance metrics are a desirable proxy for estimating end-to-end performance. We then present our automatic model building tool for generating robust and accurate performance models.

3.1 Introduction

Many storage systems are moving away from dedicated appliance-based storage model to software-defined storage (SDS), which separates software that provisions and manages storage from the hardware that provides raw physical storage [65, 113, 120]. This trend is partly driven by the tremendous growth of data and the emergence of cloud applications that operate in a multi-tenant environment with diverse workload characteristics. As a result, the rigid appliance-based model, with tightly-coupled hardware and software features, is no longer cost-effective, lacks flexibility, and does not scale well. SDS systems are increasingly abandoning centralized storage services in favor of distributed systems like Ceph [26], HDFS [8], Swift [93]. Distributed storage systems are attractive because they scale well, allowing storage services to grow or shrink, based on storage demands. They are also better suited to handle diverse multi-tenant workloads.

Providing reliable quality of service (QoS) to storage applications is critical in an SDS environment shared by multiple applications with diverse usage patterns. However, in a distributed storage environment, it is challenging to provide storage QoS in a consistent and reliable manner. Practical deployments of modern distributed storage systems like Ceph are composed of a large number of individual storage components that can interact in a complex manner.

Diverse and time-varying storage workloads and performance interference in a multi-tenant environment further complicate the reliable assurance of storage QoS. Reliable and accurate monitoring of high-level storage performance metrics (e.g. throughput and IOPS) is critical for providing storage QoS guarantees. However, monitoring end-to-end storage performance is difficult in a distributed storage service. Instrumenting user applications to measure storage performance is not always practical. Performing benchmark tests in production systems also has practical limitations since they interfere with storage application workload. Furthermore, running exhaustive benchmark experiments to cover diverse application workloads, deployment topologies, and large configuration parameter space is time-consuming and impractical in many cases. Building accurate analytical performance models, on the other hand, is also difficult for the reasons mentioned above.

This chapter proposes the idea of using low-level system metrics (e.g., CPU usage, RAM usage and network I/O) as a proxy for measuring high-level performance (e.g., end-to-end IOPS and throughput) of distributed storage applications. We design, implement and evaluate a practical tool, called *Inside-Out*, that applies machine learning techniques to the low-level metrics collected from individual components of a distributed storage system to accurately estimate high-level storage performance metrics—like throughput and IOPS—of the entire distributed storage system. We believe that a tool like Inside-Out can serve as an important component of the overall SDS architecture.

Inside-Out takes a black-box modeling approach, which does not require knowledge about distributed storage system protocol, workload characteristics, and deployment topology. Inside-Out relies upon machine learning techniques to automatically derive an accurate end-to-end performance model. We explore several well-known machine learning algorithms including linear regression, decision tree learning, and ensemble methods [89, 124], and conclude that there does not exist an one-size-fits-all algorithm that can work in all prediction cases. Hyperparameter tuning [28, 89], model selection [73] and feature selection [53, 105] all turn out to be too complicated for optimizing prediction accuracy. In contrast, Inside-Out uses a two-level learning method that automatically selects important features, boosts prediction accuracy, and achieves consistent prediction. This two-level learning method pipelines two supervised learning algorithms to eliminate irrelevant features while avoiding overfitting problems.[1]

Inside-Out offers several key benefits. Unlike traditional analytic performance modeling approach, Inside-Out is more generic, and therefore can be more easily applied to different storage services. Different from previous work, Inside-Out does not require information about system configuration and application workload [10, 69, 89, 104, 110, 124, 135]. Due to the self-learning property, *Inside-Out* improves performance prediction accuracy with more data. It

[1] Overfitting describes the situation when a model captures the relationship of noisy data but not the underlying relationship [39]. Overfitting becomes more prominent in the presence of high dimensional data

can also adapt to changes in the system by continuously learning the system behavior.

We evaluate Inside-Out using Ceph [26] running on an OpenStack-based SDS platform. The low-level performance metrics are collected from participant virtual machines running various components of a Ceph storage service.[2] Our in-depth evaluation shows that Inside-Out generates end-to-end performance models with 91.1% prediction accuracy on average. More importantly, as discussed above, Inside-Out is generic in nature as it captures the behavior of the storage system by analyzing low-level system metrics (that are protocol and application agnostic). Furthermore, we demonstrate that Inside-Out can provide reliable hints for performance monitoring tasks even in the presence of evolving workload characteristics, changing storage configuration and interfering tenants. We also show that Inside-Out is reliable in estimating end-to-end performance even when the storage system expands or shrinks. We show that Inside-Out provides reliable performance prediction when the storage system is up to four times larger than the one used for building machine learning models during the training phase. Lastly, Inside-Out is able to learn new storage behavior over time.

3.2 Mapping from Low to High

This section discusses the guiding principles and challenges in using low-level performance metrics to build accurate end-to-end performance models for a distributed storage system.

3.2.1 Important Considerations

This section describes how we use low-level performance metrics to predict high-level system performance. We discuss how we pick the metrics and how to transform metrics to meaningful features.

General low-level metrics

Since our goal is to provide a tool for estimating the end-to-end performance of a diverse set of storage systems, the inputs to our model need to be generic in nature, *i.e.*, they need to be independent of storage systems or the distributed protocols used by such applications. An SDS provider should be able to obtain the input metrics without instrumenting storage application or requiring domain knowledge about the storage application. Low-level system metrics (e.g. CPU utilization, memory usage, network IO, etc.) satisfy these requirements. DeepDive uses low-level metrics to identify performance anomaly for a running VM [90]. To the best of our

[2] Our approach is not limited to VM-based environments. It can be applied to container-based and bare-metal storage servers as well.

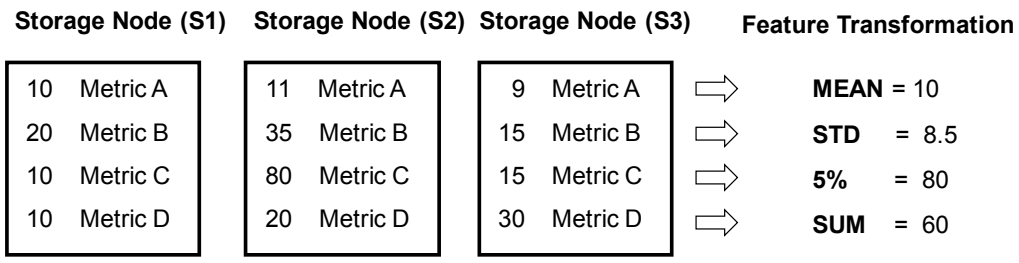


Figure 3.1 Four statistical features used in Inside-Out to capture load and internal status of a distributed storage system. The numbers and metrics represent low-level performance data collected from storage nodes.

knowledge, this work presents the first study that maps low-level system metrics to high-level end-to-end performance of a distributed storage service.

Capture important features of a distributed storage system

A distributed storage system can dynamically expand or shrink according to demand. The performance model has to capture the current scale of the deployment, the bottlenecks, and the average and variance in performance of individual components of the distributed system. For each low-level system metric collected from various components of the distributed system, we use four statistical variables to characterize the behavior of a distributed system (see Figure 3.1). The statistical variable *mean* and *std* describe whether the impact of the workload is evenly distributed among storage components. The *sum* variable represents the scale of the deployment, while the variable *5%* (top 5 percentile) captures the hot spot situations. The feature transformation from raw system metrics to these four statistical values also allows Inside-Out to apply the uniform input format for developing performance models for distributed systems at different scales.

3.2.2 Feature Selection

In this work, we collect low-level performance metrics from two components of Ceph namely monitor (MON) and Object Storage Daemons (OSD). We use *dstat*, a monitoring tool to collect resource statistics, to collect 32 low-level performance metrics in total. These measurements are then transformed using the process described in Figure 3.1 (refer to Section 3.3.1 for more details).

Selecting the "right" features from high dimensional data is a challenging task because as computation complexity increases, prediction accuracy may decreases [53, 105]. Furthermore, for our case, the right feature set is not always the identical. Table 3.1 shows the *model accuracy*

Table 3.1 Important features selected by different algorithms are not deterministic

Lasso			Ridge			Elastic Net			Decision Tree			Random Forest		
osd	network.send	sum	**osd**	**network.recv**	**mean**	osd	network.send	sum	osd	disk.read	sum	osd	disk.read	sum
osd	disk.writ	sum	**osd**	**disk.read**	**5%**	osd	disk.writ	sum	osd	network.send	sum	osd	network.send	sum
osd	**cpu.sys**	**sum**	**osd**	**load.15m**	**std**	osd	disk.read	sum	osd	network.recv	sum	osd	disk.writ	sum
osd	io.read	sum	osd	network.send	sum	osd	cpu.sys	sum	osd	disk.writ	sum	**osd**	**network.recv**	**sum**
osd	vm.minpf	mean	osd	tcp.tim	std	osd	tcp.lis	sum	**mon**	**memory.buff**	**mean**	osd	memory.cach	sum
mon	**memory.used**	**5%**	osd	network.recv	std	osd	io.read	std	osd	cpu.sys	sum	osd	memory.buff	mean
mon	**memory.cach**	**5%**	osd	load.5m	std	osd	io.read	sum	osd	vm.alloc	sum	osd	memory.buff	5%
osd	tcp.lis	sum	osd	cpu.idl	sum	osd	vm.minpf	mean	osd	vm.minpf	5%	mon	io.writ	sum
osd	io.read	std	osd	cpu.wai	sum	osd	io.writ	mean	**mon**	**cpu.idl**	**std**	osd	memory.buff	sum
osd	io.writ	mean	osd	cpu.sys	sum	**mon**	**memory.used**	**sum**	mon	memory.cach	sum	mon	vm.free	sum
96.20%			96.60%			96.18%			96.78%			96.94%		

of different learning methods when modeling read throughput. We see that all learning methods achieve high model accuracy even though they choose different features. The model accuracy was obtained using k-fold cross validation (k=10), a common technique for assessing model accuracy. The training data is partitioned into k disjoint sets. A single data partition is used for validation purpose and the remaining $k-1$ partitions are used for training data. Although all models yield good model accuracy, they perform poorly and inconsistently when the storage environment changes. In Figure 3.2, we show the prediction accuracy under three types of changes in the storage environment—increase in the size of the distributed storage system, read workload and individual storage IO request size. These algorithms (discussed later in Section 3.3.2) do not yield consistent prediction accuracy any more. For example, Lasso can still predict well when workload has changed but Decision Tree cannot. On the contrary, Decision Tree performs better than Lasso when the size of the storage system increases. We suspect this is caused by the large feature space, which leads to the overfitting problem [39, 55]. Next, we manually remove most features and select only a few with a trial-and-error strategy. As shown in Figure 3.2, we see significant improvement in some cases, but not all. Since an SDS environment can change over time, it is important for our model to provide consistent prediction accuracy under system changes such as software reconfiguration and cluster expansion.

Although Hyperparameter tuning, model selection, and feature selection have been proposed as potential solutions, it is challenging to use them in practice, not to mention the complexity of automating this task. PCA (Principle Component Analysis) is another potential solution [109]. PCA transforms original data into a lower dimension while keeping high fidelity. However, PCA has several limitations. First, PCA is not scale invariant. Not all performance metrics are comparable and therefore, there is no standard way to scale these metrics. Second, PCA assumes Gaussian distribution in data points; however, many storage workloads have Pareto distribution [71]. Third, determining a good number of components is also a challenging task. In our case, PCA does not address the problems. In fact, Figure 3.2 shows that it can further degrade prediction accuracy.

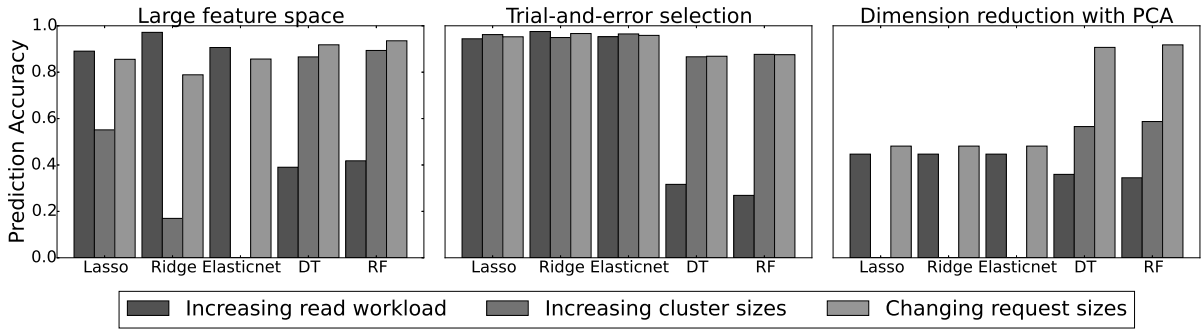


Figure 3.2 Prediction accuracy is inconsistent due to the large feature space. Learning methods fail to select the right features in some cases. Dimension reduction (PCA with 10 components) does not help in this case. In the trial-and-error case, we select a subset of metrics, e.g. *mean(disk.read)*, *sum(network.recv)* and *std(cpu.usr)*.

3.2.3 A Two-Level Approach

Instead of performing feature selection or dimension reduction, we propose a generic two-step approach that can improve the consistency of prediction accuracy. In the first step, we use some heuristic methods to filter out irrelevant features. Then, in the second step, we apply machine learning algorithms to build performance models with the reduced feature set. The intuition behind this idea is that it is difficult to determine the most important performance features but it is relatively easy to eliminate unimportant features. For example, the features which are not in the top 100 list after step one can be labeled as unimportant features.

3.3 The Inside-Out Design

In this section, we present the design of Inside-Out. We also discuss the trade-offs among a set of representative machine learning algorithms and propose a two-step learning technique for mitigating overfitting problems.

3.3.1 Collecting and Pre-Processing Low-Level Metrics

Inside-Out collects general, low-level system metrics from individual machines running the distributed storage service. However, the raw collected data suffers from various problems due to inefficiency of data collectors, system clock skews, incomparable data formats, workload outliers, bursty system anomalies, *etc.* The noisy data can lead to unstable and inaccurate performance models. Inside-Out performs a series of data pre-processing functions to address these issues.

3.3.1.1 Monitoring storage components

We collect low-level system metrics of the underlying operating system to capture resource utilization (*e.g.*, cpu, memory, disk and network usage). The low-level performance metrics are sampled with one-second granularity. Such data can be collected from *libvirt*, Ganglia, instrumented hypervisors [20] and *Ceilometer* in OpenStack. We use *dstat* monitoring tool (with option: -tcly -mg –vm -dr -n –tcp –float) to collect these data.

3.3.1.2 Data smoothing

Building a performance model with data collected at one-second granularity is challenging because system data can exhibit high variance at small time scales, *e.g.*, due to dynamic/bursty workloads and interference among co-located tenants. Furthermore, the storage IO operation needs to pass through a series of software layers between the storage client and the back-end raw physical storage device. The long storage IO path can introduce high variability in resource utilization at smaller time scales. For example, HDFS and Ceph both replicate data blocks across storage nodes distributed in physically disjoint servers, racks or even datacenters. To address the uncertainties due to complex IO path spanning several software layers, we compute the moving average of the collected performance data. We have empirically found that one-minute window for processing the moving average is sufficient to eliminate outliers from the raw data.

3.3.1.3 Timestamp alignment

Proper time synchronization among participating servers is essential to correlate data collected from those servers. We use NTP for time synchronization. The average timestamp of all nodes is taken as the basis for time alignment.

3.3.1.4 Feature transformation for a distributed storage system

As mentioned earlier, elasticity is an important feature of SDS, since it needs to adjust its size based on storage demand. Thus our model must accurately predict end-to-end performance at arbitrary deployment scales. However, the data collected from different scales may have different dimensions. For instance, Ceph with 10 Object Storage Servers (OSDs) generates 10 copies of low-level performance metrics, while Ceph with 5 OSDs generates fewer data points. This makes it hard to train and build a unified model. As mentioned in Section 3.2.1, we use *mean, sum, std, and 5%* statistical variables to capture different types of workload distribution such as *hotspot*, *load imbalance*, and *aggregate performance*.

In summary, Inside-Out collects 32 raw low-level system metrics with one-second granularity. Inside-Out applies proper time alignment and moving average with one-minute windows for

stabilizing performance data. Then it calculates *mean*, *std*, *sum* and *5%* of individual metrics collected from multiple machines. This ensures that our performance model can accept input data for systems with varying scales of deployment, while preserving important characteristics of a distributed storage system. For training and validation purposes, we measure end-to-end performance metrics (IOPS, throughput, and latency) every 5 seconds using COSbench [31], and take average over the one-minute window. Next, we describe how we build end-to-end performance models in order to capture the relationship between low-level system metrics and end-to-end throughput and IOPS.

3.3.2 Exploring Learning Methods

Our goal is to build a model that accurately predicts end-to-end throughput and IOPS by analyzing only the low-level metrics of a distributed storage system. We explore several algorithms, including statistical regression [40, 55], decision tree learning and random forests learning [55, 124]. For statistical regression, we mainly focus on linear regression techniques, which can be extended to support non-linear regression by expanding features that simulate, for example, quadratic terms [75]. We did not find this necessary in our application and exclude the discussion.

Lasso is a least square linear regression technique with L1-norm regularization. The L1 penalty function leads to a sparse solution, which has an effect of restricting the number of selected variables. This property is useful for figuring out important features, especially when the number of variables or features is large. *Ridge* is similar to Lasso but instead uses L2-norm regularization, which has the effect of group selection of variables. This property does not restrict the number of variables selected by the prediction model and therefore, the prediction accuracy might degrade and become inconsistent when the number of input features to the training model is large. *Elastic Net* combines both advantages—it does group selection while enforcing sparsity. Based on our data set, Lasso and Elastic Net have similar prediction performance, and Ridge shows larger variance. The *Decision Tree* (DT) learning uses a top-down approach and recursively partitions data to fit target values. The tree-based model is easy to interpret and scales well to large datasets. *Random Forests* (RF) is an ensemble method that uses multiple decision trees [55]. RF improves a single decision tree in many ways, e.g., accuracy, efficiency, and robustness.

To summarize, linear regression models assume a linear relationship and might oversimplify the storage behavior. Nonetheless, it has the potential to exhibit better generalization for extrapolating performance prediction for the unknown behavior case (the pattern not included in the training dataset). On the other hand, the tree-based learning can achieve good model accuracy (perfectly fits the training data), but it can easily lead to overfitting problems. Its prediction accuracy decreases, for example, under different storage workloads, as shown in

Figure 3.2.

3.3.3 Two-level Training

The fundamental challenge in building an effective prediction model from a large set of features is the overfitting problem. One way to address this problem is to perform manual feature selection. However, this approach is problematic because the right set of features depend on application types, deployment topology, resource constraint, etc.

Instead, we propose a two-level training process that filters out irrelevant features in the first step and then builds models by using the reduced set of features in the second step. To this end, Inside-Out pipelines Ridge and Lasso together, where Ridge filters features in coarse-granularity and then Lasso builds the prediction model. We choose Ridge as the filtering algorithm because it is not a sparse solution and considers all features. We then apply exhaustive grid search to find the optimized score for important features. We use $\alpha\times$ *median(coefficients)* derived from Ridge as the threshold.

For comparison, we consider Decision Tree with Lasso (Auto-DTL) and RandomForest with Lasso (Auto-RFL). Our evaluation shows Inside-Out outperforms consistently across all prediction cases, and boosts prediction accuracy in several scenarios, where the linear regression models fail to generalize the behavior of a distributed storage system. We also experimented by using Lasso and Elastic Net as the filter algorithm but did not find comparable performance with Inside-Out.

Inside-Out uses the following pseudo code to generate an end-to-end performance model. In practice, we set k to 10 for stable prediction results. The data processing part is explained in previous sections. Features are automatically selected using the Ridge algorithm with multiple thresholds. We vary α from $0.1, 0.2, ..., 1.0$. The grid search approach is used to select the best model.

3.4 Evaluation

In this section, we present a comprehensive evaluation of Inside-Out. We demonstrate that Inside-Out can accurately predict end-to-end performance, i.e., throughput and IOPS, using low-level system metrics and is applicable to a wide range of realistic scenarios.

3.4.1 Setup

We choose Ceph [26] as a target distributed storage service for our evaluation and use COSBench [31] to generate various types of storage workloads. COSBench supports several object

Algorithm 2: Inside-Out Model Building

Input: low-level performance metrics from distributed nodes
Output: an end-to-end performance model
Initialisation
1: $thresholds = \{\alpha_1, ..., \alpha_N\}$
2: $m1$ = filtering algorithm $\rightarrow$ *Ridge*
3: $m2$ = model algorithm $\rightarrow$ *Lasso*
4: k = k-fold cross validation
5: $score = 0$
Data preprocessing (refer to Section 3.3.1)
6: alignment of input data
7: calculate moving average across metrics
8: feature transformation for the distributed scenario
Grid Search
9: **for all** $t \in thresholds$ **do**
10: $features$ = execute $m1$ with threshold t
11: $score, m$ =max(crossvalidation(k, $m2$, $features$))
12: **end for**
13: **return** m with maximum *score*

storage protocols, including *librados* for Ceph, and provides a set of knobs to change storage traffic pattern. Table 3.3 lists Ceph and COSBench configurations used in our experiments.

We collected benchmarking data from an OpenStack-based SDS platform. The cluster has 16 machines, and each machine has 16 cores, 24GB memory and 250GB disk space. Each machine has 1Gbps network interface connected to a 10Gbps switch. The dataset is collected from about 5300 benchmark runs. The total dataset is composed of about 15.2 million records, each of which is a vector of 32 low-level performance data. The end-to-end performance data collected from COSBench contains 3 million records. The combined dataset is about 24GB, collected over two weeks.

3.4.2 The Comparison Method

Our goal is to find a function $f(X_t)$ that predicts the end-to-end performance, where X_t is a vector that describes the internal status at time t of a distributed storage service. We say a model is accurate if $f(X_t) = \hat{y}_t \simeq y_t$, where y_t is the ground truth (measured at the client side) and $\hat{y}_t$ is the predicted values. To interpret performance models, we are interested in four indicators: 1) the overall prediction accuracy, 2) the goodness-of-fit, 3) the consistency across diverse scenarios and 4) the consistency across prediction instances.

Table 3.2 Common scenarios that storage behavior can change in a software-define storage environment

	Scenario	Training Dataset	Prediction Dataset	Explanation
Changing Workload	Increasing users	{1, 2}	{4}	The number of client virtual machines running COSBench.
	Complex usage	{1, 2, 4, 8}	{16, 32}	The number of threads for all benchmark clients.
	Complex request	512KB	1-1024KB	The request size (either static or variable) of the workload, configured in COSBench.
	Write intensive	{50, 75, 100}	{25, 0}	The percentage of read operations the workload. The read and write percentages are 100 in total.
	Read intensive	{0, 25, 50}	{75, 100}	
	Medium write intensive	{0, 50, 100}	{25}	
	Medium read intensive	{0, 50, 100}	{75}	
Reconfiguration	Reconfigure Ceph	{1}	{2}	The number of Ceph monitor daemons.
	Scale-up instances	m1.small	m1.medium	The instance type of the virtual machines running Ceph is upgraded to a powerful one. A m1.small instance has one core and 2GB memory and m1.medium has two cores and 4GB memory. Note that in this setting, the configuration of disk I/O remains the same.
	Medium network SLO	unrestricted	500 Mbps	The network bandwidth of virtual machines is limited at 500 Mbps. We use the Linux tool *tc* for network throttling
	Low network SLO	unrestricted	250 Mbps	Network bandwidth is limited at 250 Mbps.
Elasticity	Scale out to n	{4, 6, 8, 10}	{20, 30, 40}	The total number of Ceph OSDs. Note that each OSD is running in a virtual machine and different OSDs can run on the same physical servers (10 servers in total).
	Shrink in to n	{20, 30, 40}	{4, 6, 8, 10}	Similar to the above, but the cluster size is decreased.

First, we use mean absolute percentage error (MAPE) to compute prediction accuracy as

$$max(1 - \frac{\sum_{t=1}^{n} |\frac{y_t - \hat{y_t}}{y_t}|}{n}, 0) \tag{3.1}$$

where n is the length of the observation period. We restrict the scope of prediction accuracy between 0 to 1 because the prediction accuracy can be negative (e.g. when y_t is small).

Second, we use the coefficient of determination R^2 to interpret *Goodness-of-Fit*, which is less than or equal to one [89]. Third, we examine whether a performance model can present consistent prediction in various SDS scenarios. Last, we further analyze the probability density function of prediction decisions for different categories of prediction scenarios.

We consider prediction of throughput and IOPS for both read and write operations, and use the following terms TP_r, TP_w, OP_r and OP_w for read throughput, write throughput, read IOPS and write IOPS, respectively.

Table 3.3 Ceph and COSBench settings for data collection.

Parameters	Values
Ceph version	9.2 (Infernalis)
# of physical nodes	16
Storage back end	Logic Volume (iSCSI)
# of storage nodes	{4, 6, 8, 10, 20, 30, 40}
# of drivers	{1, 2, 4}
# of workers	{1, 2, 4, 8}
Request size	{512KB, 1-1024KB}
Duration	180 sec
# of containers	{64}
# of objects	{1024}
read/write ratio	{100/0, 75/25, 50/50, 25/75, 0/100}

3.4.3 Baseline: Prediction Performance on Static Deployment

We evaluate prediction accuracy of Inside-Out under a variety of scenarios with different storage workloads and configurations listed in Table 3.2. In this subsection, we focus on a static deployment scenario with one storage tenant running on a distributed Ceph storage service that does not expand or shrink in terms of number of VMs used for running Ceph. Later, we evaluate more challenging scenarios in which the Ceph cluster expands or shrinks based on user demand, and storage traffic of multiple tenants interfere with each other.

3.4.3.1 Can Inside-Out handle diverse workloads?

An SDS application needs to handle various request volumes, object/file sizes and different ratios of read/write workloads. First we examine whether Inside-Out can achieve accurate and consistent predictions when workload changes.

Changing user behavior

We increase the number of concurrent clients to stress the Ceph cluster. The *increasing users* scenario changes the number of COSBench clients and the *complex usage* scenario increases the worker threads of each client. As shown in Figure 3.3, all prediction models perform well. The linear regression technique performs slightly better than the tree-based learning. The linearly increasing load is well captured by linear models because of proportional change in low-level metrics. When we switch to the *complex request* scenario, the variable request size slightly changes the behavior of Ceph, affecting prefetching and caching. We observe that the linear regression methods (Lasso, Ridge and Elastic Net) show drops in accuracy, e.g. 20% in the OP_r

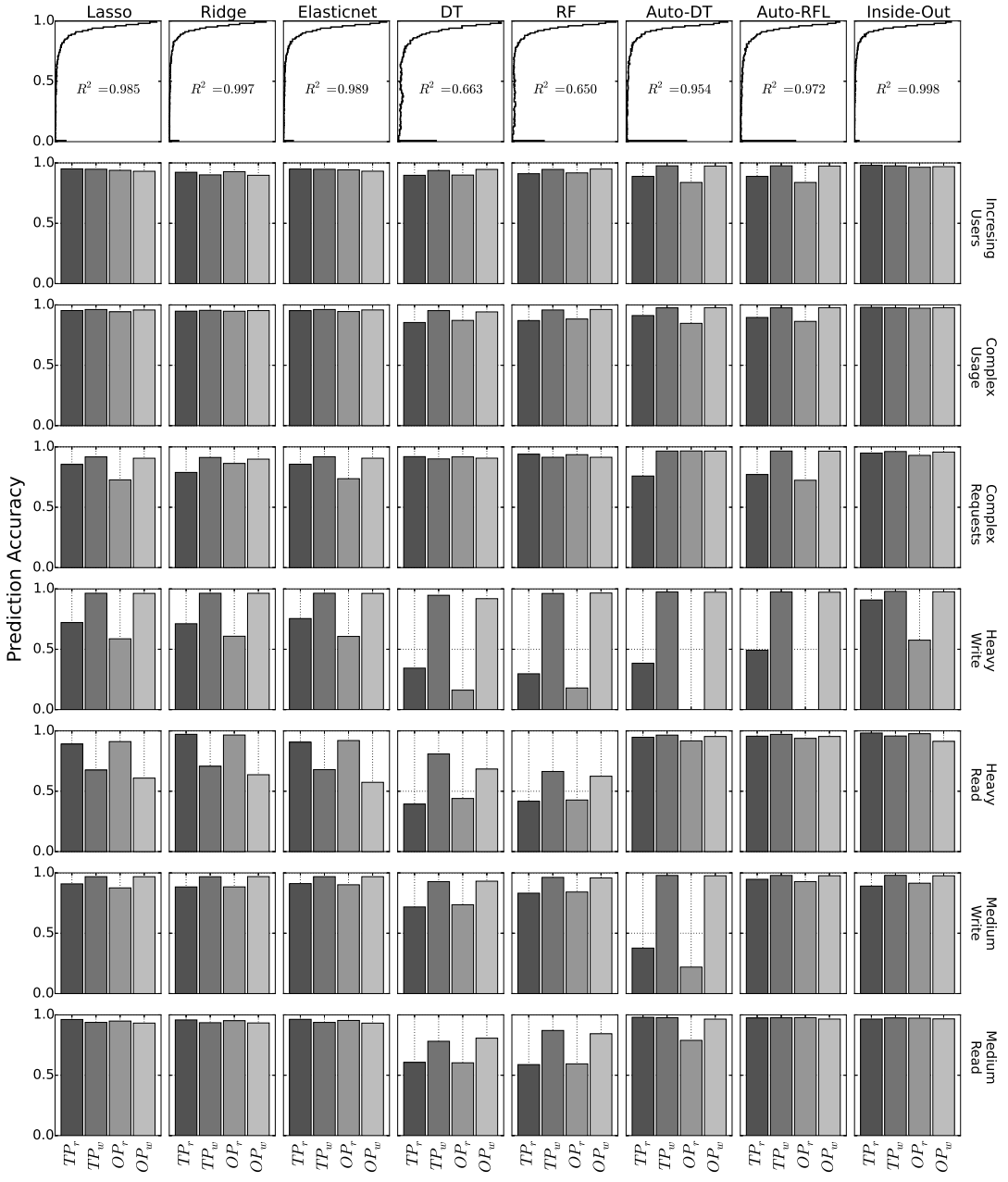


Figure 3.3 Analysis of performance models with diverse workloads. Each bar is the average prediction accuracy. The top row is the probability density function of prediction accuracy for each performance model.

case; however, Inside-Out maintains good accuracy. The tree-based learning shows comparable predictions (5-10% lower) with Inside-Out in these settings.

Varying I/O pattern

Next, we consider workloads with different ratios of read to write operations. Figure 3.3 shows that varying workload poses a big challenge to performance models. The linear regression methods (Lasso, Ridge and Elastic Net) present better prediction accuracy than tree-based models (DT, RL). In addition, we observe that several models make poor predictions of TP_r and TP_r. The reason is that read behavior is largely affected by *cache*, and large read variance contributes to low prediction accuracy. Inside-Out performs consistently well, whereas the three linear regression techniques show accuracy drops. One exception is the OP_r prediction in the *write-intensive* scenario even though TP_r prediction is accurate. As we will show later in Section 3.4.5, *over- and under-predictions* cause such behavior. The self-learning property of Inside-Out improves its prediction accuracy as it keeps learning the new storage behavior.

Summary

The linear regression models achieve high prediction accuracy, great goodness-of-fit (> 0.98) and consistency in prediction for many instances (see the distribution of prediction accuracy in Figure 3.3), but they are not consistent across all prediction scenarios. Inside-Out achieves good prediction accuracy across all cases consistently because the two-level approach filters out many irrelevant features in the first step, thereby presenting a smaller relevant feature space to the second step. The tree-based learning methods (DT and RF) do not show consistent prediction across all scenarios. Auto-DT and Auto-RFL, which use DT and RF as the filter algorithms, are not as consistent as Inside-Out.

3.4.3.2 Can Inside-Out handle different system configurations?

We study whether low-level metrics can capture the storage behavior when it is reconfigured by tenants. The results are reported in Figure 3.4.

Reconfiguring Ceph. The first change is to add one extra Ceph monitor daemon. Ridge and Elastic Net fail to generate consistent predictions, but Lasso is able to achieve around 80% to 90% prediction accuracy. DT, RF and Inside-Out have very close prediction accuracies, but Auto-DRL and Auto-RFL perform slightly worse in predicting TP_r and OP_r.

Scale-up instances. Increasing CPU and memory allocation to Ceph VM instances improves Ceph's ability to handle more requests. In this test, we change the instance type from m1.small (1 vCPU, 2GB memory) to m1.medium (2 vCPUs, 4GB memory). The linear models are unable

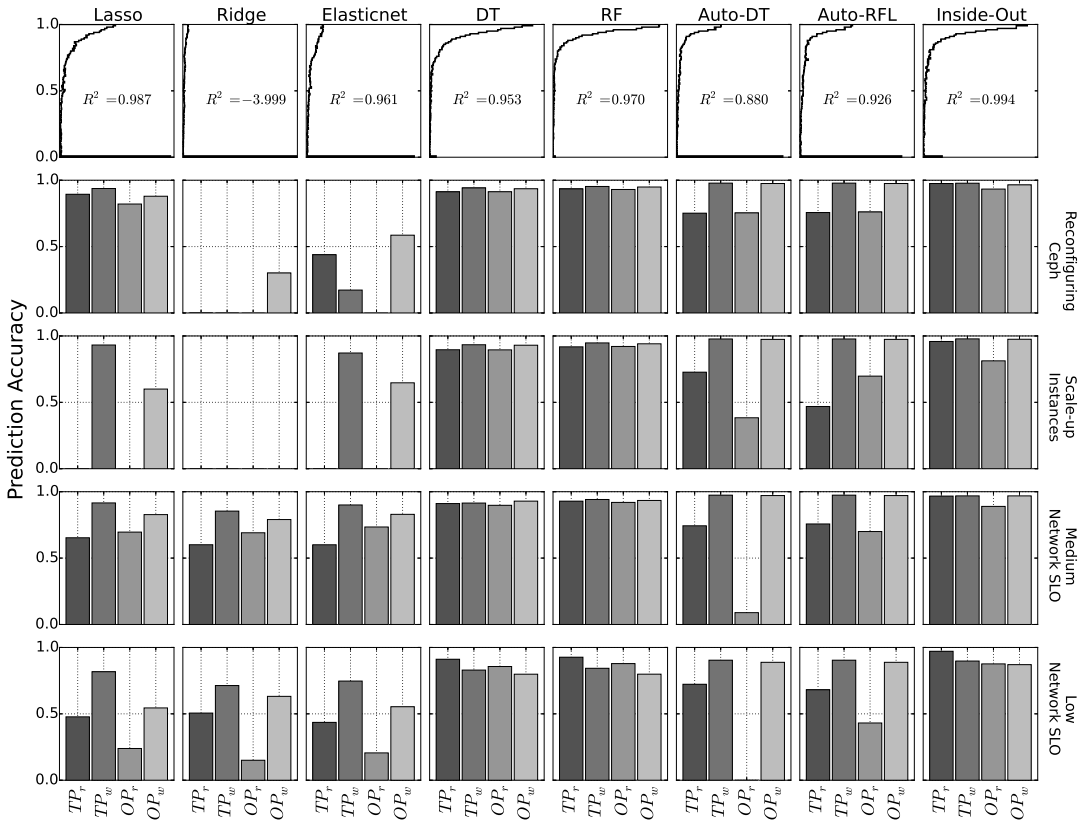


Figure 3.4 Comparison of performance models when the storage service is reconfigured: Ceph, VMs and network SLOs

to predict TP_r and OP_r, but Inside-Out's two-level learning performs well by avoiding the overfitting problem.

Network SLOs. Here we consider the case where the amount of network bandwidth allocated to Ceph VMs is limited. We use Linux network throttling tool *tc* to limit network bandwidth at 500 Mbps and 250 Mbps for medium and low bandwidth SLOs, respectively. We observe that linear models without the two-level method do not show comparable prediction accuracy across both throughput and IOPS predictions. The tree-based learning models, on the other hand, achieve 80% to 90% accuracy, comparable to Inside-Out.

Summary. Tree-based learning (DT, RF) models demonstrate promising prediction in terms of prediction accuracy and consistency. Lasso, Ridge and Elastic Net show inconsistent behavior in the above four scenarios. Inside-Out, on the other hand, provides consistent predictions and improves Lasso, from 23.9% to 87.6% in the extreme case.

3.4.4 Prediction Performance in a Multi-tenant Cloud

This section examines the modeling performance of Inside-Out. We first evaluate whether Inside-Out is able to extrapolate performance of a larger Ceph cluster. Next, we evaluate how Inside-Out performs when systems are subject to performance interference.

3.4.4.1 Elastic Storage (On-demand Scaling)

A storage service needs to grow or shrink its capacity on demand. We evaluate Inside-Out's ability to capture the storage behavior at different system scales. As shown in Figure 3.5, we use training data collected from 4, 6, 8, and 10 nodes, and then predict the performance of 20, 30, and 40 nodes. We also evaluate prediction accuracy in the *shrink-in* scenario. For both read and write throughput predictions, the linear models exhibit high variance. In the OP_r and OP_w cases, the prediction results are not even comparable to the other methods. Inside-Out, on the other hand, helps mitigate this issue, and achieves more than 90% accuracy. With increasing sizes of the storage, the prediction accuracy decreases because the prediction target becomes increasingly different from the training data. Running a benchmark test against a very large system is time-consuming. Here we demonstrate that Inside-Out can predict performance for systems that are four times larger than the system for which training data was collected.

3.4.4.2 Multi-Tenancy

Next we evaluate Inside-Out's ability to adapt to performance interference among storage tenants. We consider two cases for this evaluation. Each tenant runs a Ceph cluster with 10 OSDs separately, but tenants share the same 10 physical machines. In the first case, we restrict the bandwidth of only the first tenant at 250Mbps. In the second case, we run two concurrent

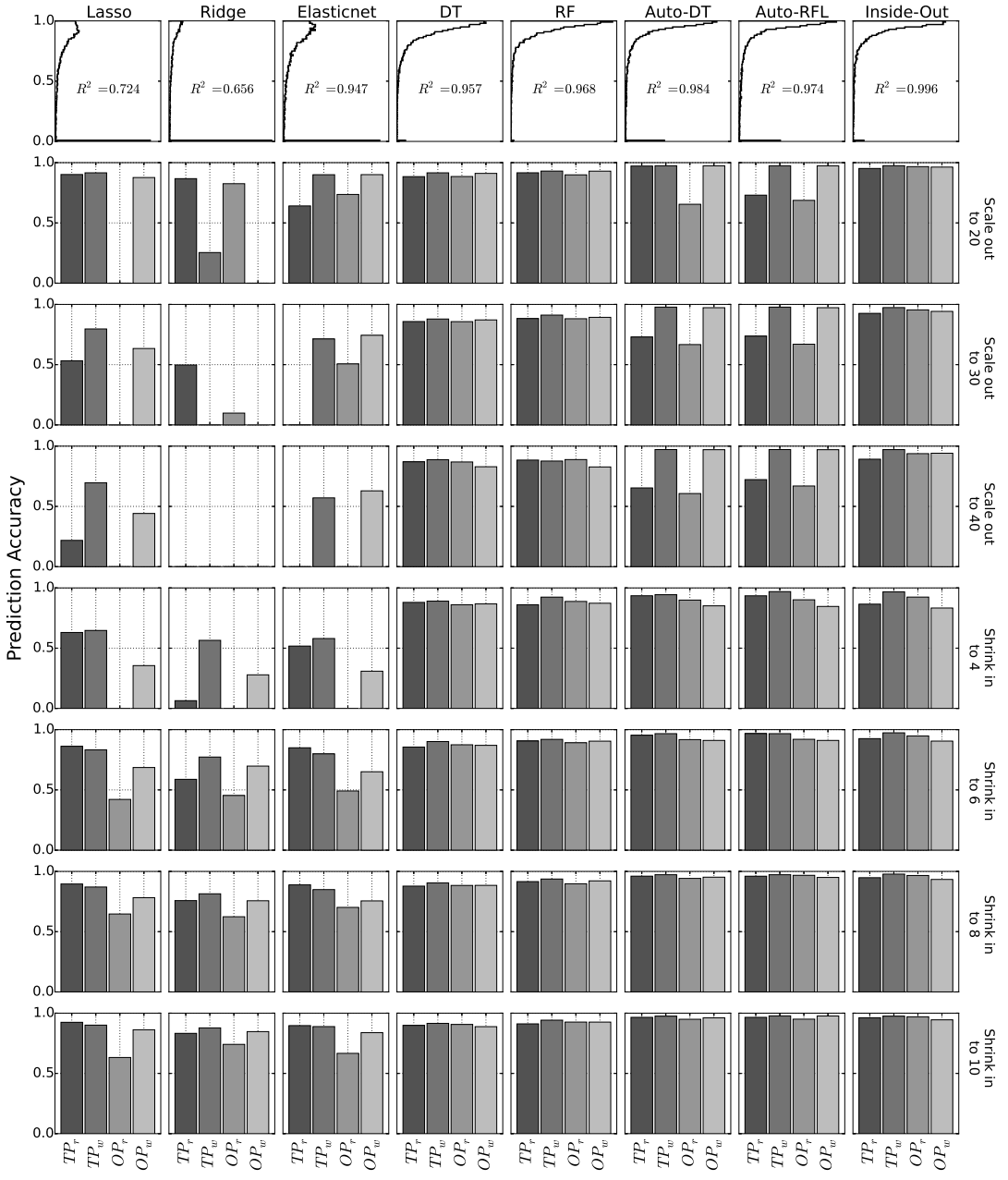


Figure 3.5 Comparison of model performance in the on-demand scaling scenario. In the scale-out scenario, a performance model trained with 10 Ceph nodes is used to predict the performance of Ceph cluster with 20, 30 and 40 nodes.

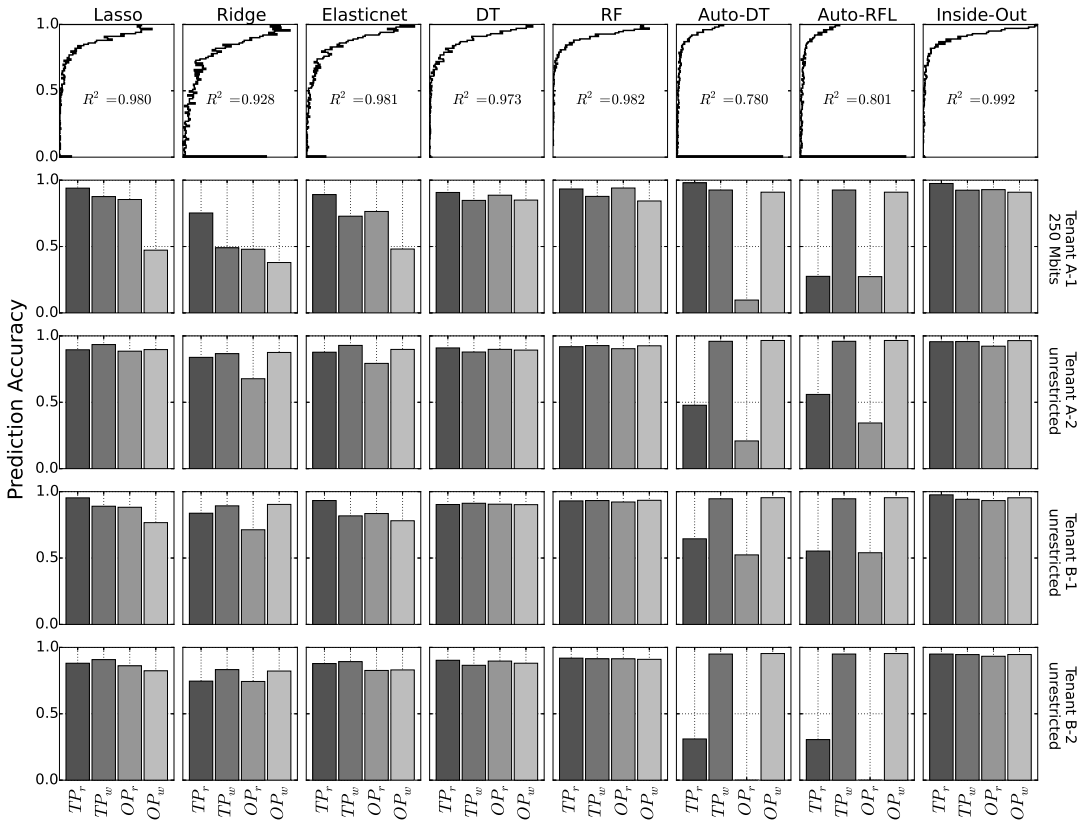


Figure 3.6 Prediction accuracy in a multi-tenancy scenario. Tenant A-1 is co-located with Tenant B-2. Tenant A-1 is throttled at 250Mbps. Tenant B-1 and B-2 are co-located without any traffic throttling.

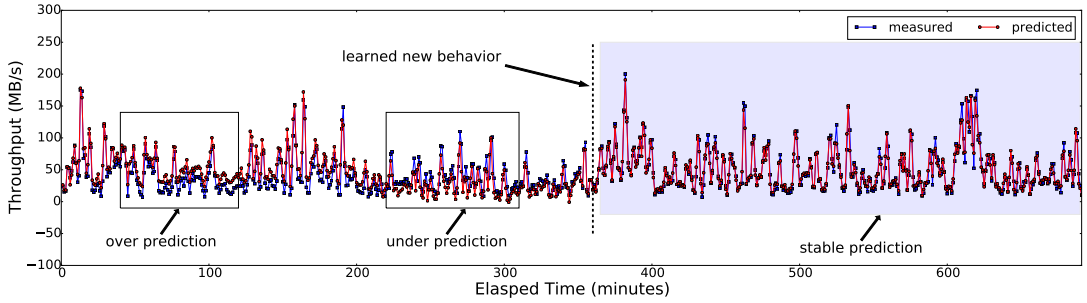


Figure 3.7 Application of Inside-Out to real time prediction of read throughput on a 10-node Ceph cluster. Inside-Out starts from a simple prediction model trained by our collected benchmarking data. Inside-Out keeps learning the storage behavior while improving prediction accuracy over time.

Ceph clusters but without network throttling. Figure 3.6 shows that most prediction models are able to achieve more than 80% accuracy. The linear models like Ridge and Elasticnet yield lower prediction accuracies in some cases; however, Inside-Out performs well consistently. Performance interference is challenging for a performance model designed for an isolated environment. This evaluation demonstrates that the low-level performance metrics are good proxies for measuring the end-to-end storage performance, even in a shared SDS environment.

3.4.5 Online Self-Learning

Next, we create several synthetic workloads with mixed read/write ratios. This synthetic workload spans 12 hours with 720 stages. Each stage is 60-second long on average, with a standard deviation of 20 seconds. We run four COSBench virtual machines for benchmarking and up to eight threads per COSBench client, with 10 Ceph OSDs and one monitor daemon. We use Inside-Out to build an initial performance model with the training dataset described in Section 3.4.1. Figure 3.7 shows the prediction result for read throughput. We can observe that the generated model can capture the overall trend, but suffers from over and under predictions. This is because our training dataset is generated from a relatively clean environment, *i.e.*, the OS memory is flushed before any benchmarking process. However, in the online prediction setting, cache is continuously consumed by non-stop client requests, which causes the real time storage behavior to be different from the training dataset. With continuous monitoring of the performance of the storage service, we use Inside-Out to generate a new performance model at the sixth hour. Figure 3.7 shows that Inside-Out learns the new storage behavior and therefore, the over- and under-prediction issues are greatly mitigated. By continuously learning the storage behavior, SDS can accurately capture performance changes and therefore is able to provide reliable storage service.

3.4.6 Discussion

We have shown that low-level performance metrics are useful to predict end-to-end throughput and IOPS. Our evaluation has shown that low-level performance metrics are good indicators of end-to-end throughput and IOPS. Most existing performance models exhibit an inconsistent prediction behavior in the presence of diverse storage scenarios, such as changing workload, storage reconfigurations, growing/shrinking storage, and multi-tenancy environments. Our proposed two-level learning method can greatly improve prediction accuracy and yield consistent behavior. Machine learning provides powerful tools, but they need to be used intelligently to achieve the best prediction accuracy. Figure 3.8 shows the kernel density function of prediction accuracy across all prediction scenarios. Inside-Out is a clear winner in terms of accuracy and consistency. More importantly, Inside-Out is able to learn new storage behavior, thereby enabling

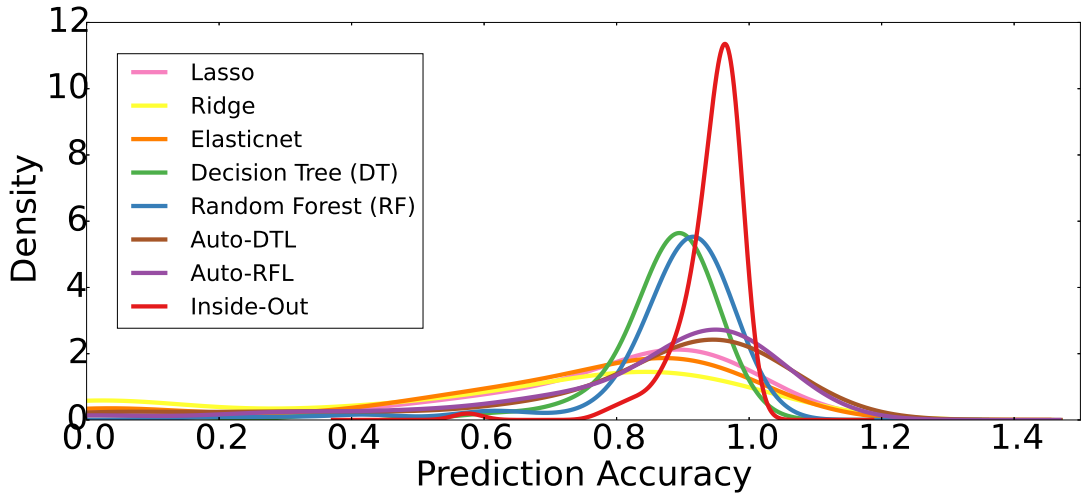


Figure 3.8 Kernel density function of prediction accuracy from Figure 3.3 to Figure 3.6. Each colored line represents the density function of a modeling approach. Inside-Out is more consistent and accurate across almost every prediction case.

the performance model to adapt to the complex SDS environment.

3.5 Conclusion

Ensuring end-to-end performance in software-defined storage (SDS) requires accurate performance models. This chapter presents Inside-Out, a tool that provides reliable and consistent prediction of end-to-end performance of distributed storage systems using low-level system metrics. Our evaluation indicates that Inside-Out is able to generate accurate prediction models even when the storage environment differs significantly from the training phase. Inside-Out is generic in nature because it does not use application or protocol specific data for building performance models. Although we used Ceph as an example distributed storage service to evaluate Inside-Out, we believe it should be applicable to other storage systems as well with minor modifications.

Chapter 4

Cloud Architecture Tuning

This chapter formulates the problem of cloud architecture tuning (CAT) and describes the challenges in CAT with large-scale evaluation conducted on AWS.

4.1 Introduction

Cloud computing is a cost-effective alternative to on-premise computing. Choosing the right VM type for a workload is essential to provide quality service while being cost effective [45, 134]. In this chapter, we describe the *cloud architecture tuning (CAT)* problem [3, 62, 123, 133]. Given a workload and a service objective, we focus on delivering the best architecture configurations, such as virtual machine (VM) types and the number of VMs. CAT is similar to hyper-parameter tuning in many aspects [37, 50, 56, 72, 108, 140]. However, the search space of CAT can be more irregular, which makes existing approach to CAT more fragile [62]. This is because the same workload can perform very differently on two similar configurations. For example, memory bottleneck can exist in one configuration but not another.

A good solution for CAT should have low search cost and find (near) optimal configurations. Additionally, the solution should be reliable, scalable, and work for wide range of applications and workloads. *Ernest* is an effective method to extrapolate workload performance on different configurations, but it is not scalable because it requires separate prediction models for distinct workloads and VM types [123]. *CherryPick* adopts Bayesian Optimization to support any kinds of workloads [3]. However, it relies on an appropriate kernel function to model the search space, which makes it fragile [62]. *PARIS* uses extensive training data to build prediction model [133]. However, it may suffer from a high variance of prediction error. Thus, no prior work fulfills all the requirements of a good solution.

4.2 Open Performance Data

To evaluate a CAT method, we need large-scale performance dataset—evaluating diverse workloads on different architectural choices. We conducted a large-scale evaluation using different workloads and software systems on Amazon Web Services (AWS) [4]. We choose Apache Hadoop (version 2.7) and Apache Spark (version 1.5 and 2.1) as our software system [8, 9]. Our evaluation includes data processing, OLAP queries, and machine learning, which are popular applications on Hadoop and Spark. We choose 18 VMs and run the 30 applications on them. Table 4.1 lists all the software systems and applications.

We also vary the input size or input parameters to the applications for creating diverse workloads [34]. When workloads are different, the optimal VM type (even for the same application) might change as well. By running workloads with different data sizes, we can observe whether a particular VM can sustain increasing resource requirements (of a workload). Our motivation (for the large-scale study) was to diversify the workloads such that we can extensively benchmark VMs. In this study, each workload is tested with three different inputs sizes. Some tests failed because smaller VM instances run out of memory. Those are excluded in our data set. In total, we measure the performance and collect the low-level information of 107 workloads on 18 different VM types. We also collect 18 workloads on 69 multi-node configurations. Table 4.2 summarizes the dataset collection.

Performance optimization requires continuous efforts to keep up with the rapid pace of cloud computing. In our experience, performance data is hard to find. A lack of performance data discourages the advances in system performance research. We believe that we will see advances in performance optimization by sharing performance data. Our large-scale performance dataset is available at https://github.com/oxhead/scout.

4.3 Challenges

Cloud providers recommend the choice of VM types [4, 51]. However, it is too coarse grain and does not apply to many workloads because resource requirement is often opaque [133]. Finding the best VM is often very challenging. The growing complexity comes from five factors.

The increasing number of VM types

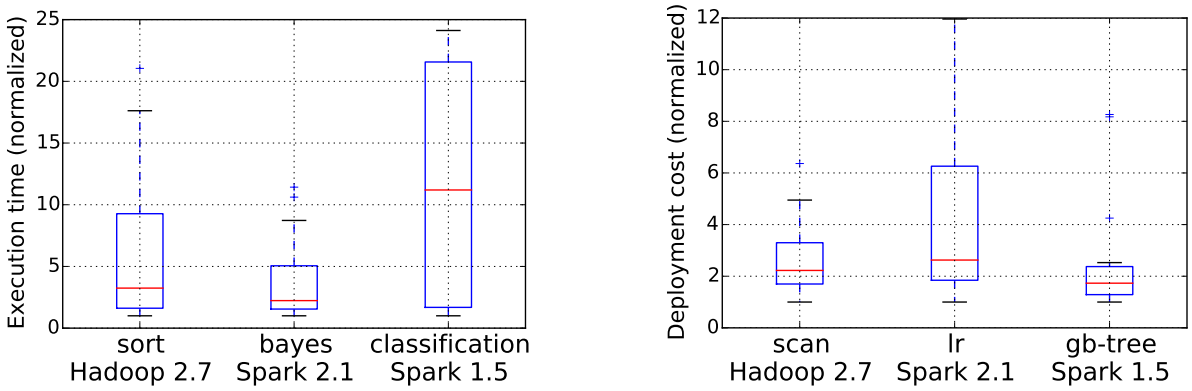
To accommodate the growing number of workloads, cloud service providers frequently adds new VM types to their already large VM portfolio. AWS, for instance, has a significant upgrade on its service two times a month on average [12]. As of December 2017, AWS provides 71 active VM types. Such a trend would make a brute-force search for the best VM type expensive. Also,

Table 4.1 The evaluated applications. In total, there are 30 applications and 107 workloads measured on Hadoop 2.7, Spark 1.5 and Spark 2.1.

Application	Description
Micro Benchmark	
sort	Sorts text input data, generated by RandomTextWriter in Hadoop.
terasort	A standard Hadoop benchmark. Data is generated from TeraGen.
pagerank	The PageRank algorithm. Hyperlinks follow the Zipfian distribution.
wordcount	Counts the frequency of words that generated by RandomTextWriter. This is a typical MapReduce job.
OLAP	
aggregation	A Hive query performing aggregation.
join	Implement the join operation in Hive
scan	Implement the scan operation in Hive
Statistics Function	
chi-feature	Chi-square Feature Selection.
chi-gof	Chi-Square Goodness of Fit Test.
chi-mat	Chi-square Tests for identity matrix.
spearman	Compute Spearman's Correlation of two RDDs.
statistics	Generate column-wise summary statistics.
pearson	Compute the Pearson's correlation of two series of data.
svd	Singular Value Decomposition, a fundamental matrix operation for finding approximate solutions.
pca	Principal Component Analysis for dimension reduction.
word2vec	Generate distributed vector presentation of words according to distance.
Machine Learning	
classification	Implement the generalized linear classification model.
regression	Generalized Linear Regression Model.
als	The Alternating Least Squares algorithm, implemented in spark.mllib. It is a collaborative filtering algorithm used for product recommendation.
bayes	Implements the Naive Bayes algorithm for the multiclass classification problem. Input documents are generated from /usr/share/dict/linux.words.ords.
lr	A popular algorithm for the classification problem.
mm	Matrix multiplication with configurable row, column and block sizes.
d-tree	A greedy algorithm for classification and regression problems.
gb-tree	Gradient Boosted Tree, an ensemble learning method for classification and regression problems.
df	The Random Forest algorithm for classification and regression problems.
fp-growth	The FP-growth algorithm to mine frequent pattern in large-scale dataset.
gmm	Gaussian Mixture Model is a clustering algorithm that uses k Gaussian distributions to find the k clusters.
kmeans	K-means is a common clustering algorithm that finds k cluster centers.
lda	Latent Dirichlet allocation is a clustering algorithm that infers topics from a collection of text documents.
pic	Power iteration clustering is a scalable algorithm for clustering.

Table 4.2 Dataset description. The benchmark programs are taken from *HiBench* [58] and *spark-perf* [114].

Applications	30	**Evaluation**	$> 12,000$ runs
Workloads	125	**Duration**	$> 1,300$ hours
Single-node	18	**Raw data size**	2.5 GB
Multi-node	68	**Data points**	6 million
Frameworks	Hadoop Spark	**Low-Level Metrics**	72 (raw) 504 (populated)



(a) Execution time

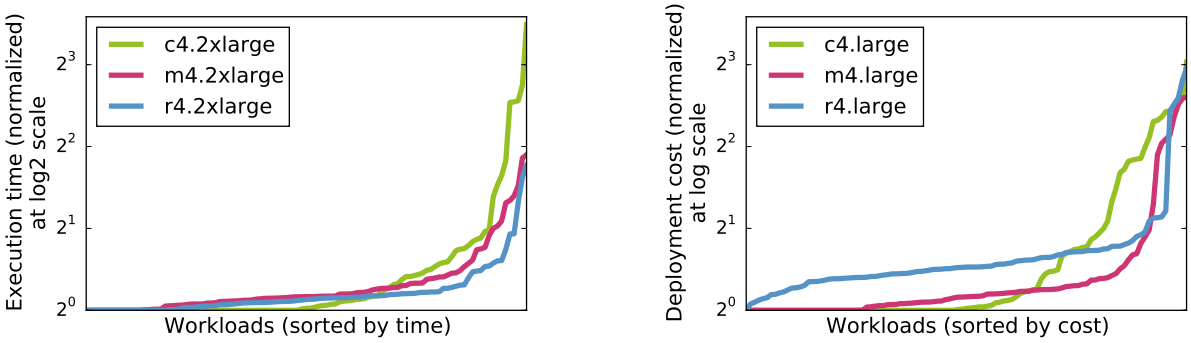
(b) Deployment cost

Figure 4.1 The execution time and deployment cost of workloads running on 18 virtual machines (different types). The execution time of classification-Spark 1.5 in the worst case is 20 times slower that the best VM type. Similarly the deployment cost of running Linear Regression on the worst VM type is 10 times more expensive than the best VM type.

it is difficult to model the performance of a workload for distinct VM types [133].

Official recommendation is insufficient

AWS recommends VM types for workloads. Even though such recommendations are beneficial for users, these recommendations should be carefully examined. For example, users are encouraged to choose compute-optimized VMs for CPU-intensive workloads and memory-optimized VMs for workloads requiring large memory. However, characterizing workloads is still considered difficult and requires expertise, which is often very expensive and sometimes unavailable. This problem is exacerbated by workloads, which regularly exercise resource components in a non-uniform manner [94]. Furthermore, it is difficult to understand the resource requirement of a workload for achieving a specific performance objective [133].



(a) Execution time on the most expensive VM types.

(b) Deployment cost on the least expensive VM types.

Figure 4.2 Performance distribution over different workloads. The performance is normalized to the optimal performance measured in the 18 virtual machines. The *x-axis* represents workloads, sorted by their normalized performance. Both choosing the most expensive and the cheapest VM types are not desirable.

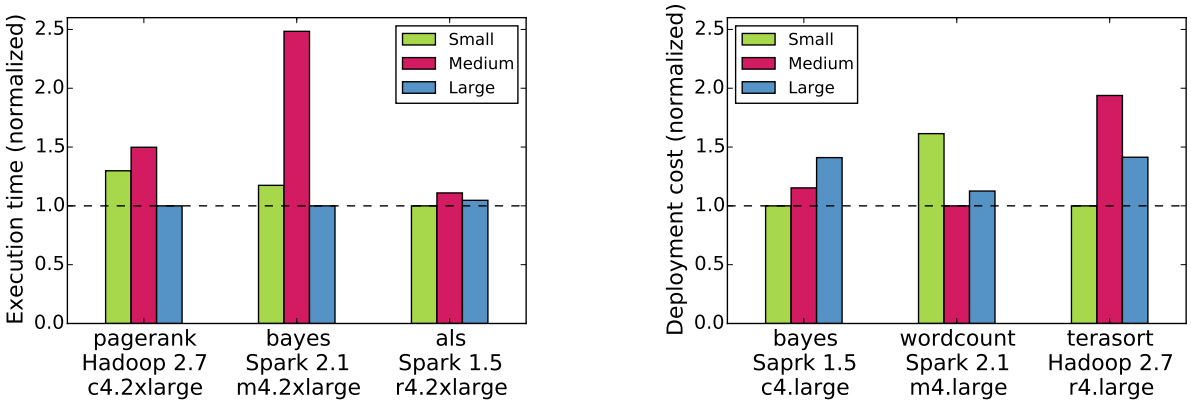
No VM rules all

Our empirical data, as shown in Figure 4.1, demonstrates that a bad choice can increase the execution time (of a workload) up to 20 times and can be ten times more costly than the optimal one. Prior work reports similar results [3, 133]. Careless selection can often end up with high deployment cost and longer (sub-optimal) execution time.

Even though users are willing to pay a higher cost in exchange for performance, choosing the most expensive VM type may not always result in optimal performance. Figure 4.2a shows the distribution of the execution time when running on the most expensive VM types (namely c4.2xlarge, m4.2xlarge and r4.2xlarge). For instance, if we look at the distribution of execution times for c4.2xlarge, we observe that c4.2xlarge is the best VM for 50% of the cases. This means for the other 50% of the workloads; the most expensive VM type does not guarantee the lowest execution time. We observe similar behavior in Figure 4.2b, where the least expensive VM, c4.large, does not ensure the lowest deployment cost.

The same application with different input sizes favors different VM types

Machine learning workloads are readily available such as the machine learning library in Apache Spark and Python [106]. It is valid to assume that similar workloads would prefer the same VM type provided the user can accurately identify similar workloads. Consequently, users can always reuse the best VM type for their workloads without testing further. However, we found that this might not always be the case. A workload with different input sizes or parameters performs very



(a) Execution time.

(b) Deployment cost.

Figure 4.3 Running application with different input sizes result in very different performance. The best performing VM types for an application can change when the input size or parameters are changed.

differently on different VMs [123]. Figure 4.3 illustrates how the performance of an application varies with different input sizes. For example, in Figure 4.3b *c4.large* is the most cost-effective VM type for running the *bayes* application with the *small* input size. However, the deployment cost increases by 40% (is no longer the optimal VM) when the input size is *large*. A possible explanation is that a larger input size creates a resource bottleneck on a smaller VM. Hence, users need to be more careful at selecting the best VM type even for the same applications.

Cost creates a level playing field

Finding a cost-effective VM type can be harder because a slower VM can be competitive in deployment cost. In Figure 4.2a, *c4.2xlarge* is the fastest VM type for over 50% of the workloads (optimal execution time is 1.0). However in Figure 4.2b, when considering deployment cost, we observe that *c4.large* is likely to be a better choice, since it is optimal VM in over 50% of the workloads.

Figure 4.4 presents the normalized execution time and deployment cost of a workload (*regression* on *Spark 1.5*). The figure demonstrates how execution time can be very different while deployment cost is similar across all VM types. For example, *m4.large* and *c4.xlarge* are comparable to *c4.2xlarge* in terms of deployment cost. When the difference between execution times of a workload in different VM types is large, choosing the best VM is easier because there is a clear winner. Incorporating cost compresses the difference. Therefore, searching for the most cost-effective VM type becomes more difficult because several inferior choices (in terms of execution time) are now competitive (in deployment cost). In Section 5.4.2, we show why

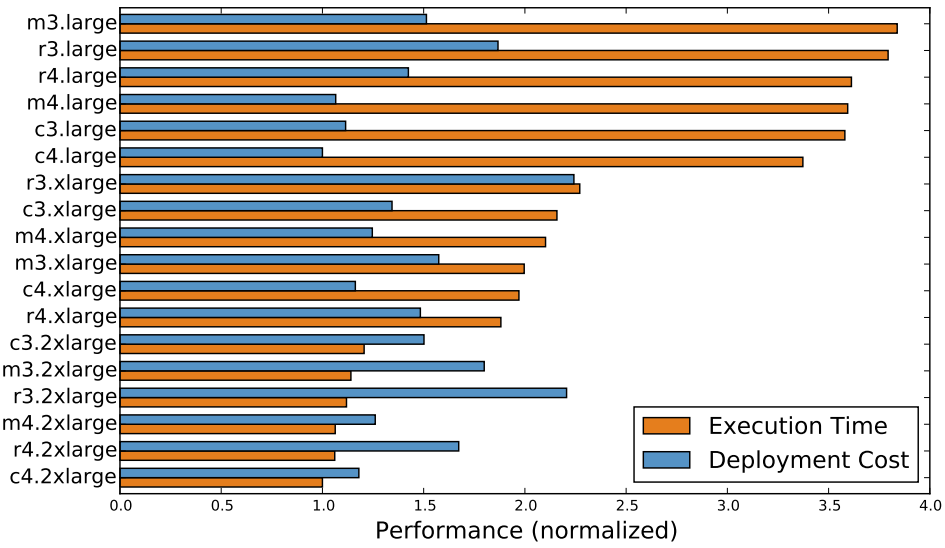


Figure 4.4 The performance of running the *regression* workload on instances with different VM types. Introducing cost creates a *level playing filed*, in which several inferior VM types in execution time are now competitive in deployment cost. This observation implies that searching for the most cost-effective configuration is harder than searching for the fastest configuration.

finding cost-effective VM type is harder than execution time.

4.4 State-of-the-Art Approaches

The CAT problem can be cast into a *learning problem*—which uses elaborate offline evaluation to generate a machine learning model that predicts the performance of workloads [123, 133] and an *optimization problem*—which successively evaluates configurations looking for one that is near optimal [3, 62].

Prediction, as proposed in PARIS [133], is not reliable because of high variance in prediction results. Prediction accuracy heavily relies on feature selection, model selection, and parameter tuning and the quality of data. Sequential Model-Based Optimization (SMBO) [37] is a search-based optimization method, which does not require an accurate model but can have a high evaluation cost (measured in terms of configurations evaluated). Bayesian Optimization (used in *CherryPick*) falls into the SMBO class of algorithms.

The state of the art techniques such as CherryPick [3] and PARIS [133] suffer from three major issues.

- **Model accuracy.** Prediction based approaches like PARIS build a model using mea-

surements. The objective of such an approach is to use an accurate model to predict, for example, execution time or running cost of workloads. This method has two major weaknesses (i) building an accurate model requires more data—which in our setting is hard to come by, and (ii) the performance of the cloud environment is susceptible to performance variability—the data collected after running the workload might not reflect the true performance [118]. As shown in PARIS, the performance of batch-processing jobs is less predictable. The inaccurate estimation of the execution time can be attributed to the non-linear relationship between resource and performance [3].

- **Cold-start** Any SMBO method requires initial measurements to seed the search process. The initial measurements are very crucial since it determines the effectiveness of the search. A poor seeding strategy can lead to wasted effort and can yield a sub-optimal cloud configuration. The effect of cold-start is more pronounced when the initial measurement cost cannot be amortized, *e.g.*,the search space is not large enough.

- **Fragility** An SMBO method is fragile as it is overly sensitive to input parameters. The success of *CherryPick* on a given workload depends on the initial points used to seed the search and the choice of the kernel function used in the performance model (Gaussian Process Model). Previous work shows that *CherryPick* sometimes fails to find near-optimal configurations and incurs longer (than expected) search path [62].

Hyper-parameter tuning shares similarity with CAT. For example, system and software performance is highly affected by configurations. StarFish is an auto-tuning system for Hadoop applications [56]. *BestConfig* proposes the Divide and Diverge Sampling strategy along with the Recursive Bound and Search method for turning software parameters [140]. A similar framework is also proposed to automate tuning system performance of stream-processing systems [16]. BOAT is a structured Bayesian Optimization-based framework for automatically tuning system performance [34] which leverages contextual information. Sampling techniques focus on reducing sampling cost while building accurate models to optimize software systems [86, 91]. Parameter tuning is also an critical in machine learning [37, 50, 72, 108].

The above methods focus on performance tuning for the same workload (or application) on the same type of architecture. It is still not clear how to leverage their approaches to support different architectural configurations in cloud computing.

4.5 Problem Formalization

Cloud architecture tuning (CAT) finds the best cloud configuration for a workload ($w \in W$)—an application and its input. *IaaS* provides a set of computation, storage, and network resources.

For example, users have to determine the type and the number of VMs to run a workload. The search space (S) is a valid set of architectural configurations for running the given workload. The size of the search space is $|S|$ configurations (the search space is the same for all workloads). For a given workload w, each configuration $s \in S$ has a corresponding cost measure $y = \phi(w, s)$. The objective function, ϕ, is user-specific. In practice, cost measures can include execution time, query throughput, running cost, etc.

A cloud service provider presents its user with several choices of VM types (VM). Let VM_i indicate the i^{th} VM type in the list of VMs. In general, each VM type has distinct characteristics (such as memory size and core counts). When a workload ($w \in W$) is run on a VM (VM_i), the low-level metrics ($l_{i,w} \in L$) can be collected from the VM. Each VM type (VM) has a corresponding performance measure $y \in Y$ (*e.g.*,time or cost). We denote the performance measure associated with a given VM type and a workload by $y_{i,w} = f(VM_{i,w})$. In this setting, $VM_{i,w}$ and $y_{i,w}$ are the independent and dependent variables, respectively.

An effective CAT method must find (near) optimal configurations and exhibit low *search cost*.

Search performance

Let y^* be the optimum (minimum) cost for a workload, *i.e.*, $y^* = \min_{\forall s \in S} \phi(w, s)$. Let $\hat{y}$ be the best workload performance that a CAT optimizer finds. The search performance of the CAT optimizer is $\hat{y}/y^*$ (lower the better). A brute force approach finds the optimal configuration for a given workload. Existing methods such as *CherryPick* and *Arrow* achieve near-optimal search performance (*e.g.*,< 1.1).

Search cost

A CAT optimizer must evaluate a workload on several architectural configurations to determine the best choice. Such evaluation is generally expensive. *CherryPick* needs to evaluate at least three configurations when running Bayesian Optimization and *PARIS* generates the fingerprint using two evaluations. Let $E \subset S$ represent the search space that a CAT optimizer evaluated. We define search cost as $|E|$—the number of evaluations needed to select a configuration. This measure is intuitive and effective to compare different CAT methods. Other possible measures are evaluation cost (dollars) and evaluation duration (time).

Our goal is to design a search method to:

1. *Minimize* the performance difference between the *best* VM (VM^*) (found by search) and the optimal VM (VM^{opt}). We find VM^* both in terms of *execution time* and *deployment cost*;

2. *Minimize* search cost—the number of measurements required to find the (near) optimal configuration.

We also look at other metrics for comparing CAT methods. First, it is important to deliver *reliable* search performance and search cost across different workloads. Some optimizers may encounter the fragility issue because the search space is hard to model [62]. Second, a CAT method must be *scalable*. *Ernest*, for example, must build a prediction model for each VM family. Last, the optimizer must use a generic approach for adapting to the rapid changes in cloud computing and software systems. Exploiting workload information and system internals can improve prediction performance but makes a CAT method less applicable to distinct systems [123, 124].

4.6 Time-Cost Trade-Offs

When hosting applications in the cloud, users can choose the *fastest* configuration regardless of cost or the *cheapest* configuration without the time constraint. Neither choice is truly piratical. There is always a trade-off between *time* and *cost*. Users are willing to spend more on resources when time is more critical (hard deadline) or the increase in cost expects reasonable decrease in execution time (soft deadline). Similarly, when the time constraint is relaxed, users accept slower configuration but with higher cost saving.

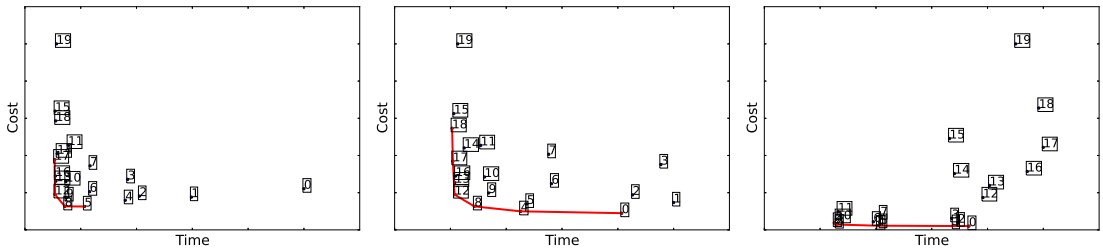
We propose using cost-delay product (CDP), similar to energy-delay product (EDP) [44], to analyze the trade-offs for choosing the cloud configurations of data-intensive applications. CDP puts the same importance on time and cost. For example, a 5% slow down in execution time is enough to justify a 5% cost saving. CD^2P and C^2DP, on the other hand, shift the importance to time and cost respectively. When the time improvement is 1% but it incurs 50% increase in cost, users will probably choose the slower configuration.

We run three real-world applications, *PageRank*, *web log analysis*, and *regression*, on Apache Spark, a distributed, large-scale data processing system [9]. We conduct a series of evaluations of the three applications against different resource costs by varying the number of CPUs (from 1 to 12) and the size of memory (from 1 to 4 GB per CPU). We measure the execution time to complete the applications. Table 4.3 lists their time and cost in detail. Figure 4.5 uses a scatter plot to show the execution time of applications running against different resource *costs*. We find that their distribution patterns are not totally alike. Therefore, there does not exist one best configuration for all applications. Even within the same applications, execution time can change dramatically. Figure 4.5 also illustrates a convex hull for outcomes bounded to a subset of the plane. When choosing the *best* configuration, the solution must be close to the convex hull.

In the PageRank case, for example, users are more likely to choose four CPUs instead of

Table 4.3 The execution time and resource cost of applications running with different numbers of CPUs and memory per CPU. The text in bold refer to the configurations on the convex hull in Figure 4.5.

Configuration			PageRank		Web Log Analysis		Regression	
ID	CPU	Memory/CPU	Time	Cost	Time	Cost	Time	Cost
0	1	1	652.2	3261.1	**179.3**	**896.3**	**765.0**	**3825.2**
1	1	2	389.2	2335.3	220.2	1321.1	706.7	4240.1
2	1	4	267.7	2141.8	187.8	1502.4	723.7	5789.7
3	1	8	240.3	2883.6	210.1	2521.1	701.0	8412.0
4	2	1	234.6	2346.3	**99.5**	**994.6**	**423.4**	**4233.6**
5	2	2	**139.1**	**1669.1**	104.3	1252.0	433.3	5200.1
6	2	4	151.3	2420.0	123.8	1980.1	407.8	6524.6
7	2	8	152.4	3657.0	121.6	2919.5	434.0	10414.2
8	4	1	**93.2**	**1864.5**	**63.0**	**1259.5**	**269.1**	**5382.2**
9	4	2	96.2	2310.0	74.2	1781.1	**261.7**	**6280.3**
10	4	4	95.7	3063.2	71.0	2270.8	268.7	8598.7
11	4	8	101.0	4846.5	68.0	3262.1	275.9	13242.1
12	8	1	**70.3**	**2811.5**	**47.3**	**1892.5**	817.8	32711.8
13	8	2	72.4	3475.4	47.5	2277.8	842.3	40431.1
14	8	4	75.4	4828.5	55.0	3515.8	711.2	45518.7
15	8	8	70.7	6783.4	46.8	4488.7	692.7	66502.3
16	12	1	71.2	4272.9	48.1	2887.0	983.5	59007.1
17	12	2	**69.6**	**5007.7**	**46.0**	**3309.2**	1043.4	75122.3
18	12	4	72.0	6912.8	**45.4**	**4357.1**	1026.6	98549.5
19	12	8	73.7	10617.9	49.9	7180.6	940.0	135362.0



(a) PageRank **(b)** Web Log Analysis **(c)** Regression

Figure 4.5 Applications' execution time and resource costs with different configurations.

eight CPUs because 25% reduction in execution time requires more than 50% extra cost. We explain the three applications' trade-off as follows.

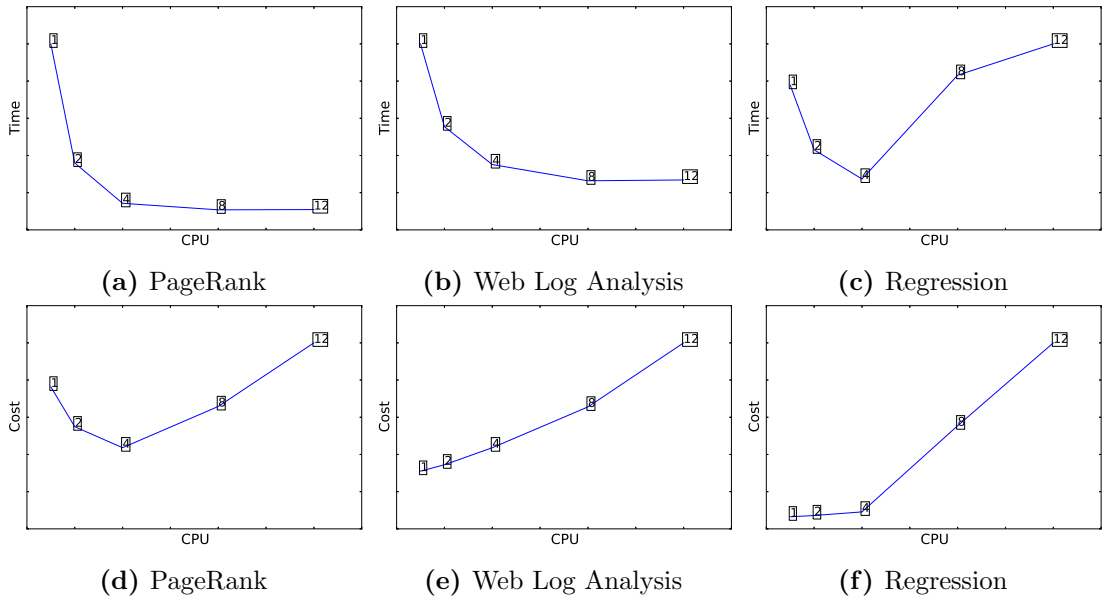
PageRank PageRank is an ranking algorithm to calculate the importance of website pages by counting the number of links to them. Our evaluation has shown that execution time decreases as we increase the number of CPUs. Although *PageRank* exhibits shorter execution time when the CPU count is greater than eight, it more makes sense to choose four CPUs as it is more cost effective, as depicted in Figure 4.6d. However, when time is more critical (CD^2P), Figure 4.6a shows that eight CPUs is a better choice.

Web Log Analysis This application tracks the query counts and aggregate bytes in a particular group. The CPU count greatly affects the execution time but memory size does not show significant impact. As shown in Figure 4.6b, a larger number of CPUs (increasing from 1 to 4) increase resource costs but time reduction is large enough to compensate the increase in cost. This is not the case when the CPU count is more than four. When time is more critical, CD^2P suggests eight CPUs is a viable configuration.

Regression The regression application generates a function that estimates the relationships among variables. Different from the above two applications, *regression* shows more CPU counts do no necessarily help reduce execution time. It is even worse. One possible explanation is the overhead by increasing parallelism overcomes the benefits by higher parallelism.

4.7 Conclusion

Cloud architecture tuning is essential for hosting applications in the cloud. In this chapter, we formulate the CAT problem and describe its challenges. We also present the state-of-the-art approaches and discuss their pros and cons. In the following chapters, we will study how to



(a) PageRank **(b)** Web Log Analysis **(c)** Regression

(d) PageRank **(e)** Web Log Analysis **(f)** Regression

Figure 4.6 The speedup and cost saving by CPU scaling (1GB memory per CPU).

incorporate low-level insights to improve a CAT method. We will demonstrate how to build a practical system that delivers an effective CAT solution.

Chapter 5

Low-Level Augmented Bayesian Optimization

Bayesian Optimization is a class of sequential model-based optimization, which is used in *CherryPick* to solve the CAT problem. In this chapter, we examine the effectiveness of Bayesian Optimization in finding the best cloud configurations. We identify a *fragility* issue in naïve Bayesian Optimization, and propose using low-level performance information to enhance Bayesian Optimization.

5.1 Introduction

In this chapter, we address the problem of finding a suitable cloud VM type for a recurring job. This problem is further aggravated by the *long execution times of the workloads* since a brute-force approach will no longer be a viable option. Furthermore, because there are evaluation costs, this *decision space* must be explored efficiently. The prior work in this area, solved this problem using two different approaches namely (1) *PARIS* [133] builds a complex performance model (using large-scale one-time benchmark data) to predict workload performance, and (2) *CherryPick* [3] uses Bayesian optimization to find the best cloud configuration. We prefer the Bayesian Optimization (BO) method because it does not require additional historical training data and supports any objective functions (essential for diverse workloads).

However, we have come across workloads where a Bayesian Optimization method is ineffective—surprisingly, we found this problem in a large number of workloads. Our large-scale empirical study, as shown in Figure 5.1, reveals that BO incurs different search cost on different workloads. We observe that Bayesian Optimization is effective in 50% of the workloads (in *Region I*) since it requires exploration of only 33% of the total search space. However, we also notice that Bayesian Optimization is not as effective at finding the optimal VM type for the other workloads

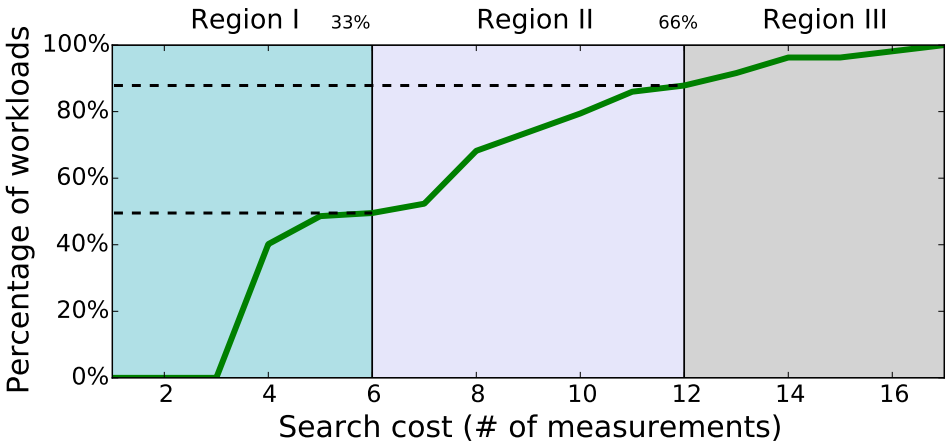


Figure 5.1 The number of measurements required by Bayesian Optimization (as used in [3]) to find the optimal VM type. We observe that 50% and 85% of the workloads (shown in dashed lines) require 6 (33% of the search space) and 12 (66% of the search space) measurements respectively. Bayesian Optimization is not always effective for any workload. The fragility problem—either incurs high search cost or yields sub-optimal solution (as in *Region II* and *Region III*).

(in *Region II* and *Region III*). This poor performance can be attributed to the insufficient information (for example core counts, memory capacity, etc.) used by Bayesian Optimization during the search process. Such VM characteristics are not sufficient to capture application behavior [34, 60, 133]. Consequently, Bayesian Optimization may fail to find the optimal VM for some workloads efficiently. Figure 5.2 shows how Bayesian Optimization is sluggish to find a 'better' VM type for a workload from the *Region III*. In summary, the lesson that we learned from the large-scale empirical study is *Bayesian Optimization is not a silver bullet to find optimal VM type* for any workloads. Furthermore, it can be *fragile*—either incurs higher search cost or yields a sub-optimal solution. Without further investigation, it is hard to claim BO is an effective method for finding the best VM type.

To further understand the fragility of Bayesian Optimization, we conducted a large-scale empirical study with three popular big data systems along with 107 different workloads and 18 different VM types (for more details refer to Section 4.2). We first observe that using rules-of-thumb (intuitions) to select the best VM type is far from ideal. There does not exist one such best VM type for all the workloads. Second, the same application with different input sizes may favor different VM types. Last, while the execution time tends to decrease with a more powerful VM, the cost per unit time goes up, which compresses the deployment costs. This creates a *level playing field*—several inferior configurations in execution time are now competitive in deployment cost. These reasons make the problem of selecting the best VM for

any given workload challenging.

To find the best VM type, *CherryPick* [3] uses Bayesian Optimization, which sequentially evaluates the VMs and moves closer to the optimal VM type. As presented before, a BO method can encounter the fragility problem. As shown in Figure 5.2, the execution time on the selected instance type after the fifth iteration is 1.75 times slower when compared to the optimal instance type. In this case, BO did not find the optimal solution until the thirteenth attempt. We argue that the fragility of BO arises from the *insufficient information*. That is, characteristics of a VM such as CPU speed, core counts, memory per core and disk capacity, are not sufficient to predict its performance. Besides, the *choice of the kernel function* (the prior) and the selection of the initial measurements are both critical to the effectiveness of BO [23, 37, 108, 111]. We believe they are also related to the fragility problem.

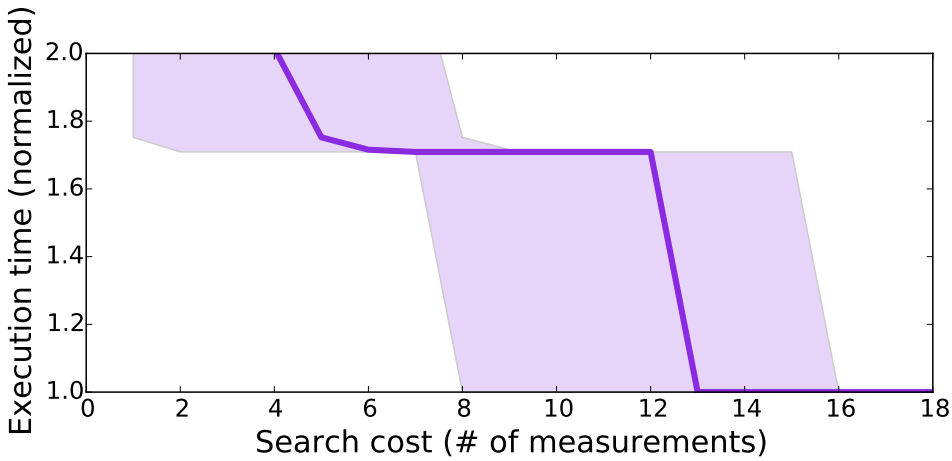


Figure 5.2 Using Bayesian Optimization to find the best VM type for running the ALS algorithm on Spark. The horizontal axis represents the search cost, and the vertical axis represents the execution time of the workload (for both lower is better). The edges of the colored area represents the 25 and the 75 percentile of the execution time. A naive Bayesian Optimization method progresses slowly towards the optimal VM type. The low-level augmented BO method alleviates the fragility problem as shown in Figure 5.6a.

Low-level performance metrics are a good proxy for estimating application and system performance [60, 133]. They are also useful to identify performance anomalies [18, 90]. We argue that *low-level performance information* such as I/O wait and memory usage well characterizes application behavior and better guides a BO method through the search process.

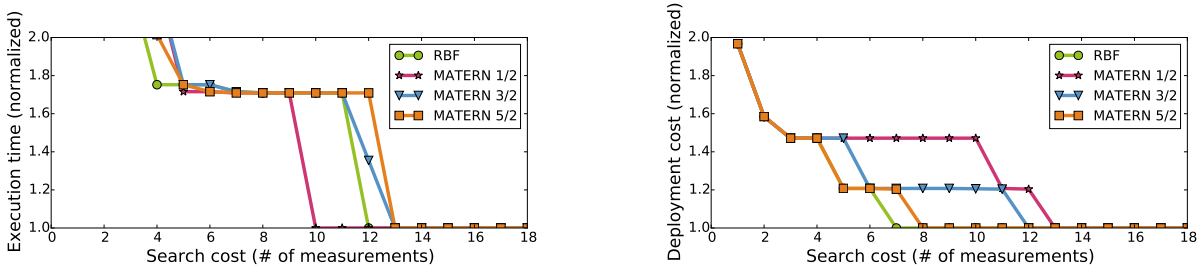
In this chapter, we propose a novel method to augment Bayesian Optimization by leveraging low-level performance information. However, embedding low-level performance information is

tricky because the (low-level) information is not available until the workload is executed on a given VM type. Our proposed modeling technique seamlessly integrates the high-level features with the low-level performance information. The prediction model estimates the workload performance in VMs (not measured) using the low-level performance information collected from previous measurements. Throughout the search process, the model keeps updating its belief based on the new measurements.

The proposed low-level augmented Bayesian Optimization (Augmented BO) outperforms the naive Bayesian Optimization (Naive BO) [3]. Our evaluation shows a reduction in search cost on 46 out of 107 applications in search for the most cost-effective configuration. Our method reduces about 20% search cost on average for cases with the fragility issue, and reaches 43% reduction for some while maintaining the same or slightly better performance in comparison to Naive BO.

5.2 The Fragility Issue

In this section, we explain why Bayesian Optimization can be fragile in CAT.



(a) Minimizing execution time for *als*

(b) Minimizing deployment cost for *bayes*

Figure 5.3 The number of actual measurements required to find the optimal VM type by Bayesian Optimization with different kernel functions. Each kernel function is tested with 100 different sets of initial points uniformly selected. The points represent the median performance from 100 runs.

Choosing the Right Kernel Function is Prone to Error

Since the choosing the covariance kernel function is critical, this section examines how different kernel functions can affect the usefulness of BO. We implement BO (as prescribed by *CherryPick*) to examine four different kernel functions. First, *RBF* (Radial Basis Function) is a widely used kernel. It considers the effects of features on the covariance equally [23], which may not be

realistic. Matérn kernel function is another family of covariance functions which incorporates a smoothness parameter such that it is flexible to model different objective functions. The smoothness parameter serves as the similarity function that determines whether two samples are alike. The most commonly used smoothness parameters are *Matérn 1/2*, *Matérn 3/2*, and *Matérn 5/2*.

Figure 5.3 shows the number of actual measurements required to find the best VM for a given workload. Figure 5.3a shows that BO with *Matérn 1/2* kernel finds the optimal VM faster thereby reducing the search cost. However, in Figure 5.3b, while trying to find a cost-effective VM, BO with *Matérn 1/2* kernel performs the worst. With the two particular examples, we want to demonstrate that choosing the appropriate kernel function affects the performance of BO. In practice, choosing the right kernel function relies on engineering and automatic model selection [23, 37, 108, 111].

Our prior experience indicates that it is possible to have a non-smooth performance outcome for a given workload on different VMs [60]. When a workload hits a resource bottleneck, *e.g.*,memory or disk, it can slow down greatly. This means that a workload might perform very differently on two VMs which are close to each other in the instance space. Therefore, we believe that architecture parameters alone are insufficient to predict the performance of cloud applications [60, 133].

No one-size-fits-all initial points

The choice of initial VMs also affects the effectiveness of BO. A common approach is a quasi-random method which uniformly selects very distinct VMs [112]. This method helps capture workload behavior, which can then be used to choose the next best VM to measure. However, in practice, we have seen that BO is sensitive to initial points (VMs in our setting) and can exhibit large variances in their outcome.

To demonstrate the effect of initial VMs on the performance of BO, we choose three very different starting points, *i.e.*,*c4.xlarge*, *m4.large* and *r3.2xlarge*, and then run BO on all the 107 workloads. We observe that about 15% applications do not find the optimal configuration within six attempts (33% of the instance space). We choose multiple combinations of initial VMs and repeat the same experiment, and we observe a similar phenomenon. This experiment shows that the performance of BO is dependent on the choice of initial VMs.

Even though there exists a set of initial points that work well on almost all applications, the optimal initial VMs are subject to change because new VMs are frequently added to the Cloud portfolios. Therefore, it is essential to design a search method that performs consistently with different initial points.

Summary

BO is a promising technique for finding the best VM for any workload. However, our large-scale evaluation shows that a BO method can be *fragile* or unstable. Without proper design, it may lead to high search cost or a sub-optimal solution. This is because the effectiveness of BO is significantly affected by choice of the kernel function and the initial VMs (used to seed the BO). However, choosing the suitable kernel function requires further analysis and in-depth study. To sum up, BO can be fragile and requires extra care while making design choices. Our objective is to make BO less fragile by (i) augmenting BO with additional (low-level) information and (ii) use variants of BO, which are less sensitive.

5.3 Approach

In this section, we introduce how to leverage low-level performance information to augment Bayesian Optimization.

Choosing the Low-Level Metrics

Prior work has shown that low-level performance metrics of workload are a good proxy for predicting performance [60, 90, 94, 133]. For example, the *memory commit size* represents the amount of memory required to handle current workload, and the *I/O wait time* may indicate a resource bottleneck. However, in practical settings, we need to analyze multiple metrics for better understanding the key factors that affect performance. System utilities on Linux, such as *sysstat*, provide a comprehensive set of performance metrics [117], which are useful to characterize workloads and identify performance bottlenecks. In this work, we use these low-level metrics to augment BO. The intuition behind this design choice stems from the fact that the published VM characteristics are inadequate to fully characterize a VM. In Figure 5.4, we show that using low-level information helps identify memory bottleneck of running Logistic Regression.

Since we focus on recurring jobs, we should use metrics that can capture the workload progress and identify resource bottleneck. The selection of low-level metrics depends heavily on workloads. If possible, we should use a comprehensive set of metrics. However, a large number of features can lead to the over-fitting problem in building predictive models. This is known as the curse of dimensionality [39]. Automatic feature selection can help address this problem [53, 60] but requires further studies. In this work, we find the following low-level metrics are effective.

- **Workload progress**: CPU utilization of user space processes and I/O operations, and the number of tasks in the task list.
- **Memory pressure**: % of commits in memory.

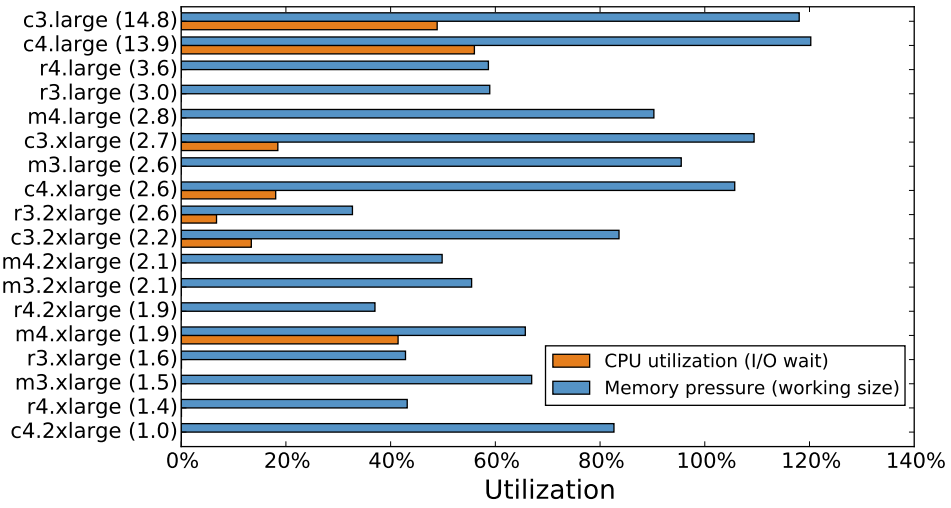


Figure 5.4 A memory bottleneck is identified by low-level performance information. The horizontal axis represents the resource utilization (%), and the vertical axis represents the VM types. The numbers in the parenthesis are the normalized execution time where 1.0 represent the best VM type. The memory of a small VM type (c3.large) is not sufficient to run Logistic Regression, which leads to 14.8 times slower than the best VM type (c4.2xlarge). This behavior is captured by memory pressure and CPU utilization.

- **I/O pressure**: disk utilization and disk wait time.

Low-Level Augmented Bayesian Optimization

Leveraging low-level information in BO requires novel modeling methods because the given workload is yet to be executed on the candidate VM. Our approach, instead, predicts the performance (cost or time) based on the VM characteristics and observed low-level metrics of the VM that is already measured. This is similar to the reasoning technique used in practical settings by experts and the table based models [7]. Experts choose to interpolate or extrapolate the workload performance using not only characteristics of VM but also the low-level performance information.

We make the following design choices to integrate low-level performance information into BO. Algorithm 3 illustrates the Augmented BO.

- **Augmented Instance Space**: Instead of using only VM characteristics (VM_i) as an input to the surrogate model, we also use low-level metrics (L_i) collected from running the workload on (VM_i). These constitute the *independent variables*. Similar to Naive BO, the performance of the workload is used as the *dependent variable*. *Decision to use low-level information allows BO to make more informed search.*

Algorithm 3: Low-Level Augmented Bayesian Optimization (similar to Algorithm 1)

Input: f, VM, S, M
Output: Near-optimal configurations

$D :=$ Initial sampling (f, VM)
for k *in* $|VM \notin D|$ **do**
  $p(y|vm, D, L) :=$ FitLowLevelModel(M, D, L)
  $vm_k := argmax_{vm \in VM} S(x, p(y|vm, L, D))$
  $y_k, L_k := f(vm_k)$
  $D := D \cup (vm_k, y_k, L_k)$
  if *meeting stopping criteria* **then**
    break
  end
end

- **Surrogate Model**: Instead of using Gaussian Process as the surrogate model, we choose a tree-based ensemble method Extra-Trees algorithm for building the surrogate model. The tree-based learning method is effective to capture complex performance behavior [60, 89, 124, 133, 135]. We choose not to use Gaussian Process in Bayesian Optimization because determining the right kernel function (as discussed in Section 5.2) requires careful evaluation, which is not practical for supporting diverse workloads. *This design choice lets us side-step one of the reasons for the fragility of Naive BO.*

- **Acquisition Function**: We replace *Expected Improvement* (EI) with *Prediction Delta* as the acquisition function. Prediction Delta can be used for selecting a VM type with better performance (shorter execution time or cheaper deployment cost). Prediction Delta can also be used as a stopping criterion—to terminate the search process if there exists no better VM type. We do not use Expected Improvement as our acquisition function because it is not useful when the kernel function cannot estimate the black-box function. *This design choice for an acquisition function which does not require a suitable kernel function.*

- **Surrogate Model Update**: When updating the surrogate model upon a new observation for workload (w) ($VM_{i,w}$, $L_{i,w}$ and $y_{i,w}$), we generate multiple pairs of input ($VM_{j,w}$, $VM_{i,w}$), where $i \neq j$ with low-level information ($L_{j,w}$), where j represent the source VM—which has been measured and i represents the destination VM—which is yet to be measured. This surrogate model answers "what is the predicted performance of $VM_{i,w}$ given the low-level performance information observed on a particular VM ($VM_{j,w}$)". For example, if we have measured the performance of workload (w) in 3 VMs ($VM_{1,w}, VM_{2,w}, VM_{3,w}$), the number of independent values for which the performance needs to be estimated would be $3 \times (18 - 3)$. It should be noted that in order to estimate the performance of a workload

in a VM (say $VM_{15,w}$), we have to consider $VM_{1,w} \rightarrow VM_{15,w}$, $VM_{2,w} \rightarrow VM_{15,w}$, and $VM_{3,w} \rightarrow VM_{15,w}$. Since multiple pairs exist, we average the estimated performance. *This design choice helps us update the surrogate model even when the low-level information of destination VM is not available.*

5.4 Evaluation

This section describes our experimental setting and evaluation method to compare Augmented BO with Naive BO.

5.4.1 Experimental Method

Workload

For evaluation, we use Apache Hadoop (v2.7) and Spark (v2.1 and v1.5). We choose distinct workloads from *HiBench* and *spark-perf*, as listed in Table 4.1. *HiBench* is a big data benchmark suite for Apache Hadoop and Spark [58]. It was designed to test batch processing jobs and streaming workloads. Similarly, *spark-perf* is a performance testing suite for Spark [114]. The testing suite provides a wide range of workloads including supervised learning such as regression and classification modeling, unsupervised learning such as K-Means clustering, and statistical tools such as correlation analysis, and Principal Component Analysis (PCA). We run 107 workloads to test their performance on 18 VM types. During the execution of the workload, a *sysstat* demon is run in the background to collect low-level performance information [117]. There is no observable overhead to this data collection.

Cloud Configurations

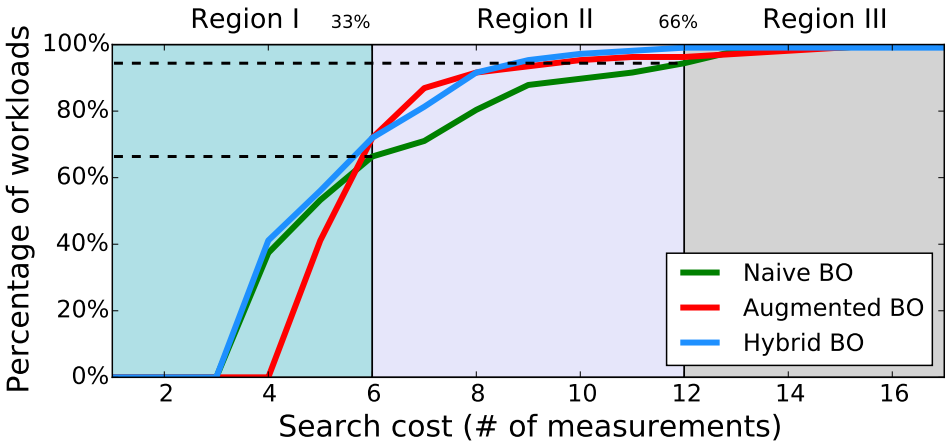
We measure the performance on six VM families (available on AWS) {*c3*, *c4*, *m3*, *m4*, *r3* and *r4*}, and three VM sizes {*large*, *xlarge* and *2xlarge*}.[1] The VM size represents the core count. For example, *c4.large* has two cores, *c4.xlarge* has four cores and *c4.2xlarge* has eight cores.

Encode Cloud Configurations

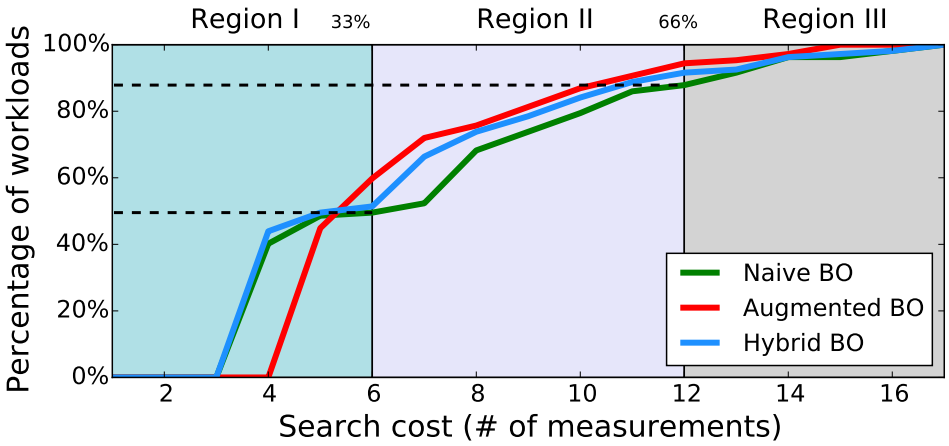
Each VM type is characterized by CPU types, core count, average RAM per core, and the bandwidth to Elastic Block Storage (EBS). We encode the four features with numerical values into $\vec{VM}$. The CPU types are encoded from one to six in order, and for the core count, we

[1]The latest generation has been upgraded from *c4* to *c5* for the compute-optimized VM and from *m4* to *m5* for the general-purpose VM after we completed our data collection.

use their actual values {2, 4, 8}. Similarly, the RAM size per core is {2, 4, 8} GB. Last, the bandwidth to EBS has three classes for different VM types encoded as {1, 2, 3}.



(a) Optimizing running time



(b) Optimizing running cost

Figure 5.5 Search cost of finding the optimal VM type across the 107 workloads. The *y-axis* represents the cumulative percentages of workloads. In *Region I*, although Augmented BO does not find the optimal VM type at the fourth step, it does find a very near optimal solution with only 4% difference. Section 5.5 provides further details.

5.4.2 Comparison

This section evaluates Naive BO and Augmented BO on the 107 workloads with randomly selected initial VMs. The above process is repeated 100 times to account for variance. Here we

minimize execution time and deployment cost individually. Figure 5.5 presents the overall result.

Can Augmented BO find optimal VMs?

Figure 5.5a shows the percentage of workloads, where Naive BO and Augmented BO find the optimal VM. The horizontal axis represents the search cost (in terms of the number of measurements), and the vertical axis represents the percentage of the workloads. In this figure the green line represent the Naive BO and the red line represent the Augmented BO. The Naive BO can find the optimal solution for 60% of the workloads by searching 33% of the search space (Region I). The performance of Augmented BO is similar to naive BO. However, Augmented BO has a slow start problem but becomes effective eventually. In Region II, Augmented BO is a clear winner as it can find optimal VMs for 96% of the workloads within ten measurements. At the same time, Naive BO can only find 80% of the workload.

We claim that the performance of Augmented BO is better than Naive BO, for regions I and II (at step 6 and 12). We also observe an interesting phenomenon—Augmented BO is outperformed by Naive BO in initial four steps. The one-step difference can be attributed to the over-fitting problem caused by the larger training features (both high-level and low-level information) in Augmented BO. This is a challenge of leveraging low-level information (for future work).

While looking at the performance of VMs selected by Augmented BO, we observe that the best VM found by Augmented BO is only 4% away from the optimal VM. In practice, this difference can be easily ignored (refer to Section 5.5). Furthermore, with the growing instance space, this difference (though we believe is little) can be amortized because the search cost will also increase.

A possible workaround to this problem can be to create a Hybrid BO (shown in blue)—which combines the best of the two methods. Figure 5.5a shows that *Hybrid BO* outperforms Naive BO in all cases. However, we choose not to focus on the hybrid method here because our primary objective is to identify the fragility of Naive BO and the advantages of leveraging low-level information. Please refer to Section 5.6 in more detail.

Can Augmented BO minimize cost?

To answer the question if Augmented BO can minimize the deployment cost, it is essential to demonstrate that Augmented BO can find optimal VM faster than Naive BO (lower search cost). In Figure 5.5b, we observe that *minimizing deployment cost is more difficult* than minimizing execution time, *i.e.*,both methods require more search cost to reach the optimal solution. Naive BO can find the best VM with six attempts for only 50% applications while Augmented BO increases this probability to 60%. We also see a clear win for Augmented BO as it can find

best VM which minimizes the deployment cost after measuring five measurements. However, we see that Augmented suffers from a slow start, which is similar to Figure 5.5a and Hybrid BO (shown in blue) is the workaround.

Is Augmented BO fragile?

In finding the best VM, Naive BO fails in 36% (minimizing time) and 50% (minimizing cost) of the workloads after measuring the performance of the six VMs (Region I). Augmented BO alleviates this problem, and this can be observed by a up to 20% increase in the number of workloads for which Augment BO found the optimal VM (step 7 in Figure 5.5b).

Stability is another important aspect of Augmented BO. As discussed in Section 5.2, initial points are critical to the performance of BO—different initial VMs can lead to very different results (performance and search cost) or high variances in results. Figure 5.6 compares the search cost and the performance found by the two methods. We present the median value (shown by line), and the interquartile range (the difference between the 3^{rd} and the 1^{st} quartile) shown by the shaded region. The three cases show that Augmented BO yields less search cost and reduces the variance. This demonstrates Augmented BO is not fragile.

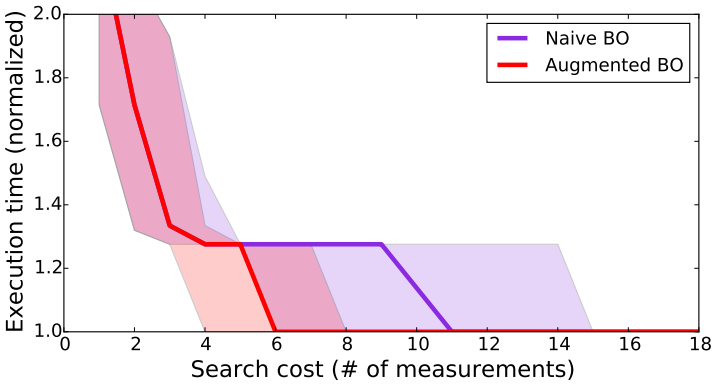
Another interesting observation is that Augmented BO not only alleviates the fragile problem in *Region II* but also moves workloads from *Region III* to *Region II*. Figure 5.6a and Figure 5.6b are example workloads in *Region III* for Naive BO. The first quartile indicates that Augmented BO finds the optimal configuration even with four or five attempts in 25% initial points that are tested.

5.5 Discussion

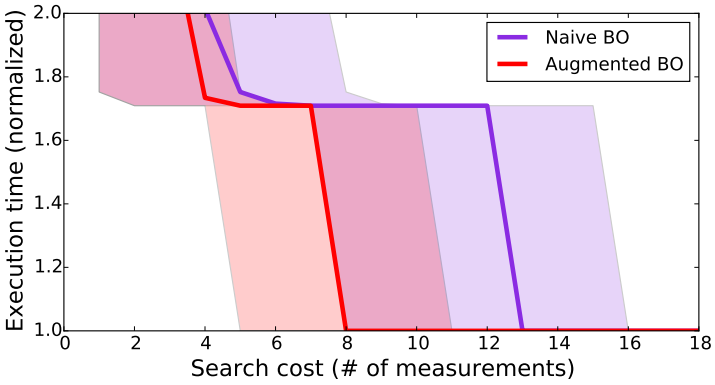
5.5.1 Bayesian Optimization in Practice

In practice, users can tolerate a loss in performance (deployment cost or execution time) in exchange for lower search cost. In this section, we examine the performance of the two methods when we (slightly) relax the definition of optimality. Due to space limitations, we only present the results of minimizing deployment cost as we have shown it is more challenging, and the conclusion is similar to minimizing execution time.

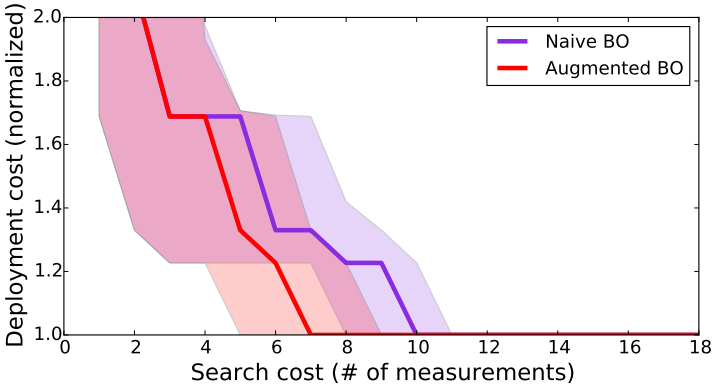
To demonstrate the performance (of BO) and search cost trade-off, we vary the stopping criteria to understand how they affect both search cost and the best VM they find. We choose EI as the stopping criteria for Naive BO (as prescribed by *CherryPick*). For Augmented BO, we use Prediction Delta and vary the thresholds from 0.9 to 1.3. We examine the three regions separately to analyze the effects of stopping criterion on different categories of workload.



(a) PageRank on Hadoop 2.7



(b) Alternating Least Squares on Spark 2.1



(c) Logistic Regression on Spark 1.5

Figure 5.6 Examples of searching for the best VM. The objective is to find the fastest VM in subfigures (a, b) and the most cost-effective VM in subfigure (c). Both the BO methods stops after they find the optimal VM type (normalized to 1.0). The line represents the median value of the execution time over 100 repeats. Each repeat used different initial points to seed BO. The shaded region represents the IQR or Interquartile range is the difference between 3^{rd} and 1^{st} quartile. A high value (larger area) of IQR indicates high variance.

In Figure 5.7a, Naive BO finds the optimal VM regardless of the stopping criteria. This is counter-intuitive because there should exist a trade-off between the deployment cost and the search cost. We hypothesize that Naive BO cannot estimate that it has found the optimal VM. Augmented BO, on the other hand, clearly shows the trade-off. Augmented BO with the thresholds 1.25 and 1.3 performs similarly to Naive BO.

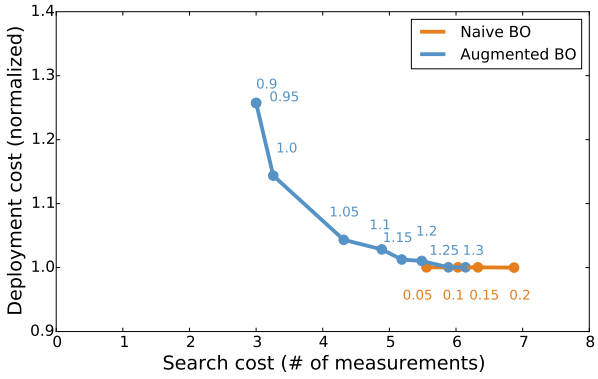
In Figures 5.7b and 5.7c, Augmented BO is the clear winner. With the 1.1 threshold, Augmented BO outperforms Naive BO in both the search cost and the deployment cost. To simplify the comparison, we choose 10% EI for Naive BO (as prescribed by *CherryPick*) and 1.1 threshold for Augmented BO. Our method yields lower search cost while achieving lower deployment cost. On average, it finds VMs with 5% lower in deployment cost while reducing search cost by 20%. This demonstrates that the low-level augmented Bayesian Optimization finds the well-suited VMs quicker and is more precise when compared to Naive BO.

Overall, we recommend using 1.1 threshold in Augmented BO since the deployment cost is comparable with Naive BO and reduces the search cost. In Figure 5.8, we present the overall comparison of the two methods with the EI and threshold described above. The horizontal axis represents the reduction in search cost, and the vertical axis represents the decrease in the deployment cost (higher the better in both). The figure shows the result for all 107 workloads represented as points. Points enclosed with lines $x \geq 0$ and $y \geq 0$ (shown in blue shade) indicates workloads, where Augmented BO can find VMs with lower deployment cost and lower search cost. For example, the workload represented in (24, 10) is the case where Augmented BO uses 24% lower search cost, and the best VM found (for that workload) has 10% lower deployment cost than the one found by Naive BO. There are 46 such workloads. Augmented BO requires higher search cost than Naive BO in five workloads (region shaded in red). But they both find the optimal solution. There are 17 workloads where Augmented BO finds VM types with higher running cost but with lower search cost—a region of trade-off.

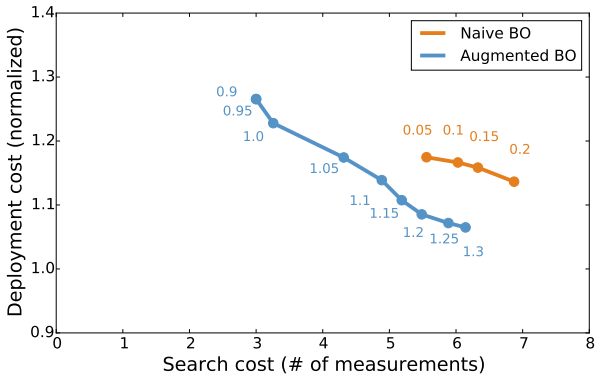
5.5.2 Time-Cost Trade-off

This section demonstrates how to adapt Augmented BO as well as Naive BO to navigate the time-cost trade-off. In practice, a user would always want a solution to reduce time as well as cost. We propose a new measure called time-cost product which is similar to an energy-time trade-off in high-performance computing [44]. Not every time-cost trade-off is desirable because a small improvement in performance may incur a higher running cost. For example, a 10% improvement in execution time requires a 50% increase in deployment cost.

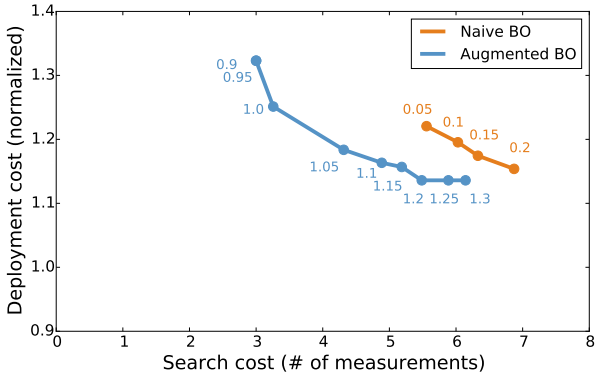
For simplicity, we assign the same importance to time and cost. That is, it is considered desirable for a 10% improvement in time and a 10% increase in cost. To support the time-cost trade-off, instead of predicting the execution time and deployment cost, the surrogate model



(a) Region I



(b) Region II



(c) Region III

Figure 5.7 Comparison between effectiveness of search with different stopping criteria. There is a trade-off between search cost and deployment cost. In *Region I*, Augmented BO is comparable with Naive BO in terms of deployment cost but can greatly reduce search cost at the expense of slight increase in deploymwnr cost. For *Region II* and *Region III*, Augmented BO outperform Naive BO for both search cost and deployment cost.

estimates the product of time and cost. Any two VMs are considered the same if their products of execution time and deployment cost are the same. Similarly, a larger product represents an undesirable choice.

Figure 5.9 presents the comparison which is similar to Figure 5.8. We observe a great reduction, *i.e.*,> 50% in search cost. Naive BO exhibits long searching process (more than six attempts) in 24% of the 107 workloads and very long searching (at least ten attempts) in 13% workloads. On the other hand, Augmented BO requires no more than 6 actual evaluation for all 107 applications. Please note that the threshold used for this experiment is 1.05, which also tells us that the stopping criteria also need to be changed based on the workload as well as the performance objective.

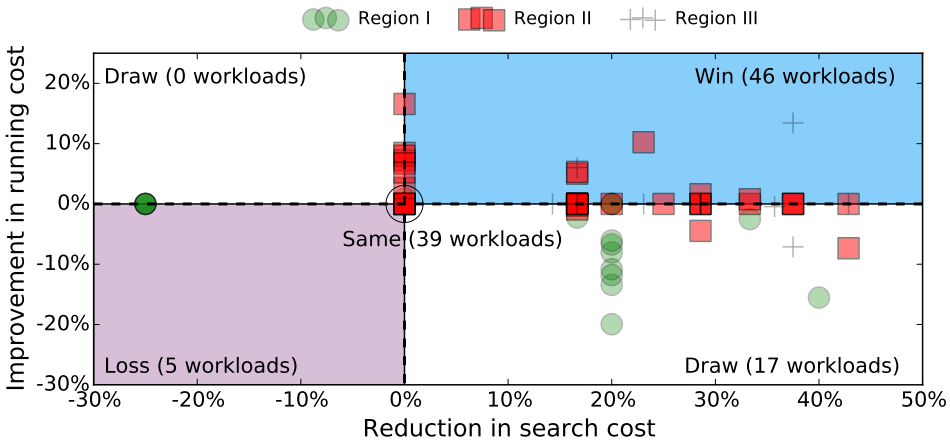


Figure 5.8 Overall comparison for the two BO methods in finding the most cost-effective VM type across the evaluated 107 workloads. The numbers are calculated as the reduction percentage in search cost and improvement in deployment cost, both higher the better. Workloads in (0,0) represent workload which achieve similar performance in both methods.

5.6 Hybrid Bayesian Optimization

Bayesian Optimization is an optimization framework that follows Sequential Model-based Optimization (SMBO). *CherryPick* implements a Bayesian Optimization method that uses only instance-level information such as core counts and memory size for building the probabilistic model [3]. We call our similar implementation *Naive BO*. Our proposed method, low-level augmented Bayesian Optimization (*Augment BO*) uses both high-level, instance-level information and low-level performance information. However, the proposed *Augment BO* encounters a "slow

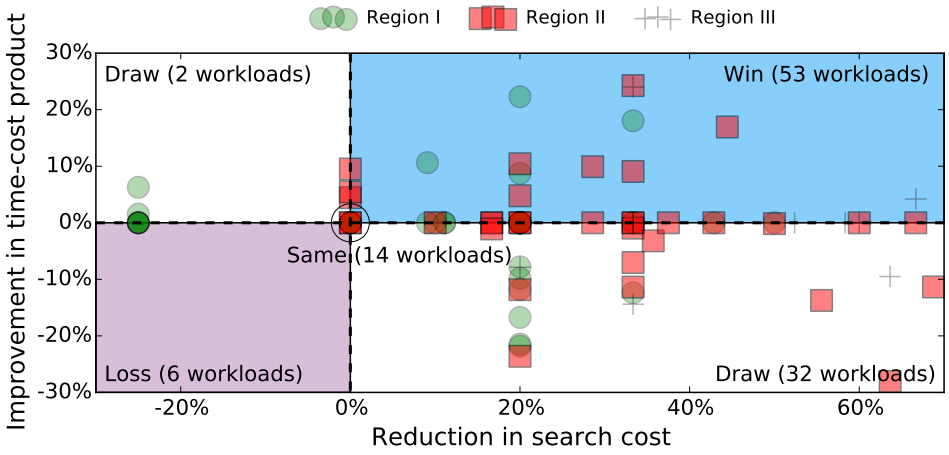


Figure 5.9 Similar to Figure 5.8, the optimization objective is to find the best configuration both in execution time and search cost. Augmented BO supports finding the best VM type, given a time-cost tradeoff.

start" issue. A possible explanation is the over-fitting problem—building a predictive model with high dimensional training data.

To remedy this problem, we propose a hybrid approach (Hybrid BO) that combines both Naive and Augmented BO. The intuition here is that *Naive BO* performs well when the instance-level information is able to characterize a given workload well. This results in a small number of measurements (search cost). When the instance-level information is not sufficient, *Naive BO* may encounter the *fragility* problem—either incurs higher search cost or yields a sub-optimal solution. On the other hand, *Augmented BO* is more stable and produces desirable outcomes for cases that *Naive BO* does not work well. The proposed *Hybrid BO* maintains two Bayesian Optimizer at the same time. Consequently, *Hybrid BO* can produce better solutions for all cases.

The key component in *Hybird BO* is the acquisition function. *Hybrid BO* uses two prediction models, one from *Naive BO* and the other one from *Augmented BO*. During the search process, *Hybrid BO* "ranks" each candidate choice. For the high-level model, the candidates are ranked by their Expected Improvement (EI), and the predicted performance (such as execution time or deployment cost) is used in the ranking process. *Hybrid BO* uses the average rank to pick the next candidate to evaluate. Algorithm 4 describes the above procedure.

In practice, a stopping criterion is required for the optimizer. Our current design is to apply separate stopping criteria to the two models. For example, we can use $EI = 10\%$ for *Naive BO* and $PD = 1.1$ (Performance Delta, as described in previous sections) for *Augmented BO*. If a candidate solution does not meet either the EI or PD constraint, it is excluded from the rank process. There exists several variants and we will leave them for future work.

Algorithm 4: Hybrid Bayesian Optimization

Input: f, VM, S, M
Output: The optimal configuration

$D :=$ Initial sampling (f, VM)
for k *in* $|VM \notin D|$ **do**
  $p(y|vm, D) :=$ FitHighLevelModel(M, D)
  $p(y|vm, D, L) :=$ FitLowLevelModel(M, D, L)
  $vm_k := argmax_{vm \in VM} S(x, p(y|vm, L), p(y|vm, L, D))$
  $y_k, L_k := f(vm_k)$
  $D := D \cup (vm_k, y_k, L_k)$
end

5.7 Conclusion

In this chapter, we identify and demonstrate the fragility of Bayesian Optimization in finding the best cloud VM type. The fragility arises from the inadequate information used to represent the instance space. This fragility affects prior work which uses only instance space to guide Bayesian Optimization. To overcome the problem of fragility, we augment the instance space with low-level performance information. We present our method, Augmented Bayesian Optimization, which seamlessly integrates the low-level metrics (obtained with negligible overhead) to the surrogate model. Additionally, we make design choices to modify existing BO to make more informed decisions. We demonstrate that Augmented BO can find the best VM type across all workloads. In 46 out of 107 workloads, Augmented BO outperforms the state-of-the-art Bayesian optimization method in terms of both performance and search-cost.

More generally, we conclude that it is often insufficient to use general-purpose off-the-shelf methods (BO in this case) for selecting the best VM without augmenting those methods with essential systems knowledge such as CPU utilization, working memory size and I/O wait time. In our future work, we plan to further augment Bayesian Optimizer with historical performance data to further reduce the search cost.

Chapter 6

Scout: System Design and Implementation

To solve the CAT problem, one-shot prediction (such as *PARIS* [133]) suffers from high variance of prediction error while Bayesian Optimization (such as *CherryPick* [3] and *Arrow* [62], described in Chapter 5) must tolerate the *cold-start* issue. In this chapter, we design and implement an effective, efficient and reliable system that recommends the best architectural configurations that satisfies performance and cost objectives to run a given workload.

6.1 Introduction

This chapter presents Scout, which identifies search regions that contain suitable architecture configurations. Scout follows sequential model-based optimization (SMBO) that converges to the best architectural configurations. This method better tolerates high variance of the prediction error in a machine learning based prediction model. Second, Scout adopts pairwise comparison for determining the next architectural configurations that are likely to improve upon the current choice. Instead of predicting workload performance directly (*e.g.*, in *CherryPick* and *PARIS*), Scout uses relaxed modeling to find the next *better* choices (relative ordering). This naturally fits into SMBO. Third, Scout uses low-level performance metrics to identify performance bottlenecks and to capture cost-performance relationship. Last, Scout uses transfer learning to acquire knowledge of CAT from other workloads. These four elements enable Scout to navigate through the search space more quickly and intelligently.

Any search-based method has two aspects.

- *Exploration:* Gather more information about the search space by executing a new cloud configuration.
- *Exploitation:* Choose the most promising configuration based on information collected.

Additional exploration incurs higher search cost and insufficient exploration may lead to sub-optimal solutions. This is the exploration-exploitation dilemma the appears in many machine learning problems [68]. For example, *CherryPick* requires a good exploration strategy to characterize the search space [3].

In this chapter, we demonstrate that it is possible to trade exploration for exploitation without settling for a sub-optimal configuration. The central insight of this work is that the cost of the search for the right cloud configuration can be significantly reduced by using information gathered during tuning. However, prior work such as *CherryPick* and *Arrow* learns from an optimization process of a single workload [3, 62]. This produces unnecessary search cost on exploration (related to the *cold-start* problem) and may eventually lead to sub-optimal choices (due to irregular search space). Scout is able to alleviate these problems with *transfer learning* [95]— the knowledge is transferred from previous (but distinct) workloads using relative ordering and low-level performance metrics, which does not require workload information.

In this chapter, we identify the key components for an effective method to the CAT problem. Scout enables practitioners to find a near-optimal cloud architecture configuration with a better search performance and a lower search cost than the state of the art. Our key contributions are:

1. we propose a novel system, Scout, that finds (near) optimal solutions and solves the shortcomings of prior work. (Section 8.3);
2. we present a novel way to represent the search space, which can then be used to transfer knowledge from historical measurements(Section 8.3);
3. we compare the performance of Scout with other state-of-the-art methods using 125 workloads and 87 architecture configurations on three different data processing systems. (Section 8.4); and

6.2 Design Choices

By analyzing the differences between the state of the art methods, we identified the following key components in solving the CAT problem: (1) a search-based method (similar to *Cherrypick* [3]) is essential since it accommodates mispredictions and performance variances in the cloud, (2) relative ordering better captures the workload-architecture-performance relationship, which creates fewer mispredictions, (3) low-level performance metrics are a good proxy of predicting system performance [60, 90, 133], (4) historical data (as used in PARIS [133]) is useful to understand the inherent preferences of a workload, and (5) transfer learning boosts search performance and improves convergence speed by minimizing exploration phase. These components together solve the CAT problem more effectively and overcome the shortcomings of the current state of the art approaches. Figure 6.1 compares and contrasts the design choices of Scout

Methods	Search-based	Relative Ordering	Low-level Metrics	Historical Data	Transfer Learning
CherryPick [3]	✓	✗	✗	✗	✗
PARIS [133]	✗	✗	✓	✓	✗
Arrow [62]	✓	✗	✓	✗	✗
Scout	✓	✓	✓	✓	✓

Figure 6.1 An overall comparison with other CAT methods. A search-based method better tolerates prediction bias. Relative ordering better captures the workload-architecture-performance relationship. Leveraging low-level metrics improves search performance. Historical data helps eliminate unnecessary exploration overhead in a search. Transfer learning greatly reduces search cost.

against prior work.

Because it is a daunting task to build an accurate model that predicts performance and cost of workloads on distinct cloud architectural configurations, we can instead build an indirect model (for improving prediction accuracy). A search-based method does not require a direct answer (which choice is the best), but an answer to "are there better choices?" We do not predict the absolute performance of a configuration but rather predict the relative performance of two configurations. That is, we can simplify the prediction model that will assist a search-based method in finding the solutions more efficiently [86]. *Learning to rank* is an important machine learning task [24, 54, 77]. We prefer relative ordering instead of total ordering for ranking architectural configurations because there does not exist a one-size-fits-all architecture for any workloads and for any objectives.

SMBO requires exploration efforts (increases search cost) to update its belief (prediction) on the search space. However, the data for the initial model need not come from the workload being evaluated. Rather, data from any workload can be used to build a useful model describing the search space. For this model to be most useful, the information must be generic—independent of workloads. This technique is inspired by [60, 133]. There are too few features (dimensions and options) in the configuration space to build a robust model that works across many workloads. Consequently, a model based only on architectural features (*e.g.*,cluster sizes and memory per core) is fragile.

To summarize, the following elements are necessary to create an effective approach.

1. Prefer the ***search-based technique***, which converges to the best solution iteratively and avoids the large penalty caused by dramatic prediction error.
2. Use a ***relaxed*** model that boosts prediction accuracy, thereby better guides a search process to find the near-optimal configurations more quickly,
3. Use ***low-level metrics*** to generate a generic representation of the search space such that it can be used by other combinations of workload and application.

4. Create a ***performance database*** so that the knowledge of optimization can be used by other optimizers to find the right cloud configuration and hence reduce the search cost.

6.3 From Observation to Action

In this section, we describe how to derive search hints from performance data to guide a search process.

6.3.1 Exploration vs. Exploitation

A search-based method navigates in the search space to find the best cloud architectural configuration. It is mostly concerned with two questions: "what are better choices?" and "what are more promising regions?" The former ensures that a search will eventually, find a near-optimal configuration while the latter determines how quickly it finds the solution (also known as convergence speed). An effective and efficient search method must answer these two questions.

To this end, we actually need only to know "what are better choices?" At each step, a search-based method aims to find a cloud configuration that is better than the current best. A higher probability of *guessing* the next step right ensures that a search process sequentially finds a better choice. A right next step also guides a search process to move towards the right direction. As long as the optimizer can move closer to the desired solution at each step, it is more likely to guarantee it will find near-optimal solutions.

To better determine the next step, a search process can learn from the observations along the search path. However, this method faces two challenges. First, it requires collecting sufficient data to build strong belief. *CherryPick* is confronted by the cold-start issue since it must first explore the search space—to identify the promising regions. Second, an insufficient number of observations leads to high bias in prediction—the method can wrongly believe that a particular region (such as VM types or cluster sizes) is more promising than the other, leading to a sub-optimal solution.

Instead of learning only from observations collected while executing the target workload, a search process also can learn from performance data of other workloads—which have been optimized in the past. This addresses the issue of high bias because a larger number of measurements (performance data) is available to create a prediction model that generalizes a performance model better. This also sidesteps the exploration problem because the search process does not need to collect observations by running workloads of the current search task. The idea of reusing the data is often tricky since the different combination of application and workload exhibit very different behavior. For example, the same application with different inputs can create very different workload behavior (such as the execution time and running cost) [62]. A performance

model, which captures this complex behavior requires more information about the search space than just the architecture level information such as VM types and cluster sizes.

6.3.2 Core Techniques

To navigate the search space efficiently, SCOUT is built on the following four ideas.

Pairwise comparison

A search-based method determines the next configuration to evaluate. That is, the method only needs to rank the set of unevaluated cloud configurations. Therefore, we can use *Pairwise Comparison* modeling scheme [126]. *CherryPick* uses Gaussian process to build prediction model, $f(x_i, K) = y_i$, where x_i is the feature vector to represent an architecture configuration and K is the covariance kernel function. With pairwise comparison, the learning task is

$$f(x_i, y_i, x_j) = y_j, \tag{6.1}$$

where $x_i \in E$ and $x_j \in S - E$. In words, it predicts the cost (y_j) of a configuration not yet evaluated (x_j) given the cost (y_i) from the best configuration yet found (x_i). This modeling technique does not require to make an assumption (K) about the search space (which is another hyper-parameter to tune), and naturally fits into SMBO in updating belief upon new observations. For the same workload, switching from a smaller VM type (*e.g.*,*large*) to a larger one (*e.g.*,*2xlarge*) may result in different performance on different VM families (*e.g.*,*c4* and *r4*). This performance relationship may change dramatically due to workload changes. Regression-based modeling needs to fit the corresponding features accurately for predicting performance [100, 129]. Pairwise comparison helps capture performance transition between architectural configurations. Although there are $P(n, 2)$ pairs (permutation), where n is the number of configurations, in practice, we do not have to obtain full pairs for training this model. This is because some pairs, *e.g.*,(*c4.large*, *r4.large*), show significant performance differences in comparison with other pairs, *e.g.*,(*m4.large*, *r4.large*).

Relative ordering

Ranking architectural configurations does not need to predict absolute performance (a harder problem). Besides, "a bad learner" sometimes can still find a good solution [86]. The idea of using an inaccurate model is useful because the effort required to build an accurate model is much higher than an inaccurate model. Based on this insight, we choose not to infer (inaccurate) performance measure but rather to infer (accurate) relative ordering— one configuration is better than another. To support relative ordering, we modify the learning task from Equation 6.1

to

$$f(x_i, x_j) = \sigma(\frac{y_j}{y_i}) \tag{6.2}$$

where σ is a ranking function. Binary ranking (*i.e.*,better or worse), for example, is the simplest form. This transformation simplifies the learning task because it does not predict absolute performance. Furthermore, a coarse-grain ranking function better tolerates performance variance in the cloud because a small variance may still result in the same ranks. Figure 6.2 shows that the prediction accuracy of relative ordering (using classification) is higher than total ordering (using regression). This experiment uses *ExtraTrees* [48] in both the regression and classification task for fair comparison. The increase in prediction accuracy greatly reduces search cost because it is more likely to avoid wrong predictions (see Section 6.5.5).

Low-Level Insight

Low-level performance metrics help identify performance problems [18, 90] and predict application and system performance [60, 133]. Understanding resource bottlenecks helps choose the right cloud configuration and helps Scout ignore not so promising cloud configurations. For example, in optimizing execution time, a memory bottleneck indicates an instance with larger memory may improve resource efficiency, thereby reducing execution time. With low-level performance information, our learning task becomes

$$f(x_i, m_i, x_j) = \sigma(\frac{y_j}{y_i}) \tag{6.3}$$

where m_i represents the low-level performance vector. This is similar to the process of troubleshooting performance problems and identifying resource bottleneck. Instead of constructing rules manually, this learning task can extract those rules implicitly. When a workload runs inefficiently on one architectural configuration, Scout observes abnormal or insufficient resource usage. This observation is translated to prediction probability implicitly. Scout ignores those configurations with low prediction probability.

Transfer learning

The accuracy of a performance model depends on the number of data points used to train the model. In CAT, the data points are expensive to compute. When building the performance models, it might be best to reference observations from other optimization processes. Researchers in transfer learning report that data from other optimization processes can yield better models than just using current data [95, 98]. In this work, workloads share the same search space S and therefore, architectural features are the same. Besides, Scout uses generic low-level metrics. These enable transfer learning possible in Scout. Although including workload information might

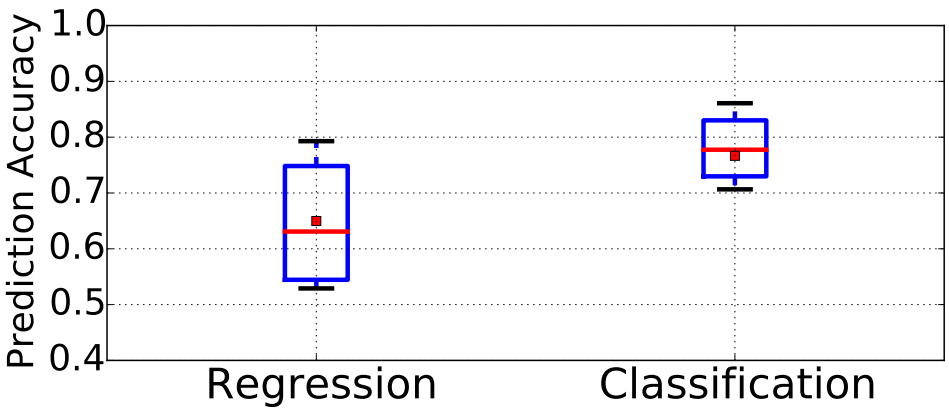


Figure 6.2 On the model selection of predicting the next step. We evaluate the ability to distinguish a good and a bad configuration. In regression, we test rank preserving as prediction accuracy [86].

improve search performance, it prevents SCOUT from transferring knowledge from historical optimization data.

6.3.3 Search Hints

This section explains how to extract search hints using probabilistic classification methods [46, 136, 137]. A probabilistic classifier predicts probability distribution over predicted classes. In Equation 6.3, the learning task ranks the relative performance measure of two architectural configurations. We can map ranks to classes. For example, the rank function σ outputs *class 1* if $\frac{y_j}{y_i} < 1.1$. Otherwise, the output is *class 0*. A binary classifier is able to answer "is x_j better than x_i?" A probabilistic classifier outputs higher probability for *class 1* if a specific workload performs better on x_j than x_i.

We use an example to illustrate this process. Consider a configuration space ($S = \{S_1, S_2, S_3, S_4\}$). An CAT optimizer starts with S_1 and $y_1 = \phi(S_1) = 10$ (the current best). Let us assume we have collected historical data for training the probabilistic classifier. The predicted probability distribution over S_1, S_2, S_3, S_4 is $[-, 0.8, 0.2, 0.2]$. The probability vector P_i represents "how likely S_j is a better choice than S_i." A higher value indicates S_j is more likely to perform better than S_i. In this example, S_2 is a better choice than the others. When the actual performance measure is better $\phi(S_2) \leq \phi(S_1)$, then CAT optimizer found a better solution than the current best.

The above example uses a binary classifier, and SCOUT uses multiclass classification. The intuition for using multiple classes is that some configurations yield similar performance, and SCOUT should not consider little improvement. For example, we can define the classes to be

"better," "fair," and "worse." SCOUT favors the configurations in the "better" class. SCOUT uses a predefined discretization policy (based on user-defined thresholds) to convert probability to discrete classes.

6.3.4 Search Strategy

During the search process, a new observation (running a workload on a selected cloud configuration) provides the necessary information to determine whether there exist better choices (see Equation 6.3). The probability vector P_i is derived for each new observation $\phi(S_i)$. A search strategy determines the choice based on the probability predictions. At each step, the search process selects the configuration S_j with the maximum probability in P_i.

This search strategy is similar to depth-first search. While *CherryPick* requires balancing exploration and exploitation, SCOUT tends to exploit—because it uses historical data. When the prediction model can generate quality predictions, this search strategy leads to quick convergence speed (the selected configuration improves over the current best). Therefore, the search process has low search cost to find near-optimal configurations.

A search process should stop when it no longer can find a better configuration. This is controlled by a predefined parameter called *probability threshold* (α) and acts as a stopping criterion. When the predicted probability P_{ij} is lower than α for all S_j, the search process is not confident that it would find better configurations in the next step. A search should also stop if it fails to find better solutions due to an inaccurate performance model. This is controlled by another parameter called *misprediction tolerance* (β) to avoid excessive search cost.

6.3.5 Putting It All Together

We have shown that the core element of a search based method is to determine the next best step. For obtaining hints to guide a search process, we propose using the probabilistic classification technique to predict improvement probability. That is similar to Expected Improvement (EI) in *CherryPick*. We choose pairwise comparison and relative ordering to deliver high prediction accuracy and to naturally into the search process. SCOUT leverages low-level performance information and extracts rules (based on resource utilization) implicitly. This improves a search process because certain types of cloud configurations can be avoided (as we will show in Section 8.4). Last, we choose a search strategy that merely picks the configuration that is most likely to be better than the current best. This strategy increases convergence speed—has low search cost.

6.4 Implementation

This section describes the implementation of SCOUT as shown in Figure 7.5. We implement the system in Python with scikit-learn [106] for machine learning libraries and Boto 3 [19] for interacting with Amazon web services.

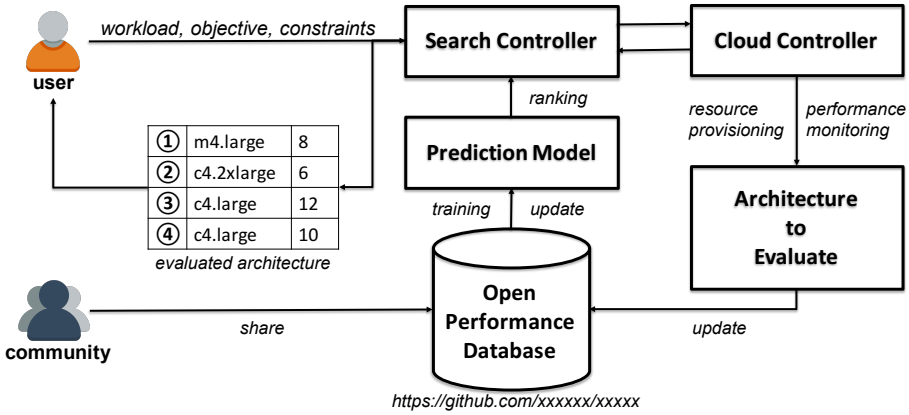


Figure 6.3 SCOUT**'s implementation.**

- **Search Controller.** This is the entry point to use SCOUT. A user submits a workload along with performance and cost objectives. The user can also specify constraints such as maximum execution time, maximum cost budget and the range of cluster sizes, etc. The performance of SCOUT is not heavily affected by the initial architectural configurations. For evaluations, SCOUT randomly selects one configuration. The search controller forwards the workload and the selected configuration to the cloud controller. Once the selected configuration is evaluated, the search controller is notified and determines the next configuration to evaluate. SCOUT selects the configuration that, according to the model, has the greatest likelihood of improving performance. This optimization process stops when the objective is fulfilled or when the evaluation budget is expended.

- **Prediction Model.** We can use a probabilistic classifier to derive the probability distribution over prediction classes [46]. Possible choices include, but are not limited to, Logistic Regression, Gaussian Process, Random Forests and neural networks [22, 46]. SCOUT uses *ExtraTree* to train a classification model for deriving relative ordering (to determine better configurations) because *ExtraTree* is more efficient and reliable with numerical performance data [48].

- **Cloud Controller.** The cloud controller bridges the search controller and cloud platforms. The controller is responsible for running the workload with a specific architectural configuration. Scout currently supports AWS but easily can be extended to other cloud platforms. For each evaluation, the cloud controller monitors the execution time and collects low-level performance information. We use the *Sysstat* package on Linux for system monitoring [117]. Since we collect only generic performance metrics, other monitoring tools should work as well. We choose a five-second sampling period. Each sample has 72 metrics from CPU, memory, cache, disk and network components. These numbers are then aggregated from various-size samples (per node) and nodes (per cluster). We follow the similar process of feature transformation as described in [60, 62].

- **Open Performance Database.** Performance data is hard to find. We believe sharing data can greatly advance the research on CAT. Moreover, cloud users benefit from the performance database because they are able to learn the workload performance on different architecture configurations, which are very expensive to obtain.

6.5 Evaluation

We evaluate Scout with three sets of big data analytics applications on 18 cloud configurations on a single-node. We further evaluate 69 cloud configuration on multiple nodes. Our evaluations show that Scout finds the optimal or near-optimal configuration more often than other methods and does so while reducing search costs.

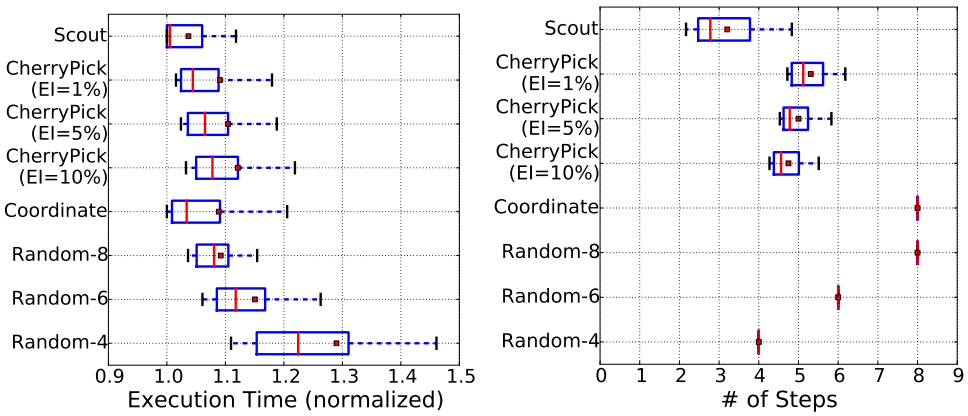
6.5.1 Experiment Setup

Workloads

We choose diverse workloads (CPU-intensive, memory-heavy, IO-intensive and network-intensive) such as PageRank, sorting, recommendation, classification and online analytical processing (OLAP). We also change the input parameters and data sizes to create a wide spectrum of workloads. These workloads run on Apache Hadoop [8] and two separate versions of Apache Spark [9] (version 1.5 and 2.1).

Deployment Choices

Our evaluation examines both single- and multiple-node settings. The single-node setting serves a comparison baseline and allows us to test more workloads (due to smaller search space). In the single node setting, we choose 18 distinct instance types or cloud configurations and 107 workloads. When evaluating different cluster sizes, we use strong scaling—fixed problem



(a) Search Performance **(b)** Search Cost

Figure 6.4 Minimizing Execution Time. The *x-axis* represents the normalized performance (to the optimal configuration), and the optimal performance is 1. SCOUT finds the near-optimal solutions (< 1.1) in 87% workloads while using much fewer steps.

size—because we are interested in how to speed up a workload rather than the efficiency of the cluster. For the multiple-node setting, we run 18 workloads on 69 cloud configurations (9 instances with various cluster sizes).

Dataset

In order to verify the effectiveness of SCOUT, we create a comprehensive performance data conversing all the combinations of workloads and architectural configurations. We collected our data on AWS. Table 4.2 present a summary. Please refer to the open performance database for further details in Section 4.2.

Parameters

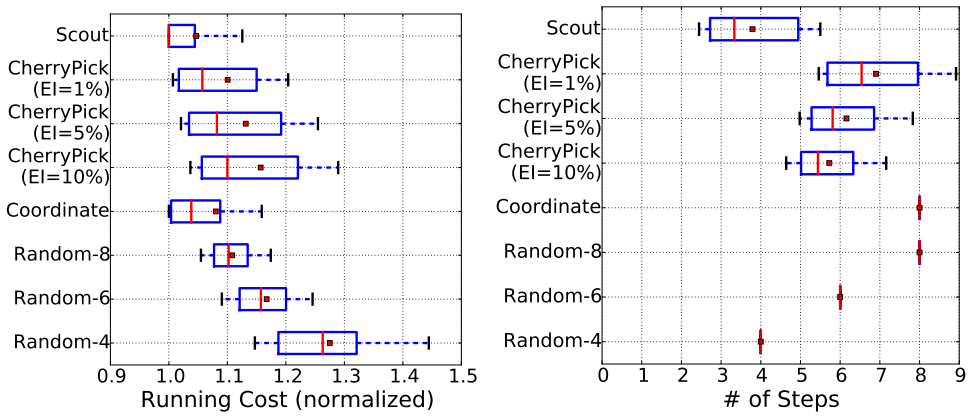
SCOUT has three important parameters: 1) labeled classes, 2) probability thresholds and 3) misprediction tolerance. For the labeled classes in classification modeling, we define five classes, "better+", "better", "fair", "worse" and "worse+", using thresholds [0.8, 0.95, 1.05, 1.2] as the cut points. Regarding the two stopping criteria, we choose 0.5 for the probability threshold and 3 and 4 for the misprediction tolerance in the single-node and multiple-node setting respectively. We examine the trade-off of these parameters in Section 6.6.

6.5.2 Comparison Method

To evaluate SCOUT, we examine the search performance in terms of *effectiveness*, *efficiency* and *reliability*. We compare SCOUT with random search, coordinate descent, and *CherryPick*.

Random search

This search method uniformly samples the configuration space. The stopping criterion is the number of configurations to evaluate. A higher number yields better solution but also incurs higher search cost. For a fair comparison, the search is repeated 100 times. Random-4, -6, -8 represent random samples of 4, 6, and 8 cloud configurations respectively. It serves as a naïve baseline method.



(a) Search Performance **(b)** Search Cost

Figure 6.5 Minimizing Running Cost. Searching for the optimal cost is more difficult because the search cost is higher than the scenario of minimizing execution time. SCOUT still finds near-optimal solutions with a small increase in search cost while *CherryPick* only finds near-optimal solutions in about 50% workloads.

Coordinate descent

This method searches one dimension (*e.g.*,CPU type and memory size) at a time. It determines the best choice of the dimension and continues to choose the best from other dimensions. This approach may suffer from local minimum due to diminishing return and irregular performance outcome [3]. This situation worsens when the number of dimension increases. In the evaluation, there are three dimensions: (1) the instance family (such as *c4* or *r4*), (2) the instance size (such as, *large* or *2xlarge*), and (3) the cluster size (the number of VMs). The results are from 100 distinct searches, in which the starting point was randomly selected.

CherryPick

We implement the approach proposed in *CherryPick* [3]. We use the same kernel function (*Matérn 5/2*) and the same stopping criteria (**EI**=10%). We uniformly sample three configurations as starting points. Since the search performance of *CherryPick* is highly dependent on the selection of the starting points, this experiment repeats 100 times to reduce artifacts and give a better picture of *CherryPick*'s capability.

We compare these approaches using three metrics. First, we evaluate the effectiveness of the methods using the *normalized performance* (to the optimal choice). It can be the *execution time* or the *deployment cost*. Second, we use the search cost— the number of cloud configurations measured to find the right cloud configuration. Last, we examine how reliable our method is across the workloads. We compare the aggregate of the normalized performance and the search cost along with their the 10^{th} and 90^{th} percentiles to observe whether our method performs well with consistency. These numbers better illustrate reliability of the methods.

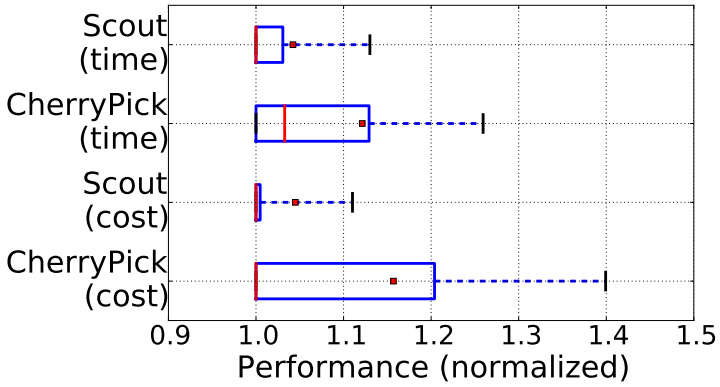


Figure 6.6 Quality of found solutions. Although both *CherryPick* and SCOUT find the near optimal-solutions in most of the time, SCOUT is less fragile.

6.5.3 Is Scout effective and efficient?

We examine search performance and search cost across 107 workloads in the single-node setting. This evaluation largely answers whether a search method is reliable.

Scout finds the near-optimal configurations (within 10% difference) for 87% workloads. Figure 6.4a and Figure 6.5a presents the best cloud configuration (normalized to the optimal performance—1.0 represent the best, higher the worse) found by SCOUT and other methods while minimizing execution time and deployment cost, respectively. Figure 6.4b

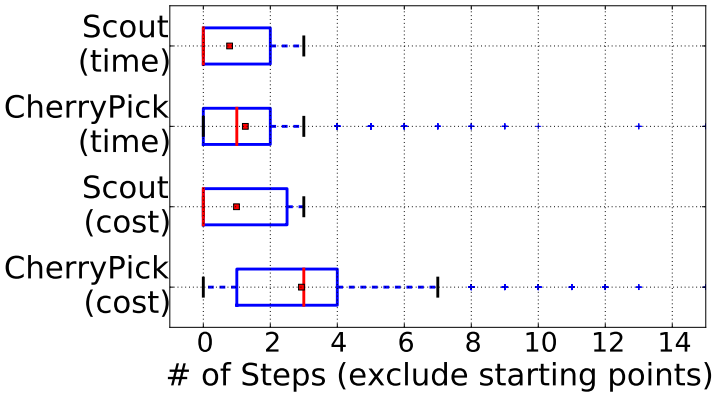


Figure 6.7 Stopping awareness. Search optimization avoids unnecessary search cost if it knows when the optimal solution is found.

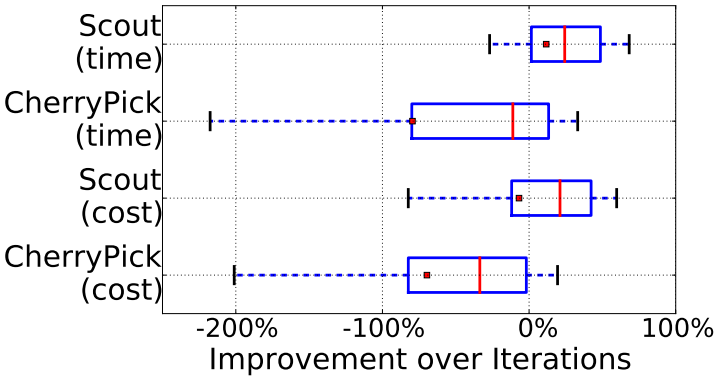


Figure 6.8 Convergence speed. SCOUT finds a better solution with 25% improvement (on average) at each iteration, which suggests SCOUT is more likely to converge.

and Figure 6.5b presents the search cost required the find the best cloud configuration which minimizes execution time and deployment cost, respectively. The figures display a box plot. The box shows the inter-quartile range (from 25^{th} to 75^{th} percentile). The vertical red line is the median, and the dot is the mean. The whiskers to left and right show the 10^{th} and 90^{th} percentiles, respectively. The horizontal axis shows execution time, and the vertical axis shows different techniques. An ideal search-based technique would find the best cloud configuration (in terms of performance) using the lower search cost. These figures show the following.

- Scout finds the best relative performance in terms of both execution time and deployment cost. The median performance of Scout, while searching for the cloud configuration which minimizes the deployment cost is 1.0, which means Scout was able to find the right cloud configuration.
- Scout is better than CherryPick across all measures (execution time, deployment cost, and search cost).
- Scout finds the best relative performance using the least search cost (fewer number of steps). Random-4 also requires low search cost, but its performance is much worse than Scout.
- The variance in the performance (in terms of execution time and deployment cost) of Scout is much lower than the other methods. The large variance of the Random methods can be attributed to their inherent randomness.

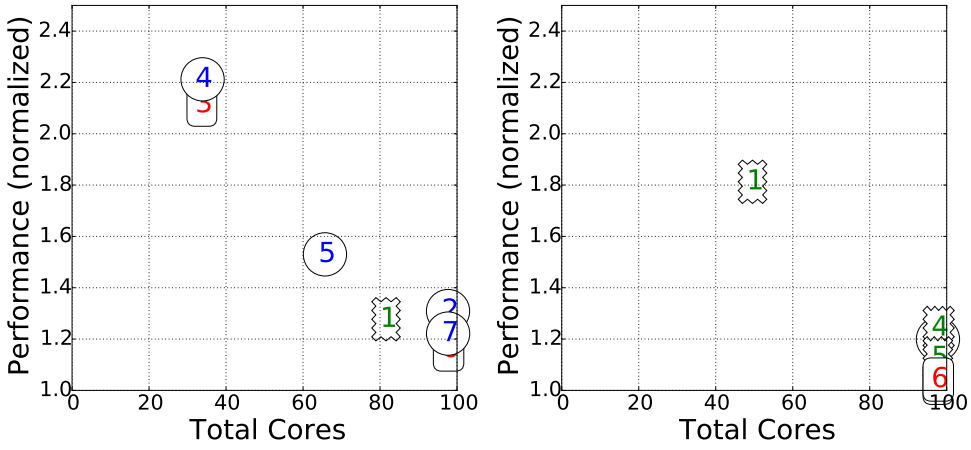
Overall, we see that Scout is that best performing method and *CherryPick*, the state of the art method, only delivers similar performance in 64% workloads while requiring 47% greater search cost (4.7 compared to 3.2 steps). We also observe that the variance in the best cloud configuration found by Scout over 100 runs across 107 workloads is much lower than the other method. Hence, we can conclude that Scout is a reliable method to find the best cloud configuration.

Cost creates a level playing field. Optimizing execution time is relatively easy because a larger, more powerful instance type is more likely to have a shorter execution time. However, the more powerful types are more expensive to execute. Consequently, a smaller instance type may run longer but cost less. Because the cost to execute an instance grow as the raw hardware performance increase, the differences in deployment cost between configurations tend to be much less than the differences between execution time. This levels the playing field for cost—many more configurations are good candidates. This leveling leads to, in general, longer searches, as shown in Figure 6.5b. *CherryPick* requires one extra step in optimizing deployment cost with a 15% decrease of workloads in which it fails to find a solution within 10% of the optimal configuration. To summarize, the performance of a search-based method is dependent on the

objective of the search. From the data, we observe that searching for the best cloud configuration in terms of cost is more challenging than finding the best cloud configurations in terms of execution time.

6.5.4 Is SCOUT reliable?

Users are willing to use a tool only when it is reliable. We evaluate the performance of *CherryPick* and SCOUT with different initial points for understanding their consistency. In BO in *CherryPick*, uses a random initial points to seed the search process and the effectiveness of CherryPick depend on these initial points. Selecting these initial points is non-trivial because (1) a good set of starting points for one workload does not work for other workloads, and (2) cloud providers frequently upgrade their instance portfolio with new instance types which make the process of selecting initial points more challenging. SCOUT is robust such that the effectiveness of SCOUT does not rely on initial points.



(a) *CherryPick* **(b)** SCOUT

Figure 6.9 Finding the fastest configuration for PageRank on Hadoop. Left & right sub-figure show the search path of CherryPick and SCOUT respectively. SCOUT identifies PageRank as a compute-intensive workload. It chooses the configurations with higher core counts and CPU speed.

To demonstrate the robustness of SCOUT, for each workload, we varied the initial points used in *CherryPick*. These points were randomly (without replacement) selected from the search space. On the other hand, SCOUT only needs one starting point, which is also selected randomly. This experiment was repeated 100 times to understand the implication of randomness. Figure 6.6 shows the variance in the normalized performance of the found solutions by both the methods. We see that

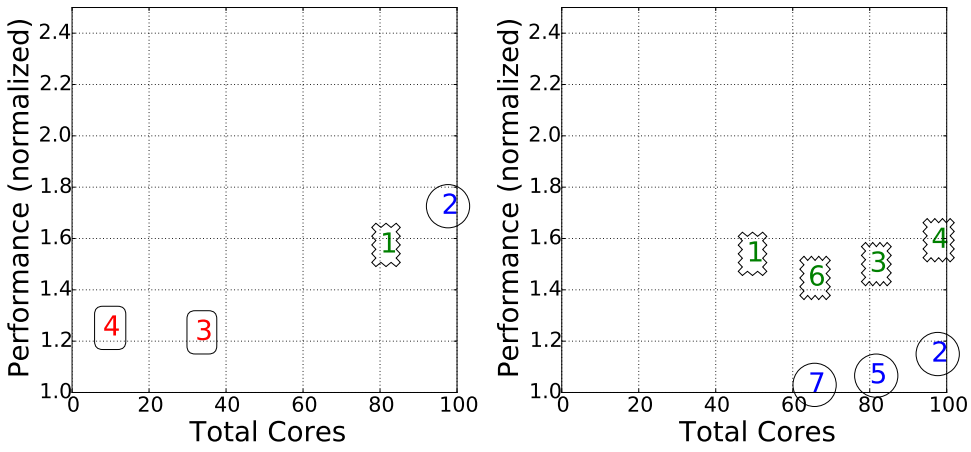
- Scout can find the optimal cloud configuration for most of the case since median performance is 1.0. However, there are some outliers which pushes the mean to 1.05. This is not a major concern since the 75^{th} percentile is less than 1.05. This goes to show that the variance in the performance of 107 workloads aggregated over 100 runs is low.
- CherryPick is also effective in finding the cloud configuration since its median performance over 107 workloads is 1.05. We notice that the variance of the performance (both in terms of search performance and time) is larger than Scout.

The variance in the results of CherryPick can be a major concern for the practitioners since a bad choice of initial points can lead to selecting either a slow or expensive configurations. Scout, on the other hand, has more stable search performance regardless of the starting point.

6.5.5 Why Scout works better?

Scout relies on quality routing policy to deliver good solutions. We find Scout effective because it knows when to stop searching and converges to better solutions.

- Scout **knows when to stop.** When an optimizer can stop as soon as it finds the optimal solution (or near-optimal solutions), it can avoid unnecessary search efforts. Figure 6.7 shows that Scout requires a fewer number of steps if the starting point is already the optimal configuration.
- **Convergence speed.** The speed of convergence of a search-based method is dependent how it selects the next cloud configuration to measure. An ideal search-based method will always find the next cloud configuration, which is better than the cloud configurations sampled previously. *Converge speed* can be defined as the average difference between the performance score (execution time or deployment cost) of the previous measurement (i^{th} step) and the current measurement ($i+1^th$ step). A positive number would indicate that the current cloud configuration is better than the previous measurement (for both deployment cost and execution time, lower is better). Figure 6.8 compares the convergence speed of CherryPick and Scout. Figure 6.8 indicates that Scout overall finds cloud configurations 50% (median) better execution time than the current cloud configuration, whereas CherryPick overall moves to cloud configuration which is 25% worse than the current best configuration. Similar behavior is seen for deployment cost. This is evidence to show that Scout uses the historical data to find the promising region in the search space and exploits that space effectively.



(a) *CherryPick* **(b)** Scout

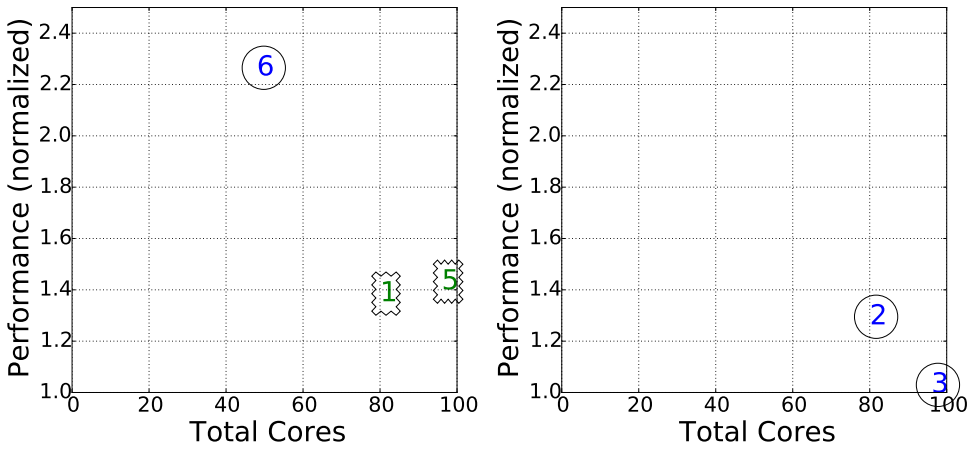
Figure 6.10 Minimizing the running cost for Naive-Bayes on Spark. This is a memory-intensive workload. Scout does not even try the *c4* family due to its small memory per core.

6.5.6 Example Search Process

This section compares and contrasts the properties of *CherryPick* and Scout. We provides four examples of optimizing execution time (in Figure 6.9 and 6.11) and running cost (in Figure 6.12 and 6.9). Different colored markers in the graphs represent different families of instances: green represent the m4 family—general purpose, blue represent the r4 family—memory optimized, and red represent the c4 family—compute optimized. We evaluate *CherryPick* and Scout on four representative workloads, selected based on diverse resource requirements (CPU intensive, Memory intensive). For *CherryPick*, we choose *20×m4.xlarge*, *48×r4.large* and *16×c4.large* as the starting points because they are wide spread in the search space. Since Scout only needs one starting point, we choose *24×m4.large* because it is the mid point of the search space. We observe that *CherryPick* can find near-optimal solutions for few workloads if not all.

Reliable exploration is difficult and generates high search cost

In Figure 6.11, 6.12, 6.9, 6.10, we observe that the search path generated by *CherryPick* involves more distinct VM types due to the need to explore the performance model. For example, in Figure 6.12, CherryPick visits each instance family at once in all examples while Scout skips some specific families. This is because Scout builds the performance model from historical data. Hence, it requires only little (or no) exploration. This phenomenon, exploration-exploitation dilemma, is studied extensively in Machine learning [68]. The cold-start issue (as described in Section 7.2 arises partly because of the requirement to explore the configuration space since Scout learns the performance behavior from historical data from workloads (previously



(a) *CherryPick* **(b)** Scout

Figure 6.11 Minimizing execution time of Regression on Spark. Since the Regression workload requires both computation and large memory, Scout directly chooses configurations with the *r4* family and larger cores.

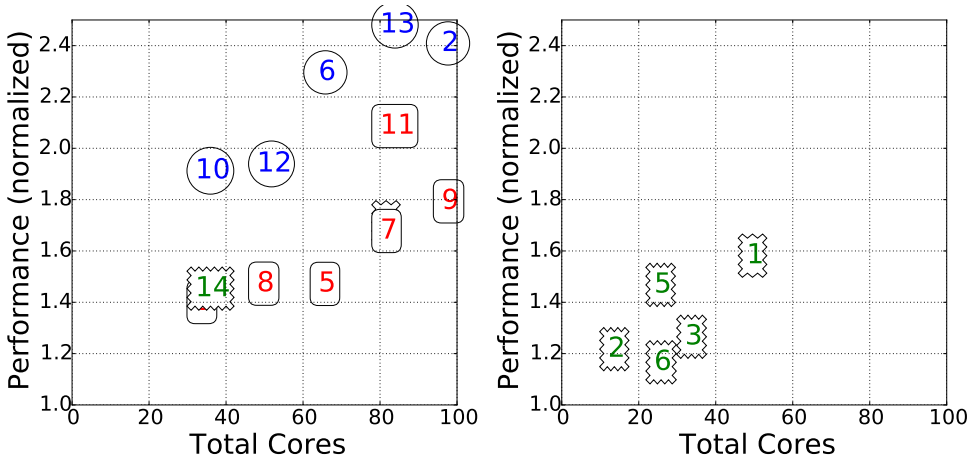
explored) can sidestep the need to explore the search space.

Fragility of CherryPick

As explained in Section 7.2, CherryPick is fragile because it is sensitive to its parameters and the starting points. In the four examples, *CherryPick* starts from the same three configurations; however, the results are very different. In Figure 6.12, *CherryPick* fails to characterize the search space, which results in long search path (and high search cost). While in Figure 6.10, *CherryPick* stops too early and only finds a local minima (the *c*4 family). These two examples show that *CherryPick* is fragile and therefore, its search performance is not stable.

Scout identifies resource requirements

When resource requirements can be articulated, a search process is more likely to find cloud configurations effectively and efficiently. In Figure 6.9, the *PageRank* workload runs faster on a larger cluster (higher core counts) and higher-frequency CPUs. The *r4* family, with larger memory but slower CPU speed, does not seem to be the best choice, hence avoided by Scout and instead prefers *c4* and *m4* family. This tendency is more clear in the other cases as well (Figure 6.11, 6.12, and 6.10).



(a) *CherryPick* **(b)** Scout

Figure 6.12 Finding the cheapest configuration for Terasort on Hadoop. The Terasort workload requires enough memory to avoid spilling data to disks. Besides, a large cluster can be insufficient due to the shuffle phase in MapReduce. Scout chooses a smaller cluster with the general-purpose VM type.

Scout captures the complex cost model

In a real-world setting, practitioners can choose either a smaller cluster built using more powerful instances or choose large cluster built using smaller or less powerful instances (a scale-out and a scale-out configuration). The performance model used by Scout can infer the size of the cluster of the best cloud configuration. In Figure 6.12, Scout chooses to run *TeraSort* on a smaller cluster to save cost. On the contrary, in Figure 6.10, Scout selects a larger cluster for efficiently running the *Naive-Bayes* workload while achieving lower cost. These two examples show that Scout captures the complex relationship between the resource metrics and the running cost.

Summary

The main difference between *CherryPick* and Scout lies how the method explores the space of possible cloud configuration options. We can see that *CherryPick* has to explore more cloud configuration options and hence have higher search cost (longer search path) while Scout searches within a relatively restricted region. This feature of Scout can be attributed to its performance model, which learns from the historical data. This also goes to show that encoding scheme, which uses low-level performance metrics, is successful in transferring knowledge from one workload to another.

6.6 Discussion

Tuning Searching Performance

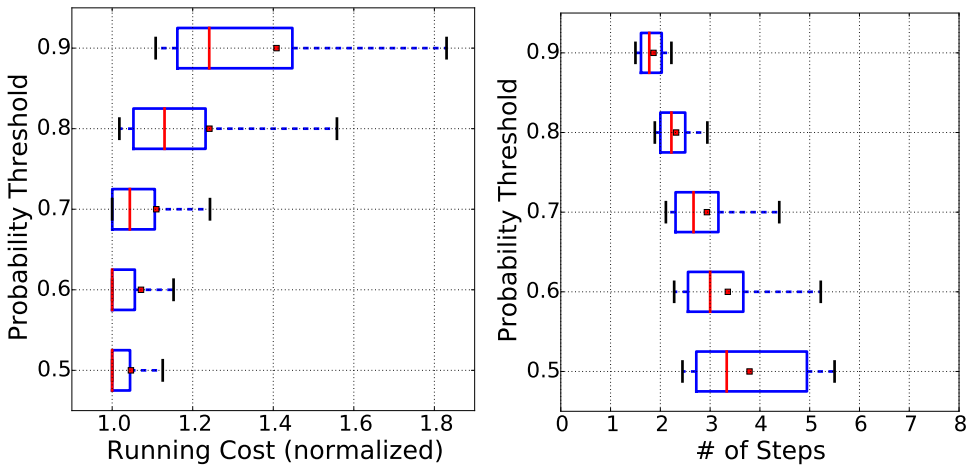
SCOUT uses "probability threshold" and "misprediction tolerance" as stopping criteria.

Probability Threshold

SCOUT chooses the next configuration to evaluate based on the probability of improvement and stops when the probability is lower than the probability threshold α. Figure 6.13 shows that a higher probability threshold is pessimistic and terminates the search process prematurely, hence, shorter search path and unstable search results). The probability threshold presents a trade-off between search performance and search cost. The right threshold must consider the reliability curve of classification methods [88].

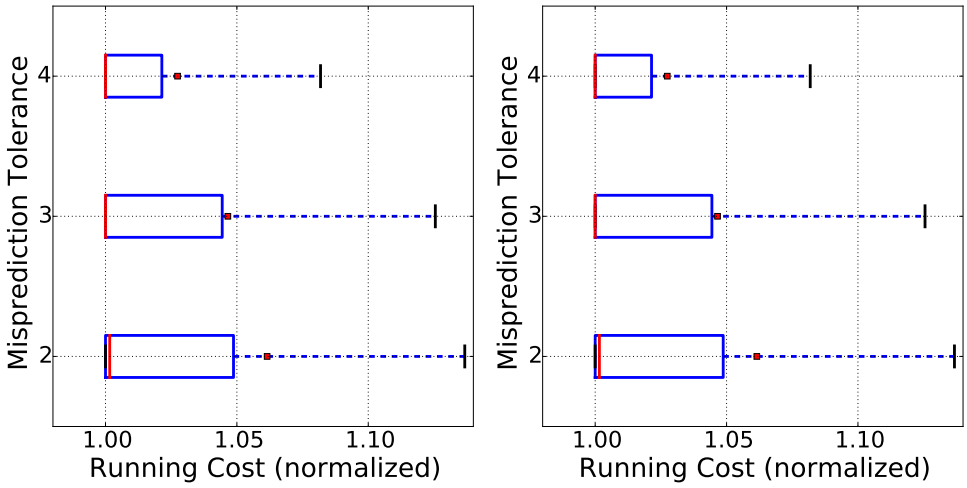
Misprediction Tolerance

SCOUT terminates the search process if the selected configurations do not improve the current best choice (considered as a misprediction). SCOUT maintains a counter of mispredictions. A higher limit tolerates more mispredictions but yields better search performance due to more chances. A proper limit should consider both the size of search space and the accuracy of prediction. In Figure 6.14, we show that a higher tolerance level leads to better search performance but higher search cost. This trade-off is similar to the probability threshold.



(a) Search Performance **(b)** Search Cost

Figure 6.13 Tuning the probability threshold. A smaller threshold generates a longer search path but ensures better search performance.



(a) Search Performance **(b)** Search Cost

Figure 6.14 Tuning the misprediction tolerance. A higher tolerance to mispredictions generates higher search cost.

Alternative search strategies

SCOUT generates the probability vector P_i for each new observation (running the workload on S_i). Our current search strategy only uses information from the latest observation. SCOUT stores historical observations and therefore, the next search step can be determined using several past observations. Given two observations on S_1 and S_2 and two unevaluated configuration S_3 and S_4, SCOUT generates prediction probability P_{13} and P_{14} from S_1, and P_{23} and P_{24} from S_2. Instead of choosing P_{23} after the second step, SCOUT should choose P_{14} when S_2 is much worse than S_1 (due to mispredictions). This strategy is more likely to avoid bad choices. On the other hand, SCOUT currently relies on offline performance modeling. Another alternative is to update the prediction model upon new observations. For unseen workloads, this update enables SCOUT to improve prediction accuracy. However, the downside is the cost of retraining the model. An online learning method might help reduce the retraining cost. The two possible alternatives remain as future work.

Universal Prediction Models. In prior work, the performance model needs to be retrained for every optimization process, which leads to wasted effort. There is a need for a modeling strategy, which becomes more accurate with experience. Transfer learning can be beneficial in our setting, where the performance model can predict a new workload using knowledge learned from optimization results of other workloads [95]. SCOUT tries to learn from other performance data so that all the experience from the past optimization process is not lost. Figure 6.15 shows

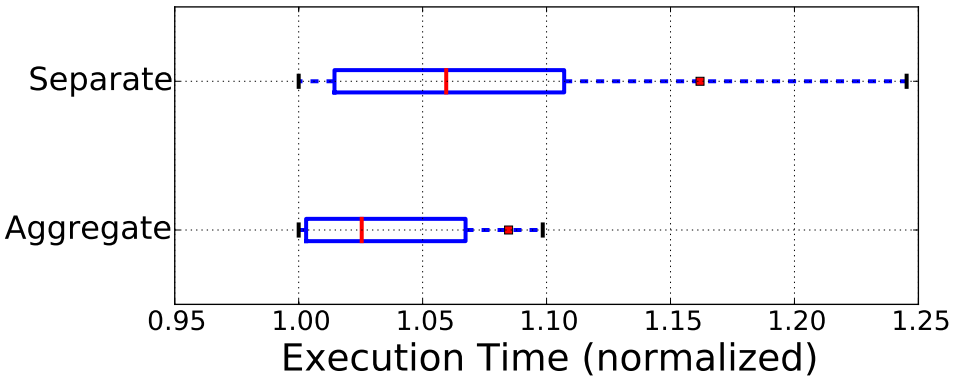


Figure 6.15 Universal performance models. Training data form multiple systems improves prediction.

how the performance model learned from more data (from different workloads) can generalize better than the performance model training for a single application. In the figure, the horizontal axis represents the execution time of the workload, and the vertical axis shows two versions of Scout. *Separate* refers to the Scout which is trained with performance data from just Hadoop workloads, whereas *Aggregate* refers to Scout trained on Hadoop as well as Spark workloads. We can see that *Aggregate* can find cloud configurations with better performance (lower execution time). *Overall, the prediction model used in* Scout *is universal and can learn from any workload.*

Time-cost trade-off

Often a user is willing to wait longer for a result if there is a big reduction in cost. For example, many might be willing to trade a 20% increase execution time for a 50% decrease in running cost. This is similar to the energy-time trade-off in high-performance computing [44]. Scout can support this scenario. In our design, we define prediction classes based on the normalized performance of a single performance measure, *i.e.*,time or cost. Previous work supports this trade-off in a similar manner [62].

6.7 Conclusion

Cloud architecture tuning (CAT) is essential to maximize the performance of an application while keeping the deployment cost down. In this chapter, we identify key elements for an effective CAT method. We design and implement a novel system Scout, which delivers *efficient*, *effective* and *reliable* search performance.

Chapter 7

A Collective CAT Optimizer for Multiple Workloads

This chapter proposes a collective optimization method for a group of workloads in the CAT setting.

7.1 Introduction

Cloud computing optimizer is a device to select the best cloud configurations (such as virtual machine (VM) types and the number of VMs) for a given workload. Choosing the right cloud configuration is essential to maximize application performance and minimize operational costs. However, such optimization task is not straightforward due to opaque resource requirement [60, 133]. To address this challenge, prior work either builds prediction models (as in *Ernest* [123] and *PARIS* [133]) or uses sequential model-based optimization (as in *CherryPick* [3]). We also choose sequential model-based optimization as in *Arrow* [62] (see Chapter 5) and as in *Scout* [61] (see Chapter 6).

While they are effective, they are only designed for a single workload. In practice, it is rare to migrate only one workload [70, 116]. Since these optimizers are expensive to run, applying them independently to workloads requires significant measurement cost and long optimization process. In this chapter, we optimize a batch of workloads altogether.

This kind of collective optimization is impossible if workloads execute very differently on different cloud configurations. Prior work reports there does not exist an one-size-fits-all VM type that is best for all workloads [3, 62, 133]. However, while analyzing the data from our large empirical study involving three different software systems and over 100 workloads, we noticed that there does exist at least a cloud configuration (*e.g.*,m4.large), which performs satisfactorily for the majority of workloads. If the above is prevalent in cloud computing, it should be possible

to simplify collective optimization. In this chapter, we exploit this phenomenon in order to further reduce optimization cost.

We call such a cloud configuration *Exemplar Configuration*, which is near-optimal or satisfactory for the majority of workloads. In our empirical study, the exemplar configuration is only 5-20% slower or more expensive than the optimal choice. In any cloud optimizer, there exists a trade-off between search performance (how far a choice is from the optimal) and measurement cost (how many tests an optimizer requires to find a suitable configuration). With the exemplar configuration, we can trade a slight decrease in search performance for a large reduction in measurement cost because redundant efforts can be reduced in collective optimization. When optimizing a group of workloads, such trade-off not only brings significant cost reduction but also shortens the optimization process as well as the migration procedure. However, finding such an exemplar configuration is not straightforward because it depends on workloads and performance objectives. Moreover, as cloud providers expand their cloud portfolio, the exemplar configuration is also likely to change. In this chapter, we focus on finding out this exemplar configuration efficiently.

To this end, we propose and evaluate a collective optimization method, *Micky*[1], which enables users to deploy a group (not one) of workloads to the cloud more efficiently (lower measurement cost). We reformulate "finding the exemplar configuration" as the multi-armed bandit problem [11, 15, 35, 67, 127]. The two problems are similar because the bandit problem aims to maximize rewards (*Micky*, for example, minimizes execution time or operational cost) in a series of decisions (to run a workload on a cloud configuration), each is associated with an unknown payoff and a known opportunity loss (whether the decision meets the performance objective). Our evaluation shows that *Micky* can find the exemplar configuration using only 12% of the total effort compared to a sophisticated single-optimizer. This cooperative style of search methods ensures that users do not need to optimize each workload separately; instead, finds the exemplar cloud configuration collectively, thereby reducing measurement cost.

Micky finds a configuration that is near-optimal for the majority of workloads. But the chosen configuration could perform unacceptably for some workloads. To remedy this issue, we integrate our previously built system Scout to identify sub-optimal cases [61]. This enables elaborate optimization for unsatisfactory workloads if strict performance is required.

We demonstrate the effectiveness of *Micky* by evaluating it on 107 real-world workloads (using three popular software systems) and show that *Micky* can find near-optimal cloud configurations by using only a fraction (12%) of the measurement cost used by the state of the art methods, at the expense of less optimal choices. There is always a trade-off between search performance and measurement cost. Based on our evaluation, we advise users not to use *Micky* only when

[1]Micky (Rosa) is a character, from the Hollywood movie 21, who founded the MIT Black Jack team of card counters.

the same workloads will repeat more than tens of runs (*i.e.*,30 times using our analysis) . To deploy a batch of workloads to cloud, we believe *Micky* is more desirable than state-of-the-art methods because the higher number of recurrence would certainly limit the applicability of cloud optimization. Furthermore, those sub-optimal choices can be eliminated through the integration between *Micky* and Scout, thereby creating a more robust solution.

7.2 Why Collective Optimization

A cloud optimizer is often evaluated with search performance and measurement cost.

- **Search performance** is the measure of the quality of the found solutions by an optimizer. For example, in searching for the most cost-effective configuration, an optimizer that finds a configuration that is only 10% more expensive than the optimal is considered better than another optimizer that can only find a configuration that yields 30% more cost. Therefore, we use normalized performance (to the optimal) for evaluation.

- **Search cost** is the total cost of running an optimizer. An optimization process is expensive because it requires to test a workload on some cloud configurations for deriving the best choice. We use the number of tests as the search cost (or measurement cost) because it is an intuitive measure. The amount of charge is another measure [3].

There is always a trade-off between measurement cost and search performance. The primary motivation for collective optimization is to reduce high measurement cost of optimizing multiple workloads. If users demand strict search performance, they better turn to single-optimizers. However, we argue that collective optimization is promising because it achieves comparable or slightly worse search performance while reducing measurement cost significantly. In the following, we discuss the benefits of having a collective optimizer.

- **Large scale cloud migration.** Cloud computing is a cost-effective solution. Enterprises are moving in-house applications to the cloud, and need a quick way for large migration [70, 116]. Elaborate optimizers are expensive (in measurement cost) and time-consuming (in optimization process).

- **Limited budgets.** The single-optimizer such as *CherryPick* and *Scout* are effective and desirable for highly recurring workloads because the measurement cost can be amortized. However, the number of budgets to run optimizers does not increase linearly with the number of workloads. To better support multiple workloads, we need to reduce measurement cost while delivering comparable search performance.

- **Expanding cloud portfolio.** Cloud providers expand their cloud portfolio more than 20 times in a year [12]. Therefore, users have to rerun optimizers to update their configurations for all workloads. Again, this is an expensive and time-consuming process.

- **Seed cloud optimizers.** All the cloud optimizers require initial measurements. It is unclear how to determine the best starting points. We aim to find the exemplar configurations, which can be used as the starting points, thereby reducing measurement cost. The exemplar configuration can be used to seed singe-optimizers such as CherryPick and Scout, which will be discussed more in Section 7.5.

In summary, users would prefer collective optimization if search performance is comparable to single-optimizers while measurement cost can be reduced greatly.

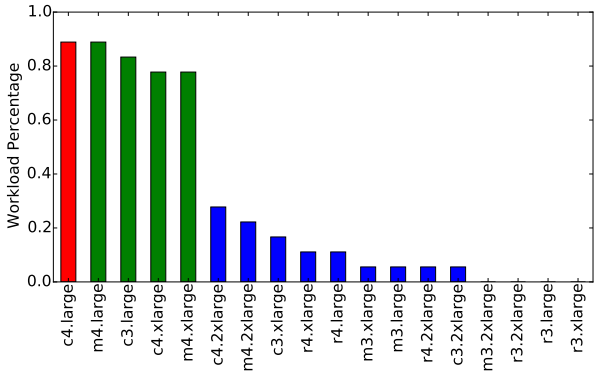
7.3 Finding the Exemplar Cloud Configuration

In this section, we first present our empirical study on investigating the potential of finding the exemplar cloud configuration. We then formulate "finding the exemplar configuration in the cloud" as the multi-armed bandit problem. Finally, we discuss the heuristics to derive the exemplar configuration.

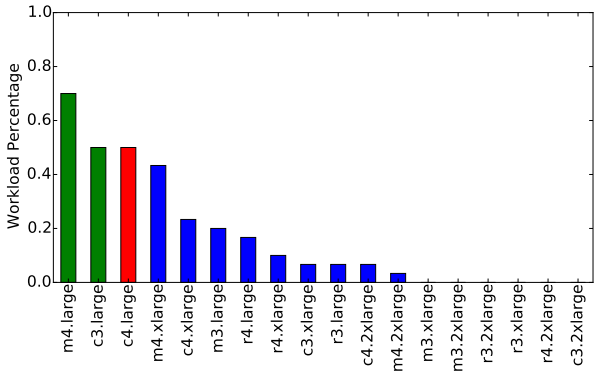
7.3.1 Empirical Study

We choose three popular software systems for cloud applications, namely Apache Hadoop 2.7, Spark 2.1 and Spark 1.5. This study includes 30 applications for diversification. They are data processing, OLAP queries, common statistics functions, and popular machine learning algorithms. Although they do not cover all the spectrum of real-world applications, they are representative of many nowadays cloud applications. When the input to applications changes, the workload behavior changes accordingly [34, 123]. We also choose three different input parameters and data sizes for each application. In total, our evaluation includes 107 workloads.

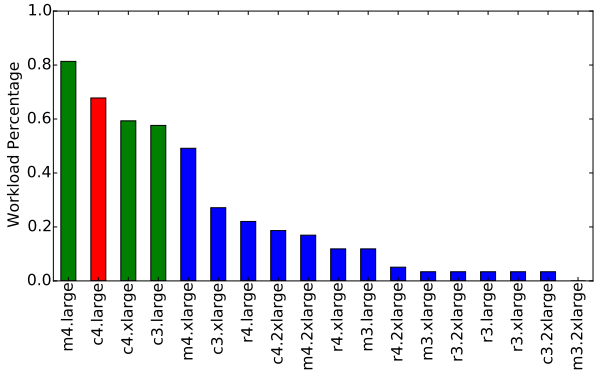
We conduct our evaluation on AWS EC2 [4]. Regarding the VM to run the workloads, we choose 18 different VM types. They include three instance families: 1) compute-optimized instances (*c3* and *c4*), 2) memory-optimized instances (*r3* and *r4*), and 3) general-purpose instances (*m3* and *m4*). For each instance family, we choose *large*, *xlarge* and *2xlarge* for the instance size. Although we only evaluate 21% VM types (AWS supports 85 kinds as of January in 2018), they reflect many use cases on AWS EC2. Besides, some VM types are designed for acceleration using GPU and FPGA, and therefore, they are less common and not included. Furthermore, it is reported that VM types with lower than 8 cores dominate VM useage on Azure [30]. We try our best to reflect the common cloud deployment. More details regrading data



(a) Hadoop 2.7



(b) Spark 2.1



(c) Spark 1.5

Figure 7.1 Opportunity to find the exemplar VM instances across workloads for reducing operational cost. The *y-axis* represents the percentage of workloads (out of 107 in three systems) that are within 30% difference with the optimal performance. The colored bars are VM types that considered the exemplar configurations for the majority of workloads (>= 50%). The red bar represents that the VM type is more likely to be the optimal choice.

collection can be found in our previous work [61, 62]. We also made our data public available for further research [92]. Please refer Section 4.2 in more detail.

7.3.2 The Exemplar Configurations

The exemplar configurations are configurations that are near-optimal or satisfactory in the majority of workloads. When the percentage is large, we can exploit the exemplar configurations to simplify collective optimization. In Figure 7.1, we present the opportunity of exploiting such configurations. We count the number of the normalized performance that is within 30% of the optimal. The colored bars are possible exemplars because they are satisfactory at least in half of the workloads. The red bar represents the VM type that is more likely to be the optimal than other configurations. These figures show there exist several exemplar configurations.

In Table 7.1, we give snippets of measurement data to better illustrate the exemplar configurations. This table presents the normalized performance of workloads on some of the VM types. A 1.0 number indicates the VM type is the optimal choice for the corresponding workload while a larger number implies a sub-optimal choice. We can observe that *c4.large* is the optimal configuration for 18 workloads (out of 35). However, it is also a sub-optimal VM type (> 1.4) in 11 workloads, which generating 1.72 normalized performance on average. On the other hand, *m4.large* seems to be a better choice because it delivers 1.45 performance on average and creates only 5 sub-optimal workloads. The above gives one way to select the exemplar configuration and in the following, we describe the challenges of selecting the exemplar.

- **Varying workloads.** In Table 7.1, we show five possible exemplar configurations for those particular workloads. The exemplars very in different sets of workloads. For example, *c4.large* is the best choice in Hadoop 2.7 while *m4.large* should be selected as the exemplar VM type in Spark 2.1.

- **Expanding cloud portfolio.** As mentioned before, cloud providers introduce new VM types regularly, which includes performance boost and price adjustment. The exemplar configurations might also change accordingly.

- **Online discovery.** We present an offline analysis of measurement data above. However, finding the exemplar configuration is an online task (for unknown workloads), which is considered a difficult learning problem. This is similar to the exploration-exploitation dilemma [68].

From the above, it would appear that there exist exemplar configurations in real-world workloads. Note that if the exemplar configuration is prevalent, it should be possible to simplify collective optimization as follows: finding the exemplar configuration instead of finding the

Table 7.1 Normalized performance on a selected group of VM types and workloads. The number 1.0 represents the optimal choice across the 18 VM types for the particular workload.

System	Workload	c3.large	c4.large	c4.xlarge	m4.large	m4.xlarge
Hadoop 2.7	aggregation	1.26	1.00	1.12	1.12	1.29
	join	1.26	1.00	1.09	1.17	1.26
	scan	1.16	1.00	1.70	1.15	1.89
	sort	1.10	1.00	1.06	1.03	1.11
	terasort	1.31	1.00	1.16	1.07	1.12
	pagerank	1.24	1.03	1.16	1.05	1.00
Spark 2.2	join	1.12	1.00	1.40	1.12	1.23
	scan	1.13	1.00	1.48	1.03	1.59
	sort	1.11	1.00	1.42	1.13	1.40
	terasort	1.30	1.19	1.66	1.34	1.46
	wordcount	1.83	1.64	1.23	1.00	1.08
	als	1.00	1.67	3.19	1.46	2.72
	aggregation	1.30	2.00	1.08	1.00	1.18
	pagerank	2.33	2.12	1.00	1.31	2.10
	bayes	3.15	3.57	1.00	1.60	1.61
	lr	6.50	5.56	1.44	1.00	2.61
Spark 1.5	chi-feature	1.19	1.00	1.32	1.29	1.53
	fp-growth	1.27	1.00	1.37	1.20	1.46
	gmm	1.19	1.00	1.27	1.25	1.36
	gb-tree	1.19	1.00	1.63	1.17	1.94
	pca	1.16	1.00	1.11	1.15	1.31
	pearson	1.19	1.00	1.11	1.19	1.11
	word2vec	1.22	1.00	1.06	1.15	1.24
	spearman	1.21	1.00	1.12	1.06	1.02
	statistics	1.15	1.00	1.43	1.08	1.56
	svd	1.16	1.00	1.02	1.07	1.09
	chi-gof	1.24	1.12	1.46	1.00	1.81
	bayes	1.27	1.15	1.19	1.25	1.35
	lda	1.66	1.36	1.10	1.00	1.31
	pic	1.53	1.39	1.00	1.15	1.31
	d-tree	1.70	1.70	1.23	1.00	1.48
	als	2.23	1.86	2.89	1.00	1.27
	regression	4.03	3.60	4.06	4.42	4.70
	classification	6.11	5.41	5.70	6.07	1.00
	kmeans	6.22	5.74	3.66	3.73	1.00
# of optimal		1	18	3	7	3
Mean		1.89	1.72	1.63	**1.45**	1.53
25%		1.18	1.00	1.11	1.04	1.15
Median		1.26	1.00	1.23	1.15	1.31
75%		1.68	1.69	1.47	**1.25**	1.58

optimal choice for each of the workload. The exemplar configurations deliver near-optimal to satisfactory performance in the majority of workloads. The rest of this work is a test of that speculation.

7.3.3 Problem Formulation

Micky attempts to find the exemplar VM type ($vm^* \in VM$) for a group of workloads (W). The workload refers to a combination of an application and the data used. The performance is measured in terms of *execution time* and *operational cost*. The cloud configuration space for workload w is referred to as ($s \in S_w$), where S_w is the set of cloud configuration options for a workload w. The size of the search space is N_w cloud configurations. In our setting, the size of the cloud configuration space is same for all workloads. For a given workload w, each configuration s has a corresponding performance measure $y_{w,s} = \phi(w, s)$.

Single-optimizers such as Cherrypick [3] searches a suitable VM type for every workload w separately. The search starts with a pool of unevaluated configuration (U_w)—the specific workload has not been run on any configuration. As the search proceeds, the cloud configuration are selected from U_w and moved to the evaluated pool (E_w). The sum of the cardinalities of U_w and E_w is equal to the cardinality of S_w ($|U_w| + |E_w| = |S_w|$). The measurement cost of the search process is $C_w = |E_w|$. When optimizing a group of workloads, single-optimizers generate a total cost $C = \sum_{w \in W} |C_w|$.

Micky is a collective optimization method. We explore the exemplar VM type vm^* so that $|E_{w_1} \cup E_{w_2} \cup \cdots \cup E_{w_n}|$ is minimized while the corresponding performance measure y_{w,vm^*} is comparable to the the ones in single-optimizers.

7.3.4 The Multi-Armed Bandit Problem

To realize collective optimization, we reformulate the problem of configuration optimization as a multi-armed bandit problem [11, 15, 101, 127]. In the problem setting, an agent (gambler) sequentially searches for a slot machine (from a group of slot machines) to maximize the total reward collected in the long run. This problem is non-trivial since the agent (gambler) cannot access the true probability of winning—all learning is carried out via the means of trial-and-error and value estimation. To find the suitable slot machine, the agent needs to acquire information about arms (exploration) while simultaneously optimizing immediate rewards (exploitation). The is referred to as the exploration-exploitation dilemma [68]. Finding the better VM type for workloads naturally fits into the multi-armed bandit problem. We describe their similarities in the following.

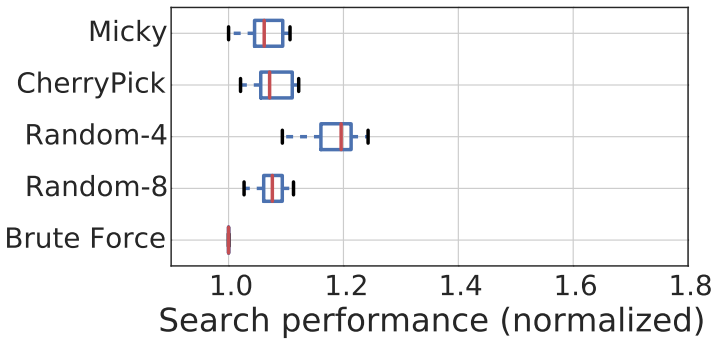
- **Slot Machine.** Each VM type is similar to a slot machine. Our objective is to find the best VM that maximizes the reward for a group of workloads.

- **Arm.** Arms are the choices of slot machines. In the cloud setting, an optimizer chooses a VM type to run a workload.
- **Pull.** A pull is one play on the slot machine. It takes coins (cost) and yields a reward. Similarly, an optimizer picks a VM type and measures the performance of a workload on the selected VM.
- **Reward.** Reward refers to the amount of money a gambler wins or loses from pulling the arms. In our setting, the reward is determined by where it meets a performance objective. We use performance delta (between the selected and the optimal choice) for calculating the reward. Please note that the optimal configuration is not known in the real-world setting.
- **Budget.** A gambler owns a certain amount to spend on the slot machines. In our setting, an optimizer requires to complete the optimization process in a limited budget. We use the number of measurements as the budget (C). In practice, the minimal budget is usually $|VM|$ and the maximum budget is $|VM| \times |W|$. The budget is determined by users. A higher budget yields a better reward.
- **Objective.** The objective of *Micky* is to find the best configuration (minimize performance delta) for multiple workloads with fewer measurements.

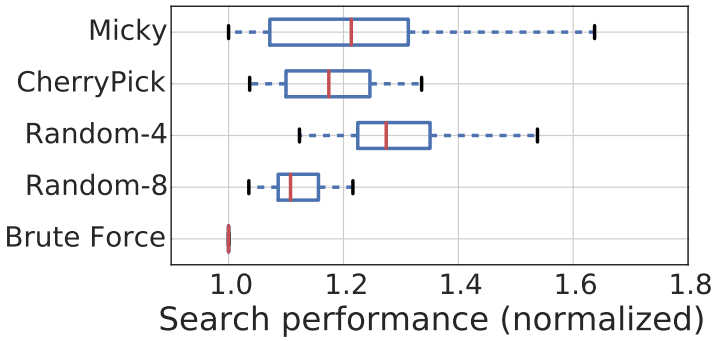
The multi-armed bandit problems have attracted attention for solving online learning problems. For example, Dambreville et al. [35] used multi-arm bandit to minimize the energy consumption of a cloud platform by using workload prediction to reallocate the set of available servers. Jiang et al. perform data-driven QoE (quality of experience) optimization for real-time exploration and exploitation [67]. While we borrow techniques from this rich literature [15], our contribution is to shed light on how to use these techniques to find the exemplar cloud configurations and to show collective optimization can solve the problem using only a fraction of measurement cost required by prior work.

7.3.5 Heuristics

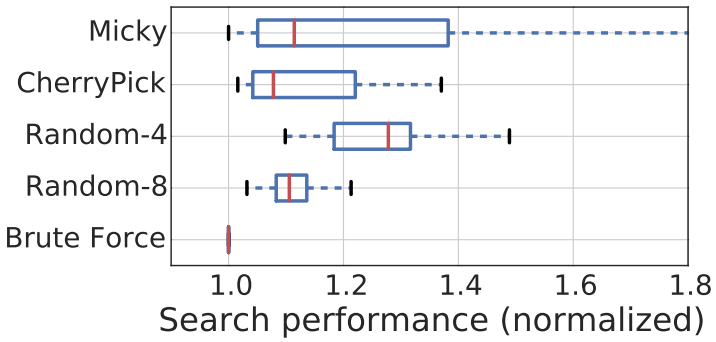
In the literature, several strategies have been proposed to find the most rewarding slot machines (the exemplar configurations) in the multi-arm bandit setting. These strategies can be divided into three major groups. First, the *Epsilon-greedy*, works by oscillating between (a) exploiting the best option which is currently known, and (b) exploring at random among all of the options available to it. Second, the probability matching strategy selects the arms according to the probability of the arm being the optimal choice. Thompson sampling or Bayesian Bandits are well-known probability matching strategies. Last, in the contextual bandit problem, strategies such as Upper Confidence Bound (UCB) builds a predictor from existing observation for making



(a) Hadoop 2.7



(b) Spark 2.1



(c) Spark 1.5

Figure 7.2 Search performance of optimization methods in search for cost-effective cloud configurations. Three software systems are evaluated. *CherryPick* finds good solutions in the three systems while *Micky* is comparable in Hadoop 2.7 but shows higher variance (sub-optimal choices). We propose a integrated system (in Figure 7.5) to detect those sub-optimal cases for improving *Micky*.

a better decision. UCB always opportunistically chooses the arm that has the highest upper confidence bound of reward, and therefore, it will naturally tend to use arms with high expected rewards or high uncertainty. The above only discusses some important methods. It is not the major focus to design the best method but to evaluate the existing methods best for collective optimization.

7.4 Evaluation

7.4.1 Comparison Method

We compare our method with *Brute Force*—measures all possible configurations and *CherryPick*—the state-of-the-art method [3]. Besides, we use *Random-4* and *Random-8*, which randomly measures 4 and 8 configurations (for each workload) respectively as straw man methods. The comparison metrics are *measurement cost* and *search performance*.

The *brute force* approach needs to test each configuration, and therefore, it generates constant measurement cost ($|S_w| \times |W|$). The measurement cost of CherryPick varies for different workloads since it uses a heuristic stopping criterion. The lower bound is $3 \times |W|$ because CherryPick uses at least three measurements as its initial points. *Micky* performs collective optimization, and hence the measurement cost is shared by a batch of workloads and therefore, expected to be much lower than the other methods.

To compare their search performance, we use normalized performance in terms of execution time (the elapsed time required to complete a workload) and operational cost (the charge for completing a workload). The *brute force* approach always finds the optimal configuration while the CherryPick is very likely to find near-optimal choices. We examine whether *Micky* can find near-optimal configurations that are comparable to the *CherryPick* approach. A method delivers better search performance when the performance of found solutions is closer to the optimal. For example, 1.05 is better than 1.15 because the former is only 5% slower than the optimal.

We demonstrate the effectiveness of *Micky* using 107 workloads on Apache Hadoop and Spark. These workloads are representative of many real-world applications. Table 7.1 lists a subset of the workloads. Please refer to our previous work for more details [61, 62, 92].

7.4.2 Experiment Setup

CherryPick—Bayesian Optimization

We encode the cloud configurations (*i.e.*,CPU types, core counts, memory size per code and the bandwidth to Elastic Block Storage) to represent the search space. For the parameters, we choose the same kernel function (*Matérn 5/2*) and the same stopping criteria (**EI**=10%),

as used in *CherryPick*. Regarding the choice of initial points, we randomly select three cloud configurations. The above process is repeated 100 times for reducing artifact and better showing the capability of *CherryPick*.

Micky—Multi-Armed Bandit

There are three common algorithms for the multi-armed bandit problems as described in Section 7.3. We choose UCB because it is more stable as compared to other bandit algorithms (will be discussed later in Section 7.4.5). *Micky* runs in two phases: (1) pure exploration, and (2) exploration along with exploitation. In the pure exploration phase, *Micky* measures the performance of VMs with random workloads for improving stability and reduces sampling bias. The α parameter represents the number of exhaustive iterations over each VM type. In the second phase, *Micky* runs the algorithm to handle the exploration and the exploitation. The behavior of this phase is controlled by the parameter β, which controls the number of measurements for finding the exemplar configurations. The measurement cost of *Micky* is $(\alpha \times |S| + \beta \times |W|)$. We have observed that the measurement cost is directly proportional to the effectiveness of *Micky*. In our experiments, we choose $\alpha = 1$ and $\beta = 0.5$.

7.4.3 Can *Micky* identify the exemplar cloud configurations?

The primary goal of *Micky* is to find the most suitable cloud configuration across all workloads. In this evaluation, we show the search performance in finding the cost-effective VM types. In Figure 7.2, we use box plot for comparison. The red line in the box represents the median value while the two sides of the box are the first and third quartile. The whiskers represent the 10 and 90 percentile respectively.

From this figure, we observe that the performance of *Micky* is comparable to CherryPick in the majority of workloads (using the median). *Micky* is only 5% worse than CherryPick on Spark 2.1 and Spark 1.5. Surprisingly, *Micky* is slightly better than CherryPick on Hadoop 2.7. The variance of *Micky* is higher because *Micky* optimizes most workloads but fails to optimize for some. We will discuss how to remedy this situation in Section 7.5.

To explain why *Micky* works, we further analyze the exemplar VM types recommended by *Micky* as listed in Table 7.2. The table shows the percentage of workloads that are within the performance thresholds. CherryPick finds good VM types (< 1.2) in 86% of workloads while *Micky* achieves the same search performance performance in 71% of workloads using only 11.6% of measurements by CherryPick.

Table 7.2 The most cost-effective VM types for 107 workloads recommended by *Micky* The number above each column label represents normalized performance (to the optimal). CherryPick finds good (< 1.2) VM types in 86% of workloads.

	$= 1.0$ **Optimal**	< 1.1 **Excellent**	< 1.2 **Good**	≤ 1.4 **Tradeoff**	> 1.4 **Unsettled**
c4.large	48%	61%	66%	70%	30%
m4.large	27%	46%	71%	84%	16%
m4.xlarge	9%	15%	32%	63%	37%

7.4.4 When not to use *Micky*?

Micky reduces measurement cost while delivering satisfactory performance. However, there exists a trade-off between the cost reduction achieved by *Micky* and its effectiveness.

First, Figure 7.3 shows that *CherryPick* is four times expensive than *Micky*. As the number of workloads increases, *Micky* is more economical because the cost of pure exploration phase of *Micky* remains constant. This is because α depends only on the number of cloud configurations. We observe that *Micky* only uses a fraction of the measurement cost when compared to the other methods. For example, when optimizing for 40 workloads, *Micky* only uses 30 measurements to find the suitable cloud configuration whereas CherryPick uses 156 measurements. Another observation is that *Micky* is more scalable because the slope of the line decreases as the number of workloads increases.

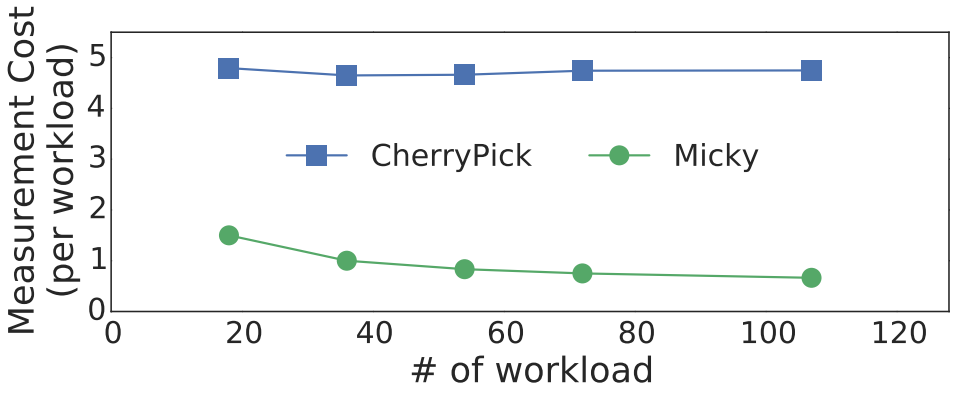


Figure 7.3 Low measurement cost in collective optimization. *CherryPick* optimizes each workload separately while *Micky* finds the exemplar cloud configuration suitable for a group of workloads.

Second, a user demands near-optimal solutions (*e.g.*,< 1.1) mostly for highly recurring workloads because its measurement cost can be amortized. In Table 7.3, we show the *knee-point* that a user should use a single-optimizer rather thatn a collective-optimizer. We calculate the knee point using $K \times f(\Delta_P, C_P) \geq g(\Delta_M, C_M)$, where K is the recurrence of a workload as the knee point, the function f represents the opportunity loss due to inferior search performance, and the function g represents the reduction of measurement cost when using collective optimization. In addition, Δ_P is the delta of normalized search performance (between a single- and collective-optimizer), Δ_M is the delta of measurement cost. C_P and C_M are cost (*e.g.*,dollars) defined by users. For simplification, we use $C_P = 10 \times C_M$ in this calculation.. For non-critical workloads (*e.g.*,recurring batch-process jobs), C_P is lower, and hence, *Micky* is more beneficial. As shown in Table 7.3, *CherryPick* is preferred only when the same workloads run more than 20 to 30 times. Otherwise, *Micky* is a more desirable solution.

Table 7.3 The knee point when *Micky* should not be used. The knee point (the number of recurrence of workloads) represents a trade-off between search performance and measurement cost.

	18	36	54	72	107
Brute Force	84.8	120.6	55.0	52.1	57.3
Random-8	37.9	51.5	33.7	36.0	44.7
Random-4	18.4	24.2	27.0	28.5	27.9
CherryPick	23.3	30.8	20.8	24.0	27.0

7.4.5 Why UCB is the preferred choice?

To select the suitable method for *Micky*, we compare three multi-armed bandit algorithms (as mentioned in Section 7.3.5). First, the behavior of Epsilon Greedy is controlled by the parameter ϵ. A larger value encourages exploration while a lower value encourages exploitation. We choose 0.1 for the epsilon parameter. Second, the Softmax algorithm uses a temperature parameter for structured exploration. The Softmax algorithm with an infinity temperature uses pure exploration while a zero value sticks to the arm (cloud configuration) with the highest estimated probability—pure exploitation. We use 0.1 for the temperature parameter. Last, the Upper Confidence Bound algorithm (UCB) tracks the confidence of rewards of arms. There are no parameters.

Figure 7.4 presents the comparison between the three methods. The parameter in the parenthesis represents the measurement budget, determined by α and β (as described in

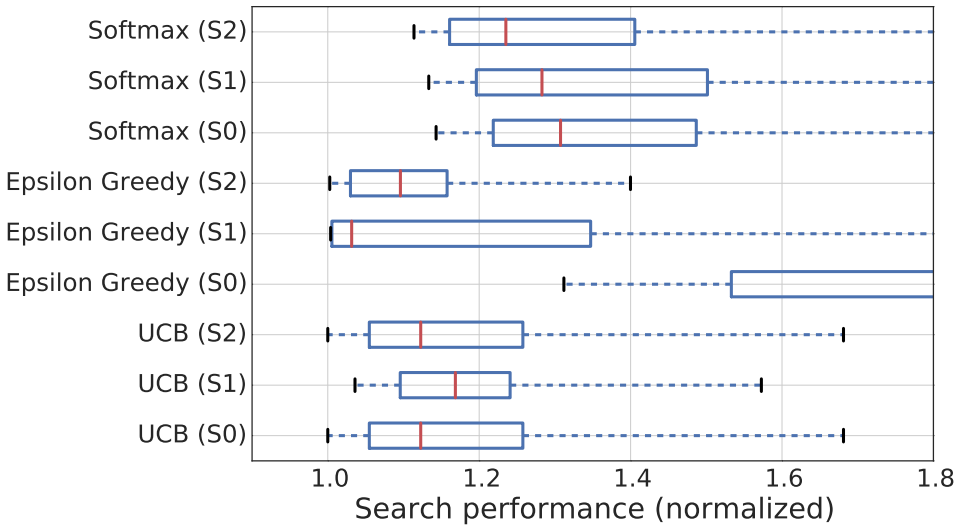


Figure 7.4 Selection of multi-armed bandit algorithms. The parameter (in the parenthesis) controls the measurement budget ($S0 < S1 < S2$).

Section 7.4.2). We choose 0, 1, 2 as the α parameter for $S0, S1, S2$ and use 0.5 for β in all. This figure shows UCB is more stable. Besides, the performance of UCB does not heavily rely on parameter tuning. Therefore, we prefer UCB to Epsilon Greedy, and *Micky* is built using UCB.

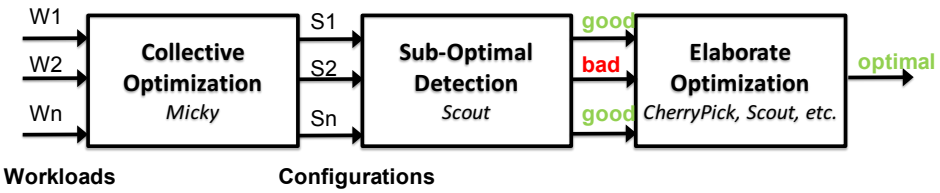


Figure 7.5 A system integration to alleviate sub-optimal choices in some workloads. Scout answers "is there a better configuration than the current choice?" [61]. An integration of *Micky* and Scout delivers a more efficient and reliable recommendation system of cloud configurations.

7.5 To Eliminate Sub-Optimal Choices

While the exemplar configuration is adequate for most workloads and reduces measurement cost significantly, it is almost inevitably that fewer workloads may suffer from sub-optimal

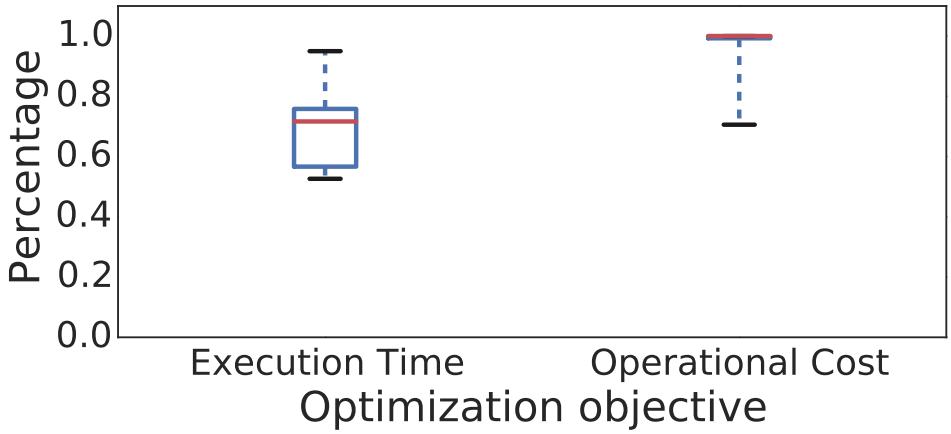


Figure 7.6 Detection of mis-predictions using SCOUT**.** The percentage represents the truth positive ratio, the probability the unsettled configurations can be identified. The two optimization objectives are to find the fast configuration and the most cost-effective VM type respectively.

performance (since we trade-off near-optimal performance for a large reduction in measurement cost). For instance, 30% workloads (*c4.large*) underperform (> 1.4) as shown in Table 7.2. Similarly, the 90 percentile in Figure 7.2. Although we have shown *Micky* is much more practical in the knee point analysis (Section 7.4.4), it would be great if we can inform users of those sub-optimal choices.

We propose a two-level approach that integrates our previously built system, SCOUT, to detect this problem for further optimization [61]. SCOUT is able to answer "is there a better configuration than the current choice?". Figure 7.5 illustrates the proposed system integration. Users get choices of optimizing those under-performed workloads. Figure 7.6 indicates that those sub-optimal choices are very likely to be identified. The detection module can detect bad performance with a median accuracy of 98%. This is promising because users benefit from low measurement cost (by *Micky*) and performance guarantee (by SCOUT). This ability enables users to further optimize for those sub-optimal workloads, which is particularly beneficial to highly recurring workloads.

7.6 Conclusion

Collective optimization is promising and yet practical for deploying multiple workloads in clouds. The collective optimization problem is similar to the multi-armed bandit problem. With existing heuristics, we are able to derive the exemplar cloud configuration that works well across a group of workloads. Collective optimization greatly reduces measurement cost while producing optimal

to satisfactory performance.

Chapter 8

Workload-Aware Data Placement

A CAT optimizer determines the best architectural choices. For example, users can scale out or shrink in a cluster size to meet workload changes. In this chapter, we focus on optimizing performance after a software system is reconfigured.

8.1 Introduction

In large-scale, distributed systems the dataset, which is too large for a single node, is partitioned among the nodes. Incoming workload (*i.e.,* requests) is routed among nodes by a load balancer. For extreme horizontal scaling to be effective, it is necessary for nearly all requests to be routed to a node containing the needed data locally, which avoids unnecessary node-to-node interactions. Consequently, data replication and data placement are components of load balancing and have a substantial impact on system performance. In the literature, for example, AUTOPLACER [102] and MET [32] optimize data placement to fit workload characteristics in NoSQL databases. Replicating hotter data in storage is a common approach to balance load [78]. In relational databases, sharding is used to distribute load by partitioning tables to achieve effective horizontal scaling [1, 27, 33, 47, 76].

Cloud computing has changed how computing resources are used. Before cloud computing, infrastructure is purchased and used for many years before it is upgraded or replaced. However, with Infrastructure as a Service (IaaS) equipment is rented for a short time, including as short as one execution of an application. This shift from buying to renting greatly increases the flexibility of infrastructure available for a given application. Therefore, rather than tune an application to run well on a specific machine, a cloud user instead can tune the infrastructure to accommodate each application on each dataset. When an infrastructure is resized in response to changes in workloads, it is necessary to replicate and place data partitions among nodes. It is still unclear what are the better strategies for this decision—prior work on data placement does not totally

address these issues [32, 78, 102].

Efficient deployment of distributed systems with an irregular workload requires both cluster sizing and data placement. Näive uniform data replication (where all data partitions have the same number of replicas) is effective only when the workload is also uniform—requests are equally distributed among all partitions. *Workload-aware* data placement replicates and places partitions to match the distribution of requests in the anticipated workload.

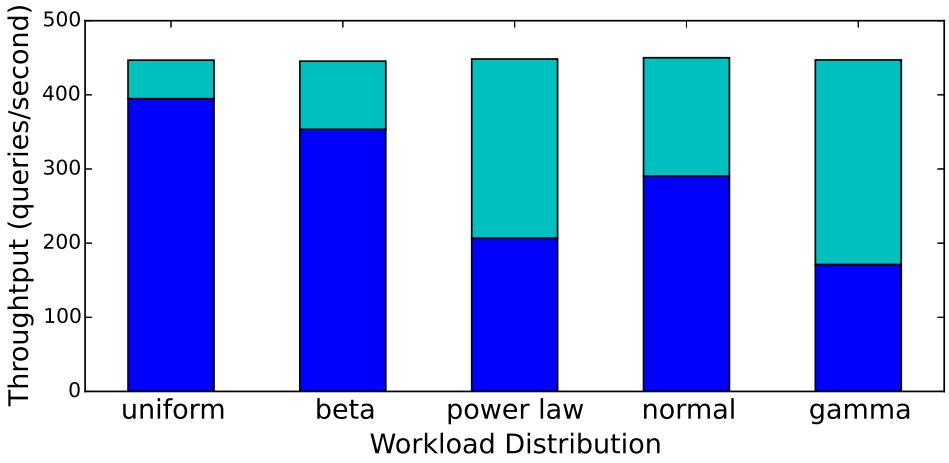


Figure 8.1 Uniform data placement is suboptimal. The lower bar is the measured throughput of uniform placement while the upper bar is the performance loss to the idealistic placement. (Data is a subset of data shown in Table 8.2 on Section 8.4.)

Figure 8.1 illustrates the cost of uniform data replication on non-uniform workloads. The results for five different workloads are presented on the x-axis; the skew (degree of imbalance) in the workload increases from left to right. The *complete* placement solution, where each node has all data, is an idealistic upper bound on the potential gains of matching data replication and workload. Because any node can process a request, new requests are sent to the least loaded node and the performance of *complete* is flat across all workloads. In this example, the cluster has twice the minimum capacity, so uniform replication has two copies of each partition. Therefore, a request must be sent to one of the two nodes that hold the data associated with the request. Throughput decreases as the workload skew increases because some nodes are over loaded and others are under utilized. While uniform placement achieves 88% of ideal on uniform workload, it is only 38% of ideal on the highly-skewed gamma workload. This example illustrates the need to properly *replicate* and *place* data on the nodes of a cluster.

This chapter explores the three dimensions that affect data placement. The first dimension

is *granularity* of data partitions. Fine-grain (more than one partition per node) placement has costs (overhead) and benefits (flexibility). The second dimension is how many *replicas* of each partition. We let the anticipated demand per partition (*i.e.*, the *workload*) determine the replication factor for each data partition. The hotter the data, the greater the replication. The number of data partitions influences how closely the replication matches the workload. However, a coarse-grain partition (one per node) is unlikely to match the workload. Last, fine-grain partitioning introduces the *placement* dimension because there are several ways to distribute the partitions among the nodes.

We present results from a simulation program that examines these three dimensions. We find that coarse-grain placement does not provide sufficient flexibility to balance non-uniform loads. A surprisingly small amount of additional partition granularity is sufficient to load balance and obtain most benefits. The work presents and evaluates the tradeoff between several fine-grain placement strategies that either increase robustness to tolerate workload deviation or reduce storage footprint.

To further examine these conclusions, our empirical study on an HPCC cluster[1] shows that proper data replication and placement affect system performance greatly. The coarse-grain scheme improves system throughput by 25% and 85% for the normal and power law distribution, respectively. A fine-grain placement strategy improves query throughput by 52% and 105%. On the most highly-skewed case, the improvement is 166% increase over the näive solution.

Because data placement relies on a prediction of the upcoming workload, which will invariably be wrong to some degree. This chapter, therefore, considers the *robustness*, which is a measure to describe how sensitive a data placement scheme is to slightly mis-predicted workloads, of several placement strategies. Results show that maximizing the number of unique partitions per node increases the robustness of a placement.

8.2 Modeling Data Replication and Placement

Our work concerns systems in which the dataset must be partitioned among the nodes because the dataset is too large to be completely replicated on each node. We replicate subsets of the whole dataset in order to increase throughput and decrease latency. While replication for availability is critically important, it is not a subject of this research.

Distributed, large-scale systems such as Apache Hadoop, Spark, Cassandra, and Ceph largely exploit data locality while reducing node-to-node communication for achieving high horizontal scaling [36, 76, 128, 138]. Data partitions are replicated as the system scales out. An inefficient data placement scheme is unable to achieve the optimal system performance and service-level

[1]HPCC Systems is an open source *data-analytics computer*—a highly scalable, distributed framework for processing and analyzing large datasets—supported by LexisNexis Risk Solutions at https://hpccsystems.com/.

objective (SLO) violations may occur [32, 102, 103, 121].

The goal of this work is to place data partitions onto nodes such that the performance is maximized for the upcoming workload. There is a large body of work supporting workload prediction [2, 21, 49]. This work assumes that a reliable (though not necessarily perfect) prediction is provided by some other work. Instead of solely relying on accurate workload prediction, systems can dynamically adjust replication factors and data locations for handling workload changes [32, 78, 121]. This work focuses on determining the optimal partition granularity, replication factors, and placement strategy. Our work is complementary to a dynamic approach.

The following model characterizes the *data replication and placement problem* in large-scale, distributed systems. Let $M > 1$ be the minimum cluster size that is sufficient to hold all data. The storage capacity is strictly limited by the amount of data it physically can store locally. In many real-world applications, M is in the hundreds of nodes. However, for this model it is only necessary that M not be equal to one, which does not require data partitioning.

Let N ($N \geq M$) be the number of nodes in a cluster. When the workload changes, the cluster expands (N increases) to meet increased demand and minimize QoS violations, or it contracts (N decreases) to reduce resources and cost. But the cluster cannot contract smaller than the M nodes needed to hold the data.

The data is partitioned into $k \geq 1$ equal-sized data partitions on each node. Thus, the dataset has $P = Mk$ unique partitions. Because the cluster has N nodes, there are $S = Nk$ *slots* for data partitions. We define the *replication factor*, R, as $R = N/M$. When $R > 1$, then $N > M$ and $S > P$ and some partitions will be replicated among the "extra" slots. *Coarse-grain* data placement occurs when there is only one partition per node, $k = 1$. *Fine-grain* data placement, $k > 1$, which has more total data partitions and partition slots, supports more distinct placements than coarse-grain providing a better opportunity to match the workload, and increases performance.

Model parameters

Minimum number of nodes:	$M > 1$
Per node granularity:	$k \geq 1$
Replication factor:	$R \geq 1$

Derived terms

Instantiated nodes:	$N = RM$
Unique data partitions:	$P = Mk$
Slots:	$S = Nk = RMk$

We present a motivating example below. The base cluster has four nodes, $M = 4$. The current demand is twice the current capacity. Figure 8.2a shows the load per partition on four nodes. The load is unevenly distributed among the partitions. In particular, in terms of node

capacity the load on the four partitions in Figure 8.2a is 3.5, 2, 1.5, and 1 (which conveniently totals 8). Figure 8.2b, shows the same aggregate demand as it is distributed among 16 partitions, four per node: $k = 4$. Fine-grain replication gives rise to a *placement* decision. Redistributing loads helps reduce load imbalance among nodes by placing highly requested partitions onto least loaded nodes. In Figure 8.2c the load is nearly balanced: two nodes have a load of 2.125 and two have 1.875. However, this alone does not help solve the overloading issue because the workload demand is twice the system capacity.

Suppose the cluster doubles in size to eight nodes exactly meeting anticipated demand: $R = 2$ and $N = 8$. Uniform replication onto eight nodes creates two replicas of each partition as shown in Figure 8.3a. The red box above the two left-most nodes shows the excess demand on those nodes as 3.5 units of node capacity are serviced by two nodes. The white regions in the four right-most nodes show the underutilization because the demand is less than the capacity.

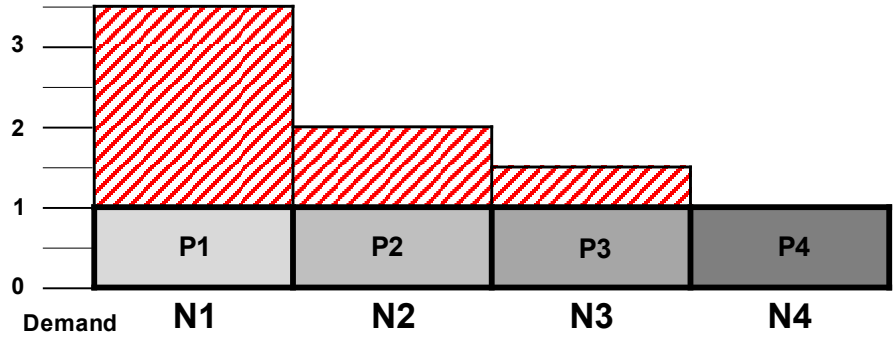
A coarse-grain, non-uniform solution is clearly better than uniform because *hotter* partitions can be replicated more than *colder* ones. For example, in Figure 8.3b the 3.5 units of P1 are distributed over three nodes. Unsurprisingly, a fine-grain solution can better match the demand to the capacity. Figure 8.3c shows that demand is perfectly matched to capacity in this idealized example.

Figure 8.3d shows a second placement that also perfectly balances load but has four unique partitions on each node. The nodes in Figure 8.3c have either one or two unique partitions. Fewer unique partitions per node reduces the footprint of both primary (memory) and secondary (disk) storage. On the other hand, more unique partitions per node increases the number of nodes that can respond to a given request, which may better share the load among nodes.

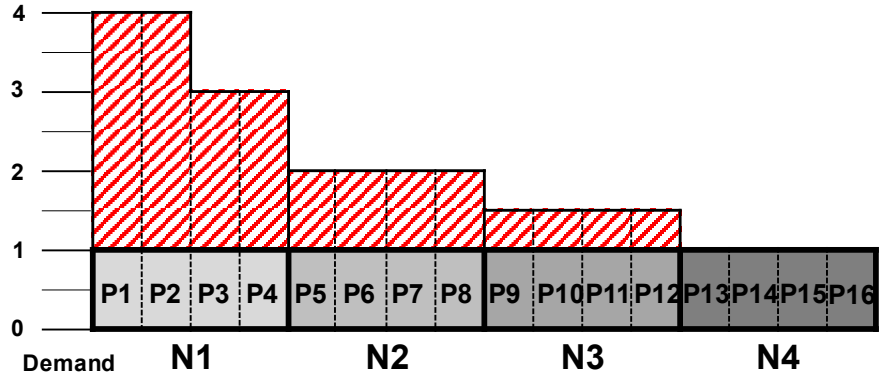
This simple example was constructed to clearly show the benefits of non-uniform, fine-grain data placement. Unless the demand is uniformly distributed among the partition (an extremely unlikely occurrence as explained in Section 8.3.1) then a näive uniform placement leads to over- and underutilization.

8.3 Workloads

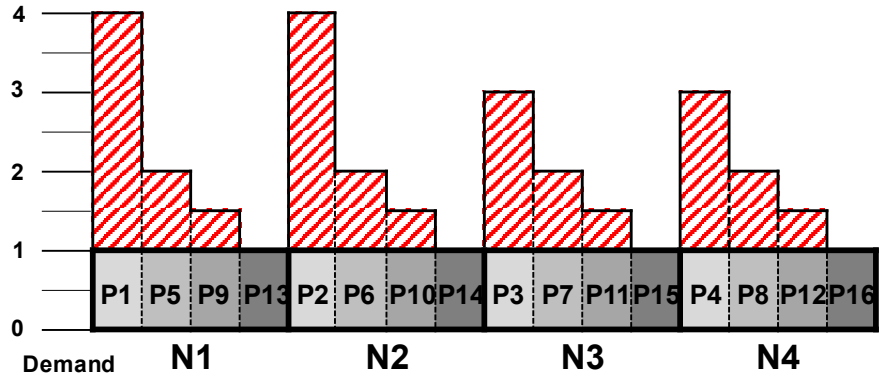
This section presents the data placement problem. First, it characterizes the workloads used in the work. Next, it expounds on the three dimensions of the problem: (1) granularity, (2) replication, and (3) placement. Last, it explains and analyzes three data placement methods, comparing performance in terms of load imbalance, storage footprint, and robustness.



(a) Coarse-Grain ($M = 4, R = 1, k = 1$)

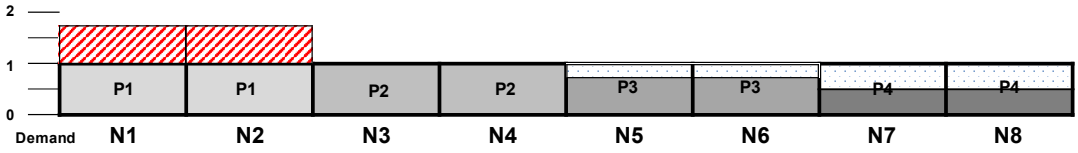


(b) Fine-Grain ($M = 4, R = 1, k = 4$)

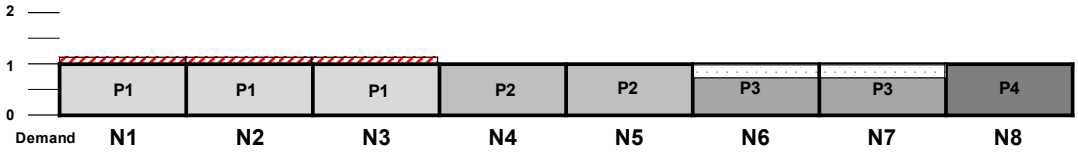


(c) Fine-Grain alternative placement

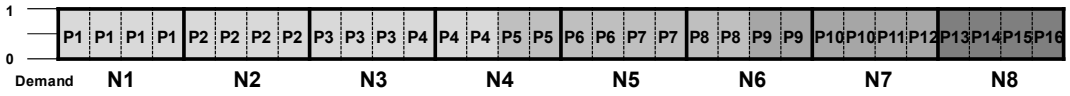
Figure 8.2 The workload demand exceeds the system capacity.



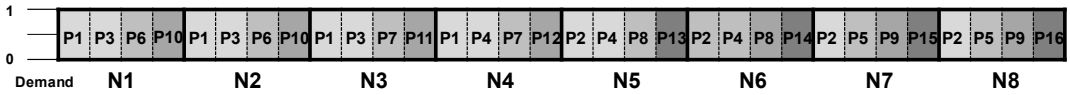
(a) Uniform data placement ($M = 4, R = 2, k = 1$)



(b) Coarse-grain data placement ($M = 4, R = 2, k = 1$)



(c) Fine-grain *compact* data placement ($M = 4, R = 2, k = 4$)



(d) Fine-grain *balanced* data placement ($M = 4, R = 2, k = 4$)

Figure 8.3 Different data placement schemes.

8.3.1 Workload Characteristics

A workload can be described with several critical characteristics, such as arrival rate and autocorrelation. Because this work considers data replication and placement, the workload characteristic that matters most is access frequency of the individual elements of the dataset, such as, the pages of a web server or the keys in an index. Other characteristics are not factored into this work because they do not have a direct impact on data replication or placement.

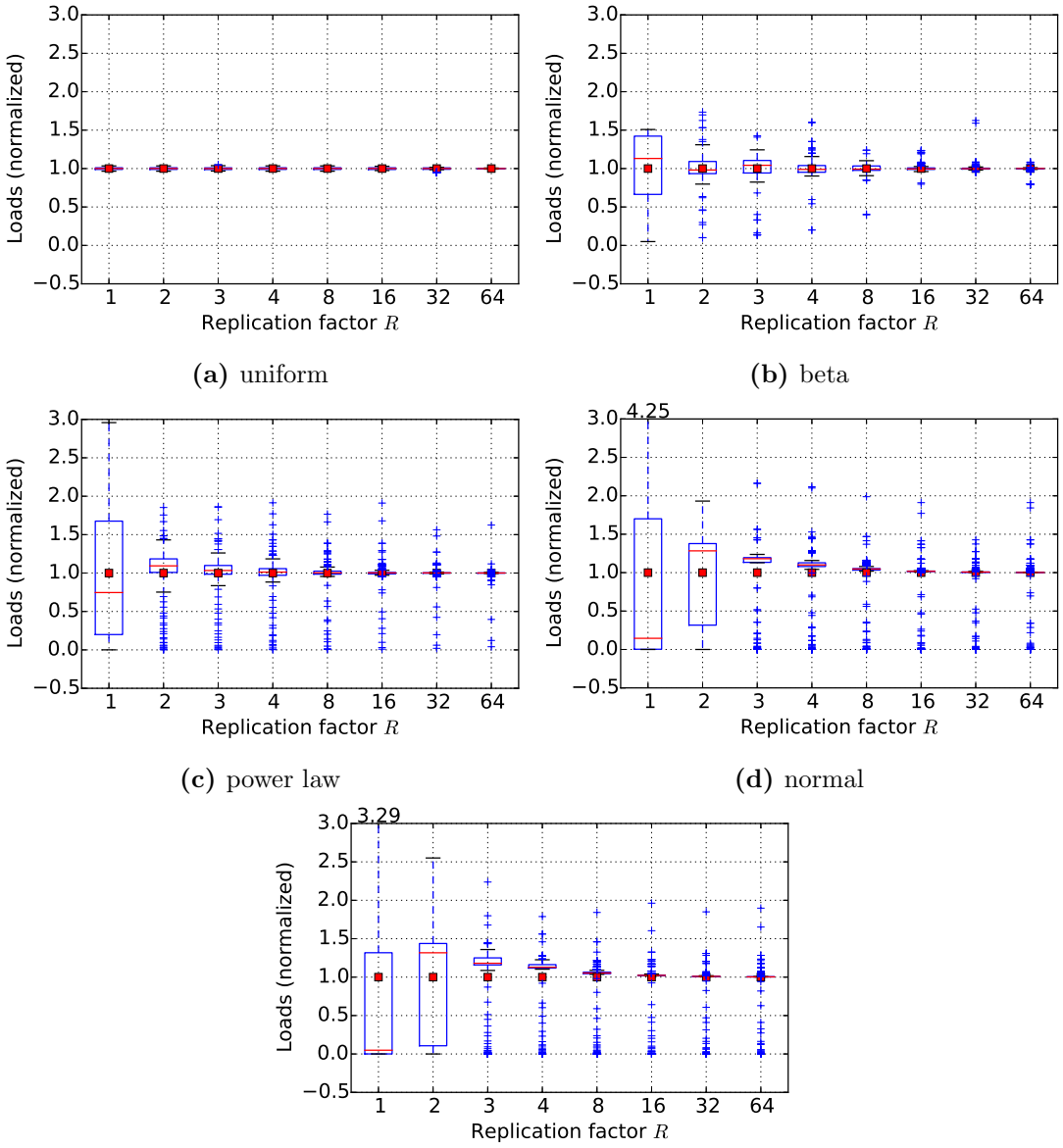
A uniform workload is atypical and likely artificial, which is unsurprising because non-uniform workloads are common in many natural settings [97]. For example, the normal (Gaussian) distribution can be observed in class score distribution, while the log-normal distribution is useful to describe the response file size in web servers [14]. In addition, the power law probability distribution is widely applicable to web hits, word frequency, personal income, *etc.* and tends to be highly skewed towards a small subset of the full dataset [87]. That is, a small number of partitions accounts the vast majority of key access and most of the partitions are touched infrequently. Several studies show that frequency of access to different pages or keys often follows a Zipf or power law distribution. This has been shown in web servers [38, 96], video streaming [115], and Wikipedia traffic [122] to name a few.

In this work, we consider the *normal* and the *power law* distribution for their wide appearance in many workloads. We also consider the *uniform* distribution for a näive baseline, *beta*, which is less skewed than *normal* and *power law*, and *gamma*, which generates the highest skewed workload and has been used in modeling workloads in storage systems [131].

Table 8.1 shows the five workload distributions used in the work. This table presents the distribution of the requests on each of four partitions ($M = 4$, $k = 1$). There are 30,000 unique requests among the 1024 keys, and the average is 7,500 requests per partition. The requests-per-node values are ordered in decreasing magnitude. As expected there are approximately the same number of requests for each partition in the uniform distribution. The *max:mean* column shows the ratio of the maximum number of requests to the average. Because the total running time is largely dependent on the slowest or most heavily loaded node, the max-mean ratio foretells the performance penalty for each workload on a uniform distribution. The ratio for uniform is 1.01, meaning the maximum is 1% greater than the average. But the highly-skewed gamma distribution has one partition that receives no requests and one has more than three times the average. It is clear from Table 8.1 that in non-uniform workloads the maximally loaded partition demands more resources than the other partitions. The final column presents access imbalance using the *skew* metric [97].

This work evaluates the problem using synthetic workloads. An alternative is to evaluate using real-world traces. But such traces are in short supply. Moreover, a trace represents a very specific situation that may not be representative of a general class. Additionally, a trace has

many characteristics that are hard to control. Using synthetic workloads enables us to evaluate more distributions and confine the observed effects to the change in distribution.

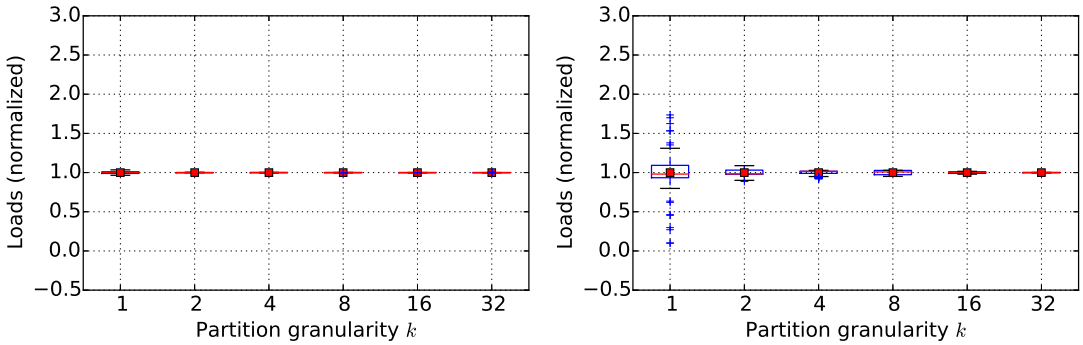


(a) uniform **(b)** beta

(c) power law **(d)** normal

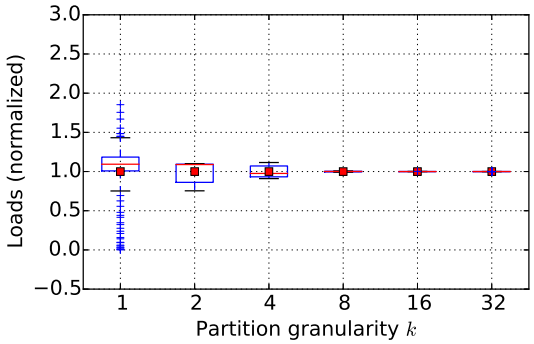
(e) gamma

Figure 8.4 The load distribution among nodes under the coarse-grain data placement ($M = 64, k = 1$).

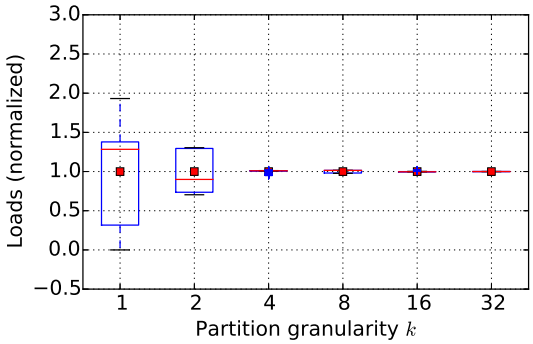


(a) uniform

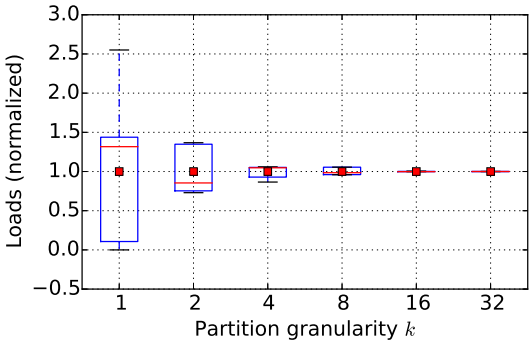
(b) beta



(c) power law



(d) normal



(e) gamma

Figure 8.5 The load distribution among nodes under the fine-grain data placement with various k ($M = 64, R = 2$).

Table 8.1 Load-imbalance of workloads.

Distribution	Requests for individual partitions				max:mean	skew
Uniform	**7,565**	7,548	7,449	7,438	1.01	0.15
Beta	**10,313**	10,288	4,715	4,684	1.38	0.39
Power law	**17,344**	8,795	3,361	500	2.31	0.60
Normal	**14,882**	14,827	149	142	1.98	0.74
Gamma	**23,542**	6,329	129	0	3.14	0.77

8.3.2 Data Placement Steps

A data placement method requires determining partition granularity, replication factors, and placement schemes. Partition granularity represents the smallest unit for replicating data and calculating loads. The coarsest granularity is one partition per node ($k = 1$), and the finest is one partition per key. A small k decreases the likelihood of balancing a non-uniform workload. However, a large k increases management overhead.

Once the partition granularity k is determined, the next step in deriving a solution is determining the number of replicas for each partition based on the expected workload. For example, suppose there are four partitions ($P = 4$) with a replication factor of four ($R = 4$), then there are sixteen slots for these partitions ($S = 16$) in a coarse-grain solution. Given a uniform expected workload the replication factor vector would be [4, 4, 4, 4], that is there are four copies of each of the four partitions. For a workload with a normal distribution the replication factor vector might be [2, 6, 6, 2] and for power law it might be [1, 2, 4, 9]. In general, there is no perfect match between the vector and the anticipated workload.

Assuming that the predicted load on each partition is λ_i and the total load is $\Lambda = \sum_{i=1,P} \lambda_i$. The replication problem is to determine the *replication vector*, $\vec{R}, R_i \in \mathcal{I}$, such that $R_i \geq 1 \; \forall i$ and $S = \sum_{i=1,P} R_i$. In words, the replication vector contains the number of replicas (an integer value) of each partition, such that the total number of replicas equals the number of slots available. The replication error is $E = \sum_{i=1,P} |R_i/S - \lambda_i/\Lambda|$, which is the accumulation of difference between the actual relative replication of each partition (R_i/S) and the anticipated relative workload per partition (λ_i/Λ).

The last step is assigning the replicas to the slots on the nodes. For coarse-grain, the number of replicas equals the number of slots and there is only one possible placement. But for fine-grain, $k > 1$, there are many possible placements. One placement strategy is to minimize the number of unique partitions on each node in order to reduce dataset footprint on the nodes. We call

this strategy *compact*. If there are multiple choices, it picks the node with the least load. The opposite strategy to *compact* is *full*, which maximizes the number of different partitions on each node and also picks the node with the least load. The third strategy is placing partitions in order to balance loads as much as possible. We call this strategy *balanced*. We show that although *full* and *balanced* have different goals, the resultant placements and effects are similar. Furthermore, *compact* and *full* balance loads among the nodes with their best efforts within their given constraints. Consequently, all three strategies achieve good load balancing, of course *balanced* is a slightly better at it. To reiterate: The goal of *compact* is reducing the number of unique partitions on each node, which reduces the footprint of the data set. On the other hand, the goal of *full* is to distribute replicas for the same partition to as many nodes as possible—balancing the load—with the intent of increasing the availability of hot partitions.

Our data placement procedure is described in Algorithm 5. This is a framework and therefore, different placement strategies can be used. The complexity for generating the replication vector is proportional to the number of partitions, $\mathcal{O}(Mk)$. Different placement strategies implement distinct *pick_node* functions, which is $\mathcal{O}(N)$. This function is executed for each slot (Nk). Therefore, the total complexity is of the placement algorithm is $\mathcal{O}(kN^2)$. Although it is quadratic, it is in the number of nodes that, even in a very large cluster, is tractable. A straightforward Python implementation executes in under few seconds when $N = 256$ and $k = 16$. Furthermore, this solution is a heuristic, it does not find the optimal solution. However, the solution is nearly optimal and because it is based on a prediction of anticipated load optimal is unnecessary.

Algorithm 5: Data Placement Procedure

```
   Input: an historical workload
   Output: partition placement on each node
 1 M := the minimum number of nodes that hold all data
 2 k := the selected partition granularity
 3 R := the replication factor
 4 loads := the predicted loads λ_i of partitions
 5 replicas := the replication vector (see Section 8.2)
 6 for p_i in replicas do
 7 |   for j=1; j <= p_i; j = j + 1 do
 8 |   |   /* strategy is compact, balanced or full */
 9 |   |   n := pick_node(strategy)
10 |   |   assign p_i to n
11 |   end
12 end
```

8.3.3 Tradeoffs in Placement Strategy

Our data placement framework is shown in Algorithm 5. This framework yields several variants of data placement when choosing different placement strategies. This section compares their effects on load balancing and storage footprint.

8.3.3.1 Load Balancing

The primary goal of data placement is distributing workloads evenly among nodes. A highly skewed system is more likely to encounter performance bottlenecks. Therefore, a well-balanced system achieves a higher system throughput. To explore the benefit of fine-grain replication and placement, we evaluate the effectiveness of our data placement schemes in balancing the anticipated workload. We generated 100 instances of workloads, of 300,000 queries, for each of the five distributions. The workload instances vary because the access counts are generated probabilistically. Each generated workload is considered a prediction of the upcoming workload.

First, we evaluate how the replication factor R affects load balancing. This simulation is conducted with $M = 64$ and $k = 1$. We use the box plot, as shown in Figure 8.4, to analyze the loads distributed among nodes. The bottom and top of the rectangle represent the first and third quartile of loads. The red line is the median, and the red dot is the mean. To facilitate comparison, the loads are normalized so that the means equal one. Above the box is the whisker that is calculated by adding 1.5 times the interquartile range (IRQ) to the third quartile. Similarly, the whisker below the box is the first quartile minus 1.5 times the IRQ. The plus signs represent the data points that have the loads beyond the two whiskers. These is the standard representation for a box plot. In a well balanced system, the median and mean values will be close and the box will be small.

This figure clearly shows that increasing the replication factor reduces load imbalance to a degree. However, this alone is insufficient to eliminate under-utilized loads totally because increasing the number of replicas only reduces the loads. The only way to reduce under utilization is to overlap under and over utilized partitions, which is not possible in the coarse grain method. Therefore, the replication factor alone is not sufficient for balancing loads because it does not reduce under-utilization. Take the *gamma* distribution for example, when $R = 1$, most of the nodes are under utilized (low median) and few nodes have extremely excess loads (large size box). Doubling the replication factor creates more overloaded nodes because $R = 2$ is not enough to distribute loads. As R increases, the median value moves closer to the mean and the variance (the size of box) also decreases, which suggests over-utilization is mostly addressed. However, increasing R is still insufficient because there are still many outliers (beyond the two whiskers).

Next, we examine how much the partition granularity can reduce load imbalance. We run a set of simulations with $M = 64$, $R = 2$, and various k values, and choose *balanced* as the

placement scheme. Figure 8.5 clearly shows that fine partition granularity greatly reduces load imbalance. Even in the most skewed workload (gamma), $k = 16$ is able to almost perfectly balance loads. For most workloads, $k = 4$ is sufficient to reduce the load imbalance below 10%. When workloads are highly skewed (the *normal* and *gamma* case), their load imbalance (the size of box in the figure) drops greatly from $k = 1$ to $k = 2$ because finer partitioning provides higher flexibility to mix over- and under-utilized partitions on the same node. Increasing partition granularity eventually leads to (for all practical purposes) perfectly balanced loads.

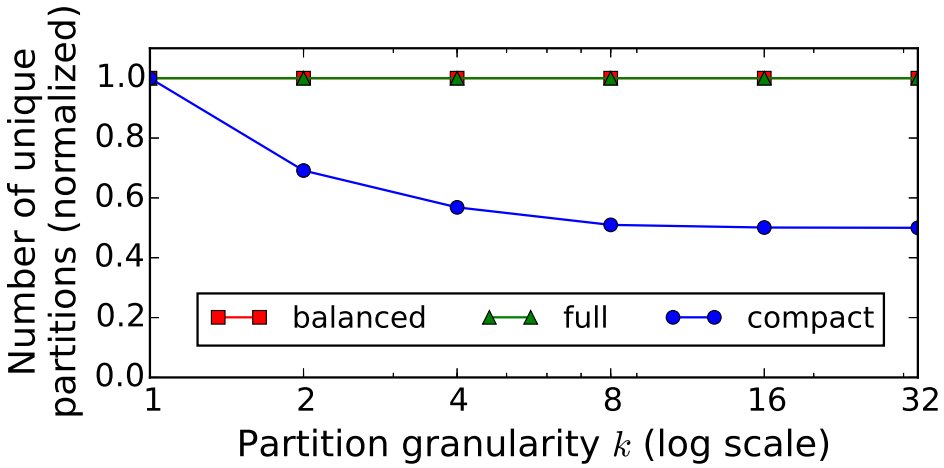


Figure 8.6 The number of unique partitions per node (storage footprint) under different placement schemes.

8.3.3.2 Storage Footprint

There are $S = Nk$ choices for placing a partition replica. When placing multiple replicas of the same partition on the same node, it reduces the number of unique partitions per node. When the number of unique partitions per node is lower, it generally requires lower storage footprint. A lower storage footprint reduces memory pressure, and may lead to higher cache efficiency. We run a set of simulations with $M = 64$, $R = 2$ and various k. This work only presents the results of the *power law* workloads. Other workload cases show very similar effects.

Figure 8.6 shows the average number of unique partitions on each node. By definition all partitions in *full* are unique and it is always 1 (normalized). *Balanced* is very nearly 1 in all cases. On the other hand, as k increases, *compact* tends to 0.5, which is $1/R$. We further investigate how the replication factor affects storage footprint. Figure 8.7 shows that the number of unique

partitions tends to $1/R$ as k increases, which is the desired outcome of the *compact* scheme.

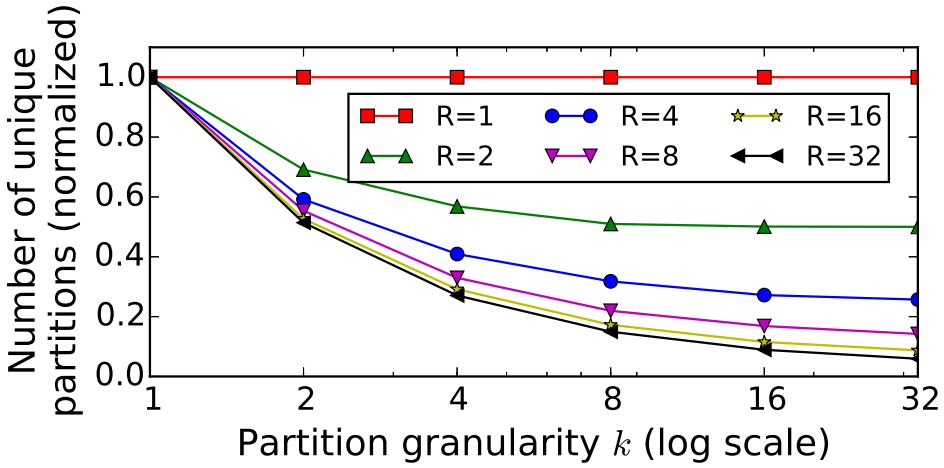


Figure 8.7 The number of unique partitions per node under the *compact* method with various k. The number converges at $k = 32$, which is equal to $1/R$.

8.4 Evaluation

This section presents our evaluation. We introduce our experimental setup and benchmark design. We then evaluate different placement schemes by measuring query throughput and testing their robustness to slight workload mispredictions.

8.4.1 Experiment Setup

We conducted our evaluation on Virtual Computing Lab (VCL), a cloud platform provided by NC State University. All servers are equipped with 2-core Intel Xeon CPU and 8GB memory and connected to a 10 Gbit switch. VCL only provides limited storage capacity of local disks or *instance storage*. This is also the case for many cloud configurations. In fact, a majority of EC2 instance types in Amazon Web Services (AWS) do not have any instance storage. Instead one must mount a remote volume served by a enterprise-level storage system. (AWS calls this *elastic block storage*–EBS.) We evaluate our approach using both instance storage (local disks) and remote volumes provided by network file system (NFS), backed by NetApp 2554 filer with dual controllers.

We evaluate our placement schemes on high performance computing cluster (HPCC) [83].

HPCC is an open source data analytics computer developed by LexisNexis Risk Solutions for processing big data. They maintain several clusters with more than 100 nodes, the largest with more than 500, to provide services to their clients. Experiments were conducted on a HPCC Roxie cluster. Roxie is a *data delivery engine* that responds to queries. It finds the answers to requests in an index that is partitioned and, if desired, replicated across the nodes. Roxie is optimized to handle massive amounts of concurrent requests with low latency.

Data replication and placement that fit workload demands have direct performance impact on performance. Roxie clusters partition and distribute data with two replicas per partition by default. We modified Roxie to incorporate our data placement schemes. Our approaches are not specific to Roxie. They should be able to apply to Apache Hadoop, HBase, Cassandra, and Ceph, providing benefit when workloads are not uniformly distributed across keys, partitions or nodes.

8.4.2 Workload Generator and Benchmark Suite

To evaluate the results learned in our simulation, we developed a distributed benchmark tool that is able to issue a large volume of concurrent queries to Roxie. This benchmark tool adopts the master-slave architecture, where the the master node generates workload according to a workload profile, and the slave nodes execute the query requests. This tool is customizable and supports any number of workload distributions. This benchmark suite is written in Python, and designed for testing query performance at large scale.

In our evaluation, we are interested in how placement schemes with different levels of partition granularity respond to different types of workload distribution. We consider *uniform*, *beta*, *power law*, *normal*, and *gamma* distributions. The beta distribution is defined on the interval $[0, 1]$ with two shape parameters, α and β. We choose $\alpha = 2$ and $\beta = 2$ for the base case of beta distribution. The power law distribution is controlled by the *shape* parameter and we choose 3 for the base case. Regarding the normal (or Gaussian) distribution it has a *mean* and a *standard deviation* parameter, which is 0 and 1 in our case. The gamma distribution also has a *shape* parameter and the base case uses 5. A single instance of each workload is used in all the empirical tests of Roxie so that results can be compared across multiple runs and different configurations. The specific workloads used are those shown in Table 8.1.

8.4.3 Benchmark Steps

To best measure the performance, our benchmark service runs one worker node for each Roxie server, which eliminates the performance impact at the client side. We use separate machines from the Roxie cluster on the VCL for the benchmark service. The Roxie controller node dispatches requests and synchronizes with workers. Worker nodes request jobs and execute them

Table 8.2 Steady-State Throughput Comparison (instance storage)

	Uniform	Beta	Power Law	Normal	Gamma
base	394.8	353.5	206.6	290.4	171.2
coarse	-	367.8 (4.0%)	381.9 (84.9%)	364.0 (25.3%)	309.7 (80.9%)
compact	375.4 (−4.9%)	377.5 (6.8%)	383.2 (85.5%)	374.6 (29.0%)	374.2 (118.6%)
balanced	408.1 (3.4%)	412.6 (16.7%)	422.8 (104.7%)	442.5 (52.4%)	455.9 (166.3%)
complete	446.8 (13.2%)	445.4 (26.0%)	448.3 (117.0%)	450.1 (55.0%)	447.0 (161.1%)

unit: queries/second

as soon as possible.

We generate the five workload distributions with different access counts to keys. All datasets are 128GB. Next, we specify the smallest cluster size M, the replication factor R (which determines the cluster size N), and the partition granularity k. In our evaluation, M is equal to 4. The coarse-grain schemes replicate data on a node basis. Fine-grain schemes, on the other hand, divides the data on a node into 32 equal-size partitions (1GB per partition).

We compare five placement schemes in total. First, *base* represents the uniform data placement. It is coarse grain ($k = 1$) and not workload aware. The *coarse* scheme is also coarse-grain but replicates partitions based on anticipated workload. For the fine-grain schemes ($k = 32$), we consider *compact* to reduce storage footprint while maximizing cache locality, and *balanced* to minimize load imbalance among machines. In our evaluation, we found that the *balanced* and *full* scheme have comparable performance. Due to the page limit, we report the results of *balanced* in most cases. Last, the *complete* is an "idealistic" placement where each node holds the entire dataset. It represents an upper bound.

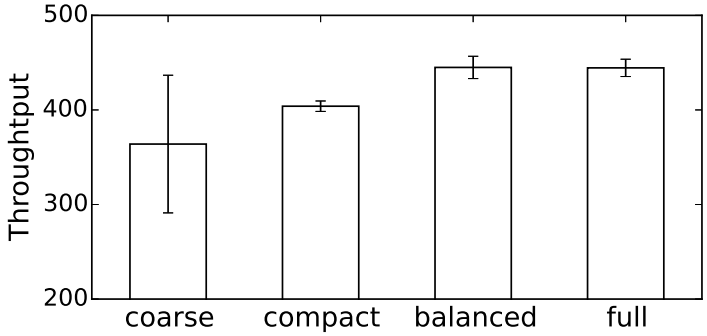
Our next step is to change the data layout in Roxie to reflect the desired data placement decision. The Roxie cluster is restarted to load the new data layout. To avoid cache interference, the file system cache is cleared before every benchmark run.

Last, a workload profile is submitted to the benchmark controller. The controller node generates the query plan accordingly. In this way, the same stream of requests is presented for each benchmark, which allows us to verify results with multiple identical runs and to compare results from different placements. We collect query throughput during the entire benchmark process.

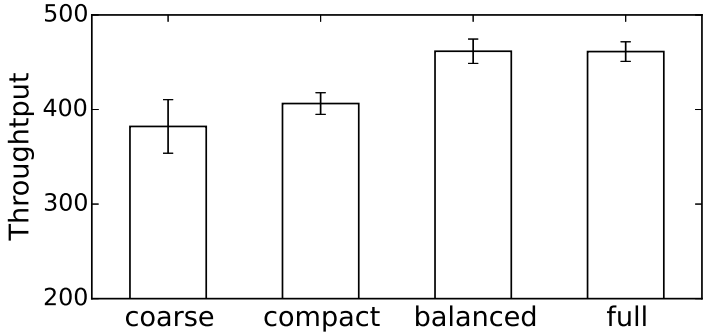
8.4.4 Steady-State Throughput

We conduct this evaluation to test steady-state throughput. We generate 30,000 requests for each of the five workload distributions. We then calculate the average throughput over the sampling period (the first and last 10% period are not included.) This measurement ensures we

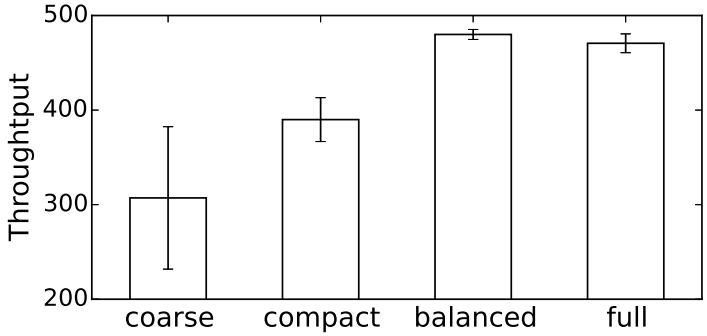
capture the stable throughput, but not the warm-up period (low throughput) and the long-tail period (system is not saturated).



(a) Beta



(b) Power Law



(c) Gamma

Figure 8.8 Compare robustness under slight workload mispredictions. The y-axis represents queries per second, and starts from 200 for better presentation to tell performance difference.

Table 8.3 Steady-State Throughput Comparison (NFS)

	Uniform	Beta	Power Law	Normal	Gamma
base	446.1	383.1	220.2	393.3	176.4
coarse	-	396.4 (3.5%)	415.3 (88.6%)	379.6 (29.4%)	327.9 (85.9%)
compact	403.7 (−9.5%)	416.2 (8.7%)	401.3 (82.3%)	407.1 (38.8%)	407.8 (131.1%)
balanced	447.6 (0.3%)	440.8 (15.1%)	454.0 (106.2%)	469.7 (60.1%)	485.9 (175.5%)
complete	484.2 (9.0%)	485.6 (26.8%)	492.4 (123.7%)	495.0 (68.8%)	490.0 (177.8%)

unit: queries/second

8.4.4.1 Local Storage

Our evaluation starts with storing data required for Roxie queries on local disks. This evaluation involves 8 Roxie nodes: $M = 4$ and $R = 2$. Table 8.2 shows the throughput of proposed replication and placement schemes under different workloads. The values in parenthesis are the speedup relative to *base* performance at the top of each column. The *base* placement strategy is uniform. It does not perform well as the skewness of workload increases. For example, the power law workload in the uniform data placement can only achieve 52.3% of the throughput of a uniform workload. The second strategy is also coarse grain but replicates according to anticipated workload. On a uniform workload this is the same as *base*. It out performs *base* on skewed workloads. For example, it achieves 84.9% more throughput than *base* on *power law*.

Two fine-grain approaches, *compact* and *balanced*, which further improve performance over *coarse*, are also shown. In the normal workload case, *compact* and *balanced* improve on *coarse* by an additional 10.6 and 78.5 queries per second. In the gamma case, *balance* adds 146.2 queries, a 47.2% improvement over *coarse*. Workload-aware data placement is preferable for non-uniform workloads. The fine-grain strategies out perform *coarse* on all the skewed workloads. This is attributed to better load balancing. As skew increases, the benefit from fine-grain increases (because the load imbalance in *coarse* increases).

The *complete* solution out performs all others, including both fine-grain solutions. This is because while the workload was probabilistically generated over 30,000 requests. The workload for each small window of requests does not always reflect the overall workload. In such cases, *complete* performs better. However, *complete* is generally not feasible when dataset is too large to fit into one node.

Overall, workload-aware data placement significantly increases query throughput. Using fine partition granularity better balances the load. *Balanced* performs better than *compact*, indicating that the benefit of a smaller footprint is less than the cost of poorer load balancing. The *balanced* scheme is occasionally competitive with *complete*.

8.4.4.2 Remote Storage

Next, we evaluate our proposed schemes against data storing on remote storage. The Roxie cluster size and the number of benchmark clients remain the same with the local storage case. Table 8.3 details the throughput numbers. This evaluation confirms the general observations seen in the instance storage test. However, the throughput is higher using remote storage. While somewhat counter intuitive, it is not unheard. This occurs because local storage uses plain commodity disks and the filer uses high-performance disks as well as aggressive caching. Moreover, the I/O demand does not exceed the capacity of the NFS server. Therefore, the additional network traffic is not creating a performance bottleneck.

8.4.5 Robustness Comparison

We are interested in how sensitive a placement scheme is to minor deviations in the anticipated workload. (Tables 8.2 and 8.3 show performance degradation for major deviations.) We say a placement scheme is more *robust* when the scheme works well even when the actual workload is slightly different from the anticipated workload. We pick different parameters for generating slight workload variance. For example, we change the shape parameter in the power law distribution. Therefore, it becomes either less or more skewed. We create two less and two more skewed workloads for each type.

Figure 8.8 shows how different placement schemes react to workload shifts. The figure shows the average throughput and the standard deviation of the placement schemes under the four "shifted" workloads The figure indicates that the coarse-grain scheme under performs in both average (lower) and deviation (greater) compared to the fine-grain schemes.

The *compact* scheme is better than *coarse* but its performance is not as good as the *balanced* scheme. The *balanced* scheme overall exhibits higher throughput than *coarse* and *compact*. More importantly, *balanced* shows consistent standard deviation in three workloads. The highest performance degradations in each workload are 2%, 6% and 3% while the *compact* scheme shows 3%, 5% and 14% (increasing as skewness increases) degradation respectively. The above suggests than *balanced* is more robust then *compact*, which is robust than *coarse*. There is little difference between *balanced* and *full* in either average throughput or standard deviation. This is because there is little difference in the placement of partitions—that is, the balanced scheme tends to have a high degree of unique partitions on each node.

8.4.6 Micro Benchmark

We have presented the steady-state throughput in Section 8.4.4. In this section, we further examine why different placement schemes lead to large performance difference.

We investigate resource utilization of different placement schemes for understanding the tradeoff between the *compact* and *balanced* scheme. We collect system statistics (*dstat* [130] and *cachestat* [52]) during the entire benchmark runs. Table 8.4 presents the system statistics, and metrics are normalized to the smallest value in each metric group, except the *max:mean* ratio. This normalization better shows the difference between placement schemes. Except *mean %CPU*, a system is more efficient when the metrics listed are small. These metrics are collected from the benchmark runs under the gamma workload. Other workloads present very similar trends.

First, we examine CPU utilization across all Roxie nodes. The average CPU utilization indicates whether Roxie is fully saturated, and the *max:mean* metric tells whether loads are well balanced among Roxie nodes. In the *base* scheme, CPU utilization is the lowest and load imbalance is the highest, which explains why uniform data placement under performs. Workload-aware replication eliminates load imbalance while improving CPU utilization. Fine-grain partition further reduces load imbalance, as in the *balanced* scheme.

Table 8.4 Normalized System Statistics of Roxie Servers

	Metrics	*base*	*coarse*	*compact*	*balanced*	*complete*
Load Balancing	% CPU (mean)	**1.00** (13%)	2.29	2.37	3.11	2.97
	% CPU (max:mean)	2.01	1.34	1.23	1.1	1.14
Cache Locality	cache misses (sum)	1.13	1.24	**1.00** (659K)	1.26	1.22
	dirty pages (sum)	1.20	1.47	**1.00** (200K)	1.52	1.29
	cache sizes (max)	2.11	1.51	**1.00** (825MB)	1.17	1.15
Efficiency	I/O wait (mean)	1.39	6.73	**1.00** (2.35%)	2.48	2.55
	TCP connections (mean)	2.40	1.53	1.31	1.10	**1.00** (1357)

Second, we examine the benefits of packing multiple replicas into the same node, as in the *compact* scheme. Table 8.4 shows that cache misses and dirty pages are significantly lower in the *compact* scheme. Besides, *compact* has much lower cache sizes, 17% lower than *balanced* and 51% less than the *coarse* scheme. Although the *compact* scheme outperforms others in cache locality and requires less cache, it does not generate the highest query throughput. A possible explanation is that requests do not greatly benefit from better cache locality. We suspect the *compact* scheme is useful especially when query applications require costly read operations.

Third, we compare I/O wait time and the number of TCP connections for comparing their efficiency. A lower I/O wait time indicates that systems do not waste much time on slow I/O operations. The *compact* scheme, with maximum cache locality, has the lowest I/O wait value. The number of TCP connections at a given time is related to processing efficiency. We observe

that the *balanced* scheme yields a lower number of TCP connections than the other schemes, suggesting that requests complete more quickly.

8.4.7 Summary

Our evaluation provides empirical data to support our findings in the simulation. First, workload-aware data placement, both the coarse and fine grain methods, reduces load imbalance, thereby improving system throughput. However, the coarse-grain approach is insufficient when workload is highly skewed. Finer partitioning further balances the loads and in many cases, the *balanced* scheme has comparable performance to *complete* (an "idealistic" placement). Furthermore, both *balanced* and *full* are robust while *compact* reduces storage footprint to $1/R$.

8.5 Conclusion

Efficient deployment of large-scale, distributed systems with an irregular workload requires both cluster sizing and data placement. We show the uniform distribution is (unsurprisingly) poor for typical, non-uniform workloads. This work further shows that coarse-grain replication can reduce over-utilization but is unable to address under-utilization. Finer partition granularity reduces both under- and over-utilization. With fine-grain partitioning there is a placement decision. Maximizing the number of unique partitions per node increases robustness to workload misprediction, while minimizing the number reduces storage footprint. Our empirical study using an HPCC Roxie cluster shows the benefit of footprint reduction does not offset cost due to poorer load balancing. However, we do not believe this is universally true.

This work focuses on the dimension and tradeoffs of various data replication and placement strategies. For our future work, we plan to implement an elastic controller that incorporates our proposed data placement schemes. In an elastic system, the gains of the optimal placement must be offset by the cost of data movement. Calculating this data movement cost remains as a future work.

Chapter 9

Conclusions and Future Work

This dissertation focuses on using data-driven approaches to optimize workload and system performance in the cloud. We study cloud architecture tuning (CAT), which brings great benefits of cloud computing—tuning architecture for a workload at a time. Our prototype system Scout consists of guided search, relative ordering, low-level insights and transfer learning—enabling an *efficient*, *effective* and *reliable* CAT method in an *automatic* and *scalable* way. This Chapter concludes this dissertation and discuss its future work.

9.1 A Practical Guide to Cloud Optimizer

To pick an optimizer, we should compare its search performance and measurement cost, and understand their assumptions and constraints. In Table 9.1, we derive a practical guide for selecting an optimizer. This guide is derived based on extended literature review and our extensive experimentation.

Historical data are execution records of workloads on cloud configurations. *CherryPick* and *Arrow* do not use historical data (from other workloads) and therefore, require significant initial measurements for building prediction models while *PARIS* and Scout uses historical data.

Low-level Metrics are runtime information (such as CPU utilization, memory usage, and I/O rates) for better characterizing workloads. If such low-level information is accessible, users should choose optimizers that leverage low-level performance information.

Reliability represents whether a CAT method can deliver consistent search performance and maintain reasonable search cost across diverse workloads. Some cloud optimizers may suffer from the fragility issue or high prediction error. They are considered less reliable. When using these optimizers, users should be more careful because they do not know whether the recommended configurations by the optimizers are near-optimal or sub-optimal.

In this dissertation, our proposed SCOUT is *effective*, *efficient* and *reliable* for any single workload. On the other hand, we propose a collective optimizer *Micky* to further reduce search cost while delivering comparable search performance for a group of workloads. To address the sub-optimal choices in some workloads, we propose an integration with SCOUT for further optimization.

Table 9.1 A practical guide to choosing the right optimization method. *CheeryPick* works for any workloads without historical and low-level performance data [3]. *Arrow* uses low-level metrics to augment Bayesian Optimization (used in *CherryPick*) [62]. *PARIS* requires low-level and historical data for predicting execution time and running cost of workloads on different VM types [133]. SCOUT leverages a learning model and sequential model-based optimization (SMBO) to deliver efficient, effective and reliable recommendation [61]. *Micky*, different from others, applies collective optimization to largely reduce measurement cost.

CAT Optimizer	Solution	Historical Data	Low-Level Metrics	Reliability
Single	CherryPick [3]	✗	✗	✗
	Brute Force	✗	✗	✓
	Arrow [62]	✗	✓	✓
	PARIS [133]	✓	✓	✗
	SCOUT [61]	✓	✓	✓
Collective	Micky [63]	✗	✗	✗
	Micky + Arrow	✗	✓	✓
	Micky + Scout [63]	✓	✓	✓

9.2 Conclusion

- Low-level performance information is essential to characterize workloads predict systems performance. With low-level performance insights, we are able to understand workload behavior and system behavior without expert knowledge—not to mention the increasing complexity in software systems and applications.
- Although machine learning tools are readily available, off-the-shelf methods are far from ideal. For better predicting workload and system performance, we need to avoid generalization error. We propose leveraging low-level performance information to improve generalization capability. We also design a two-level learning method that reduces the overfitting problem.

- Bayesian Optimization is a powerful device for optimizing any black-box functions—well suited to CAT problem. However, high-level features—such as core counts, memory size, and cluster sizes—are not sufficient to accurately predict performance. Our novel optimization method alleviates the fragility issue by embedding low-level performance insights. Together with relative ordering and transfer learning, we can find near-optimal architectural choices with only a handful of trials.
- When a cluster is resized to reflect the workload changes, we need to optimize data placement to release the full capacity of the resized cluster. We find that workload-aware data placement with fine-grained partitioning better balances the loads and tolerates mispredictions of loads.

9.3 Future Work

This dissertation has shown that data-driven approaches are very promising to optimize system performance. To further improve this line of research, we should focus on building accurate prediction models and understanding the relationship between architecture, workload, and performance.

- **Synthetic performance data.** The efficacy of data-driven approaches relies on quality data. A key element to quality data is to ensure that comprehensive workloads are included in the performance data. However, data collection from real-world applications is difficult to guarantee such comprehensiveness. In this dissertation, we assume we have obtained comprehensive data by using a large set of diverse workloads. However, it is impossible to claim to have comprehensive data unless we can define the whole spectrum of workload behavior. To solve this problem, we should be able to generate performance data using synthetic program that simulates comprehensive workload behavior such as CPU-intensive, memory-intensive and I/O-intensive. With the synthetic program, we can control the ratio of computation, memory access, and I/O characteristics. Training a prediction model with comprehensive performance data will improve prediction quality.
- **Performance extrapolation.** Generalizing performance behavior beyond the training data is very difficult. This dissertation does not examine performance extrapolation—when the target cluster size is several times larger. This is particularly important, for example, to accelerate big data analytics jobs. *Ernest* exploits workload and system knowledge for building accurate prediction models [123]. However, Ernest is very expensive because it requires separate training processes for different workloads and different virtual machine types. We believe that low-level performance insights and comprehensive synthetic performance data might help extrapolate scaling behavior better.

Performance optimization is a long-standing research problem. This dissertation examines several challenges of applying machine learning techniques to optimizing system performance in the cloud. As cloud computing becomes more prominent, hosted applications become more complex, and architectural choices become more customizable, data-driven approaches—due to its black-box feature and its prediction capability—will be a vital device to optimize workload and system performance in cloud computing.

REFERENCES

[1] Adya, A. et al. "Slicer: Auto-sharding for Datacenter Applications". *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*. OSDI'16. Savannah, GA, USA: USENIX Association, 2016, pp. 739–753.

[2] Akdere, M. et al. "Learning-based Query Performance Modeling and Prediction". *Proceedings of the 2012 IEEE 28th International Conference on Data Engineering*. ICDE '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 390–401.

[3] Alipourfard, O. et al. "Cherrypick: Adaptively Unearthing the Best Cloud Configurations for Big Data Analytics". *Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation*. NSDI'17. Boston, MA, USA: USENIX Association, 2017, pp. 469–482.

[4] *Amazon Web Services*. https://aws.amazon.com.

[5] Ananthanarayanan, G. et al. "Scarlett: Coping with Skewed Content Popularity in Mapreduce Clusters". *Proceedings of the Sixth Conference on Computer Systems*. EuroSys '11. Salzburg, Austria: ACM, 2011, pp. 287–300.

[6] Ananthanarayanan, R. et al. "Cloud Analytics: Do We Really Need to Reinvent the Storage Stack?" *Proceedings of the 2009 Conference on Hot Topics in Cloud Computing*. HotCloud'09. San Diego, California: USENIX Association, 2009.

[7] Anderson, E. *Simple table-based modeling of storage devices*. Tech. rep. Technical Report HPL-SSP-2001-04, HP Laboratories, 2001.

[8] *Apache Hadoop*. http://hadoop.apache.org/.

[9] *Apache Spark*. http://spark.apache.org/.

[10] Ardagna, D. et al. "Quality-of-service in cloud computing: modeling techniques and their applications". *Journal of Internet Services and Applications* **5**.1 (2014), p. 11.

[11] Audibert, J.-Y. & Munos, R. "Introduction to Bandits: Algorithms and Theory". *ICML Tutorial on bandits* (2011).

[12] *AWS EC2 Document History*. http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/DocumentHistory.html.

[13] Babu, S. & Herodotou, H. "Massively Parallel Databases and MapReduce Systems". *Foundations and Trends in Databases* **5**.1 (2013), pp. 1–104.

[14] Barford, P. & Crovella, M. "Generating Representative Web Workloads for Network and Server Performance Evaluation". *Proceedings of the 1998 ACM SIGMETRICS Joint Inter-*

national Conference on Measurement and Modeling of Computer Systems. SIGMETRICS '98/PERFORMANCE '98. Madison, Wisconsin, USA: ACM, 1998, pp. 151–160.

[15] Bergemann, D. & Valimaki, J. "Bandit problems". *Cowles Foundation Discussion Paper No. 1551* (2006).

[16] Bilal, M. & Canini, M. "Towards Automatic Parameter Tuning of Stream Processing Systems". *Proceedings of the 2017 Symposium on Cloud Computing*. SoCC '17. Santa Clara, California: ACM, 2017, pp. 189–200.

[17] Bodik, P. et al. "Automatic Exploration of Datacenter Performance Regimes". *Proceedings of the 1st Workshop on Automated Control for Datacenters and Clouds*. ACDC '09. Barcelona, Spain: ACM, 2009, pp. 1–6.

[18] Bodik, P. et al. "Characterizing, Modeling, and Generating Workload Spikes for Stateful Services". *Proceedings of the 1st ACM Symposium on Cloud Computing*. SoCC '10. Indianapolis, Indiana, USA: ACM, 2010, pp. 241–252.

[19] *Boto 3*. https://github.com/boto/boto3.

[20] Bowman, M. et al. "An Analysis of Performance Interference Effects in Virtual Environments". *2007 IEEE International Symposium on Performance Analysis of Systems and Software(ISPASS)*. Vol. 00. San Jose, CA, USA, 2007, pp. 200–209.

[21] Box, G. E. et al. *Time Series Analysis: Forecasting and Control*. John Wiley & Sons, 2015.

[22] Breiman, L. "Random Forests". *Machine Learning* **45**.1 (2001), pp. 5–32.

[23] Brochu, E. et al. "A Tutorial on Bayesian Optimization of Expensive Cost Functions, with Application to Active User Modeling and Hierarchical Reinforcement Learning". *ArXiv e-prints* (2010). arXiv: 1012.2599 [cs.LG].

[24] Cao, Z. et al. "Learning to Rank: From Pairwise Approach to Listwise Approach". *Proceedings of the 24th International Conference on Machine Learning*. ICML '07. Corvalis, Oregon, USA: ACM, 2007, pp. 129–136.

[25] *Capacity Scheduler*. http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/CapacityScheduler.html.

[26] *Ceph*. http://ceph.com.

[27] Chang, F. et al. "Bigtable: A Distributed Storage System for Structured Data". *ACM Transactions on Computer Systems (TOCS)* **26**.2 (2008), 4:1–4:26.

[28] Chapelle, O. et al. "Choosing Multiple Parameters for Support Vector Machines". *Machine Learning* **46**.1 (2002), pp. 131–159.

[29] Chen, Y. et al. “SLA Decomposition: Translating Service Level Objectives to System Level Thresholds”. *Proceedings of the Fourth International Conference on Autonomic Computing*. ICAC ’07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 3–.

[30] Cortez, E. et al. “Resource Central: Understanding and Predicting Workloads for Improved Resource Management in Large Cloud Platforms”. *Proceedings of the 26th Symposium on Operating Systems Principles*. SOSP ’17. Shanghai, China: ACM, 2017, pp. 153–167.

[31] *COSBench*. https://github.com/intel-cloud/cosbench.

[32] Cruz, F. et al. “MeT: Workload Aware Elasticity for NoSQL”. *Proceedings of the 8th ACM European Conference on Computer Systems*. EuroSys ’13. Prague, Czech Republic: ACM, 2013, pp. 183–196.

[33] Curino, C. et al. “Schism: A Workload-driven Approach to Database Replication and Partitioning”. *Proc. VLDB Endow.* **3**.1-2 (2010), pp. 48–57.

[34] Dalibard, V. et al. “BOAT: Building Auto-Tuners with Structured Bayesian Optimization”. *Proceedings of the 26th International Conference on World Wide Web*. WWW ’17. Perth, Australia: International World Wide Web Conferences Steering Committee, 2017, pp. 479–488.

[35] Dambreville, A. et al. “Load Prediction for Energy-Aware Scheduling for Cloud Computing Platforms”. *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*. Vol. 00. 2017, pp. 2604–2607.

[36] Dean, J. & Ghemawat, S. “MapReduce: Simplified Data Processing on Large Clusters”. *Proceedings of the 6th Conference on Symposium on Opearting Systems Design and Implementation - Volume 6*. OSDI’04. San Francisco, CA: USENIX Association, 2004, pp. 10–10.

[37] Dewancker, I. et al. *Bayesian Optimization Primer*. https://sigopt.com/static/pdf/SigOpt_Bayesian_Optimization_Primer.pdf. 2015.

[38] Dilley, J. et al. “Web server performance measurement and modeling techniques”. *Performance Evaluation* **33**.1 (1998). Tools for Performance Evaluation, pp. 5 –26.

[39] Domingos, P. “A Few Useful Things to Know about Machine Learning”. *Communications of the ACM* **55**.10 (2012), pp. 78–87.

[40] Efron, B. et al. “Least Angle Regression”. *The Annals of statistics* **32**.2 (2004), pp. 407–499.

[41] Eltabakh, M. Y. et al. “CoHadoop: Flexible Data Placement and Its Exploitation in Hadoop”. *Proceedings of the VLDB Endowment* **4**.9 (2011), pp. 575–585.

[42] *Fair Scheduler*. http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/FairScheduler.html.

[43] Foster, I. *Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995.

[44] Freeh, V. W. et al. "Analyzing the Energy-Time Trade-Off in High-Performance Computing Applications". *IEEE Transactions on Parallel and Distributed Systems* **18**.6 (2007), pp. 835–848.

[45] Frey, S. et al. "Search-based Genetic Optimization for Deployment and Reconfiguration of Software in the Cloud". *Proceedings of the 2013 International Conference on Software Engineering*. ICSE '13. San Francisco, CA, USA: IEEE Press, 2013, pp. 512–521.

[46] Friedman, J. et al. *The Elements of Statistical Learning*. Vol. 1. Springer series in statistics New York, 2001.

[47] George, L. *HBase: the Definitive Guide: Random Access to Your Planet-Size Data*. " O'Reilly Media, Inc.", 2011.

[48] Geurts, P. et al. "Extremely randomized trees". *Machine Learning* **63**.1 (2006), pp. 3–42.

[49] Gmach, D. et al. "Workload Analysis and Demand Prediction of Enterprise Data Center Applications". *Proceedings of the 2007 IEEE 10th International Symposium on Workload Characterization*. IISWC '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 171–180.

[50] Golovin, D. et al. "Google Vizier: A Service for Black-Box Optimization". *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. KDD '17. Halifax, NS, Canada: ACM, 2017, pp. 1487–1495.

[51] *Google VM rightsizing service*. https://cloud.google.com/compute/docs/instances/apply-sizing-recommendations-for-instances.

[52] Gregg, B. *perf-tools*. https://github.com/brendangregg/perf-tools.

[53] Guyon, I. & Elisseeff, A. "An introduction to variable and feature selection". *Journal of Machine Learning Research* **3** (2003), pp. 1157–1182.

[54] Harrell, F. E. "Ordinal Logistic Regression". *Regression Modeling Strategies*. Springer, 2001, pp. 331–343.

[55] Hastie, T. et al. "The Elements of Statistical Learning: Data Mining, Inference, and Prediction". *The Mathematical Intelligencer* **27**.2 (2005), pp. 83–85.

[56] Herodotou, H. et al. “Starfish: A Self-tuning System for Big Data Analytics”. *The 5th Biennial Conference on Innovative Data Systems Research (CIDR 2011)*. Vol. 11. 2011. 2011, pp. 261–272.

[57] Hey, T. et al. *The Fourth Paradigm: Data-Intensive Scientific Discovery*. Microsoft Research, 2009.

[58] *HiBench*. https://github.com/intel-hadoop/HiBench.

[59] *High performance computing cluster (HPCC)*. https://hpccsystems.com/.

[60] Hsu, C. et al. “Inside-Out: Reliable Performance Prediction for Distributed Storage Systems in the Cloud”. *2016 IEEE 35th Symposium on Reliable Distributed Systems(SRDS)*. Vol. 00. 2016, pp. 127–136.

[61] Hsu, C.-J. et al. “Scout: An Experienced Guide to Find the Best Cloud Configuration”. *ArXiv e-prints* (2018). arXiv: 1803.01296 [cs.DC].

[62] Hsu, C.-J. et al. “Arrow: Low-Level Augmented Bayesian Optimization for Finding the Best Cloud VM”. *The 38th IEEE International Conference on Distributed Computing Systems (ICDCS 2018)*. 2018.

[63] Hsu, C.-J. et al. “Micky: A Cheaper Alternative for Selecting Cloud Instances”. *The 11th IEEE International Conference on Cloud Computing (IEEE CLOUD 2018)*. 2018.

[64] Isard, M. et al. “Dryad: Distributed Data-parallel Programs from Sequential Building Blocks”. *Proceedings of the 2Nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*. EuroSys ’07. Lisbon, Portugal: ACM, 2007, pp. 59–72.

[65] Jalaparti, V. et al. “Bridging the Tenant-provider Gap in Cloud Services”. *Proceedings of the Third ACM Symposium on Cloud Computing*. SoCC ’12. San Jose, California: ACM, 2012, 10:1–10:14.

[66] Jamshidi, P. & Casale, G. “An Uncertainty-Aware Approach to Optimal Configuration of Stream Processing Systems”. *2016 IEEE 24th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)*. 2016, pp. 39–48.

[67] Jiang, J. et al. “Pytheas: Enabling Data-driven Quality of Experience Optimization Using Group-based Exploration-exploitation”. *Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation*. NSDI’17. Boston, MA, USA: USENIX Association, 2017, pp. 393–406.

[68] Kaelbling, L. P. et al. “Reinforcement Learning: A Survey”. *Journal of Artificial Intelligence Research* **4** (1996), pp. 237–285.

[69] Kelly, T. et al. *Inducing Models of Black-Box Storage Arrays Inducing Models of Black-Box Storage Arrays*. Tech. rep. HP Laboratories, 2004.

[70] Khajeh-Hosseini, A. et al. "Cloud Migration: A Case Study of Migrating an Enterprise IT System to IaaS". *2010 IEEE 3rd International Conference on Cloud Computing*. 2010, pp. 450–457.

[71] Kim, Y. et al. "Workload characterization of a leadership class storage cluster". *2010 5th Petascale Data Storage Workshop (PDSW '10)*. Vol. 00. 2011, pp. 1–5.

[72] Klein, A. et al. "Fast Bayesian Optimization of Machine Learning Hyperparameters on Large Datasets". *Proceedings of the 20th International Conference on Artificial Intelligence and Statistics (AISTATS 2017)*. Vol. 54. Proceedings of Machine Learning Research. PMLR, 2017, pp. 528–536.

[73] Kohavi, R. "A Study of Cross-validation and Bootstrap for Accuracy Estimation and Model Selection". *Proceedings of the 14th International Joint Conference on Artificial Intelligence - Volume 2*. IJCAI'95. Montreal, Quebec, Canada: Morgan Kaufmann Publishers Inc., 1995, pp. 1137–1143.

[74] Kouzes, R. T. et al. "The Changing Paradigm of Data-Intensive Computing". *Computer* **42**.1 (2009), pp. 26–34.

[75] Kundu, S. et al. "Application performance modeling in a virtualized environment". *2010 The Sixteenth International Symposium on High-Performance Computer Architecture (HPCA-16)*. 2010, pp. 1–10.

[76] Lakshman, A. & Malik, P. "Cassandra: A Decentralized Structured Storage System". *ACM SIGOPS Operating Systems Review* **44**.2 (2010), pp. 35–40.

[77] Li, P. et al. "McRank: Learning to Rank Using Multiple Classification and Gradient Boosting". *Proceedings of the 20th International Conference on Neural Information Processing Systems*. NIPS'07. Vancouver, British Columbia, Canada: Curran Associates Inc., 2007, pp. 897–904.

[78] Lim, H. C. et al. "Automated Control for Elastic Storage". *Proceedings of the 7th International Conference on Autonomic Computing*. ICAC '10. Washington, DC, USA: ACM, 2010, pp. 1–10.

[79] Malewicz, G. et al. "Pregel: A System for Large-scale Graph Processing". *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*. SIGMOD '10. Indianapolis, Indiana, USA: ACM, 2010, pp. 135–146.

[80] Maltzahn, C. et al. "Ceph as a scalable alternative to the Hadoop Distributed File System". *;login:* **35**.4 (2010), pp. 38–49.

[81] Melnik, S. et al. “Dremel: Interactive Analysis of Web-scale Datasets”. *Proceedings of the VLDB Endowment* **3**.1-2 (2010), pp. 330–339.

[82] Mesnier, M. P. et al. “Modeling the Relative Fitness of Storage”. *Proceedings of the 2007 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*. SIGMETRICS ’07. San Diego, California, USA: ACM, 2007, pp. 37–48.

[83] Middleton, A. & Chala, A. “HPCC systems: Introduction to HPCC (High-Performance Computing Cluster)”. *White paper, LexisNexis Risk Solutions* (2011).

[84] Mihailescu, M. et al. “MixApart: Decoupled Analytics for Shared Storage Systems”. *Proceedings of the 4th USENIX Conference on Hot Topics in Storage and File Systems*. HotStorage’12. Boston, MA: USENIX Association, 2012, pp. 2–2.

[85] Moler, C. “Matrix computation on distributed memory multiprocessors”. *Hypercube Multiprocessors* **86**.181-195 (1986), p. 31.

[86] Nair, V. et al. “Using Bad Learners to Find Good Configurations”. *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ESEC/FSE 2017. Paderborn, Germany: ACM, 2017, pp. 257–267.

[87] Newman, M. “Power laws, Pareto distributions and Zipf’s law”. *Contemporary Physics* **46**.5 (2005), pp. 323–351.

[88] Niculescu-Mizil, A. & Caruana, R. “Predicting Good Probabilities with Supervised Learning”. *Proceedings of the 22nd International Conference on Machine Learning*. ICML ’05. Bonn, Germany: ACM, 2005, pp. 625–632.

[89] Noorshams, Q. et al. “Predictive Performance Modeling of Virtualized Storage Systems Using Optimized Statistical Regression Techniques”. *Proceedings of the 4th ACM/SPEC International Conference on Performance Engineering*. ICPE ’13. Prague, Czech Republic: ACM, 2013, pp. 283–294.

[90] Novaković, D. et al. “DeepDive: Transparently Identifying and Managing Performance Interference in Virtualized Environments”. *Proceedings of the 2013 USENIX Conference on Annual Technical Conference*. USENIX ATC’13. San Jose, CA: USENIX Association, 2013, pp. 219–230.

[91] Oh, J. et al. “Finding Near-optimal Configurations in Product Lines by Random Sampling”. *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ESEC/FSE 2017. Paderborn, Germany: ACM, 2017, pp. 61–71.

[92] *Open Performance Dataset*. https://github.com/oxhead/scout.

[93] *OpenStack*. urlhttp://www.openstack.org.

[94] Ousterhout, K. et al. “Monotasks: Architecting for Performance Clarity in Data Analytics Frameworks”. *Proceedings of the 26th Symposium on Operating Systems Principles*. SOSP ’17. Shanghai, China: ACM, 2017, pp. 184–200.

[95] Pan, S. J. & Yang, Q. “A Survey on Transfer Learning”. *IEEE Transactions on Knowledge and Data Engineering* **22**.10 (2010), pp. 1345–1359.

[96] Panteleenko, V. V. & Freeh, V. W. “Web server performance in a WAN environment”. *The 12th International Conference on Computer Communications and Networks (ICCCN 2003)*. IEEE. 2003, pp. 364–369.

[97] Pavlo, A. et al. “Skew-aware Automatic Database Partitioning in Shared-nothing, Parallel OLTP Systems”. *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*. SIGMOD ’12. Scottsdale, Arizona, USA: ACM, 2012, pp. 61–72.

[98] Peters, F. et al. “LACE2: Better Privacy-preserving Data Sharing for Cross Project Defect Prediction”. *Proceedings of the 37th International Conference on Software Engineering - Volume 1*. ICSE ’15. Florence, Italy: IEEE Press, 2015, pp. 801–811.

[99] Porter, G. “Decoupling Storage and Computation in Hadoop with SuperDataNodes”. *ACM SIGOPS Operating Systems Review* **44**.2 (2010), pp. 41–46.

[100] Rasmussen, C. E. “Gaussian Processes in Machine Learning”. *Advanced Lectures on Machine Learning: ML Summer Schools 2003, Canberra, Australia, February 2 - 14, 2003, Tübingen, Germany, August 4 - 16, 2003, Revised Lectures*. Ed. by Bousquet, O. et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 63–71.

[101] Robbins, H. “Some Aspects of the Sequential Design of Experiments”. *Bulletin of the American Mathematical Society* (1985), pp. 527–535.

[102] Rodrigues, J. P. et al. “AUTOPLACER: Scalable Self-Tuning Data Placement in Distributed Key-value Stores”. *Proceedings of the 10th International Conference on Autonomic Computing (ICAC 2007)*. San Jose, CA, USA, 2013, pp. 119–131.

[103] Ruan, X. et al. “Improving MapReduce performance through data placement in heterogeneous Hadoop clusters”. *2010 IEEE International Symposium on Parallel and Distributed Processing, Workshops and Phd Forum (IPDPSW)*. Vol. 00. 2010, pp. 1–9.

[104] Ruemmler, C. & Wilkes, J. “An Introduction to Disk Drive Modeling”. *Computer* **27**.3 (1994), pp. 17–28.

[105] Saeys1, Y. et al. “A review of feature selection techniques in bioinformatics”. *Bioinformatics* **23** (19 2007), pp. 2507–2517.

[106] *scikit-learn*. http://scikit-learn.org.

[107] Shafer, J. "A Storage Architecture for Data-Intensive Computing". PhD thesis. Rice University, 2010.

[108] Shahriari, B. et al. "Taking the Human Out of the Loop: A Review of Bayesian Optimization". *Proceedings of the IEEE* **104**.1 (2016), pp. 148–175.

[109] Shlens, J. "A tutorial on principal component analysis: derivation, discussion and singular value decomposition" (2003), pp. 1–16.

[110] Shriver, E. et al. "An Analytic Behavior Model for Disk Drives with Readahead Caches and Request Reordering". *Proceedings of the 1998 ACM SIGMETRICS Joint International Conference on Measurement and Modeling of Computer Systems*. SIGMETRICS '98/PERFORMANCE '98. Madison, Wisconsin, USA: ACM, 1998, pp. 182–191.

[111] Snoek, J. et al. "Practical Bayesian Optimization of Machine Learning Algorithms". *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 2*. NIPS'12. Lake Tahoe, Nevada: Curran Associates Inc., 2012, pp. 2951–2959.

[112] Sobol, I. M. "On quasi-Monte Carlo integrations". *Mathematics and Computers in Simulation* **47**.2 (1998), pp. 103–112.

[113] *Software-Defined Storage*. http://www.research.att.com/articles/featured_stories/2015_09/software-defined-storage.html.

[114] *spark-perf*. https://github.com/databricks/spark-perf.

[115] Sripanidkulchai, K. et al. "An Analysis of Live Streaming Workloads on the Internet". *Proceedings of the 4th ACM SIGCOMM Conference on Internet Measurement*. IMC '04. Taormina, Sicily, Italy: ACM, 2004, pp. 41–54.

[116] Sripanidkulchai, K. et al. "Are clouds ready for large distributed applications?" *ACM SIGOPS Operating Systems Review* **44**.2 (2010), pp. 18–23.

[117] *sysstat*. http://sebastien.godard.pagesperso-orange.fr.

[118] Tang, L. et al. "The Impact of Memory Subsystem Resource Sharing on Datacenter Applications". *Proceedings of the 38th Annual International Symposium on Computer Architecture*. ISCA '11. San Jose, California, USA: ACM, 2011, pp. 283–294.

[119] Tantisiriroj, W. et al. "On the Duality of Data-intensive File System Design: Reconciling HDFS and PVFS". *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*. SC '11. Seattle, Washington: ACM, 2011, 67:1–67:12.

[120] Thereska, E. et al. “IOFlow: A Software-defined Storage Architecture”. *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. SOSP ’13. Farminton, Pennsylvania: ACM, 2013, pp. 182–196.

[121] Trushkowsky, B. et al. “The SCADS Director: Scaling a Distributed Storage System Under Stringent Performance Requirements”. FAST’11 (2011), pp. 12–12.

[122] Urdaneta, G. et al. “Wikipedia workload analysis for decentralized hosting”. *Computer Networks* **53**.11 (2009), pp. 1830–1845.

[123] Venkataraman, S. et al. “Ernest: Efficient Performance Prediction for Large-scale Advanced Analytics”. *Proceedings of the 13th Usenix Conference on Networked Systems Design and Implementation*. NSDI’16. Santa Clara, CA: USENIX Association, 2016, pp. 363–378.

[124] Wang, M. et al. “Storage Device Performance Prediction with CART Models”. *Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems*. SIGMETRICS ’04/Performance ’04. New York, NY, USA: ACM, 2004, pp. 412–413.

[125] Wang, Z. & Jegelka, S. “Max-value Entropy Search for Efficient Bayesian Optimization”. *Proceedings of the 34th International Conference on Machine Learning*. Ed. by Precup, D. & Teh, Y. W. Vol. 70. Proceedings of Machine Learning Research. International Convention Centre, Sydney, Australia: PMLR, 2017, pp. 3627–3635.

[126] Wauthier, F. L. et al. “Efficient Ranking from Pairwise Comparisons”. *Proceedings of the 30th International Conference on International Conference on Machine Learning - Volume 28*. ICML’13. Atlanta, GA, USA: JMLR.org, 2013, pp. III–109–III–117.

[127] Weber, R. “On the Gittins index for multiarmed bandits”. *The Annals of Applied Probability* (1992), pp. 1024–1033.

[128] Weil, S. A. et al. “Ceph: A Scalable, High-performance Distributed File System”. *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*. OSDI ’06. Seattle, Washington: USENIX Association, 2006, pp. 307–320.

[129] Wettschereck, D. et al. “A Review and Empirical Evaluation of Feature Weighting Methods for a Class of Lazy Learning Algorithms”. *Artificial Intelligence Review* **11**.1-5 (1997), pp. 273–314.

[130] Wiers, D. *Dstat: Versatile resource statistics tool*. http://dag.wiee.rs/home-made/dstat/.

[131] Wilkes, J. “Traveling to Rome: QoS Specifications for Automated Storage System Management”. *Proceedings of the 9th International Workshop on Quality of Service*. IWQoS ’01. Berlin, Heidelberg: Springer-Verlag, 2001, pp. 75–91.

[132] *Windows Azure*. http://www.windowsazure.com/.

[133] Yadwadkar, N. J. et al. “Selecting the Best VM Across Multiple Public Clouds: A Data-driven Performance Modeling Approach”. *Proceedings of the 2017 Symposium on Cloud Computing*. SoCC ’17. Santa Clara, California: ACM, 2017, pp. 452–465.

[134] Yao, Y. et al. “Complexity vs. Performance: Empirical Analysis of Machine Learning As a Service”. *Proceedings of the 2017 Internet Measurement Conference*. IMC ’17. London, United Kingdom: ACM, 2017, pp. 384–397.

[135] Yin, L. et al. “An Empirical Exploration of Black-Box Performance Models for Storage Systems”. *Proceedings of the 14th IEEE International Symposium on Modeling, Analysis, and Simulation*. MASCOTS ’06. Washington, DC, USA: IEEE Computer Society, 2006, pp. 433–440.

[136] Zadrozny, B. & Elkan, C. “Obtaining Calibrated Probability Estimates from Decision Trees and Naive Bayesian Classifiers”. *Proceedings of the Eighteenth International Conference on Machine Learning*. ICML ’01. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2001, pp. 609–616.

[137] Zadrozny, B. & Elkan, C. “Transforming Classifier Scores into Accurate Multiclass Probability Estimates”. *Proceedings of the Eighth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. KDD ’02. Edmonton, Alberta, Canada: ACM, 2002, pp. 694–699.

[138] Zaharia, M. et al. “Spark: Cluster Computing with Working Sets”. *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*. HotCloud’10. Boston, MA: USENIX Association, 2010, pp. 10–10.

[139] Zaharia, M. et al. “Resilient Distributed Datasets: A Fault-tolerant Abstraction for In-memory Cluster Computing”. *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*. NSDI’12. San Jose, CA: USENIX Association, 2012, pp. 2–2.

[140] Zhu, Y. et al. “BestConfig: Tapping the Performance Potential of Systems via Automatic Configuration Tuning”. *Proceedings of the 2017 Symposium on Cloud Computing*. SoCC ’17. Santa Clara, California: ACM, 2017, pp. 338–350.

[141] Zuluaga, M. et al. “ε-PAL: An Active Learning Approach to the Multi-Objective Optimization Problem”. *The Journal of Machine Learning Research* **17**.1 (2016), pp. 3619–3650.