A Practical Auto-Tuning Framework for Storage Systems

A Dissertation Presented

by

Zhen Cao

to

The Graduate School

in Partial Fulfillment of the

Requirements

for the Degree of

Doctor of Philosophy

in

Computer Science

Stony Brook University

Technical Report FSL-19-01

January 2019

Abstract

A Practical Auto-Tuning Framework for Storage Systems

by

Zhen Cao

Doctor of Philosophy
in
Computer Science

Stony Brook University
2019

Storage systems come with a large number of configurable parameters that control their behavior. Tuning such parameters can provide significant gains in performance, but is challenging due to huge spaces and complex, non-linear system behavior. Auto-tuning with black-box optimization have shown promising results in recent years, thanks to its obliviousness to systems' internals.

However, previous work all applied only one or few optimization methods, and did not systematically evaluate them. Therefore, in this thesis, we first apply and then perform comparative analysis of multiple black-box optimization techniques on storage systems from various aspects such as their ability to find near-optimal configurations, convergence time, and instantaneous system throughput during auto-tuning, etc. We also provide insights into the efficacy of these automated black-box optimization methods from a system's perspective.

During our auto-tuning experiments, we noticed that sometimes multiple runs of the same workload—in a carefully controlled environment—produced widely different performance results. So next, we undertook a study to characterize the amount of variability in modern storage systems. We analyzed these variations and found that there was no single root cause: it often changed with the workload, hardware, or software configuration in the storage system. In several of those cases we were able to fix the cause of variation and reduce it to acceptable levels.

We believe several critical features are still missing from traditional black-box optimization methods. In this thesis we designed a practical framework for auto-tuning storage systems. We propose an efficient parameter-selection algorithm, Spectra, to eliminate unimportant parameters. It successfully identified important parameters for all file systems and showed that importance varies with different workloads. We demonstrated Spectra's efficiency by testing it with a small fraction of our dataset. We co-developed a suitable visual analytic tool, Interactive Configuration Explorer (ICE), to help explore large parameter spaces, identify critical parameters, and quickly zero in on optimal parameter settings. We added a workload modeler in the framework and showed that the feasibility of workload characterization using hundreds of collected block traces. Our framework categorizes storage parameters, to account for costly configuration changes. We also compared different initialization and stopping methods for our auto-tuning framework.

It is our thesis that auto-tuning storage systems is important, promising, and feasible with a carefully designed framework to include missing yet critical features. This can improve systems' performance efficiency, and save energy and human resources in the long term.

Contents

List of Figures

List of Tables

Acknowledgments

This thesis cannot be accomplished without the help of the following people. I thank my parents and my fiancee Xin Li for supporting and accompanying me in these years. They've made the long journey easier and meaningful for me.

I'd like to thank my advisor Prof. Erez Zadok for advising me on this thesis and gave me invaluable advice throughout the last five years. He helped me in every aspect of my PhD study, including research methodology, paper writing, technical presentation, etc. I also appreciate his patience and kindness when I made mistakes.

I also thank the external collaborators, Vasily Tarasov from IBM Research and Dead Hildebrand from Google. They provided guidance and comments on many aspects of the projects, and gave me memorable internship experience as well.

I'd like to thank FSL students who have contributed to this work. They are Umit Ibrahim Akgun, Tyler Estro, Ankit Aggarwal, Saish Sali, Prateek Roy, Sachin Tiwari, Hari Prasath Raman, Akhilesh Chaganti, Sonam Mandal, Aashray Arora, Arvind Chaudhary. They helped me a lot in many aspects of these projects. I also specially appreciate the help from Cong Xie and Anjul Tyagi of VAL lab. We coordinated a lot on the visualization of our datasets.

I'd like to thank Prof. Scott Stoller, Prof. Klaus Mueller, and Prof. Geoff Kuenning from Harvey Mudd College for serving on the committee of my dissertation. They are very knowledgeable and helped this thesis significantly with insightful comments and valuable feedbacks.

Chapter 1

Introduction

Storage is a critical element of computer systems and key to data-intensive applications. Storage systems come with a vast number of configurable parameters that control a system's behavior. Ext4 alone has around 60 parameters with whopping 10^{37} unique combinations of values. Default parameter settings provided by vendors are often suboptimal for a specific user deployment; previous research showed that tuning even a small subset of parameters can improve power and performance efficiency of storage systems by as much as 9× [170].

Traditionally, system administrators pick parameter settings based on their expertise and experience. Due to the increased complexity of storage systems, however, manual tuning becomes intractable, error-prone, and has a low chance of finding an optimal configuration. A myriad of file systems with diverse goals and designs have been developed [59, 103, 110, 161, 185]. Newer types of devices (SSDs [77, 138], SMR drives [2, 3], PCM [99, 208]) and more layers (LVM, RAID) are added. Storage systems expand from one or few identical nodes to hundreds of highly heterogeneous environments [67, 167]. Tuning results from one workload are often inapplicable in another [28, 198]. Furthermore, the composition of hardware and workload in a modern environment changes at a fast pace that prohibits timely manual tuning.

In recent years, several attempts were made to automate the tuning of computer systems in general and storage systems in particular [181, 198]. Black-box auto-tuning is an especially popular approach thanks to its obliviousness to system's internals [213]. The basic mechanism behind black-box auto-tuning is to iteratively try different configurations, measure an objective function's value—and based on the previously learned information—select the next configurations to try. For storage systems, objective functions can be throughput, I/O latency, energy consumption, purchase cost, or even a formula combining multiple metrics [123, 181]. Many black-box auto-tuning algorithms exist and some were applied to systems. Genetic Algorithms (GA) were applied to optimize the I/O performance of HDF5-based applications [12]. Bayesian Optimization (BO) was used to find a near-optimal configuration for Cloud VMs [5]. Other methods include Evolutionary Strategies [163], Smart Hill-Climbing [209], and Simulated Annealing [53]. Although these methods were originally proposed in different scientific disciplines, they all maintain a trade-off among three behavioral dimensions: (1) *Exploration*: how much the technique searches the space randomly. (2) *Exploitation*: how much the technique leverages the "neighborhood" of the current candidate or previous search history to find even better configurations. (3) *History*: how much data from previous evaluations is kept and utilized in the overall search process. For this dissertation, we investigated and designed a general framework based on black-box optimization, which can efficiently auto-tune storage systems.

To demonstrate black-box optimization's ability to find optimal (or at least near-optimal) storage configurations, we started by exhaustively evaluating several storage systems under four workloads on two servers with different hardware and storage devices; the largest system consisted of 6,222 unique configurations. Over a period of 3+ years, we executed 500,000+ experimental runs, with 26 different combination of workload and hardware settings. We stored all data points in a relational database for query convenience, including hardware and workload details, throughput, energy consumption, running time, etc. In this thesis, we mainly focused on optimizing for throughput, but our methodology and observations are applicable to other metrics as well. We released and will continue updating our dataset publicly to facilitate more research into auto-tuning and better understanding of storage systems.

Despite some appealing results in auto-tuning, there is no deep understanding how exactly these black-box optimization methods work, their efficacy and efficiency, and which methods are more suitable for which problems. Previous works picked algorithms somewhat arbitrarily and evaluated only one algorithm at a time. Therefore, in this thesis, for the first time and to the best of our knowledge, we apply and analytically compare *multiple* black-box optimization techniques on storage systems. We applied several popular techniques to the collected dataset to find optimal configurations under various hardware and workload settings: Simulated Annealing (SA), Genetic Algorithms (GA), Bayesian Optimization (BO), and Deep Q-Networks (DQN). We also tried Random Search (RS) in our experiments, which showed surprisingly good results in previous research [15]. We compared these techniques from various aspects, such as their ability to find near-optimal configurations, convergence time, and instantaneous system throughput during auto-tuning. For example, we found that several techniques were able to converge to good configurations given enough time, but their efficacy differed a lot. GA and BO outperformed SA and DQN on our parameter spaces, both in terms of convergence time and instantaneous throughputs. Surprisingly, RS was also able to identify good configurations, sometimes even more efficiently than sophisticated optimization methods. We further compared the techniques across the aforementioned three behavioral dimensions: *exploration*, *exploitation*, and *history*. Based on our experimental results and domain expertise, we also provide explanations of efficacy of such black-box optimization methods from a storage perspective. We observed that certain parameters would have a greater effect on system performance than others, and the set of dominant parameters depends on file systems and workloads.

During our auto-tuning experiments, we noticed that sometimes multiple runs of the same workload—in a carefully controlled environment—produced widely different performance results. In one experiment setting, over 18% of 6,222 different storage configurations that we tried exhibited a standard deviation of performance larger than 5% of the mean, and a range value (maximum minus minimum performance, divided by the average) that exceeding 9%. In a few extreme cases, the standard deviation exceeded 40% even with numerous repeated experiments. This motivated us to conduct a more detailed study of storage system performance variation and seek its root causes, as performance stability is important for the success of auto-tuning and more broadly is critical in modern storage systems. Therefore, in this thesis we conducted experiments on three local file systems (Ext4, XFS, and Btrfs) which are used in many modern local and distributed environments. We benchmarked over 100 configurations using different workloads and repeated each experiment 10 times to balance the accuracy of variation measurement with the total time taken to complete these experiments. We then characterized performance variation from several angles: throughput, latency, temporally, spatially, and more. We found that performance variation depends heavily on the specific configuration of the storage system. We then further dove into the details, analyzed and explained certain performance variations. For example, we found that unpredictable layouts in Ext4 could cause over 16–19% of performance variation in some cases. Finally, we analyzed latency variations from various aspects, and proposed a novel approach for quantifying the impacts of each operation type on overall performance variation.

The huge parameter space is one of the challenges in auto-tuning storage systems. In machine learning and information theory, dimensionality reduction is often applied to explosively sized datasets [17, 146]. We believe it can also be applied to storage-parameter selection. By eliminating the less important parameters, the parameter search space—and thus the number of configurations that need to be considered by either humans or algorithms—can be massively reduced [84]. Given these observations, we decided to investigate the practicality of parameter selection for storage systems and to design Spectra, a system that uses a variance-based metric to quantify the importance of storage parameters, applying a greedy algorithm that can *automatically* and *efficiently* identify important parameters while evaluating only a small number of configurations. We then combined Spectra with Latin Hypercube Sampling (LHS) [109, 143], allowing Spectra to identify the set of important parameters using only a small number of experimental runs that explored only a fraction of all configurations. For instance, among all 1,000 repeated runs, Spectra was able to find the two most important parameters for Ext4 using only 32 evaluations.

We believe traditional black-box optimization techniques still lack several critical features to achieve practical, auto-tuning in storage systems. To address these limitations, we propose our practical auto-tuning framework, based on all previous work, and adding several new features. We prototyped a workload modeler, which can extract features from system-collected metrics and characterize the running workload based on them. We categorize each parameter based on its *changing cost*, and showed how our auto-tuning framework will optimize storage systems within certain categories of parameters. We also compared the efficacy of multiple initialization methods and stopping criteria with our framework. Our auto-tuning framework is also equipped with a visual analytic tool, the Interactive Configuration Explorer (ICE), which is designed to help system administrators understanding auto-tuning results and the complex behavior of storage systems.

The rest of this dissertation is organized as follows. Chapter 2 describes challenges of auto-tuning storage systems and background knowledge on black-box optimization. Chapter 3 discusses related work. We list our experimental settings in Chapter 4. In Chapter 5 we perform a comparative analysis on multiple optimization methods. Chapter 6 provides our characterization work on performance variation in modern storage stocks. Chapter 7 describes our parameter selection algorithm Spectra and its evaluation results. Chapter 8 explains the design of ICE and several case studies that we conducted with it. Chapter 9 discusses several missing yet important components from traditional black-box optimization. Based on it, we added several features and summarized our auto-tuning framework for storage systems. Chapter 10 concludes this thesis.

Chapter 2

Background

In this thesis we use "storage systems" to refer file systems, underlying storage hardware and any layers between them. Storage systems have always been a critical component of most computer systems, and are the foundation for many data-intensive applications. Usually they come with a large number of configurable options that could affect or even determine the systems' performance [28,188], energy consumption [170], and other aspects [120,181]. Here we define a *parameter* as one configurable option, and a *configuration* as a certain combination of parameter values. For example, the *journal_mode* is one *parameter* for Ext4, with 3 possible values: *data=writeback*, *data=ordered*, and *data=journal*. Two other common parameters are *block_size* and *inode_size* with several possible numeric values (e.g., 4K, 8K). *[journal_mode="data=writeback", block_size=4K, inode_size=4K]* is one *configuration* with 3 specific parameters: *journal mode*, *block size*, and *inode size*. All possible configurations form a *parameter space*.

When configuring storage systems, users often stick with the default configurations provided by vendors because

- it is nearly impossible to know the impact of every parameter across multiple layers; and
- vendors' default configurations are trusted to be safe and "good enough".

However, previous studies [170] showed that tuning even a tiny subset of parameters could improve the performance and energy efficiency for storage systems by as much as 9×. As Moore's law slows down, it becomes even more important to squeeze every bit of performance out of deployed storage systems.

The rest of this chapter is organized as follows. We first discuss the challenges of storage system tuning in Section 2.1. Then, Section 2.2 briefly introduces several black-box optimization techniques that we explore in this thesis. Section 2.3 discusses how Machine Learning (ML) techniques can help in auto-tuning storage systems. Section 2.4 provides a unified view of these optimization methods.

2.1 Problem Statement

The tuning task for storage systems is difficult, due to the following four challenges.

(1) Large parameter space Modern storage systems are fairly complex and easily come with hundreds or even thousands of tunable parameters. This makes it impossible to explore even a small fraction of the parameter space exhaustively. Even human experts or file-system developers cannot know the exact impact of every parameter and thus have little insight into how to optimize them. For example, Ext4 + NFS alone would result in a parameter space consisting of more than 10^{22} unique configurations. IBM's General Parallel File System (GPFS) [167] contains more than 100 tunable parameters, and hence 10^{40} configurations. From the hardware perspective, which also constitutes part of parameter space, SSDs [77, 138, 148, 168], SMRs [2, 3, 81, 118], and PCM [99, 208] are gaining popularity and more layers (LVM, RAID) are added to storage systems.

(2) Discrete and non-numeric parameters Among storage system parameters, some can take a continuous spectrum of values, while many others are discrete and take only a limited set of values. Some parameters do not even have numeric values (e.g., I/O scheduler name or file system type). These types of parameters make gradient-based information for objective functions (e.g., linear regression) unavailable.

(3) Non-linearity A system is *non-linear* when the output is not directly proportional to the input. Many computer systems are non-linear [41], including storage systems [188]. For example, Figure 2.1 shows the average operation latency of GPFS under a typical database server workload while changing only the value of the parameter *pagepool* and setting all the others to their default. We changed the *pagepool* size from 32MB to 128MB in steps of 8MB. Clearly the average latency is not directly proportional to the *pagepool* size. In fact, through our experiments, we have seen many more parameters with similar behavior. Worse, parameter spaces for storage systems are often sparse, irregular, and contains multiple peaks. This makes optimization even more challenging, as it has to avoid getting stuck in a local optima [90].

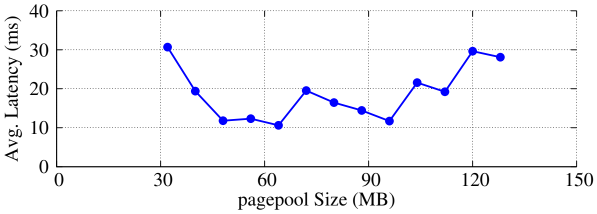


Figure 2.1: Storage systems are non-linear

(4) Non-reusable results Previous studies have shown that evaluation results of storage systems [28, 170] and databases [198] are dependent on the specific hardware and workloads. One good configuration might perform poorly when the environment changes. Figure 2.2 shows the I/O throughput under 4 different workloads with default configurations for Ext4, XFS, Btrfs, and Reiserfs—all on the same hardware. Under the *Mail Server* workload, the default XFS configuration performs best among these four configurations; but with the *Database Server*, Btrfs produces the highest throughput. In addition, these four configurations show similar results under the *Web Server* workload. We observed similar behavior when the hardware changed.

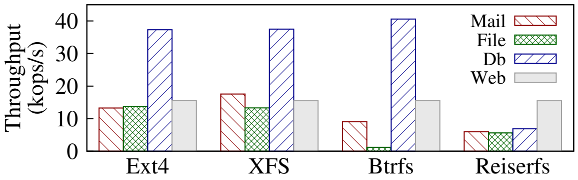


Figure 2.2: Evaluation results depend on workloads

Given these challenges, manual tuning of storage systems becomes nearly impossible while efficient automatic tuning is challenging. In this thesis we propose to design a practical auto-tuning framework for storage systems. We treat auto-tuning storage configurations as an optimization problem, and use the terms *"auto-tune"* and *"optimize"* interchangeably. Our framework is general enough to optimize for any user-specified objective, as long as possible outputs of the objective function form a totally ordered set. Examples of optimization objectives include maximizing throughput, minimizing average latency, minimizing energy consumption, etc. It can even be a complex formula combining several metrics together [123]. In this thesis we will mainly focus on auto-tuning storage systems for maximizing throughput, but our methodology and observations are applicable to other objectives as well.

Many previous efforts have been made and various techniques have been applied to parameter tuning problems. Control Theory (CT) was historically used to manage linear system parameters. CT builds a controller for a system, called the plant, so its output follows a desired control signal, called the reference [82, 111]. CT has been applied to database systems [50] and storage systems [93, 112] to provide QoS guarantees. However, CT has been shown to have the following three problems: 1) CT tends to be unstable in controlling non-linear systems [121, 122]. Although some variants were proposed for non-linear ones, they do not scale well. 2) CT cannot handle non-numeric parameters; and 3) CT requires an expensive learning phase, called *identification* to build a good controller, which requires having lots of data to learn from.

Supervised Machine Learning (ML) have been applied in black-box storage device modeling and prediction [201]. However, a well-known problem for supervised ML techniques is that they usually require a long training period and a large amount of data to build models; the models' quality depends heavily on the quality and amount of training data [201]. This data is not available or impossible to collect for large parameter spaces such as ours. Moreover, once the environment changes, the training data collected before it becomes invalid.

Based on the above reasons, we feel that neither CT nor supervised ML can be *directly and efficiently* applied for auto-tuning storage systems in its current state. Still, it was shown that many optimization techniques share some similarities with supervised Machine Learning [213]. Moreover, sub-disciplines of ML, including Online Learning [7, 175] and Active Learning [172], are evolving and gaining interests. They are practical and useful in solving certain problems where data becomes available incrementally. We believe ML techniques can still play an important role in our auto-tuning framework. Therefore, We provide a general introduction to ML in Section 2.3.

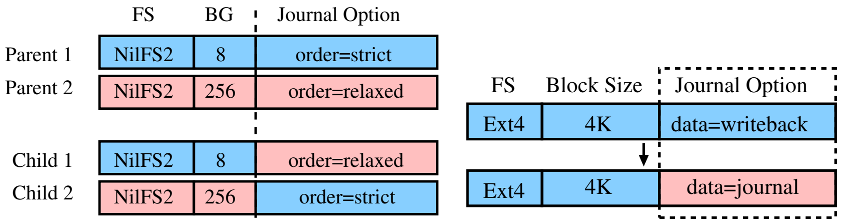
2.2 Black-box Optimization

Several classes of algorithms have been proposed for optimization tasks, including automated tuning of hyper-parameters of machine learning systems [14, 15, 156] and optimization of physical systems [5, 198]. Examples include Genetic Algorithms (GA) [45, 84], Simulated Annealing (SA) [32,101], Bayesian Optimization (BO) [22,174], etc. Although these methods were proposed originally in different scholarly fields, they can all be characterized as black-box optimizations. In this section we introduce several of these techniques that we successfully applied in auto-tuning storage systems.

Simulated Annealing (SA) is inspired by the annealing process in metallurgy. Annealing involves the heating and controlled cooling of a material to get to a state with minimum thermodynamic free energy to enhance, e.g., metal conductivity. When applied to storage systems, a *state* corresponds to one *configuration*. *Neighbors* of a state refer to new configurations achieved by altering only one parameter value of the current state. The thermodynamic free energy is analogous to user-defined optimization objectives. SA works by maintaining the *temperature* of the system, which determines the probability of accepting a certain move. Instead of always moving towards better states as hill-climbing methods do, SA defines an *acceptance probability distribution*, which allows it to accept some bad moves in the short run, that can lead to even-better moves later on. The system is initialized with a high temperature, and thus has high probability of accepting worse states in the beginning. The temperature is gradually reduced based on a pre-defined *cooling schedule*, thus reducing the probability of accepting bad states over time. SA has been applied in various areas and proved efficient in solving different types of problems, including the Traveling Salesman Problem (TSP) [1, 132, 199], Very Large Scale Integration (VLSI) design [169, 207], and network design [63, 64, 89].

Genetic Algorithms (GA) were proposed in 1975 [84] and inspired by the process of natural selection. GA maintains a population of *chromosomes* (configurations) and applies several genetic operators to them. *Crossover* takes two parent chromosomes and generates new ones. As Figure 2.3(a) illustrates, two parent Nilfs2 configurations are cut at the same *crossover* point, and then the subparts after the crossover point are exchanged between them to generate two new child configurations. Better chromosomes will have a higher probability to "survive" in future *selection* phases. *Mutation* randomly picks a chromosome and mutates one or more parameter values, which produces a completely different chromosome. Figure 2.3(b) illustrates such mutation, where the journal option is randomly mutated from *writeback* to *journal*. GA and its variants have been widely applied to various areas including the Traveling Salesman Problem (TSP) [71, 73, 108, 145, 159, 180], VLSI Design [16, 42, 126, 133], High-Performance Computing [11], and system design [36, 46, 128].

Bayesian Optimization (BO) [22,174] is a popular framework to solve optimization problems. It models the objective function as a stochastic process, with the argument corresponding to one storage configuration. In the beginning, a set of prior points (configurations) are given to the algorithm to get a fair estimate of the entire parameter space. BO works by computing the *confidence interval* of the objective function according to previous evaluation results. Here the *confidence interval* is the range of values that the evaluation result is most likely to fall into (e.g., with 95% probability). The next configuration is selected based on a pre-defined *acquisition function*. Both confidence intervals and the acquisition function are updated with each new evaluation. BO has been successfully applied in various areas, including hyper-parameter optimization [44] and sys-



(a) Crossover (b) Mutation

Figure 2.3: Crossover and mutation in a Genetic Algorithm

tem configuration optimization [5]. BO and its variants differ mainly in their form of probabilistic models and acquisition functions. In this thesis our evaluation results focus mainly on Gaussian priors and an Expected Improvement acquisition function [174].

Other promising black-box optimization techniques include Tabu Search [68–70], Particle Swarm Optimization [40,97,98], Ant Colony Optimization [51,52], and Memetic Algorithms [106, 137], etc. Most of them are nature-inspired as they have been developed based on the successful evolutionary behavior of natural systems. In this thesis, we focused on several representative algorithms, SA, GA, and BO. We plan to experiment with more techniques in the future (part of our future work). In fact, as detailed in §2.4, most of these techniques actually share similar traits.

2.3 Machine Learning

As we enter the era of big data, Machine Learning (ML) has becoming more popular in the last few decades. We can define ML as a set of methods that can automatically detect patterns in data, and then use the discovered patterns to predict future behavior, or to perform other kinds of decision making under uncertainty [146]. Generally, there are three types of ML techniques: *Supervised Learning*, *Unsupervised Learning*, and *Reinforcement Learning*.

Supervised Learning Supervised Learning is sometimes also called *predictive learning*, and its goal is to learn the mapping from the inputs $\overrightarrow{x}$ to outputs y, based on a labeled set of input-output pairs $D = \{(\overrightarrow{x_i}, y_i)\}_{i=1}^{N}$. D is often referred as a *training set* consisting of N training examples. In the training set, each input $\overrightarrow{x_i}$ is usually a multi-dimensional vector, and the elements in the vector are called *features* or *attributes*. Depending on whether the output y is *categorical* or *real-valued*, supervised learning can be further classified into two categories, *classification* and *regression*.

Unsupervised Learning Unsupervised Learning (or *descriptive learning*) is another main type of Machine Learning, where the dataset is unlabeled: $D = \{(\overrightarrow{x_i})\}_{i=1}^{N}$. The goal of Unsupervised Learning is often to find certain patterns existing on the dataset; that is why it is also called *knowledge discovery*. Unsupervised Learning is arguably more typical of human and animal learning behaviors. It is also more widely applicable than supervised learning, since it does not require a human expert to manually label the data [146]. Approaches of Unsupervised Learning include *clustering*, *Latent Variable Modeling*, etc.

Reinforcement Learning Reinforcement Learning (RL) [184] is an area of machine learning inspired by behaviorist psychology. RL explores how software agents take actions in an environment to maximize the defined cumulative rewards. Most RL algorithms can be formulated as a model consisting of: (1) A set of environment states; (2) A set of agent actions; and (3) A set of scalar rewards. In case of storage systems, *states* correspond to *configurations*, *actions* mean changing to a different configuration, and *rewards* are differences in evaluation results. The agent records its previous experience (history), and makes it available through a *value function*, which can be used to predict the expected reward of state-action pairs. The *policy* determines how the agent takes action. A simple example is ϵ-policy. For each action the agent may can take a random action with probability ϵ; otherwise it will exploit the current value function and take the best action to maximize the rewards. The value function's history can be stored in a tabular form, but this does not scale well to many dimensions. Function approximation is one way for generalization when the state and/or action spaces are large or continuous. However, most approximation methods are still known to be unstable or even divergent. With recent advances in Deep Learning [72], deep convolutional neural networks, termed Deep Q-Networks (DQN), were proposed to parameterize the value function, and have been successfully applied in solving various problems [141, 142]. Many variants of DQN have been proposed [119]; in this thesis we applied its original version [142]. Another interesting fact here is that many RL algorithms, including DQN, also maintains a tradeoff between exploitation, exploration, and history. In the early stages of execution, when the agent knows little about the environment, it will explore the space and try unknown actions. When it interacts enough with the environment, it will tend to choose the actions that it knows will receive the higher rewards.

2.4 Unified Framework

Algorithm	Origin	Exploration	Exploitation	History
Simulated Annealing (SA)	Annealing technology in metallurgy	Allowing moving to worse neighbor states	Neighbor function	N/A
Genetic Algorithms (GA)	Natural evolution	Mutation	Crossover and selection	Current population
Deep Q-Networks (DQN)	Behaviorist psychology and neuroscience	Taking random actions	Taking actions based on action-reward function	Deep convolutional neural network
Bayesian Optimization (BO)	Statistics and experimental design	Selecting samples with high variances	Selecting samples with high mean values	Acquisition function & probabilistic model

Table 2.1: Comparison and summaries of optimization techniques

Most optimization techniques are known to follow the *exploration-exploitation* dilemma [56, 119, 174, 200]. Here we summarize the aforementioned methods by extending the unified framework with a third factor, the *history*. Our unified view thus defines three factors or dimensions:

- **Exploration** defines how the technique searches unvisited areas. This often includes a combination of pure random and also guided search.
- **Exploitation** defines how the technique leverages current neighborhood or *history* to find next sample.
- **History** defines how much data from previous evaluations is kept. History information can be used to help guide both future exploration and exploitation (e.g., avoiding less promising regions, or selecting regions that have never been explored before).

Table 2.1 summarizes how the aforementioned techniques work by maintaining the balance among these three key factors. For example, GA keeps the evaluation results from the last generation, which corresponds to the concept of *history* in our unified framework. GA then *exploits* the stored information, applying selection and crossover to search nearby areas and pick the next generation. Occasionally, it also randomly mutates some chosen parameters, which is the idea of *exploration*. The trade-off among exploration, exploitation, and history largely determines the effectiveness and efficiency of these optimization techniques.

Chapter 3

Related Work

This chapter describes related previous work and compare them with our project.

3.1 Auto-tuning in Computer Systems

In recent years, several attempts were made to automate the tuning of storage systems. Gaonkar et al. [65] apply GAs to design dependable data storage systems for multi-application environments, with the goal of minimizing the overall cost of the system while meeting business requirements. Strunk et al. [181] proposed to use utility functions combining different system metrics and applied GA to automate storage system provisioning. Babak et al. [12] utilized GA to optimize I/O performance of HDF5 applications. Kimberly et al. [96] formulate the data recovery scheduling problem as an optimization problem. They aim at finding the schedule that minimizes the financial penalties due to downtime, data loss, and vulnerability to subsequent failures. GAs are applied and compared with several other heuristics. Xue et al. [211, 212] propose an autonomic technique that learns the intensity patterns of user workload in tiered storage systems over long time-scales using a probabilistic model. They use the model to predict the coming workload patterns and proactively stop/start bulky internal system work. MINERVA [6], is a suite of tools for automating storage system design, which uses declarative specifications of application requirements and device capabilities; constraint-based formulations of the various sub-problems; and simple bin-packing heuristics to explore the search space of possible solutions. More recently, Deep Q-Networks has been successfully applied in optimizing performance for Lustre [117].

Auto-tuning is also a hot topic in other computer systems: Bayesian Optimization was applied to find near-optimal configurations for databases [198] and Cloud VMs [5]. Other applied techniques include Evolutionary Strategies [163], Simulated Annealing [63, 89], Tabu Search [165], and more.

However, previous work all focused on a single algorithm or technique. One contribution of our work is to provide the first comparative study of multiple, applicable optimization methods and compare them for their efficacy in auto-tuning storage systems from various aspects. We also provide some insights into the working mechanism of auto-tuning. More importantly, we propose to design a more intelligent and practical framework for auto-tuning storage systems.

3.2 Hyper-parameter tuning

Esteban et al. [156] applied Evolutionary Algorithms to hyper-parameter optimization for neural networks, and achieved state-of-art results on certain data-sets. Bergstra and Bengio [15] found that randomly chosen trials are more efficient for hyper-parameter optimization than trials on a grid, and explained the cause as the objective function having a low effective dimensionality. In addition, Reinforcement Learning [13] and Bayesian Optimization [55] were also applied to hyper-parameter optimization. Another direction of research focuses on eliminating all hyper-parameters and tries to propose non-parametric versions of optimization methods. Examples of this include GA [79, 127] and BO [174]

In this work, we investigated the impacts of hyper-parameters on various optimization techniques, when applied to auto-tune storage systems.

3.3 Workload Modeling

A few efforts have been made on modeling or characterizing storage workloads. Bumjoon et al. [171] tried to model storage workloads on HDDs from a data-mining point of view. They use a unique clustering method for feature selection that reduces computational time on a list of 20 features available through blktrace and use a hierarchy of clustering and classification to label a workload based on access patterns. Busch et al. [26] proposed to design an automated approach for extracting workload models in virtualized environments. Features used include average file size, file set size, average request size, etc. Li et al. [114] attempted to better define sequential I/O. They focused on LBA and I/O size, and concluded that “consecutive bytes accessed” should be taken into consideration. Shen et al. [176] characterized workloads with the goal of improving performance debugging by separating their model into OS caching, prefetching, OS I/O Scheduling, and storage devices. Wang et al. [201] used CART models to predict per-request response time based on workload characteristics, and provides detailed explanations about how CART models work and why they are suitable for this problem. Riska et al. [160] tried to characterize workloads based on their environment: enterprise, desktop, or consumer electronics.

We feel that most previous work were either vague on what features to pick for characterizing workload, or they limited the model built to one or few use cases. In this work, we target at finding the minimum set, or a small-enough set of features (out of many), which is general and can characterize most storage workload. The feature engineering work will utilize cutting-edge ML and data mining techniques, but we will explain out observations from storage perspective as well.

Chapter 4

Experimental Settings

In this chapter we detail the experimental environments, parameter spaces, and our implementations of several optimization algorithms.

4.1 Hardware

We performed experiments on two sets of machines with different hardware categorized as low-end (*S1*) and mid-range (*S2*). We list the details of these two sets of machines in Table 4.1. We also use Watts Up Pro ES power meters to measure the energy consumption. During our experiments on characterizing storage performance variation (Chapter 6), to maintain realistically high ratio of the dataset size to the RAM size and ensure that our experiments produce enough I/O, we limited the RAM size on all machines to 4GB. We denote this hardware setting as *S3*. We have one type of storage device on S1 and four others on S2 and S3, which will be denoted as *HDD1*, *HDD2*, *HDD3*, *HDD4*, and *SSD* for short in this thesis.

4.2 Workload

We used *Filebench* [62,190] to generate various workloads in our experiments. In each experiment, if not stated otherwise, we formatted and mounted the storage devices with a file system and then ran Filebench. We mainly experimented with the four pre-configured Filebench macro-workloads that exhibit the following significantly different I/O properties:

- **Mailserver** emulates the I/O workload of a multi-threaded email server. It generates sequences of I/O operations that mimic the behavior of reading emails (open, read the whole file, and close), composing emails (open/create, append, close, and fsync) and deleting emails. It uses a flat directory structure with all the files in a single directory, and thus exercises the ability of file systems to support large directories and fast lookups.

- **Fileserver** emulates the I/O workload of a server that hosts users' home directories. Here, each thread represents a user, which performs create, delete, append, read, write, and stat operations on a unique set of files. It exercises both the metadata and data paths of the targeted file system.

Setting	S1	S2	S3
Model	Dell PowerEdge SC1425	Dell PowerEdge R710	Dell PowerEdge R710
CPU	Intel Xeon single-core 2.8GHz CPU × 2	Intel Xeon quad-core 2.4GHz CPU × 2	Intel Xeon quad-core 2.4GHz CPU × 2
Memory	2GB	24GB	4GB (set by *mem=* in */etc/default/grub*)
Storage	HDD1 (73GB Seagate ST373207LW SCSI drive) × 2	HDD2 (146GB Seagate ST9146853SS SAS HDD), HDD3 (500GB Seagate ST9500430SS SAS HDD), HDD4 (200GB Intel SSDSC2BA200G3 SATA HDD), SSD (250GB Fujitsu MHZ2250BKG2 SATA HDD)	HDD2 (146GB Seagate ST9146853SS SAS HDD), HDD3 (500GB Seagate ST9500430SS SAS HDD), HDD4 (200GB Intel SSDSC2BA200G3 SATA HDD), SSD (250GB Fujitsu MHZ2250BKG2 SATA HDD)
Partition	100GB	100GB	Full size
OS	Ubuntu 14.04 with kernel 3.13	Ubuntu 14.04 with kernel 3.13	Ubuntu 14.04 with kernel 4.4

Table 4.1: Details of experiment machines.

- **Webserver** emulates the I/O workload of a typical static Web server with a high percentage of reads. Files (Web pages) are read sequentially by multiple threads (users); each thread appends to a common log file (Web log). This workload exercises fast lookups, sequential reads of small files and concurrent data and metadata management.

- **Dbserver** mimics the behaviors of Online Transaction Processing (OLTP) databases. It mainly consists of random asynchronous writes, random asynchronous reads and moderate synchronous writes to the log file. It exercises the ability of large file management, extensive concurrency, and random read/write operations.

Table 4.2 shows the detailed settings of our workloads. The first four workloads are the default workload profiles provided by Filebench (named as **-def*), while the last four workloads are modified workloads with larger working dataset size (at least 2X RAM size). We explain our choices below.

Default Workloads It is well known that the working set size has a significant impact on the duration of an experiment [188]. In our auto-tuning experiments, the goal was to explore a large set of parameters and values quickly (though it still took us over two years to search some spaces exhaustively). We therefore decided to trade the working set size in favor of increasing the number of configurations we could explore in a practical time period. We mainly experimented with the default settings provided by Filebench. We did not perform a separate cache warm-up phase, since performance usually become relatively stable within a short time given the default dataset size.

Workload	Avg. File Size	Avg. Dir Width	# Files	Running Time (s)	Num. of Threads	R/W Ratio	Filebench Version
fileserver-def	128KB	20	10,000	100	50	1:2	1.4.9
mailserver-def	16KB	1,000	1,000	100	16	1:1	1.4.9
webserver-def	16KB	20	1,000	100	100	10:1	1.4.9
dbserver-def	10MB	1,024	10	100	10 + 1	10:1	1.4.9
fileserver-heavy	128KB	20	80,000	800	50	1:2	1.5.0
mailserver-heavy	16KB	1,000,000	640,000	2,000	16	1:1	1.5.0
webserver-heavy	16KB	20	640,000	800	100	10:1	1.5.0
dbserver-heavy	1GB	1,024	10	100	10 + 1	10:1	1.5.0

Table 4.2: Filebench workload characteristics.

Intensive Workloads For studying performance variations, nearly all workload characteristics were set to Filebench's default values, except for the number of files and the running time. As the average file size is an inherent property of a workload and should not be changed [190], the dataset size is determined by the number of files. We increased the number of files such that the dataset size is 10GB—or 2.5× the machine RAM size (*S3* in Table 4.1). By fixing the dataset size, we normalized the experiments' set-size and run-time, and ensured that the experiments run long enough to produce enough I/O. With these settings, our experiments exercise both in-memory cache and persistent storage devices [189]. We did not perform a separate cache warm-up phase in our experiments because in this study we were interested in performance variation that occurred *both* with cold and warm caches [189]. The default running time for Filebench is too short to warm the cache up. We therefore conducted a calibration phase to pick a running time that was long enough for the cumulative throughput to stabilize. We ran each workload for up to two hours for testing purposes, and finally picked the running time as shown in Table 4.2. We also let Filebench output the throughput (and other performance metrics) every 10 seconds, to capture and analyze performance variation at a finer time granularity. We also experimented these intensive workload for auto-tuning experiments, for *Storage V3* and *Storage V4* (Section 4.3).

4.3 Parameter Space

To test the efficacy of auto-tuning algorithms, ideally we wanted our storage parameter spaces to be large and complex enough. Alas, evaluations for storage systems take a long time. Considering experimentation on multiple hardware settings and workloads, we decided to experiment with a reasonable subset of the most relevant storage system parameters. We selected parameters in close collaboration with several storage experts that have either contributed to storage system designs or have spent years tuning storage systems in the field. We experimented with 7 Linux file systems that span a wide range of designs and features: *Ext2* [30], *Ext3* [196], *Ext4* [59], *XFS* [185], *Btrfs* [161], *Nilfs2* [103], and *Reiserfs* [158].

Our experiments were mainly conducted on two sets of parameters, termed as *Storage V1* and *Storage V2*. We started with a relatively smaller set of 7 parameters, and refer it as *Storage V1*. It contains the following common file system parameters: *file system type*, *block size*, *inode size*, *blocks per group*, *mount options*, *journal options*, and *special options*. We tested *Storage*

Param.	Abbr.	Values
File System	FS	Ext2, Ext3, Ext4, XFS, Btrfs, Nilfs2, Reiserfs
Block Size, Leaf Size	BS	1K, 2K, 4K
Inode Size, Sector Size	IS	n/a, 128, 256, 512, 1024, 2048, 4096, 8192
Block Group, Alloc. Group	BG	n/a, 2, 4, 8, 16, 32, 64, 128, 256
Journal Option	JO	n/a, order=strict, order=relaxed, data=journal, data=ordered, data=writeback
Atime Option	AO	relatime, noatime
Special Option	SO	n/a, compress, nodatacow, nodatasum, notail
I/O Scheduler	I/O	noop, cfq, deadline

Table 4.3: Details of Parameter Spaces

V1 with *Setting S1*. After some preliminary experiments, we extended our search space with one more parameter, the *I/O Scheduler*, and refer it as *Storage V2*. Experiments with *Storage V2* were conducted with *Setting S2*. We list all the aforementioned parameters and their values in Table 4.3. Note that certain combinations of parameter values could produce invalid configurations. For example, for Ext2, the journaling options make no sense because Ext2 does not have a journal. To handle this, we added a value *n/a* to the existing range of parameters. Any parameter with *n/a* value is considered invalid. Invalid configurations will always come with evaluation results of zero (i.e., no throughput); this ensures they are purged in an upcoming optimization process. There are 2,074 valid configurations in *Storage V1* and 6,222 in *Storage V2*.

Parameters)	Values
File System	Ext4
Block Size	1024, 2048, 4096
Inode Size	128, 512, 2048
Flex Block Group	4, 16, 64
Journal Option	journal, ordered, writeback
Inode Readahead	16, 32, 64
I/O Scheduler	noop, cfq, deadline
Dirty Background Ratio	5, 10
Dirty Ratio	10, 20, 40

Table 4.4: Parameters and their values in Storage V3.

As we described in Chapter 9, parameters are not equal and changing values of certain parameters may associate with overhead. Therefore, we carefully designed *Storage V3* (Table 4.4) and *Storage V4* (Table 4.5) to represent each category of parameters. They were mainly applied in Chapter 7 and Chapter 9.

4.4 Experiments and Implementations

Our experiments and implementation consist of two parts. First, we exhaustively ran all configurations for each workload on the S1 and S2 machines, and stored the results in a relational database.

Parameters)	**Values**
File System	XFS
Block Size	1024, 2048, 4096
Inode Size	512, 2048
Allocation Group Count	2, 32, 128, 512
Sector Size	512, 2048, 4096
Log Buffer Count	2, 8
Log Buffer Size	32k, 256k
Allocation Size	64k, 256k
I/O Scheduler	noop, cfq, deadline
Dirty Background Ratio	5, 10
Dirty Ratio	10, 20, 40

Table 4.5: Parameters and their values in Storage V4.

We collected the throughput in terms of I/O operations per second, as reported by Filebench, the running time (including setup time), as well as power and energy consumption. To acquire more accurate and stable results, we evaluated each configuration under the same environment for at least 3 runs, resulting in more than 500,000 total experimental runs. This data collection benefited our evaluation on auto-tuning as we can simply simulate a variety of algorithms by just querying the database for the evaluation results for different configurations, without having to rerun slow I/O experiments. The exhaustive search also let us know exactly what the global optimal configurations are, so that we can better understand how each optimization method performs.

Second, we simulated the process of auto-tuning storage systems by running the desired optimization method and querying the database for the evaluation results of the targeted storage configurations. We focused on optimizing for throughput in this thesis. Our implementations of optimization methods are mostly based on open-source publicly-available libraries. We use Pyevolve [153] for Genetic Algorithms, Scikit-Optimize [178] for Bayesian Optimization, and TensorFlow [191] for the DQN implementation. We implemented a simple version of Simulated Annealing, with both linear and geometric cooling schedules. (We also fixed bugs in Pyevolve and plan to release our patches.) Most of our implementation was done by applying storage-related concepts into algorithm-specific ones. For example, for GA, we defined each storage parameter as a *gene*, and each configuration as a *chromosome*. For DQN we provided storage-specific definitions for states, actions, and rewards. The complete implementation uses around 10,000 lines of code, consisting of Python and Shell scripts.

Chapter 5

Towards Better Understanding of Black-box Auto-Tuning: A Comparative Analysis for Storage Systems

In this chapter we apply several popular techniques to the collected dataset to find optimal configurations under various hardware and workload settings: Simulated Annealing (SA), Genetic Algorithms (GA), Bayesian Optimization (BO), and Deep Q-Networks (DQN). We also tried Random Search (RS) in our experiments, which showed surprisingly good results in previous research [15]. We compared these techniques from various aspects, such as the ability to find near-optimal configurations, convergence time, and instantaneous system throughput during auto-tuning. We also showed that hyper-parameter settings of these optimization algorithms, such as mutation rate in GA, could affect the tuning results. We compared the techniques across three behavioral dimensions: (1) *Exploration*: how much the technique searches the space randomly. (2) *Exploitation*: how much the technique leverages the "neighborhood" of the current candidate or previous search history to find even better configurations. (3) *History*: how much data from previous evaluations is kept and utilized in the overall search process. Based on our evaluation results, we show that all techniques employ these three key concepts to varying degrees and the trade-off among them plays an important role in the effectiveness and efficiency of the algorithms.

Most black-box optimization methods lack solid theoretical understanding, partially due to the large variety of problems that they were proposed to solve [213]. Based on our experimental results and domain expertise, we provide explanations of efficacy of such black-box optimization methods from a storage perspective. We observed that certain parameters would have a greater effect on system performance than others, and the set of dominant parameters depends on file systems and workloads. This allows us to provide more insights into the auto-tuning process.

Part of the results from this chapter was published in ATC 2018.

The chapter is organized as follows. Section 5.1 overviews the datasets that we collected for over two years. Section 5.2 compares five popular optimization techniques from several aspects. Section 5.3 uses GA as a case study to show that hyper-parameters of these methods could also impact the auto-tuning results.

5.1 Overview of Datasets

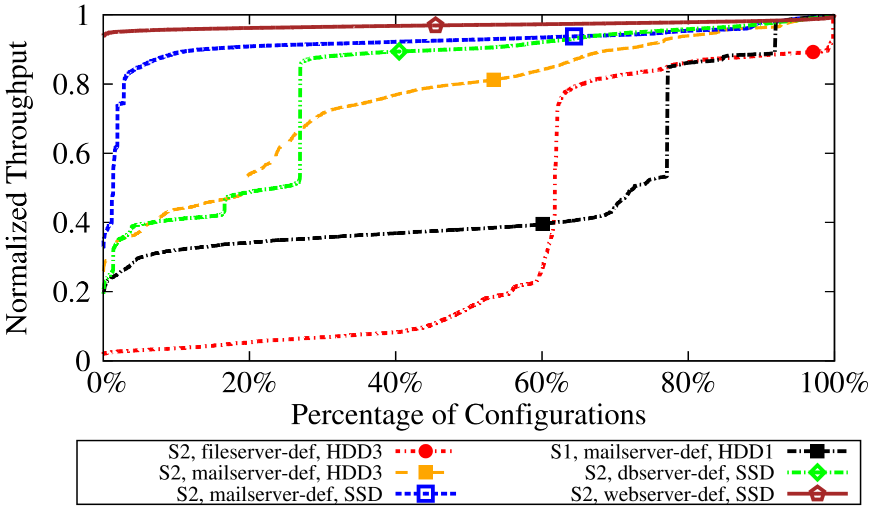


Figure 5.1: Throughput CDF with different hardware and workloads, with symbols marking the default configurations.

As per Chapter 4, our experimental methodology is to first exhaustively run all configurations under different workloads and test machines. We stored the results in a database for future use. This data collection benefits future experiments as we can simulate a variety of algorithms by querying the database for the evaluation results of different configurations.

Figure 5.1 shows the throughput CDF among all configurations for each hardware setting and workload. Due to space limits, we show only 6 representative datasets out of 18 here. The Y axis is normalized by the maximum throughput under each experiment setting. The symbols on each line mark the default configurations. As seen, for most settings, throughput values vary across a wide range. The ratios of the worst throughput to the best one are mostly between 0.2–0.4. In one extreme case, for *fileserver-def* on *S1* machines and with HDD1 device, the worst configuration only produces 1% I/O operations per unit time, compared with the global optimal one. This underlines the importance of tuning storage systems: an improperly configured system could be remarkably under-utilized, and thus wasting a lot of resources. However, *S2, webserver-def, SSD* shows a much narrower range of throughput, with the worst-to-best ratio close to 0.9. This is attributed mainly to the fact that *webserver-def* consists of mostly sequential read operations that are processed similarly by different I/O stack configurations. Another useful observation from Figure 5.1 is that default configurations are always sub-optimal and, under most settings, ranked lower than the top 40% configurations. For *S1, fileserver-def, HDD1*, the default configuration shows a normalized throughput of 0.39, which means that the optimal configuration performs 2.5 times better.

We list the optimal configurations for each hardware setting and workload from our datasets in Table 5.1. As we can see, optimal configurations depend on the specific hardware as well as the running workload. For *mailserver-def* with *S1* machines and the *HDD1*, the global best is a *Nilfs2* configuration. However, if we fix the workload and change the hardware to *S2-HDD3*, the optimum becomes an *Ext4* configuration. Similarly, fixing the hardware to *S2-SSD* and experimenting under

Hardware Workload-Device	**File System**	**Block Size**	**Inode Size**	**BG Count**	**Journal Options**	**Atime Options**	**Special Options**	**I/O Scheduler**	**Through- put (IOPS)**
S1-Mail-HDD1	Nilfs2	2K	n/a	256	order=relaxed	relatime	n/a	-	3,677
S2-Mail-HDD3	Ext2	4K	256	32	n/a	relatime	n/a	noop	18,744
S2-Mail-SSD	Ext2	4K	256	8	n/a	relatime	n/a	noop	18,845
S2-File-SSD	Btrfs	4K	4,096	n/a	n/a	relatime	nodatacow	deadline	16,587
S2-DB-SSD	Ext4	1K	128	2	data=ordered	noatime	n/a	noop	41,948
S2-Web-SSD	Ext4	4K	128	4	data=ordered	noatime	n/a	noop	16,185

Table 5.1: Global optimal configurations with different settings and workloads. Workloads are abbreviated. Db: dbserver-def; File: fileserver-def; Mail: mailserver-def; Web: webserver-def.

different workloads leads to different optimal configurations. This proves our early claim that performance (and other metrics) are sensitive to the environment (i.e., hardware, configuration, and workloads); this actually complicates the problem as results from one environment cannot be directly applied in another.

It is known that the working set size has a significant impact on the duration of an experiment [188]. Our goal in this study was to explore a large set of parameters and values quickly (though it still took us over two years). We therefore decided to trade the working set size in favor of increasing the number of configurations we could explore in a practical time period. In our experimental results, this trade-off sometimes manifests itself since SSD configurations produce comparable throughputs as HDD ones (see Table 5.1). The experiments, however, do demonstrate a wide range of performance numbers and, therefore, are valid for evaluating different optimization methods. We plan to include the working set size in the set of optimization parameters in the future.

5.2 Comparative Analysis

Many optimization techniques have been applied to various auto-tuning tasks [181, 198]. However, previous efforts picked algorithms somewhat arbitrarily and evaluated only one algorithm at a time. Here we provide the first comparative study of multiple black-box optimization techniques on auto-tuning storage systems. As discussed in §2.2, we focus our evaluations on a representative set of optimization methods, and their common hyper-parameter settings, including 1) Simulated Annealing (SA), with a linear cooling schedule; 2) Genetic Algorithms (GAs) with population size of 8, mutation rate of 2%; 3) Deep Q-Networks (DQN) with experience replay [142] and $\epsilon = 0.2$; and 4) Bayesian Optimization (BO) with Expected Improvement (EI) and Gaussian prior. 5) Random Search (RS), which merely performs random selection without replacement. We provide more discussion on the impact of hyper-parameters in Section 5.3. Note that SA, DQN, and RS experiments start with the default Ext4 configuration. GA and BO require several initial configurations (*prior points*), which we set to default configurations of all seven file systems. This allows us to simulate real-world use cases, where users often deploy their system with the default settings (and may manually optimize starting from the defaults).

Figure 5.2 presents one simulated run of each optimization method on *S2, mailserver-def, HDD3*; the Y axis shows the throughput value of the best configuration found so far, and the

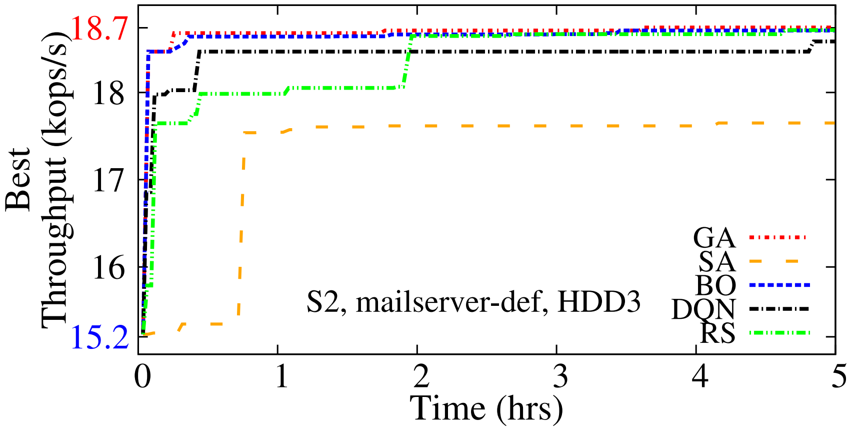


Figure 5.2: Highest throughput found over time, zooming in the $Y \in [15 : 19]$ range. The blue number (15.2) on the Y axis shows the default, and the red one (18.7) shows the optimal.

X axis is the running time. All time-related metrics in this section are based on the actual running time of evaluating each storage configuration, which is stored in our database. This includes both setup time and benchmarking time. We are not comparing the running costs (including any necessary training phases) for optimization methods here, which is our future work. Figure 5.2 is plotted by zooming in the range of $Y \in [15 : 19]$, with the blue number (15.2) on Y axis represents the default, while the red one (18.7) shows the global optimal. It shows that all five methods were able to gradually find better configurations, but their effectiveness and efficiency differed a lot. SA performed the worst, and got stuck in a configuration with throughput value of less than 18K IOps. DQN was able to converge to a good configuration, but spent more time to achieve that than RS. GA and BO performed best out of these five tested optimization methods. They both successfully identified a near-optimal configuration within one hour. Interestingly, we observed that pure Random Search (RS) produced better results than some other optimization methods. This is because not all storage parameters have significant impact on system performance, resulting in an *effective* search space that is much smaller than the original one. Similar results were observed in hyper-parameter optimization for neural networks [14]. We discuss this further in §5.4.

Since exploration is one critical component of all optimization methods (see §2.4), their evaluation results could also exhibit some degree of randomness. To compare them more thoroughly, we ran each optimization technique on the same environment (*S2, HDD3*) for 1,000 runs. Figure 5.3 shows the results, which evaluate the techniques' probability to find good and near-optimal configurations. Here we define a near-optimal configuration as one with throughput higher than **99%** of the global optimal value. The Y axis shows the percentage of total runs that found a near-optimal configuration within a certain time (X axis). Under *mailserver-def* workload, seen in the upper part of Figure 5.3, SA had the lowest probability among 5 algorithms.

Even after 5 hours, only around 80% of its runs found one near-optimal configuration, which suggests that SA can sometimes get stuck in a local optima. For other optimization methods, given enough time, over 90% of their runs converged to a near-optimal configuration, with BO outperforming GA, and GA outperforming DQN. RS shows the highest probability of finding near-optimal configurations when approaching 5 hours. This is reasonable because given enough time, a random selection will eventually hit near-optimal points. However, when conducting the

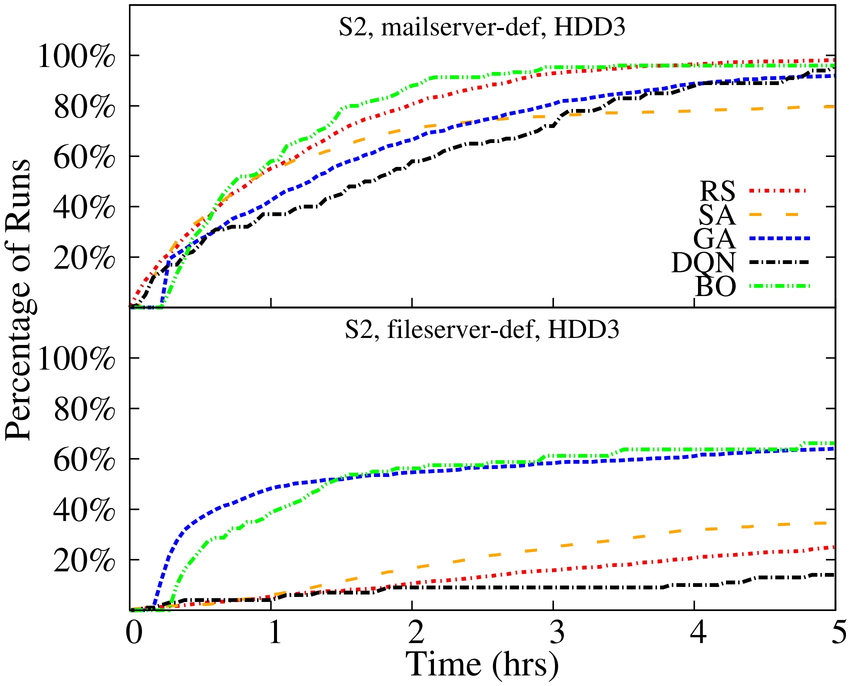


Figure 5.3: Comparing optimization methods' efficacy in finding near-optimal configurations. The Y axis shows the percentage of total runs (1,000) that found near-optimal configurations within certain time (X axis).

same experiments under the *fileserver-def* workload, it becomes more difficult to find near-optimal configurations. GA and BO are still the best, though only 65% of their runs were able to find near-optimal configurations within 5 hours. SA, RS, and DQN have a probability of lower than 40% to do so, with DQN perform the worst. This is because the global optimum under *fileserver-def* is a Btrfs configuration (see Table 5.1). It is more difficult for optimization algorithms to pick such configurations for the following reasons: 1) Few Btrfs configurations reside in the neighborhood of the default Ext4 configurations; 2) Fewer than 2 % of all valid configurations are Btrfs ones, which make them less likely to be selected through mutation.

The above results all focused on finding near-optimal configurations. However, another important aspect to compare is the system's performance *during* the auto-tuning process. This is especially important if the targeted system is deployed and online. Some randomness (exploration) is necessary when searching a complex parameter space, but ideally optimization algorithms should spend less time on bad configurations. To compare this, in Figure 5.4 we plotted the instantaneous throughput (Y axis) over time (X axis) for one run with each method under *S2, mailserver-def, HDD3*.

BO and GA are still the best two methods in terms of instantaneous throughput.

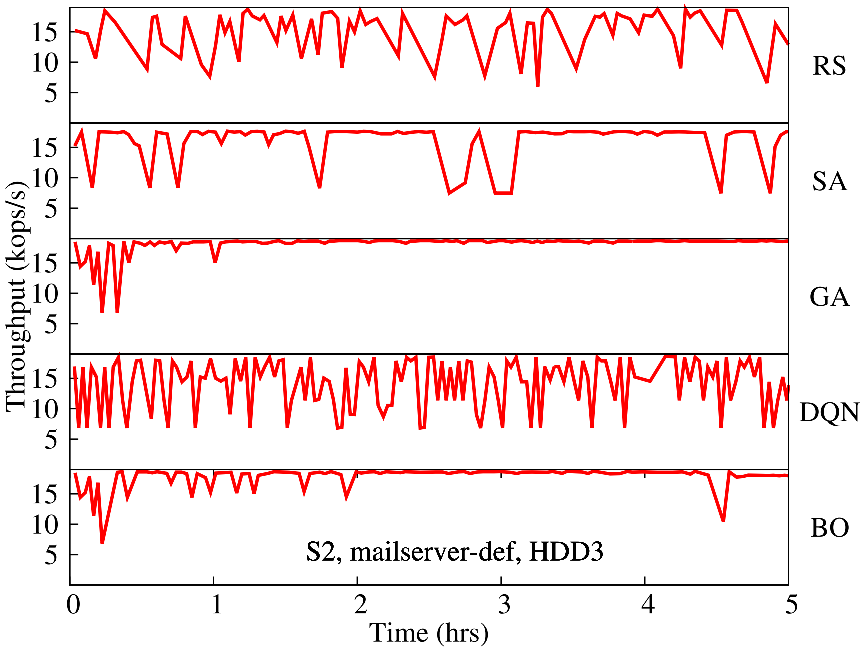


Figure 5.4: Comparing optimization methods' instantaneous performance (Y axis) over time (X axis).

During the tuning process, occasionally they will pick a worse configurations than the current one. However, they both possess the ability to quickly discard these unpromising configurations. GA achieves this by assigning the probability of surviving to next generation based on the fitness values (i.e., throughput). Configurations with low throughput values have a lower chance to be picked as parents, and thus their genes (parameter values) have a lower chance of appearing in configurations of the next generation (i.e., "survival of the fittest"). The reason for stable instantaneous throughputs with BO is that it uses an intelligent acquisition function to guide the selection of the next generation, with the goal of maximizing the potential gain; this makes BO less likely to choose a bad configuration. In contrast, SA performs poorly possibly because it lacks a history to guide the exploitation and exploration phases, and only uses its neighborhood information (and current temperature) to pick the next configuration. DQN shows similar results with RS, which is likely caused by the fact that DQN was originally designed as an agent interacting with an unknown environment, and thus a lot of exploration (randomness) occurs in the training phase [117, 142].

In conclusion, BO and GA perform best among the 5 tested methods, on either the ability to converge to near-optimal configurations or in maintaining stable instantaneous performance during the tuning process. DQN and SA can find good configurations, although they were less efficient and less stable. Surprisingly, Random Search sometimes can produce better results than some traditional optimization methods, given enough time. We provide more explanations on these methods in Section 5.4.

5.3 Impact of Hyper-Parameters

Many optimization methods' efficacy depend on the specific hyper-parameter settings, and choosing the right hyper-parameters has caused headache to researchers for a long time [14, 15]. In this section we use GA as a case study, and show the impact of one hyper-parameter, the *mutation rate*, on auto-tuning results.

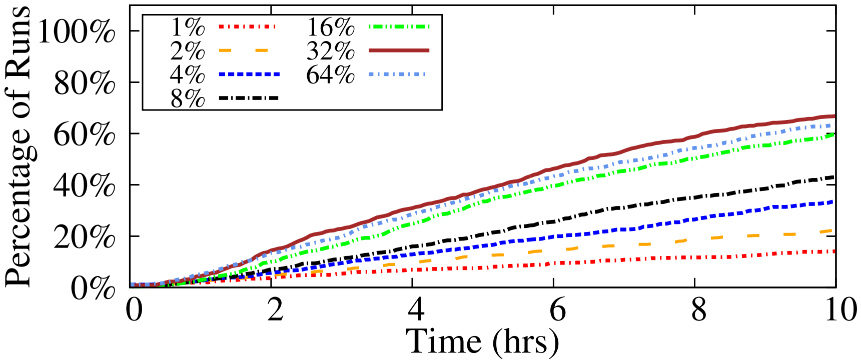


Figure 5.5: Impact of mutation rates on GA.

The mutation rate controls the probability of randomly mutating one parameter to a different value, and aligns with the idea of *exploration*, as per §2.4.

Figure 5.5 shows the results from 7 sets of GA experiments with different mutation rates (from 1% to 64%) under *S2, mailserver-def, HDD3*. Each experiment was repeated for 1,000 runs.

It is similar to Figure 5.3, but with the goal of finding near-optimal configurations whose throughput values are higher than **99.5%** of the global optimal. This makes the optimization more challenging, as GA already performs quite well on easier tasks (Section 5.2). As shown in the figure, when increasing the mutation rate, GA has a higher probability to converge to near-optimal configurations within a shorter time period. This is because GA works by identifying promising combination of alleles (parameter values) for the subset of *effective* genes (parameters). We define effective parameters as those having a higher impact on performance than all others. A higher mutation rate means a higher chances of exploration, and thus finding combinations of effective alleles within a shorter time. We explain this effect more in Section 5.4. However, a mutation rate of 64% actually performs worse than 32%. This is because in order to reach near-optimal configurations, GA needs both exploration and exploitation. Exploration lets GA identify processing subspaces (i.e., combinations of certain parameter values) while exploitation helps GA search within promising subspaces. In this case, with a mutation rate of 64%, GA spends too much time on exploration (too much randomness), resulting in fewer chances for exploitation.

Note that in this section we are only using GA mutation rates as an example showing the impact of hyper-parameters on the efficacy of optimization methods. There are other hyper-parameters for nearly all techniques, such as the cooling schedule and initial temperature in SA, the acquisition function in BO, the population size and selection method in GA, etc. In the future, we plan to conduct more experiments on all these hyper-parameters.

5.4 Peering into the Black Box

Despite some successful applications of black-box optimization on auto-tuning system parameters, few have explained how and why some techniques work better than others for certain problems. Here we take the first step towards unpacking the "black box" and provide some insights into their internals based on our evaluation results and storage domain knowledge.

Our attempts for explanations stem from a somewhat unexpected but beneficial behavior of GA in the experiments. We found that as GA runs, there is often a small set of alleles (parameter values) that dominate the current population and are unlikely to change. We present and explain this observation in Figure 5.6. The experiment was conducted on a parameter space consisting of 2,208 Ext3 configurations under *S2, fileserver-def, SSD*. The X axis shows 5 genes (parameters) separated by major ticks, while one cell represents one allele (parameter value). The parameters are denoted with their abbreviations from Table 4.3. The Y axis shows the generation number, and we only plotted the first 10 generations. Cells were colored based on the number of alleles in each generation. More frequent alleles are colored with darker colors. In the first generation, the gene's alleles (parameter values) were quite diverse. For example, there were 3 alleles (1K, 2K, 4K) for the *Block Size* gene, and 3 alleles (journal, ordered, writeback) for the *Journal Option* gene. However, the diversity of alleles decreased in later generations, and several genes began to dominate and even converged to a single allele. For the *Block Size* gene, only the 4K allele

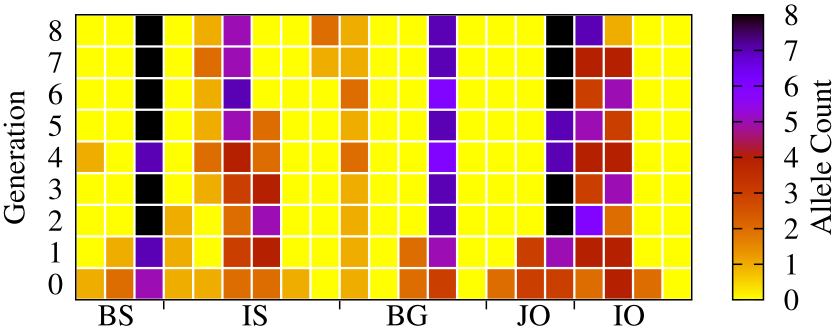


Figure 5.6: Number of alleles (parameter values) in the first 10 generations from one GA experiment run, with more frequent ones colored with darker colors.

survived and other two became extinct. Since GA was proposed by simulating the process of natural selection, where alleles with better fitness are more likely to survive, this suggests that GA works by identifying the combination of good alleles (storage parameter values), and producing offspring with these alleles. As shown in Figure 5.6, in the 10^{th} generation, all configurations have a *Block Size* of 4K and *Journal Option* of writeback.

To confirm the above observations, in Figure 5.7 we plotted all Ext3-SSD configurations under *fileserver-def* workload, with one dot corresponding to one configuration. Configurations are separated based on the *Journal Option*, shown as the X axis, and colored based on their *Block Size*. To clearly see all points within each X-axis section, we ordered configurations by their unique identification number in our database. The Y axis represents throughput values. This resulted in the formation of nine "clusters" on the graph, each corresponding to a fixed $\langle$*Journal Option*, *Block Size*$\rangle$ pair. We can see that configurations with *data=ordered* tend to produce higher throughput than those with *data=journal*, and *data=writeback* produces the best throughput. This is somewhat expected from a storage point of view, as Ext3's more fault tolerant journal option (*data=journal*) may hurt throughput by writing data as well as meta-data to the journal first.

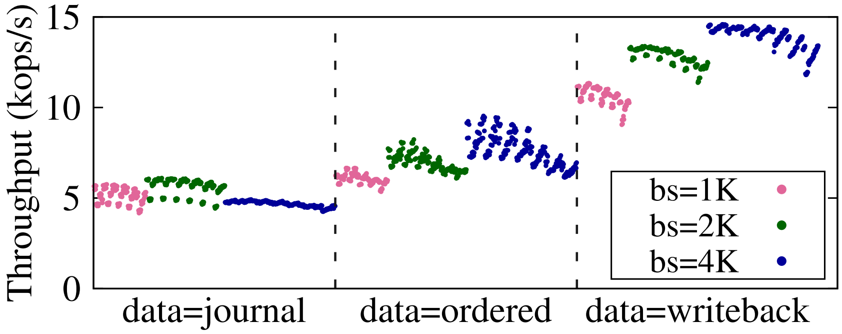


Figure 5.7: Scatter plot for all Ext3-SSD configurations under fileserver-def workload, with one dot corresponding to one configuration.

Moreover, among journal configurations with *data=writeback*, those with a 4K *Block Size* turn out to produce the highest throughput. This aligns with our observation from Figure 5.6 that GA works by identifying a subset of genes that have a greater impact on performance—*Block Size* and *Journal Option*—and finding the best alleles for them ([*4K, data=writeback*]).

Based on these observations, one interesting question to ask is whether the conclusion that a subset of parameter have greater impact on performance than other parameters, also holds for other file systems and workloads. To answer this question, we quantified the correlation between parameter values and the throughput. As most of our parameters are categorical or discrete numeric, whereas the throughput is continuous, we took a common approach to quantify the correlation between categorical and continuous variables [31]. We illustrate with the *Block Size* parameter as an example. Since it can take 3 values, we convert this parameter to three binary variables x_1, x_2, and x_3. If the *Block Size* is 1K, we assign $x_1 = 1$ and x_2 and x_3 are set to 0. Let Y represent the throughput values. We then do a linear regression with ordinary least squares (OLS) on Y and x_1, x_2, x_3. R^2 is a common metric in statistics to measure how the data fits a regression line. In our approach, R^2 actually quantifies the correlation between the selected parameter and throughput. We consider $R^2 > 0.6$ as an indication that the parameter has significant impact on performance, as is common in statistics [31]. The same calculation is applied to all parameters among SSD configurations under the *fileserver-def* and *dbserver-def*. Parameters with the highest R^2 values are colored in yellow background in Table 5.2. If all R^2 values are below 0.6, we simply leave the entries blank, meaning no highly correlated parameters were found. To find the second important parameter, the same process is applied to the remaining parameters, but with the value of the most important one fixed (to isolate its effect on the remaining parameters' importance). Taking Ext4 as an example, we calculate R^2 values for all other parameters among configurations with the same *Journal Option*. For one parameter, 3 *Journal Options* lead to three R^2 values; we then take the maximum one as the R^2 value for this parameter. We color the parameter with the highest R^2 in Table 5.2 with a green background.

Workload	**FS**	**BS**	**IS**	**BG**	**JO**	**AO**	**SO**	**I/O**
fileserver-def	**Ext2**	-	-	-	-	-	-	0.68
	Ext3	0.84	-	-	0.90	-	-	-
	Ext4	0.92	-	-	0.99	-	-	-
	XFS	0.94	-	0.82	-	-	-	-
	Btrfs	-	-	-	-	-	-	-
	Nilfs2	0.99	-	-	-	-	-	0.94
	Reiserfs	-	-	-	0.74	-	-	0.99
dbserver-def	**Ext2**	-	-	-	-	-	-	-
	Ext3	0.72	-	-	0.96	-	-	-
	Ext4	-	-	-	0.96	0.68	-	-
	XFS	-	-	-	-	-	-	-
	Btrfs	-	-	-	-	-	-	-
	Nilfs2	0.62	-	-	-	-	-	0.80
	Reiserfs	-	-	-	0.99	-	-	-

Table 5.2: Importance of parameters (measured by R^2) among SSD configurations, with the most important one colored in yellow and second in green.

We can see that the correlated parameters are quite varied, and depend a lot on file systems. For example, under *fileserver-def*, the two most important parameters for Ext3 (in descending order) are *Journal Option* and *Block Size*; this aligns with our observation in Figure 5.6 and 5.7. However, for Reiserfs, the top 2 changes to *I/O Scheduler* and *Journal Option*. Interestingly, all parameters for Btrfs come with low R^2 values, which indicates that no parameter has significant impact on system performance under *fileserver-def* with Btrfs. Correlation of parameters can also depend on the workloads. For instance, the two dominant parameters for XFS under *fileserver-def* are *Block Size* and *Allocation Group*. When the workload changes to *mailserver-def*, all parameters for XFS seem to have minor impact on performance. Note that here we are isolating the impact of each parameter, thus assuming that their effect on throughput is independent; in future work we plan to investigate whether parameters have inter-dependencies.

The fact that parameters have varied impact on performance can also help explain the auto-tuning results in Section 5.2. Although our parameter space comes with 8 parameters, only a subset of them are correlated with performance. The number of dominant parameters is termed as *effective dimension*, and has also been observed in hyper-parameter optimization problems [14]. In our experiments (Section 5.2), Random Search (RS) is actually searching in a smaller effective space than the original one, and thus can find good configurations within a short time. GA's efficacy comes from assigning a higher chance of survival to configurations with a certain combination of values for the effective parameters. BO stores its previous search experience (history) in a probabilistic surrogate model that it is building, which eventually encodes the combination of dominant parameter values that can result in good throughput values. SA does not work as well because it lacks history information to identify the dominant parameters: it wastes time on changing less useful parameters and converges slowly. Similarly, DQN also spends lots of its effort on exploring unpromising spaces, which slows its ability to find near-optimal configurations.

5.5 Limitations

In this chapter we provided the first comparative analysis of applying multiple optimization methods on auto-tuning storage systems. However, auto-tuning is a complex topic and more effort is required. We list some limitations of this comparative work below. ■ **(1)** We assume that changing parameter values come at no cost. In reality, parameters like *Block Size* may need reformatting file systems. We addressed this in Chapter 9 by associate cost functions with each parameter. ■ **(2)** Previous studies [15], as well as our results from Section 5.3, suggest that the choice of hyper-parameter settings could have a significant impact on the efficacy of optimization algorithms. We plan to further explore the impact of hyper-parameters on optimization algorithms.

Chapter 6

On the Performance Variation in Modern Storage Systems

6.1 Motivations

Predictable performance is critical in many modern computer environments. For instance, to achieve good user experience, which notably impacts the revenues, interactive Web services require stable response time [47, 88, 115]. In cloud environments users pay for computational resources. Therefore, achieving predictable system performance, or at least establishing the limits of performance variation, is of utmost importance for the clients' satisfaction [183,210]. In a broader sense, humans generally expect repetitive actions to yield the same results and take the same amount of time to complete; conversely, the lack of performance stability, is fairly unsatisfactory to humans.

Performance variation is a complex issue and can arise from nearly every layer in a computer system. At the hardware level, CPU, main memory, buses, and secondary storage can all contribute to overall performance variation [47, 115]. At the OS and middleware level, when background daemons and maintenance activities are scheduled, they impact the performance of deployed applications. More performance disruptions come into play when considering distributed systems, as applications on different machines have to compete for heavily shared resources, such as network switches [47].

In this chapter we focus on characterizing and analyzing performance variations arising from benchmarking a typical modern storage system that consists of a file system, a block layer, and storage hardware. Storage have been proven to be a critical contributor to performance variation [80, 166, 188]. Furthermore, among all system components, the storage system is the cornerstone of data-intensive applications, which become increasingly more important in the big data era [34, 91]. Although our main focus here is reporting and analyzing the variations in benchmarking processes, we believe that our observations pave the way for understanding stability issues in production systems.

Historically, many experienced researchers noticed how workloads, software, hardware, and the environment—even if reportedly "identical"—exhibit different degrees of performance variations in repeated, controlled experiments [33,47,57,115,125]. We first encountered such variations in exhaustive search experiments (see Chapter 4) with Ext4: multiple runs of the same workload in a carefully controlled environment produced widely different performance results. Over a period of two years of collecting performance data, we later found that such high performance variations were not confined to Ext4. Over 18% of 6,222 different storage configurations on 4 different storage devices that we tried exhibited a standard deviation of performance larger than 5% of the mean, and a range value (maximum minus minimum performance, divided by the average) exceeding 9%. In a few extreme cases, standard deviation exceeded 40% even with numerous repeated experiments. The observation that some configurations are more stable than others motivated us to conduct a more detailed study of storage system performance variation and seek its root causes, as performance stability is critical for storage systems and important in achieving the success of auto-tuning.

To the best of our knowledge there are no systematic studies of performance variation in storage systems. Thus, our first goal was to characterize performance variation in different storage configurations. However, measuring this for even a single storage configuration is time consuming; and measuring all possible configurations is time-prohibitive. Even with our Storage V2 (see Section 4.3), it could take more than 2 years of evaluation time. Therefore, in this study we combined two approaches to reduce the configuration space and therefore the amount of time to run the experiments: (1) we used domain expertise to select the most relevant parameters, and (2) we applied a Latin Hypercube Sampling (LHS) to the configuration space. Even for the reduced space, it took us over 33 clock days to complete these experiments alone.

We focused on three local file systems (Ext4, XFS, and Btrfs) which are used in many modern local and distributed environments. Using our expertise, we picked several widely used parameters for these file systems (e.g., block size, inode size, journal options). We also varied the Linux I/O scheduler and storage devices, as they can have significant impact on performance. We benchmarked over 100 configurations using different workloads and repeated each experiment 10 times to balance the accuracy of variation measurement with the total time taken to complete these experiments. We then characterized performance variation from several angles: throughput, latency, temporally, spatially, and more. We found that performance variation depends heavily on the specific configuration of the system. We then further dove into the details, analyzed and explained certain performance variations. For example: we found that unpredictable layouts in Ext4 could cause over 16–19% of performance variation in some cases. We discovered that the magnitude of variation also depends on the observation window size: in one workload, 40% of XFS configurations exhibited higher than 20% variation with a window size of 60s, but almost all of them stabilized when the window size grew to 400s. Finally, we analyzed latency variations from various aspects, and proposed a novel approach for quantifying the impacts of each operation type on overall performance variation.

We summarize key contributions of our performance variation study as follows: ■ **(1)** To the best of our knowledge, we are the first to provide a detailed characterization of performance variation occurring in benchmarking a typical modern storage system. We believe our study paves the way towards the better understanding of complex storage system performance variations, in both experimental and production settings. ■ **(2)** We conducted a comprehensive study of storage system performance variation. Our analysis includes throughput and latency, and both spatial and temporal variations. ■ **(3)** We offer insights into the root causes of some performance variations, which could help anyone who seeks stable results from benchmarking storage systems, and encourage more follow-up work in understanding variations in production systems.

This study has been published in FAST 2017 [28]. The rest of the chapter is organized as follows. Section 6.2 explains background knowledge. Section 6.3 describes our experimental methodology. Section 6.4 covers related work on storage performance variation. We list our experimental settings in Section 6.5. Section 6.6 evaluates performance variations from multiple dimensions. .

6.2 Background

The storage system is an essential part of modern computer systems, and critical to the performance of data-intensive applications. Often, the storage system is the slowest component and thus is one of the main contributors to the overall variability in a system's performance. Characterizing this variation in storage system performance is therefore essential for understanding overall system-performance variation.

We first define common performance metrics and notations used in this chapter. *Throughput* is defined as the average number of I/O operations completed per second. Here we use a "*Throughput-N*" notation to represent the throughput within the last N seconds of an observation. There are two types of throughput that are used most frequently in our analysis. One is *cumulative* throughput, defined as the throughout from the beginning to the end of the experiment. In this chapter, cumulative throughput is the same as *Throughput-800* or *Throughput-2000*, because the complete runtime of a single experiment was either 800 or 2,000 seconds, depending on the workload. The other type is called *instantaneous* throughput, which we denote as *Throughput-10*. Ten seconds is the smallest time unit we collected performance for, in order to avoid too much overhead.

6.2.1 Measures of Variation

Since our goal is to characterize and analyze collected experimental data, we mainly use concepts from *descriptive statistics*. *Statistical variation* is closely related to *central tendency*, which is an estimate of the *center* of a set of values. *Variation* (also called *dispersion* or *variability*), refers to the spread of the values around the central tendency. We considered the most commonly used measure for central tendency—the *mean*.

$$\bar{x} = \sum_{i=1}^{N} x_i. \tag{6.1}$$

Here, x_i is the value number i and we have N such values in total (e.g., collected from experiments).

In descriptive statistics, a measure of variation is usually a non-negative real number that is zero if all readings are the same and increases as the measurements become more dispersed. To reasonably compare variations across datasets with different mean values, it is common to normalize the variation by dividing any absolute metric of variation by the mean value. There are several different metrics for variation. We initially considered two that are most commonly used in descriptive statistical analysis:

- *Relative Standard Deviation (RSD)*: the RSD, (or *Coefficient of Variation (CV)*) is

$$RSD = \frac{\sqrt{\frac{1}{N-1}\sum_{i=1}^{N}(x_i - \bar{x})^2}}{\bar{x}} \tag{6.2}$$

- *Relative Range*: this is defined as the difference between the smallest and largest values:

$$RelativeRange = \frac{max(X) - min(X)}{\bar{x}} \tag{6.3}$$

Because a range uses maximum and minimum values in its calculation, it is more sensitive to outliers. We did not want to exclude or otherwise diminish the significance of performance outliers. We found that even a few long-running I/O operations can substantially worsen actual user experience due to outliers (which are re-producible). Such outliers have real-world impact, especially as more services are offloaded to the cloud, and customers demand QoS guarantees through SLAs. That is one reason why researchers recently have begun to focus on tail latencies [47, 78, 80]. In considering the two metrics above, we felt that the RSD hides some of the magnitudes of these variations—because using square root tends to "compress" the outliers' values. We therefore decided to use the *Relative Range* as our main metric of variation in the rest of this chapter.

6.3 Methodology

Although we encountered storage system performance variations in past projects, we were especially struck by this issue in our recent experiments on automated recognition of optimal storage configurations. We found that multiple runs of the same workload in a carefully controlled environment could sometimes produce quite unstable results. We later observed that performance variations and their magnitude depend heavily on the specific configuration of the storage system. Over 18% of 24,888 different storage configurations that we evaluated (repeatedly over several workloads) exhibited results with a relative range higher than 9% and relative standard deviation higher than 5%.

Workloads also impact the degree of performance variation significantly. For the same configuration, experiments with different workloads could produce different magnitudes of variation. For example, we found one Btrfs configuration produces variation with over 40% relative range value on one workload but only 6% for another. All these findings led us to study the characteristics and analyze performance variations in benchmarking various storage configurations under multiple workloads. Due to the high complexity of storage systems, we have to apply certain methodologies in designing and conducting our experiments.

Reducing the parameter space In this chapter we focus on evaluating *local* storage systems (e.g., Ext4, Linux block layer, SSD). This is a useful basis for studying more complex distributed storage systems (e.g., Ceph [203], Lustre [139], GPFS [167], OpenStack Swift [151]). Even a small variation in local storage system performance can result in significant performance fluctuations in large-scale distributed system that builds on it [47, 131, 144].

Despite its simple architecture, a local storage system can still have a large number of parameters at every layer, resulting in a vast number of possible configurations. For instance, common parameters for a typical local file system include block size, inode size, journal options, and many more. It is prohibitively time consuming and impractical to evaluate every possible configuration exhaustively. As shown in Table 6.1, Ext4 has 59 unique parameters that can have anywhere from 2 to numerous allowed values each. If one experiment runs for 15 minutes and we conduct 10 runs

Parameter Space	# Unique Parameters	# Unique Configurations	Time (years)
Ext4	59	2.7×10^{37}	7.8×10^{33}
XFS	37	1.4×10^{19}	4.1×10^{15}
Btrfs	54	8.8×10^{26}	2.5×10^{23}
Expert Space	10	1,782	1.52
Sample Space	10	107	33.4 days

Table 6.1: Comparison for parameter spaces. Time is computed by assuming 15 minutes per experimental run, 10 runs per configuration and 3 workloads in total.

for each configuration, it will take us 7.8×10^{33} years of clock time to finish evaluating all Ext4 configurations.

Therefore, our first task was to reduce the parameter space (as compared with *Storage V2* in Table 4.3) for our experiments by carefully selecting the most relevant storage system parameter.. This selection was done in close collaboration with several storage experts that have either contributed to storage system designs or have spent years tuning storage systems in the field. We experimented with three popular file systems that span a range of designs and features. ■ **(1) Ext4** [59] is a popular file system that inherits a lot of internal structures from Ext3 [27] and FFS [136]) but enhances performance and scalability using extents and delayed allocation. ■ **(2) XFS** [173, 185] was initially designed for SGI's IRIX OS [185] and was later ported to Linux. It has attracted users' attention since the 90s thanks to its high performance on new storage devices and its high scalability regarding large files, large numbers of files, and large directories. XFS uses B+ trees for tracking free extents, indexing directory entries, and keeping track of dynamically allocated inodes. ■ **(3) Btrfs** [25, 161] is a complex file system that has seen extensive development since 2007 [161]. It uses copy-on-write (CoW), allowing efficient snapshots and clones. It has its own LVM and uses B-trees as its main on-disk data structure. These unique features are garnering attention and we expect Btrfs to gain even greater popularity in the future.

For the three file systems above we experimented with the following nine parameters. ■ **(1) Block size**. This is a group of contiguous sectors and is the basic unit of space allocation in a file system. Improper block size selection can reduce file system performance by orders of magnitude [80]. ■ **(2) Inode size**. This is one of the most basic on-disk structures of a file system [9]. It stores the metadata of a given file, such as its size, permissions, and the location of its data blocks. The inode is involved in nearly every I/O operation and thus plays a crucial role for performance, especially for metadata-intensive workloads. ■ **(3) Journal mode**. Journaling is the write-ahead logging implemented by file systems for recovery purposes in case of power losses and crashes. In Ext4, three types of journaling modes are supported: *writeback*, *ordered*, and *journal* [60]. The *writeback* mode journals only metadata whereas the *journal* mode provides full data and metadata journaling. In *ordered* mode, Ext4 journals metadata only, but all data is forced directly out to the disk prior to its metadata being committed to the journal. There is a trade-off between file system consistency and performance, as journaling generally adds I/O overhead. In comparison, XFS implements metadata journaling, which is similar to Ext4's *writeback* mode, and there is no need for journaling in Btrfs because of its CoW nature. ■ **(4) Allocation Group (AG) count**. This parameter is specific to XFS which partitions its space into regions called Allocation Groups [185]. Each AG has its own data structures for managing free space and inodes within its boundaries. ■ **(5) Nodatacow** is a Btrfs mount-time option that turns the CoW feature on or off for data blocks. When

File System	Parameter	Value Range
Ext4	Block Size	1024, 2048, 4096
	Inode Size	128, 512, 2048, 8192
	Journal Mode	data=journal, ordered, writeback
XFS	Block Size	1024, 2048, 4096
	Inode Size	256, 512, 1024, 2048
	AG Count	8, 32, 128, 512
Btrfs	Node Size	4096, 16384, 65536
	Special Options	nodatacow, nodatasum, default
All	atime Options	relatime, noatime
	I/O Scheduler	noop, deadline, cfq
	Storage Devices	HDD (SAS, SATA), SSD (SATA)

Table 6.2: List of parameters and value ranges.

data CoW is enabled, Btrfs creates a new version of an extent or a page at a newly allocated space [161]. This allows Btrfs to avoid any partial updates in case of a power failure. When data CoW is disabled, partially written blocks are possible on system failures. In Btrfs, *nodatacow* implies *nodatasum* and compression disabled. ■ **(6) Nodatasum** is a Btrfs mount-time option and when specified, it disables checksums for newly created files. Checksums are the primary mechanism used by modern storage systems to preserve data integrity [9], computed using hash functions such as SHA-1 or MD5. ■ **(7) atime Options**. These refer to mount options that control the inode access time. We experimented with *noatime* and *relatime* values. The *noatime* option tells the file system not to update the inode access time when a file data read is made. When *relatime* is set, atime will only be updated when the file's modification time is newer than the access time or atime is older than a defined interval (one day by default). ■ **(8) I/O scheduler**. The I/O Scheduler manages the submission of block I/O operations to storage devices. The choice of I/O scheduler can have a significant impact on storage system performance [18]. We used the *noop*, *deadline*, and *Completely Fair Queuing (CFQ)* I/O schedulers. Briefly explained, the *noop* scheduler inserts all incoming I/O requests into a simple FIFO queue in order of arrival; the *deadline* scheduler associates a deadline with all I/O operations to prevent starvation of requests; and the *CFQ* scheduler try to provide a fair allocation of disk I/O bandwidth for all processes that requests I/O operations. ■ **(9) Storage device**. The underlying storage device plays an important role in nearly every I/O operation. We ran our experiments on three types of devices: two HDDs (SATA vs. SAS) and one (SATA) SSD.

Table 6.2 summarizes all parameters and the values used in our experiments.

Latin Hypercube Sampling Reducing the parameter space to the most relevant parameters based on expert knowledge resulted in 1,782 unique configurations ("Expert Space" in Table 6.1). However, it would still take more than 1.5 years to complete the evaluation of every configuration in that space. To reduce the space further, we intelligently sampled it using *Latin Hypercube Sampling (LHS)*, a method often used to construct computer experiments in multi-dimensional parameter spaces [86, 135]. LHS can help explore a search space and discover unexpected behavior among combinations of parameter values; this suited our needs here. In statistics, a *Latin Square* is defined as a two-dimensional square grid where each row and column have only one sample;

Latin Hypercube generalizes this to multiple dimensions and ensures that each sample is the only one in the axis-aligned hyper-plane containing it [135]. Using LHS, we were able to sample 107 representative configurations from the Expert Space and complete the evaluation within 34 days of clock time (excluding lengthy analysis time). We believe this approach is a good starting point for a detailed characterization and understanding of performance variation in storage systems.

6.4 Related Work

To the best of our knowledge, there are no systematic studies of performance variation of storage systems. Most previous work focuses on long-tail I/O latencies. Tarasov et al. [188] observed that file system performance could be sensitive to even small changes in running workloads. Arpaci-Dusseau [8] proposed an I/O programming environment to cope with performance variations in clustered platforms. Worn-out SSDs exhibit high latency variations [49]. Hao et al. [77] studied device-level performance stability, for HDDs and SSDs.

For long-tail latencies of file systems, He et al. [80] developed Chopper, a tool to explore a large input space of file system parameters and find behaviors that lead to performance problems; they analyzed long-tail latencies relating to block allocation in Ext4. In comparison, our goal is broader: a detailed characterization and analysis of several aspects of storage system performance variation, including devices, block layer, and the file systems. We studied the variation in terms of both throughput and latency, and both spatially and temporally. Tail latencies are common in network or cloud services [47,115]: several tried to characterize and mitigate their effects [78,88,183,210], as well as exploit them to save data center energy [197]. Li et al. [115] characterized tail latencies for networked services from the hardware, OS, and application-level sources. Dean and Barroso [47] pointed out that small performance variations could affect a significant fraction of requests in large-scale distributed systems, and can arise from various sources; they suggested that eliminating all of them in large-scale systems is impractical. We believe there are possibly many sources of performance variation in storage systems, and we hope this work paves the way for discovering and addressing their impacts.

6.5 Experimental Setup and Workloads

All experiments from this chapter were conducted on *S3* machines (see Table 4.1). We characterized variations on three storage devices, HDD2, HDD4, and SSD in Table 4.1. We use *SAS-HDD* to refer HDD2, and *SATA-HDD* for *HDD4*. When discussing results on both HDD devices, we just refer them together as *HDD* for short. Workload settings were described in Table 4.2, denoted as “*-heavy”. As the average file size is an inherent property of a workload and should not be changed [190], the dataset size is determined by the number of files. We increased the number of files such that the dataset size is 10GB—2.5× the machine RAM size. By fixing the dataset size, we normalized the experiments’ set-size and run-time, and ensured that the experiments run long enough to produce enough I/O. With these settings, our experiments exercise both in-memory cache and persistent storage devices [189].

We did not perform a separate cache warm-up phase in our experiments because in this study we were interested in performance variation that occurred *both* with cold and warm caches [189].

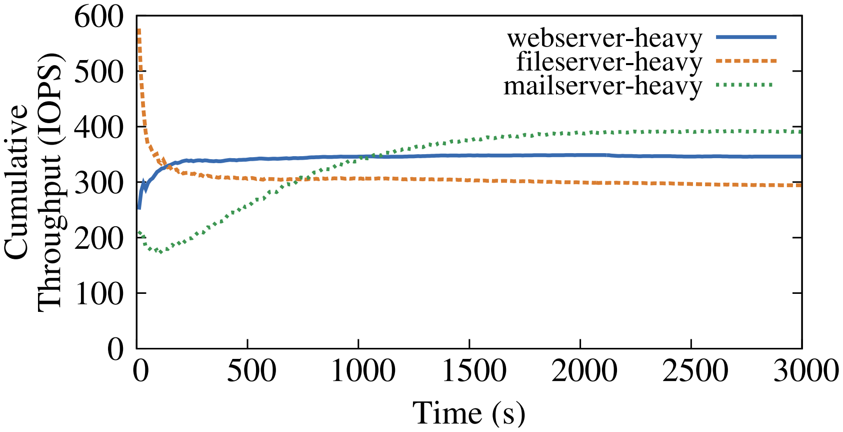


Figure 6.1: Cumulative throughput over time for one Ext4 configuration under multiple workloads. Each workload ran for 7,200s; only the first 3,000s are plotted.

The default running time for Filebench is set to 60 seconds, which is too short to warm the cache up. We therefore conducted a "calibration" phase to pick a running time that was long enough for the cumulative throughput to stabilize. We ran each workload for up to 2 hours for testing purposes. To find a suitable run time, we ran each workload for 7,200 seconds, and measured its cumulative throughput. Figure 6.1 shows the first 3,000 seconds for Ext4 configurations. In this chapter we define the cumulative throughput as the average number of I/O operations completed per second since the start of the experiment. We can see that Fileserver and Webserver took around 600 seconds to achieve stable cumulative throughputs, and Mailserver took about 1,800 seconds. We ran the same experiments multiple times, for all file systems (Ext4, XFS, and Btrfs), and we found similar behavior. Therefore, if not stated otherwise, we set the default running time to 800 seconds for Fileserver and Webserver, and to 2,000 seconds for Mailserver. We have other choices of running time in several supplement experiments as well. We also let Filebench output the throughput (and other performance metrics) every 10 seconds, to capture and analyze performance variation from a short-term view.

6.6 Evaluation

In this section we are characterizing and analyzing storage performance variation from a variety of angles. These experiments represent a large amount of data, and therefore, we first present the information with brief explanations, and in subsequent subsections we dive into detailed explanations. Section 6.6.1 gives an overview of performance variations found in various storage configurations and workloads. Section 6.6.2 describes a case study by using Ext4-HDD configurations with the Fileserver workload. Section 6.6.3 presents temporal variation results. Here, temporal variations consist of two parts: changes of throughput over time and latency variation.

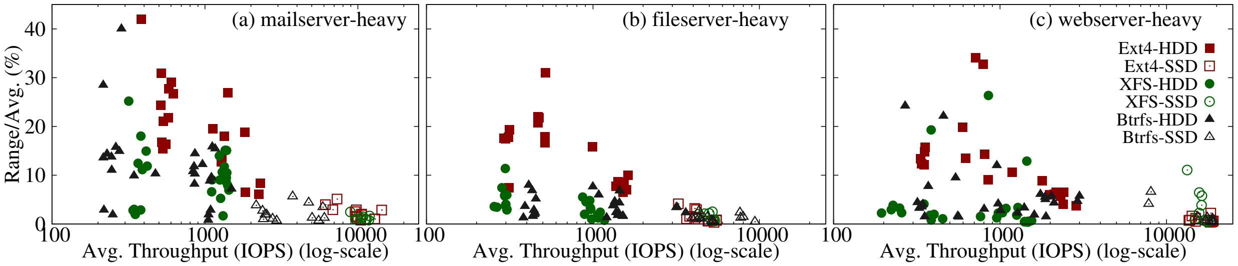


Figure 6.2: Overview of performance and its variation with different storage configurations under three workloads: (a) maileserver-heavy, (b) fileserver-heavy, and (c) webserver-heavy. The X axis represents the mean of throughput over 10 runs; the Y axis shows the relative range of cumulative throughput. Ext4 configurations are represented with squares, XFS with circles, and Btrfs with triangles. HDD configurations are shown with filled symbols, and SSDs with hollow ones.

6.6.1 Variation at a Glance

We first overview storage system performance variation and how configurations and workloads impact its magnitude. We designed our experiments by applying the methodology described in Section 6.3. We benchmarked configurations from the Sample Space (see Table 6.1) under three representative workloads from Filebench. The workload characteristics are shown in Table 4.2. We repeated each experiment 10 times in a carefully-controlled environment in order to get unperturbed measurements.

Figure 6.2 shows the results as scatter plots broken into the three workloads: *mailserver-heavy* (Figure 6.2(a)), *fileserver-heavy* (Figure 6.2(b)), and *webserver-heavy* (6.2(c)). Each symbol represents one storage configuration. We use squares for Ext4, circles for XFS, and triangles for Btrfs. Hollow symbols are SSD configurations, while filled symbols are for HDD. We collected the cumulative throughput for each run. As described in Section 6.2, we define the cumulative throughput as the average number of I/O operations completed per second throughout each experiment run. This can also be represented as *Throughput-800* for *fileserver-heavy* and *webserver-heavy*, and *Throughput-2000* for *mailserver-heavy*, as per our notation. In each subfigure, the Y axis represents the relative range of cumulative throughputs across the 10 runs. As explained in Section 6.2, here we use the relative range as the measure of variation. A higher relative range value indicates higher degree of variation. The X axis shows the mean cumulative throughput across the runs; higher values indicate better performance. Since performance for SSD configurations is usually much better than HDD configurations, we present the X axis in log_{10} scale.

Figure 6.2 shows that HDD configurations are generally slower in terms of throughput but show a higher variation, compared with SSDs. For HDDs, throughput varies from 200 to around 2,000 IOPS, and the relative range varies from less than 2% to as high as 42%. Conversely, SSD configurations usually have much higher throughput than HDDs, ranging from 2,000 to 20,000 IOPS depending on the workload. However, most of them exhibit variation less than 5%. The highest range for any SSD configurations we evaluated was 11%.

Ext4 generally exhibited the highest performance variation among the three evaluated file systems. For the *mailserver-heavy* workload, most Ext4-HDD configurations had a relative range higher than 12%, with the highest one being 42%. The *fileserver-heavy* workload was slightly better, with the highest relative range being 31%. Half of the Ext4-HDD configurations show variation

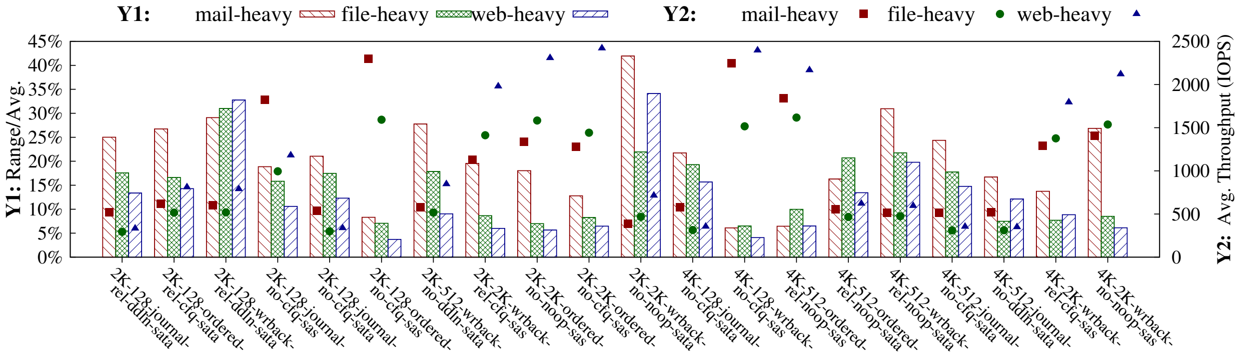


Figure 6.3: Storage system performance variation with 20 sampled Ext4-HDD configurations under three workloads. The range is computed among 10 experiment runs, and is represented as bars corresponding to the Y1 (left) axis. The mean of throughput among the 10 runs is shown with symbols (squares, circles, and triangles), and corresponds to the Y2 (right) axis. The X axis represents configurations formatted by ⟨block size - inode size - journal - atime - I/O scheduler - device⟩.

higher than 15% and the rest between 5–10%. For *webserver-heavy*, the Ext4-HDD configuration varies between 6–34%. All Ext4-SSD configurations are quite stable in terms of performance variation, with less than 5% relative range.

Btrfs configurations show a moderate level of variation in our evaluation results. For *mailserver-heavy*, two Btrfs-HDD configurations exhibited 40% and 28% ranges of throughput, and all others remained under 15%. Btrfs was quite stable under the *fileserver-heavy* workload, with the highest variation being 8%. The highest relative range value we found for Btrfs-HDD configurations under *webserver-heavy* is 24%, but most of them were below 10%. Similar to Ext4, Btrfs-SSD configurations were also quite stable, with a maximum variation of 7%.

XFS had the least amount of variation among the three file systems, and is fairly stable in most cases, as others have reported before, albeit with respect to tail latencies [80]. For *mailserver-heavy*, the highest variation we found for XFS-HDD configurations was 25%. In comparison, Ext4 was 42% and Btrfs was 40%. Most XFS-HDD configurations show variation smaller than 5% under *fileserver-heavy* and *webserver-heavy* workloads, except for one with 11% for *fileserver-heavy* and three between 12–23% for *webserver-heavy*. Interestingly, however, across all experiments for all three workloads conducted on SSD configurations, the highest variation was observed on one XFS configuration using the *webserver-heavy* workload, which had a relative range value of 11%.

Next, we decided to investigate the effect of workloads on performance variation in storage systems. Figure 6.3 compares the results of the same storage configurations under three workloads. These results were extracted from the same experiments shown in Figure 6.2. Although we show here only all Ext4-HDD configurations, we have similar conclusions for other file systems and for SSDs. The bars represent the relative range of 10 repeated runs, and correspond to the left Y1 axis. The average throughput of 10 runs for each configuration is shown as symbols, and corresponds to the right Y2 axis. The X axis consists of configuration details, and is formatted as the six-part tuple ⟨*block size* - *inode size* - *journal option* - *atime option* - *I/O scheduler* - *device*⟩. We can see that some configurations remain unstable in all workloads. For example, the configuration *2K-128-writeback-relatime-deadline-SATA* exhibited high performance variation (around 30%) under all three workloads. However, for some configurations, the actual workload played an important role in the magnitude of variation. For example, in the configuration *2K-2K-writeback-noatime-noop-SATA*, the *mailserver-heavy* workload varies the most; but in the configuration *4K-512-ordered-relatime-noop-SATA*, the highest range of performance was seen on *fileserver-heavy*. Finally, configurations with SAS HDD drives tended to have a much lower range variation but higher average throughput than SATA drives.

6.6.2 Case Study: Ext4

Identifying root causes for performance variation in the storage system is a challenging task, even in experimental settings. Many components in a modern computer system are not isolated, with complex interactions among components. CPU, main memory, and secondary storage could all contribute to storage variation. Our goal was not to solve the variation problem completely, but to report and explain this problem as thoroughly as we could. We leave to future work to address these root causes from the source code level [194]. At this stage, we concentrated our efforts solely on benchmarking local storage systems, and tried to reduce the variation to an acceptable level. In this section we describe a case study using four Ext4 configurations as examples. We focused on Ext4-HDD (SATA) here, as this combination of file systems and device types produced the highest variations in our experiments (see Figures 6.2 and 6.3).

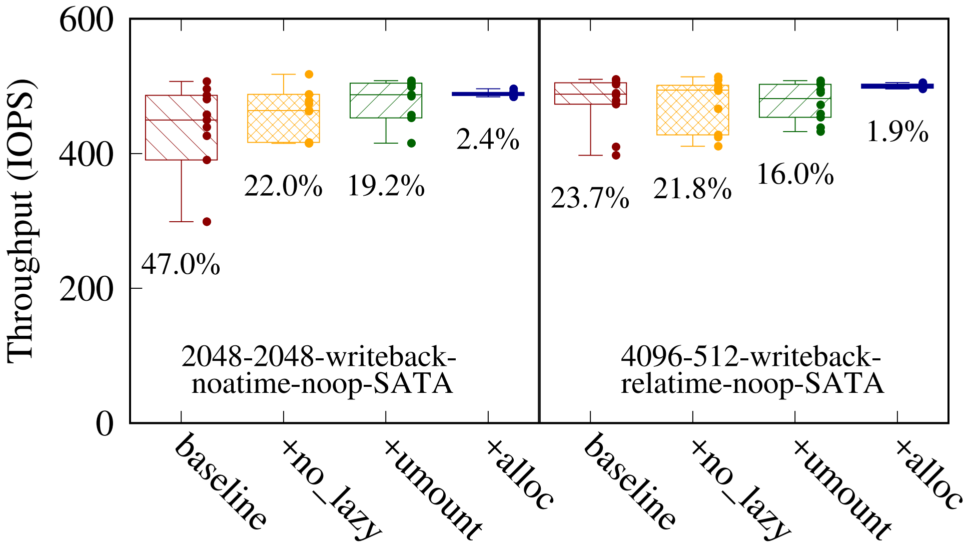


Figure 6.4: Performance variation for 2 Ext4-HDD configurations with several diagnoses. Each experiment is shown as one box, representing a throughput distribution for 10 identical runs. The top border line of each box marks the 1^{st} quartile; the bottom border marks the 3^{rd} quartile; the line in the middle is the median throughput; and the whiskers mark maximum and minimum values. The dots to the right of each box show the exact throughputs of all 10 runs. The percentage numbers below each box are the relative range values. The bottom label shows configuration details for each figure.

Figure 6.4 shows results as two boxplots for the *fileserver-heavy* workload, where each box plots the distribution of throughputs across the 10 runs, with the relative range shown below. The top border represents the 1^{st} quartile, the bottom border the 3^{rd} quartile, and the line in the middle is the median value. Whiskers show the maximum and minimum throughputs. We also plotted one dot for the throughput of each run, overlapping with the boxes but shifted to the right for easier viewing. The X axis represents the relative improvements that we applied based on our successive investigations and uncovering of root causes of performance variation, while the Y axis shows the cumulative throughput for each experiment run. Note that the improvement label is prefixed with a "+" sign, meaning that an additional feature was added to the previous configuration, cumulatively. For example, *+umount* actually indicates *baseline* + *no_lazy* + *umount*. We also added labels on the bottom of each subfigure showing the configuration details, formatted as ⟨*block size - inode size - journal option - atime option - I/O scheduler - device*⟩.

After addressing all causes we found, we were able to reduce the relative range of throughput in these configurations from as high as 47% to around 2%. In the rest of this section, we detail each root cause and how we addressed it.

Baseline The first box for each subfigure in Figure 6.4 represents our original experiment setting, labeled *baseline*. In this setting, before each experimental run, we format and mount the file system with the targeted configuration. Filebench then creates the dataset on the mounted file system. After the dataset is created, Filebench issues the *sync* command to flush all dirty data and metadata to the underlying device (here, SATA HDD); Filebench then issues an *echo 3 > /proc/sys/vm/drop_caches* command, to evict non-dirty data and metadata from the page cache. Then, Filebench runs the Fileserver workload for a pre-defined amount of time (see Table 4.2). For this baseline setting, both Ext4-HDD configurations show high variation in terms of throughput, with range values of 47% (left) and 24% (right).

Lazy initialization The first contributor to performance variation that we identified in Ext4-HDD configurations is related to the lazy initialization mechanism in Ext4. By default, Ext4 does not immediately initialize the complete inode table. Instead, it gradually initializes it in the background when the created file system is first mounted, using a kernel thread called *ext4lazyinit*. After the initialization is done, the thread is destroyed. This feature speeds up the formatting process significantly, but also causes interference with the running workload. By disabling it during format time, we reduced the range of throughput from 47% to 22% for Configuration *2048-2048-writeback-noatime-noop-SATA*. This improvement is labelled *+no_lazy* in Figure 6.4.

Sync then umount In Linux, when *sync* is called, it only guarantees to *schedule* the dirty blocks for writing: there is often a delay until all blocks are actually written to stable media [149, 186]. Therefore, instead of calling *sync*, we *umount* the file system each time after finishing creating the dataset and then *mount* it back, which is labelled as *+umount* in Figure 6.4. After applying this, both Ext4-HDD configurations exhibited even lower variation than the previous setting (disabling lazy initialization only).

Block allocation and layout After applying the above improvements, both configurations still exhibited higher than 16% variations, which could be unacceptable in settings that require more predictable performance. This inspired us to try an even more strictly-controlled set of experiments. In the *baseline* experiments, by default we re-created the file system before each run and then Filebench created the dataset. We assumed that this approach would result in identical datasets among different experiment runs. However, block allocation is not a deterministic procedure in Ext4 [80]. Even given the same distribution of file sizes and directory width, and also the same number of files as defined by Filebench, multiple trials of dataset creation on a freshly formatted, clean file system did not guarantee to allocate blocks from the same or even near physical locations on the hard disk. To verify this, instead of re-creating the file system before each run, we first created the file system and the desired dataset on it. We then dumped out the entire partition image using *dd*. Then, before each run of Filebench, we used *dd* to restore the partition using the image, and mounted the file system back. This approach guaranteed an identical block layout for each run. Figure 6.4 shows these results using *+alloc*. We can see that for both Ext4-HDD configurations, we were able to achieve around 2% of variation, which verified our hypothesis that block allocation and layout play an important role in the performance variation for Ext4-HDD configurations.

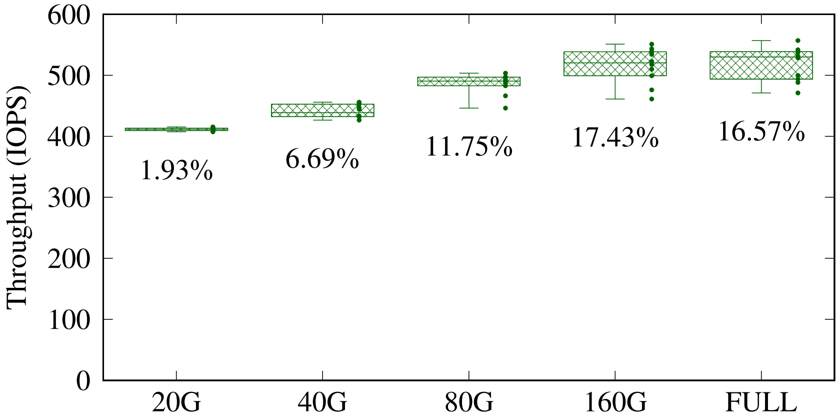


Figure 6.5: Performance variation for Ext4-HDD configuration under the Fileserver workload with different partition sizes from inner tracks of disks

After further investigation, we found this nondeterminism for Ext4 block allocation was caused by the fact that *Ext4 always tries to spread first-level directories* [61]. In the meanwhile, Filebench puts its dataset in one directory (with pre-defined directory width and depth distribution), directly under the mount point of the targeted file system. To prove this, we conducted a set of experiments by varying the partition size of the underlying hard disk. As shown in Figure 6.5, we experimented with *20G*, *40G*, *80G*, *160G* and *Full-disk* partitions. All partitions start from inner tracks of disks. We repeated each experiment for 10 runs. The meanings of boxes, whiskers, and dots are the same with those of Figure 6.4. Remember the dataset size in our experiments is 10G (see Section 6.5). When the partition size is 20G, the difference in physical positions of allocated files among 10 experiment runs could be quite small, which results in a relative range of 1.9% in final throughput values. They all clustered in inner tracks of disks. As we increase the partition size, the relative range of throughput also increases. This is because with larger partition sizes, in different experiment runs Ext4 could allocate all blocks in physically different "clusters" across the disks. Datasets allocated in the outer tracks will result in higher final throughputs, while inner tracks produce lower results. This also explains the increasing trend of the average throughput among these experiments.

Storing the images of file systems using the *dd* command, however, could be too costly in prac-

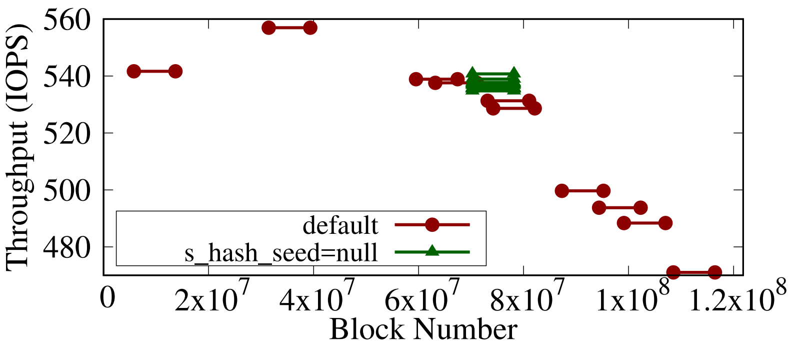


Figure 6.6: Physical blocks of allocated files in Ext4 under the Fileserver workload. The X axis represents the physical block number of each file in the dataset. Since the *Fileserver* workload consists of small files, and one extent per file, we use the starting block number for each file here. The Y axis is the final cumulative throughput for each experiment run. Note that the Y axis does not start from 0. Lines marked with solid circles are experiment runs with the default setting; lines with triangles represent experiment runs where we set the field s_hash_seed in Ext4s's superblock to null.

tice, taking hours of clock time. We found a faster method to generate reproducible Ext4 layouts by setting the *s_hash_seed* field in Ext4's superblock to *null* before mounting. Figure 6.6 shows the distribution of physical blocks for allocated files in two sets of *fileserver-heavy* experiments on Ext4. This workload consists of only small files, resulting in exactly one extent for each file in Ext4, so we used the starting block number (X axis) to represent the corresponding file. The Y axis shows the final cumulative throughput for each experiment run. Here the lines starting and ending with solid circles are 10 runs from the experiment with the full-disk partition. The lines with triangles represent the same experiments, but here we set the *s_hash_seed* field in Ext4's superblock to *null*. We can see that files in each experiment run are allocated into one cluster within a small range of physical block numbers. In most cases, experimental runs with their dataset allocated near the outer tracks of disks, which correspond to smaller block numbers, tend to produce higher throughput. As shown in Figure 6.6, with the default setting, datasets of 10 runs clustered in 10 different regions of the disk, causing high throughput variation across the runs. By setting the Ext4 superblock parameter *s_hash_seed* to *null*, we can eliminate the non-determinism in block allocation. This parameter determines the group number of top-level directories. By default, *s_hash_seed* is randomly generated during format time, resulting in distributing top-level directories all across the LBA space. Setting it to *null* forces Ext4 to use the hard-coded default values, and thus the top-level directory in our dataset is allocated on the same position among different experiment runs. As we can see from Figure 6.6, for the second set of experiments, the ranges of allocated block numbers in all 10 experiment runs were exactly the same. When we set the *s_hash_seed* parameter to *null*, the relative range of throughput dropped from and 16.6% to 1.1%. Therefore, setting this parameter could be useful when users want stable benchmarking results from Ext4. In addition to the case study we conducted on Ext4-HDD configurations, we also observed similar results for Ext4 on other workloads, as well as for Btrfs. For two of the Btrfs-HDD configurations, we were able to reduce the variation to around 1.2%, by using *dd* to store the partition image. We did not try to apply any improvements on XFS, since most of its configurations were already quite stable (in terms of cumulative throughput) even with the *baseline* setting, as shown in Figure 6.2.

6.6.3 Temporal Variation

In Section 6.6.1 and 6.6.2, we mainly presented and analyzed performance variation among repeated runs of the same experiment, and only in terms of throughput. Variation can actually manifest itself in many other ways. We now focus our attention on *temporal* variations in storage system performance—the variation related to time. Section 6.6.3.1 discusses temporal throughput variations and Section 6.6.3.2 focuses on latency variations.

6.6.3.1 Throughput over Time

After finding variations in cumulative throughputs, we set out to investigate whether the performance variation changes over time within single experiment run.

To characterize this, we calculated the throughput within a small time window. As defined in Section 6.2, we denote throughput with window size of N seconds as *Throughput-N*. Figure 6.7 shows the *Throughput-120* value (Y axis) over time (X axis) for Btrfs-HDD, XFS-HDD, and Ext4-HDD configurations using the *Fileserver* workload.

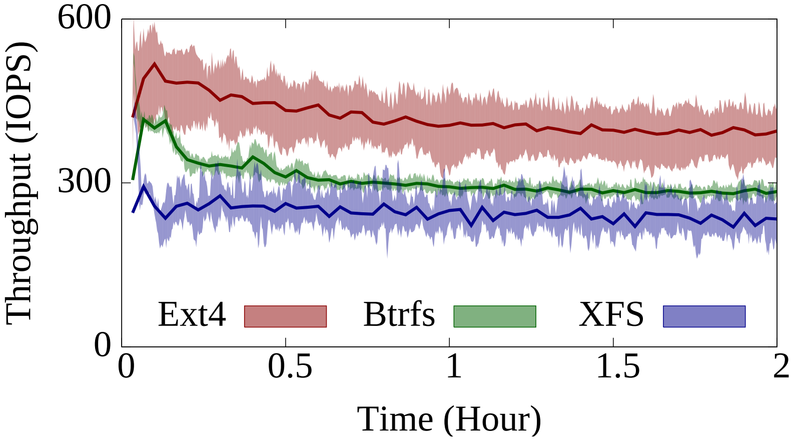


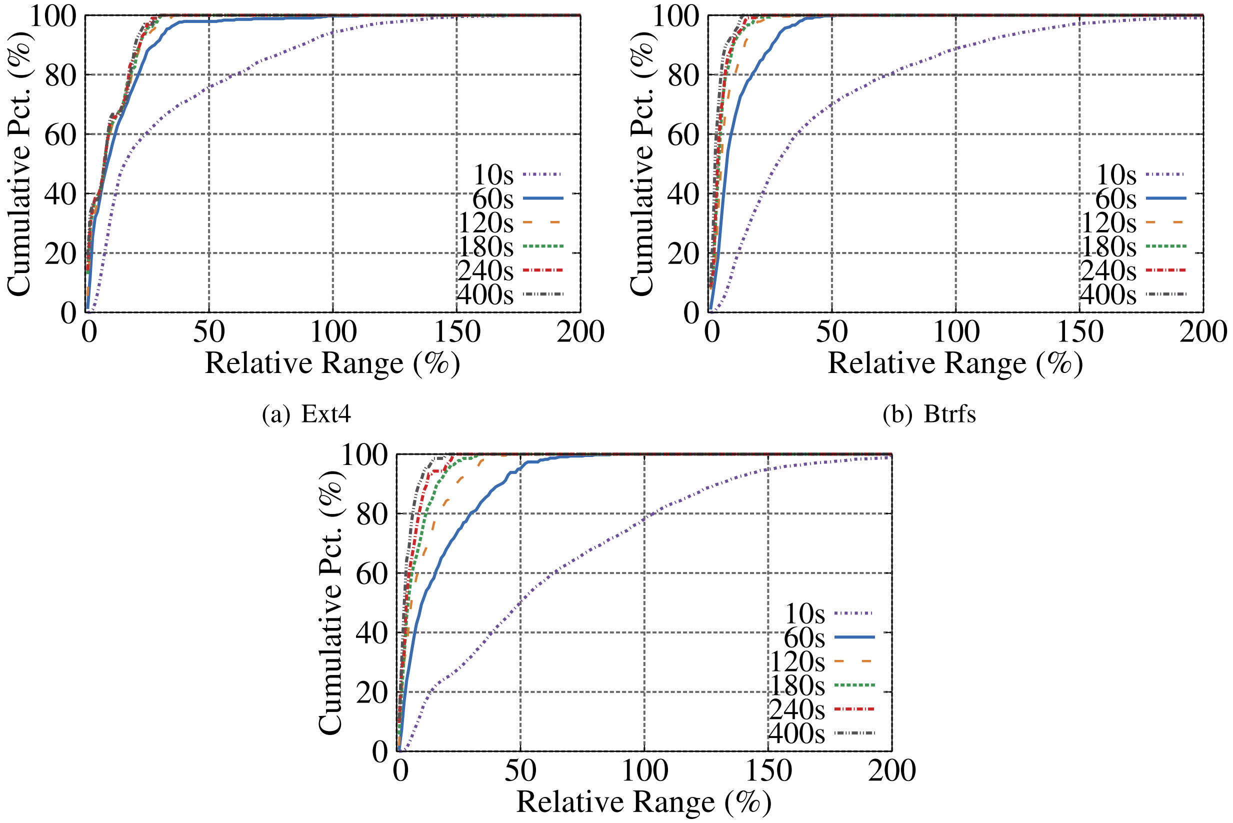
Figure 6.7: *Throughput-120* over time for Btrfs, XFS, and Ext4 HDD configurations under the Fileserver workload. Each configuration was evaluated for 10 runs. Two lines were plotted connecting maximum and minimum throughput values among 10 runs. We fill in colors between two lines, green for Btrfs, red for Ext4, and blue for XFS. We also plotted the average *Throughput-120* among 10 runs as a line running through the band. The maximum relative range values of *Throughput-120* for Ext4, Btrfs, and XFS are 43%, 23%, and 65%, while the minimum values are 14%, 2%, and 7%, respectively.

Here we use a window size of 120 seconds, meaning that each throughput value is defined as the average number of I/O operations completed per second with the latest 120 seconds. We also investigated other window sizes, which we discuss later. The three configurations shown here exhibited high variations in the experiments discussed in Section 6.6.1. Also, to show the temporal aspect of throughput better, we extended the running time of this experiment set to 2 hours, and we repeated each experiment 10 times. Two lines are plotted connecting the maximum and minimum throughput values among 10 runs. We fill in colors between two lines, this producing a color band: green for Btrfs, red for Ext4, and blue for XFS. The line in the middle of each band is plotted by connecting the average *Throughput-120* value among 10 runs. We observed in Figure 6.2(b) that for the *fileserver-heavy* workload, Ext4-HDD configurations generally exhibited higher variations than XFS-HDD or Btrfs-HDD configurations in terms of final cumulative throughput. However, when it comes to *Throughput-120* values, Figure 6.7 leads to some different conclusions. The Ext4-HDD configuration still exhibited high variation in terms of short-term throughout across the 2 hours of experiment time, while the Btrfs-HDD configuration is much more stable. Surprisingly, the XFS-HDD configuration has higher than 30% relative range of *Throughput-120* values for most of the experiment time, while its range for cumulative throughput is around 2%. This suggests that XFS-HDD configurations might exhibit high variations with shorter time windows, but produces more stable results in longer windows. It also indicates that the choice of window sizes matters when discussing performance variations.

We can see from the three average lines in Figure 6.7 that performance variation exists even within one single run—the short-term throughput varies as the experiment proceeds. For most experiments, no matter what the file system type is, performance starts slow and climbs up quickly in the beginning phase of experiments. This is because initially the application is reading cold data and metadata from physical devices into the caches; once cached, performance improves. Also, for some period of time, dirty data is kept in the cache and not yet flushed to stable media, delaying any impending slow writes. After an initial peak, performance begins to drop rapidly and then declines steadily. This is because the read performance already reached its peak and cached dirty data begins to be flushed out to slower media. Around several minutes in, performance begins to stabilize, as we see the throughput lines flatten.

The unexpected difference in variations for short-term and cumulative throughput of XFS-HDD configurations lead us to investigate the effects of the time window size on performance variations. We calculated the relative range of throughput with different window sizes for all configurations within each file system type. We present the CDFs of these range values in Figure 6.8. For example, we conducted experiments on 39 Btrfs configurations. With a window size of 60 seconds and total running time of 800 seconds, the corresponding CDF for Btrfs is based on $39 \times \frac{800}{60} = 507$ relative range values. We can see that Ext4's unstable configurations are largely unaffected by the window size. Even with *Throughput-400*, around 20% of Ext4 configurations produce higher than 20% variation in terms of throughput. Conversely, the range values for Btrfs and XFS are more sensitive to the choice of window size. For XFS, around 40% of the relative range values for *Throughput-60* are higher than 20%, whereas for *Throughput-400*, nearly all XFS values fall below 20%. This aligns with our early conclusions in Section 6.6.1 that XFS configurations are relatively stable in terms of cumulative throughput, which is indeed calculated based on a window size of 800 seconds; whereas XFS showed the worst relative range for *Throughput-60*, it stabilized quickly with widening window sizes, eventually beating Ext4 and Btrfs.

All the above observations are based on the throughput within a certain window size. Another approach is to characterize the *instant throughput* within an even shorter period of time. Figure 6.9 shows the instantaneous throughput over time for various configurations under the *fileserver-heavy* workload. We collected and calculated the throughput every 10 seconds. Therefore we define instantaneous throughput as the average number of I/O operations completed in the past 10 seconds. This is actually *Throughput-10* in our notation. We normalize this by dividing each value by the maximum instantaneous throughput value for each run, to compare the variation across multiple



(a) Ext4

(b) Btrfs

(c) XFS

Figure 6.8: CDFs for relative range of throughput under Fileserver workload with different window sizes. For window size N, we calculated the relative range values of throughput for all configurations within each file system type, and then plotted the corresponding CDF.

experimental runs. The X axis still shows the running time.

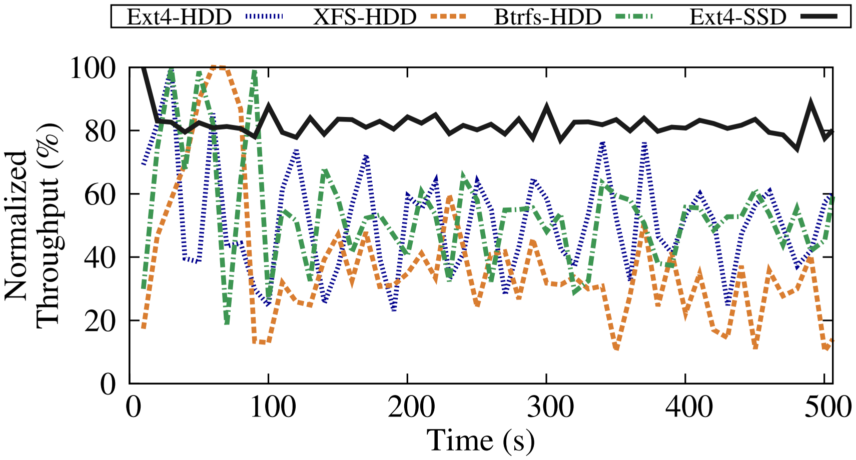


Figure 6.9: Normalized instantaneous throughput (*Throughput-10*) over time for experiments with various workloads, file systems, and devices. The Y axis shows the normalized values divided by the maximum instantaneous throughput through the experiment. Only the first 500s are presented for brevity.

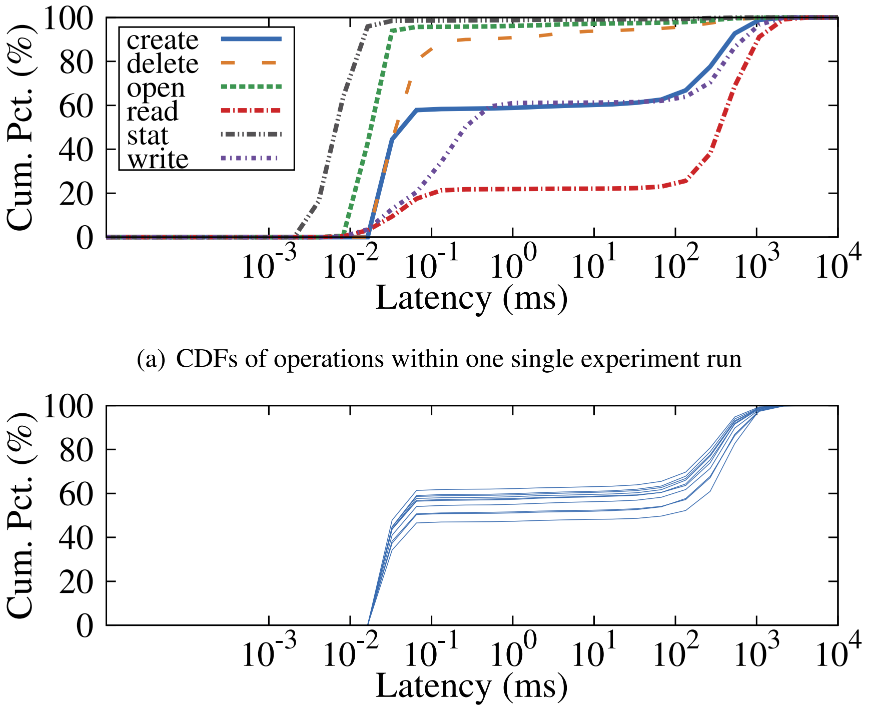
We picked one illustrative experiment run for each configuration (Ext4-HDD, XFS-HDD, Btrfs-HDD, and Ext4-SSD). We can see from Figure 6.9 that for all configurations, instantaneous performance fluctuated a lot throughout the experiment. For all three HDD configurations, the variation is even higher than 80% in the first 100 seconds. The magnitude for variation reduces later in the experiments, but stays around 50%.

The throughput spikes occur nearly every 30 seconds, which could be an indicator that the performance variation in storage systems is affected by some cyclic activity (e.g., kernel flusher thread frequency). For SSD configurations, the same up-and-down pattern exists, although its magnitude is much smaller than for HDD configurations, at only around 10%. This also confirms our findings from Section 6.6.1 that SSDs generally exhibit more stable behavior than HDDs.

6.6.3.2 Latency Variation

Another aspect of performance variation is latency, defined as the time taken for each I/O request to complete. Much work has been done in analyzing and taming long-tail latency in networked systems [88, 115] (where 99.9^{th} percentile latency is orders of magnitude worse than the median), and also in local storage systems [80]. Throughout our experiments, we found out that long-tail latency is not the only form of latency variation; there are other factors that can impact the latency distribution for I/O operations.

A *Cumulative Distribution Function (CDF)* is a common approach to present latency distribution. Figure 6.10(a) shows the latency CDFs for 6 I/O operations of one Ext4-HDD configuration under the *fileserver-heavy* workload. The X axis represents the latency in log_{10} scale, while the Y axis is the cumulative percentage. We can see that for any one experimental run, operations can have quite different latency distribution. The latencies for *read*, *write*, and *create* form two clusters. For example, about 20% of the *read* operation has less than 0.1ms latency while the other



(a) CDFs of operations within one single experiment run

(b) CDFs of *create* operation among repeated experiment runs

Figure 6.10: Latency CDF of one Ext4-HDD configuration under *Fileserver* workload.

80% falls between 100ms and 4s. Conversely, the majority of *stat*, *open*, and *delete* operations have latencies less than 0.1ms. The I/O operation type is not the only factor that impacts the latency distribution. Figure 6.10(b) presents 10 CDFs for *create* from 10 repeated runs of the same experiment. We can see for the 60^{th} percentile, the latency can vary from less than 0.1ms to over 100ms.

Different I/O operations and their latencies impact the overall workload throughput to a different extent. With the empirical data that we collected—per-operation latency distributions and throughput—we were able to discover correlations between the speed of individual operations and the throughput. We first defined a metric to quantify the difference between two latency distributions. We chose to use the Kolmogorov-Smirnov test (K-S test), which is commonly used in statistics to determine if two datasets differ significantly [192]. For two distributions (or discrete dataset), the K-S test uses the maximum vertical deviation between them as the distance. We further define the range for a set of latency distributions as the maximum distance between any two latency CDFs. This approach allows us to use only one number to represent the latency variation, as with throughput. For each operation type, we calculated its range of latency variation for each configuration under all three workloads. We then computed the Pearson Correlation Coefficient (PCC) between the relative range of throughput and the range of latency variation.

Figure 6.11 shows our correlation results. The PCC value for any two datasets is always between [-1,+1], where +1 means total positive correlation, 0 indicates no correlation, and –1 means total negative correlation. Generally, any two datasets with PCC values higher than 0.7 are considered to have a strong positive correlation [162], which we show in Figure 6.11 with a horizontal dashed red line. The Y axis represents the PCC value while the X axis is the label for each opera-

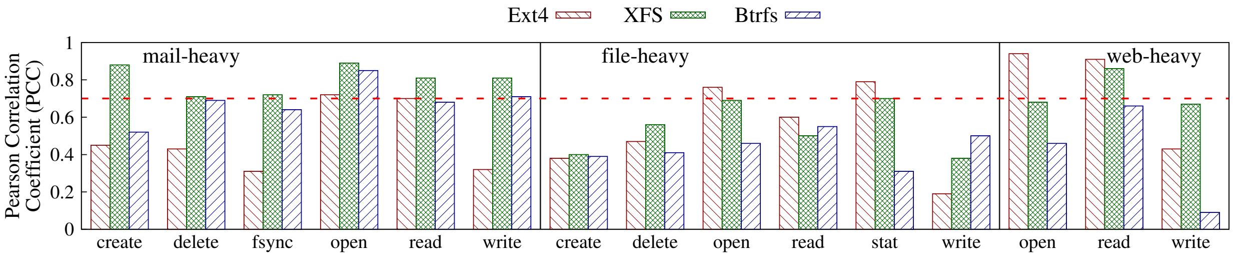


Figure 6.11: Pearson Correlation Coefficient (PCC) between throughput range and operation types, for three workloads and three file systems. The horizontal dashed red line at Y=0.7 marks the point above which a strong correlation is often considered to exist.

tion. We separate workloads with vertical solid lines. As most SSD configurations are quite stable in terms of performance, we only considered HDD configurations here. For Ext4 configurations, *open* and *read* have the highest PCC values on both *mailserver-heavy* and *webserver-heavy* workloads; however, on *fileserver-heavy*, *open* and *stat* have the strongest correlation. These operations could possibly be the main contributors to performance variation on Ext4-HDD configurations under each workload; such operations would represent the first ones one might tackle in the future to help stabilize Ext4's performance on HDD. In comparison, *write* has a PCC value of only around 0.2, which indicates that it may not contribute much to the performance variation. Most operations show PCC values larger than 0.4, which suggest weak correlation. This is possibly because I/O operations are not completely independent with each other in storage systems.

For the same workload, different file systems exhibit different correlations. For example, under the *webserver-heavy* workload, Ext4 show strong correlation on both *read* and *open*; but for XFS, *read* shows a stronger correlation than *open* and *write*. For Btrfs, no operation had a strong correlation with the range of throughput, with only *read* showing a moderate level of correlation.

Although such correlations do not always imply direct causality, we still feel that this correlation analysis sheds light on how each operation type might contribute to the overall performance variation in storage systems.

Chapter 7

Spectra: Finding Important Parameters in Storage Systems

7.1 Introduction

Storage systems are critical components of modern computer systems and have significant impact on application performance and efficiency. Most storage systems have many configurable parameters that control and affect their overall behavior. For example, Linux's Ext4 [60] offers about 60 parameters, representing over 10^{37} potential configuration states. The default settings are often sub-optimal; previous research has shown that tuning storage parameters can improve system performance by a factor of as much as 9× [170].

To cope with the vast number of possible configurations, system administrators usually focus on using their domain expertise to tune a few frequently used and well-studied parameters that are *believed* to significantly impact system performance. However, this manual-tuning approach does not scale well in the face of increasing complexity. Modern storage systems sport different file system types [59,110,161,185], new hardware (SSDs [77,138], SMR [2,3], NVM [99,208]), multi-tier and hybrid storage, and more virtualization layers (e.g., LVM, RAID). Storage systems range from one or a few identical nodes to hundreds of highly heterogeneous environments [67, 167]. Worse, tuning results depend heavily on hardware and the running workloads [28,29,198].

Recently, several black-box optimization methods have been used to auto-tune storage systems, achieving good performance improvements within reasonable time frames [29, 117]. These auto-tuning techniques model the storage system as a black box, iteratively trying different configurations, measuring an objective function's value, and—based on previously learned information—selecting new configurations to try. However, all previous auto-tuning efforts have focused on only a limited set of parameters, often pre-selected by storage experts. Wang *et al.* have shown that many black-box techniques have difficulty scaling to high dimensions [174]. Therefore, the problem of dealing with the vast number of storage-parameter configurations remains unsolved.

In machine learning and information theory, dimensionality reduction is often applied to explosively sized datasets [17, 146]. We believe it can also be applied to storage-parameter selection. Cao *et al.* have demonstrated that certain storage parameters have greater performance impact than others [29]. By eliminating the less important parameters, the parameter search space—and thus the number of configurations that need to be considered by either humans or algorithms—can be massively reduced [84]. Given these observations, we decided to investigate the practicality of parameter selection for storage systems and to design Spectra, a system that uses a variance-based metric to quantify the importance of storage parameters, applying a greedy algorithm that can *automatically* and *efficiently* identify important parameters while evaluating only a small number of configurations.

To evaluate Spectra, we first provide a thorough study of storage parameters' importance. We conducted the study on experimental data collected from 7 file systems under 4 workloads over the past three years. For each file system, we picked 8–10 frequently tuned parameters and their values, and exhaustively evaluated all possible storage configurations resulting from combinations of these values. The exhaustive dataset simplified our study because we can accurately calculate and compare parameter importance, which serves as a "ground truth" when evaluating Spectra's efficacy on a small proportion of the dataset. Our data set consists of more than 500,000 experimental runs (data points) in total. In this paper storage parameter importance was primarily evaluated in terms of I/O throughput, and we ignore other aspects such as reliability. Our approach applies equally well to other quantifiable objectives such as latency, energy consumption, and even composite cost functions [123]. We quantified parameter importance using variance-based metrics inspired by regression trees [20]. We show that in all our datasets there is always a small set of parameters that have significantly more impact on throughput than all the others. For example, under a *Fileserver* workload, the two most important parameters for Ext4 are *Journal Option* and *I/O Scheduler*. However, we found that the set of important parameters varies with different workloads. In the above example, the two important Ext4 parameters become *Block Size* and *Inode Size* when the workload changes to *Dbserver*. We also observed interactions between storage parameters and that choosing good values for all interacting parameters can significantly improve performance.

Based on these observations, Spectra uses a greedy algorithm to select storage-system parameters, which we evaluate using our collected datasets. We then combined the algorithm with Latin Hypercube Sampling (LHS) [109,143], allowing Spectra to identify the set of important parameters using only a small number of experimental runs that explored only a fraction of all configurations. For instance, among all 1,000 repeated runs, Spectra was able to find the two most important parameters for Ext4 using only 32 evaluations. The algorithm's efficiency in finding the most important parameters quickly and accurately is critical and promising, since (1) the technique can be applied to new storage systems or environments, and (2) such parameter findings can then be used by storage experts or auto-tuning algorithms to further optimize the system.

The key contributions of this chapter are:

1. We provide a thorough quantitative analysis of the effects of storage parameters on system performance, for 7 different file systems across 4 representative workloads.

2. We designed Spectra, which uses a variance-based metric of storage-parameter importance to drive an intelligent and efficient algorithm that can select the most important parameters using only a small number of experimental runs.

3. We observed that many storage parameters interact with each other. Spectra can identify and take advantage of these interactions.

7.2 Background

Tuning storage configurations can be modeled as an optimization problem:

$$\vec{x}^* = \text{argmax } f(\vec{x}), \quad \vec{x} = (x_1, \cdots, x_n)$$

Here $x_1, \cdots, x_n$ denote various parameters and $f(x)$ is the optimization objective. For storage systems, common objectives include maximizing I/O throughput or minimizing latency. If desired, the objective can be defined as a complex function of several metrics [123, 181]. Other disciplines use somewhat different terminology (e.g., parameters are analogous to *features* in machine learning, *independent variables* in regression analysis, and *dimensions* in mathematics); optimization objectives can be called *dependent variables* or *target variables*. When discussing prior techniques (§), we use the field-appropriate terms.

7.2.1 Motivation and Challenges

Given the challenges discussed in Section 2.1, manually tuning storage systems has become nearly impossible, and automatic tuning can be computationally infeasible. Recent efforts, including our work in Chapter 5, have used black-box optimization techniques to auto-tune storage configurations [29, 117], addressing several of the above challenges and achieving useful performance improvements. However, we believe that the challenge of tuning storage systems is far from being solved. All previous work has tuned only a small set of parameters, often pre-selected by experts. Several of these black-box optimization techniques have scalability problems in high-dimensional spaces [174]. Therefore, directly applying them to tuning systems with hundreds or thousands of parameters would be ineffective or impractical.

In machine learning and information theory, *dimensionality reduction* is a common technique for coping with explosively sized datasets [17, 146]. We propose that it can also be effective in storage systems. Previous work has demonstrated that not all storage parameters have equally important performance impact: a few have much greater effect than others [29]. Eliminating less-important parameters can massively reduce the search space [84], making it much easier for humans or algorithms to tune storage systems. Therefore, in this paper we propose Spectra, which uses a variance-based metric for storage-parameter importance and an efficient algorithm that can automatically select a subset of parameters that have a significant impact on performance.

7.2.2 Dimensionality Reduction

One critical issue when dealing with high-dimensional data is the *curse of dimensionality*, which refers to the fact that data become sparse in high-dimensional spaces and thus make algorithms designed for low-dimensional spaces less effective. Dimensionality reduction is a powerful way to address this issue; it can be categorized into two main components: *feature extraction* and *feature selection* [76, 116].

Feature extraction refers to projecting high-dimensional data into low-dimensional spaces; the newly constructed features are usually linear or nonlinear combinations of the originals. Common feature-extraction methods include Principal Component Analysis (PCA) [177], Independent Component Analysis [85], and Linear Discriminant Analysis [140]. One major drawback of feature extraction is that the physical meaning of each feature is lost by the projection and the nonlinear combination of many dimensions into fewer ones [116]. Common feature-extraction techniques thus conflict with our goal in this paper, which is to select a few features that can be understood and interpreted.

Conversely, *feature selection* directly selects a subset of features from the original ones, with the intention of finding only those that are important. Feature-selection methods can be classified as *supervised* or *unsupervised* [116]. Unsupervised feature selection, such as PCA [129], chooses a subset that contains most of the essential information based on relationships among features. It does not consider the impact of features on optimization objectives during the selection phase. In contrast, supervised feature selection chooses a subset that can discriminate between or approximate the target variables. Examples include Lasso [193] and decision-tree based algorithms [94]. Since we are interested in finding parameters that have significant impact on our optimization objectives, such as I/O throughput, supervised feature selection best fits our needs.

Several intrinsic properties of our project also limit our choice of feature-selection methods. Many storage parameters are discrete or categorical (see §§ 7.2.1 and. The performance of storage systems is usually presented as I/O throughput or latency, which are continuous. Therefore, an ideal feature-selection method should work with categorical features and continuous targets. Although there are discretization techniques that can break continuous target variables into discrete sections, feature-selection results depend heavily on the quality of discretization [116]. One common approach for dealing with categorical features is to transform each of them into dummy binary parameters that take values of 0 or 1. For instance, *io_scheduler* with three possible values (*noop*, *deadline*, and *cfq*) can be converted into three binary features: *"io_scheduler = noop" is 0/1*, *"io_scheduler = deadline" is 0/1*, and *"io_scheduler = cfq" is 0/1*. This approach is unsatisfactory because it selects the individual binary features instead of the original categorical ones. Moreover, converting each categorical parameter with N values into N separate binary parameters would *expand* the parameter space by 2^N. For this reason, we feel that Lasso [193] is not suitable for our problem, even though it has been successfully applied to selecting important knobs for databases [198]. Although Group Lasso has been proposed to partially address this deficiency [38, 100, 214], the computational cost of the Lasso-based methods is still high [116].

Another popular category of feature-selection methods has been built upon information theory [23, 58, 94, 116]. These approaches usually define a metric for the *homogeneity* of the target variable within certain subsets. Commonly used metrics include Gini impurity [116] and Entropy [17] for discrete target variables, and Variance [20] for continuous variables. Spectra applies a variance-based metric for parameter importance, as detailed in §.

7.3 Spectra: Algorithmic Parameter Selection

Spectra uses a variance-based metric (Section 7.3.1) and Latin Hypercube Sampling (Section 7.3.2) to construct an efficient algorithm for finding important parameters (Section 7.3.3).

7.3.1 Measuring Parameter Importance

Spectra proposes a variance-based metric for storage-parameter importance. The variance of a set S of storage configurations is defined as usual:

$$\mathrm{Var}(S) = \frac{1}{|S|}\sum_{i=1}^{|S|}(y_i - \mu)^2, \tag{7.1}$$

where y_i is the throughput of the i-th configuration; $|S|$ is number of configurations in S; and μ is the average throughput within S. Inspired by CART (Classification and Regression Tree) [20], we use the *reduction in variance* to measure parameter importance. We extend CART's original definition to support categorical parameters taking an arbitrary but finite number of values, as compared with only two in CART.

We define the parameter importance PI of a parameter P that can take a finite number of categorical values, $\{p_1, ..., p_n\}, n > 1$, as:

$$PI(P) = \mathrm{Var}(S) - \sum_{i=1}^{n}\frac{|S_{P=p_i}|}{|S|}\mathrm{Var}(S_{P=p_i}) \tag{7.2}$$

Here S is the original set of configurations, and $S_{P=p_i}$ is the subset of configurations with the parameter P taking the value p_i. Intuitively, an important parameter P divides a set S of configurations into multiple subsets, and the weighted sum of variances within each subset should be much smaller than the variance of S. Thus, a high PI indicates a parameter that has a significant effect on performance. As discussed in Section 7.1, we repeatedly choose the parameter with the highest PI until a given stopping criterion is met.

As described in Section, storage parameters sometimes interact with each other. Therefore, we define the *conditional parameter importance* for a parameter Q, given $P = p$ as:

$$CPI(Q|P = p) = \mathrm{Var}(S_{P=p}) - \sum_{j=1}^{m}\frac{|S_{Q=q_j,P=p}|}{|S_{P=p}|}\mathrm{Var}(S_{Q=q_j|P=p}) \tag{7.3}$$

where $S_{Q=q_j,P=p}$ denotes the set of configurations with parameters P and Q taking values p and q_j, respectively. Similar to Equation 7.2, given $P = p$, the next most important parameter Q divides $S_{P=p}$ into multiple subsets, and if Q is important then the weighted sum of variance within each subset will be much smaller than variance of $S_{P=p}$. To remove the restriction to a given value p, we define $CPI(Q|P)$ as the maximum of $CPI(Q|P = p_i)$ over all possible values $p_i \in \{p_1, ..., p_n\}$ that parameter P can take:

$$CPI(Q|P) = \max_{i=1}^{n} CPI(Q|p = p_i) \tag{7.4}$$

Note that in this paper we use only variance-based metrics to measure parameter importance and select the most critical subset. We leave storage-performance prediction, which requires a large amount of training data [201], for future work.

7.3.2 Sampling

Given the large parameter space and the time needed to evaluate a single storage configuration, we must limit the number of experimental runs required to select important parameters. Therefore, Spectra needs an exploratory method that can cover the space uniformly and comprehensively, yet sparsely. In this work, we chose Latin Hypercube Sampling (LHS) [135].

LHS is a stratified sampling method [31]. In two dimensions, a square grid containing samples is a *Latin Square* iff there is only one sample in each row and each column. A *Latin Hypercube* is the generalization of a Latin Square to higher dimensions, where each sample is the only one in each axis-aligned hyperplane containing it [109]. LHS has been shown to be more effective in exploring parameter spaces than random sampling [135] and Monte Carlo sampling [39]. It has been successfully applied in sampling configurations of storage [80] and cloud systems [124].

Previous work has also applied Plackett-Burman (P&B) Design [155] to evaluate the impact of parameters in storage benchmarks [152] and databases [48]. However, P&B design requires each parameter to have only two possible values, and the target variable must be a monotonic function of the input parameters. Neither requirement holds in our problem.

7.3.3 Parameter-Selection Algorithm

Based on the proposed measurements of parameter importance and Latin Hypercube Sampling (LHS), the pseudo-code for Spectra's parameter selection algorithm is as follows:

Algorithm 1 Parameter-Selection Algorithm

```
Require: $P$: set of parameters, $S$: initial set of configurations; stop(S, selected): user-defined
  stopping function.
  $selected \leftarrow \{\}$
  $S^* \leftarrow LHS(S)$
  repeat
    $p^* \leftarrow \text{argmax}\, CPI(p|selected), p \in P$
    $selected.insert(p^*)$
    $P.remove(p^*)$
  until $stop(S, selected)$ is true
Ensure: $selected$
```

Our algorithm takes a set of initial parameters P and configurations S. It greedily selects the current most important parameter, based on its *conditional parameter importance* given the set of previously selected parameters, and continues to select parameters until the *stop function* succeeds. A naïve stop function could be $sizeof(selected) \geq N$, which would select the N most important parameters. An alternative variance-based stopping function might stop when the variances of subsets of configurations (given the current *selected* parameters) are below a certain threshold ϑ. This stopping condition indicates that by setting the values of the *selected* parameters, the system throughput already falls into a small enough range that there is little potential gain from additional tuning. In our experiments, we apply this idea and use Relative Standard Deviation (RSD) [31], or Coefficient of Variation, to define our stopping condition. The RSD of a set S of configurations is defined as:

$$\mathrm{RSD}(S) = \frac{1}{\mu}\sqrt{\frac{\mathrm{Var}(S)}{N-1}} \tag{7.5}$$

where N is the number of configurations and μ is the mean throughput of configurations within S. We chose RSD because it is normalized by the mean throughput and is represented as a percentage; that way the same threshold can be used across different datasets. We used a threshold of 2% in our experiments; as seen in §, parameters selected by this criterion give us near-optimal and stable throughput.

7.4 Experiment Settings

Table 7.1 lists all our file systems, their parameters, and the number of possible values that each parameter can take. Note that under *S2* we conducted default Filebench workloads (**-def*) on four storage devices, and we treat the device as one of the parameters. Under *S3* we focused on Ext4 and XFS experiments with an SSD, but evaluated a wider variety of parameters. Experiments were run with modified Filebench workloads of 10GB working dataset size (*-heavy). Machine and workload details are described in Table 4.1 and Table 4.2 in Chapter 4. Cells with "–" mean the parameters are invalid for certain file systems. Cells with "def" mean we used the default value for that parameter, and they were not considered during the parameter-selection phase.

7.5 Evaluation

We evaluated Spectra experimentally by benchmarking real hardware in a realistic parameter space (§ 7.4).

7.5.1 Parameter Importance: an Overview

We exhaustively collected experimental data from 9 different parameter spaces (Table 7.1) under 4 representative workload types. Having the exhaustive datasets allowed us to accurately calculate and evaluate the importance of different storage parameters. This can serve as the ground truth when evaluating our Spectra, whose goal is to explore only a small fraction of the parameter space yet find the same subset of important parameters as if we had explored it all. In this section, we first provide an overview of the importance of storage parameters.

Figure 7.1 shows the top 3 most important parameters for Ext4 under *S2, fileserver-def.* A parameter with highest importance was evaluated and selected by its Parameter Importance (PI), as defined in § 7.3.1. The second most important parameter was measured by its Conditional Parameter Importance (CPI) given the most important one, in this case $\mathit{CPI}(X|journal)$. Similarly, the 3rd most important parameter was evaluated by comparing its $\mathit{CPI}(X|journal, device)$. Note that the Y axis scales in the three sub-figures are different (but higher is always better). The X axis shows the Ext4 parameters that we experimented with. As shown in the top subfigure in Figure 7.1 *Journal Option* turns out to be the most important parameter for Ext4 under *S2, fileserver-def.* It has the highest variance reduction, 2.7×10^7. In comparison, the PI of *Device* is around 10^6, while

Setting - File System	S2 - Ext2	S2 - Ext3	S2 - Ext4	S2 - XFS	S2 - Btrfs	S2 - Nilfs2	S2 - Reiserfs	S3 - Ext4	S3 - XFS
Workloads	*-def	*-def	*-def	*-def	*-def	*-def	*-def	*-heavy	*-heavy
Block Size	3	3	3	3	-	3	def	3	3
Inode Size	7	7	7	5	5	-	-	3	2
Block Group	6	6	6	-	-	9	-	def	-
Journal Option	-	3	3	-	-	2	3	3	-
Flex Group	-	-	def	-	-	-	-	3	-
Inode Readahead	-	-	def	-	-	-	-	3	-
Sector Size	-	-	-	def	-	-	-	-	3
Allocation Count	-	-	-	9	-	-	-	-	4
Log Buffer Count	-	-	-	def	-	-	-	-	2
Log Buffer Size	-	-	-	def	-	-	-	-	2
Allocation Size	-	-	-	def	-	-	-	-	2
Node Size	-	-	-	-	3	-	-	-	-
Special Option	-	-	-	-	4	-	-	-	-
Atime TOption	2	2	2	2	2	2	2	def	def
I/O Scheduler	3	3	3	3	3	3	3	3	3
Dirty Bg Ratio	def	def	def	def	def	def	def	2	2
Dirty Ratio	def	def	def	def	def	def	def	3	3
Device	4	4	4	4	4	4	4	SSD	SSD
Total Config.	2,208	6,624	6,624	2,592	288	1,944	192	3,888	5,184

Table 7.1: Details of parameter spaces. Each cell gives the number of settings we tested for the given parameter and file system; empty cells represent parameters that are inapplicable to the given file system. We exhaustively evaluated 29,544 configurations in total under four workloads, and each experiment was repeated $3+$ times.

all other parameters are under 5×10^4. Similarly, the second and third importance parameters are *Device* and *Block Size*, respectively, both with a much higher *CPI* value than other parameters.

We discovered that parameter importance depends heavily on file system types and on the running workload. Table 7.2 lists the top 4 important parameters for Ext4, XFS, and Btrfs under various workload types; the column header *#N* identifies the N$^{\text{th}}$ most important parameter. We also applied the stopping criterion described in § 7.3.3. Cells marked as "–" here indicate that no parameter gave a large reduction in variance; and thus no parameter was considered important. Due to space limits, we only list 3 file systems under 4 workloads here, and we show only the top 4 ranked parameters under each case.

As we can see in Table 7.2, the important parameters are quite diverse and depend significantly on the file system types and workloads. For Ext4 under *S3* and *dbserver-heavy*, the top 4 ranked parameters are *Block Size*, *Inode Size*, *I/O Scheduler*, and *Journal Option*. When the workload changes to *webserver-heavy*, the top 4 parameters become *Inode Size*, *Flex BG*, *Block Size*, and *Journal Option*. For *fileserver-heavy* under Ext4, we only found three important important parameters. This indicates that fixing the values of these three parameters already resulted in quite stable throughputs. We explain this observation in more detail in § 7.5.2. We found similar results on XFS: the values and number of importance parameters depend heavily on workloads. Interest-

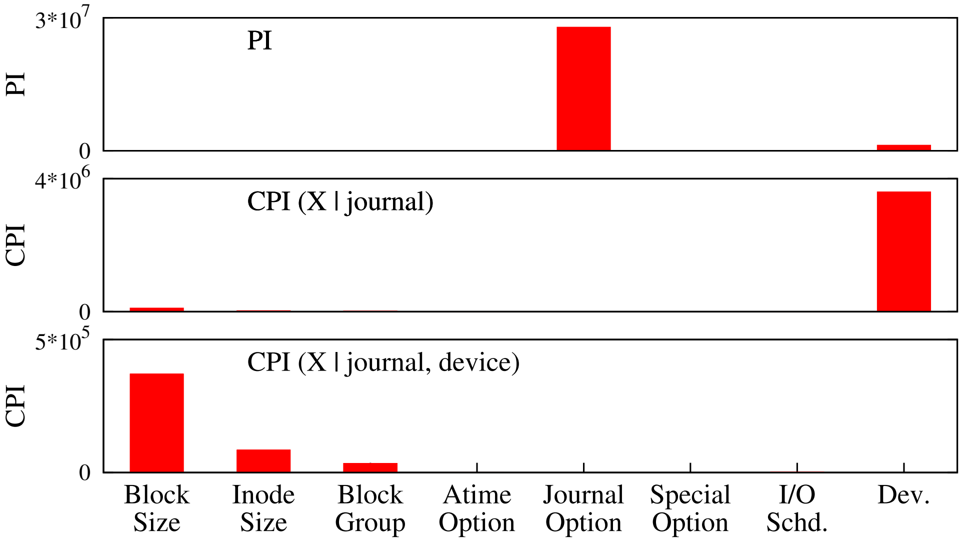


Figure 7.1: Top 3 most important Ext4 parameters under S2, fileserver-def. The most important parameter is measured by its PI; the second and third parameters are evaluated by their CPI given higher-ranked parameters. The Y-axis scales in the three subfigures are different.

ingly, for Btrfs under *S2, webserver-def*, we did not find any important parameters. This is because *webserver-def* is a workload consisting of mostly read operations, and the default working-set size used by Filebench is small. All Btrfs configurations actually produce quite similar throughput values under *webserver-def*. For this reason, we also collected datasets from workloads with a much larger working-set size (10GB), as mentioned in § 7.4.

7.5.2 Parameter Interactions

In our experiments we observed that the importance of some storage parameters can interact with the particular values of other parameters. For example, in Figure 7.1, the variance reduction of the parameter *Device* is 1.2×10^6, while the variance reduction of *Device* with *Journal Option* fixed is 3.6×10^6. We observed similar behavior in nearly all of our datasets: the importance of certain parameters might only appear or be amplified when one or more other parameters have certain values.

We further demonstrate this observation in Figure 7.2. Each point in the figure represents the set of Ext4 configurations under *S2, fileserver-def* by fixing the values of N parameters. For $N = 1$, we have 28 points, which equals the sum of possible value counts for each parameter, as shown in Table 4.3. There are 374 points for $N = 2$. We use different point colors and sizes for different number of parameters. We only plot up to $N = 2$ here; we extend to $N = 4$ in Figure 7.3. Larger points are used for smaller N values, since fixing fewer parameter values would result in a larger number of configurations. For example, fixing *journal_option = ordered* in our datasets leads to a set of 2,208 configurations; fixing *journal_option = ordered, device=ssd* reduces that number to 552.

In Figure 7.2, performance is measured by the average throughput of a configuration within each set of configurations, as presented in the X axis. The Y axis shows the stability, measured by

Setting	WL	F/S	#1	#2	#3	#4
S2	File -10GB	Ext4	Journal Option	I/O Scheduler	Inode Size	–
S2	Db -10GB	Ext4	Block Size	Inode Size	I/O Scheduler	Journal Option
S2	Mail -10GB	Ext4	I/O Scheduler	Inode Size	Journal Option	Block Size
S2	Web -10GB	Ext4	Inode Size	Flex BG	Block Size	Journal Option
S2	File -10GB	XFS	I/O Scheduler	Inode Size	Alloc Grp Cnt	–
S2	Db -10GB	XFS	Block Size	Log Buf Size	Dirty Ratio	Alloc Grp Cnt
S2	Mail -10GB	XFS	Inode Size	I/O Scheduler	Log Buf Size	Alloc Size
S1	File -def	Btrfs	Special Option	Inode Size	Device	–
S1	Mail -def	Btrfs	Inode Size	Device	–	–
S1	Web -def	Btrfs	–	–	–	–

Table 7.2: Top-ranked important parameters for various file systems. The column header #N identifies the N$^{\text{th}}$ most important parameter.

the Relative Standard Deviation (RSD) of throughput within each set. We chose to use the RSD rather than variance because the figure shows sets of varying numbers of configurations; RSD is normalized by the configuration count as well as the average throughput, and thus is easier to compare. Our goal is to maximize throughput *while* minimizing RSD; therefore the best points should cluster in the bottom-right quadrant of Figure 7.2, and the addition of parameters should move us from the upper-left to the bottom-right quadrants (diagonally).

As we can see from the figure, fixing just one parameter value (purple dots) causes the mean throughput to range from 2.5K to around 15K, and the RSD ranges from around 7% to 76%. The upper-left purple point (2,500, 76%) represents the configurations gotten by setting *Journal Option* to *journal*. The other two points representing a *Journal Option* of *ordered* or *writeback* turn out to be the best among all purple points. They are both seen near the bottom right with mean throughput of around 15K and an RSD value of 7%. Clearly, the *Journal Option* parameter has the highest impact on performance; setting it to an improper value could lead to low throughput and high RSD, while setting it correctly provides significant benefits. The points with $N = 2$ form several clusters. All points with mean throughput less than 9K result from setting *Journal Option* to *journal* (and with another parameter set to various valid values). Conversely, all points with mean throughput larger than 14K result from a *Journal Option* of *ordered* or *writeback*. *Journal Option* is also the most important parameter selected by our definition of Parameter Importance, PI, supporting the validity of that definition. Another interesting observation from Figure 7.2 is that the best point, either the highest mean throughput or the lowest RSD that we can get by fixing N parameter values, tends to improve with the increase in N. This phenomenon arises because

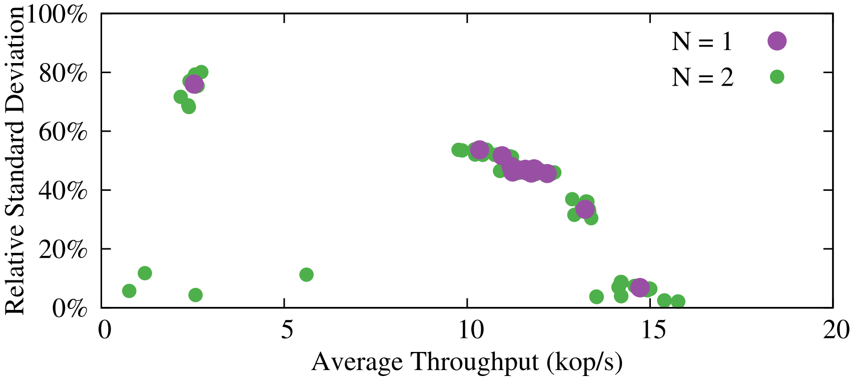


Figure 7.2: Impact of parameters on performance and stability (Ext4, S2, fileserver-def). Each dot represents a set of configurations created by fixing N parameters, while different dot sizes and colors are used for different values of N. Performance is measured by the average throughput (X axis) of all possible configurations within each set; stability is measured by relative standard deviation (Y axis, lower is better) of the throughput within each set.

reducing the configuration space reduces variance, which in turn reduces the RSD and thus allows the mean throughput to rise.

To probe this further, we zoomed into the bottom-right part of Figure 7.2 and added points for $N = 3, 4$, as shown in Figure 7.3. The X and Y axes are similar but with narrower ranges (and the X axis starts at 14K). The label “Max” on the X axis, with a small tick mark, shows the global maximum throughput of all Ext4 configurations within the parameter space. For each N, we only plot the point(s) with the highest average throughput or lowest RSD. The labels around each point show the associated parameter values, ordered by (*Journal Option*, *Device*, *Block Group*, and *Inode Size*). The black triangle marks the point with highest mean throughput, gotten by fixing the values of the three most important parameters selected by Spectra. For $N = 1$, the best two points resulted from setting *Journal Option* to either *ordered* or *writeback*. These two points overlap with each other in this figure, as they share nearly identical mean throughput and RSD values. Only one point is plotted for $N = 2$, since the point *(journal_option=ordered, device=ssd)* shows both the highest throughput and the lowest RSD among all $N = 2$ points; the same is true for $N = 3$. For $N = 4$, the left red point shows the lowest RSD value while the right red point shows the highest average throughput. We can see from Figure 7.3 that the impact of parameters on performance depends on the parameters selected and their order. For example, with two parameters, the best average throughput is 15.7K, which results from *journal_option=ordered, device = ssd*. The average throughput achieved by setting only *journal_option* is 14.7K, as shown in the figure. The throughput value for *device=ssd* is even worse. We have similar conclusions for $N = 3$ and $N = 4$. As explained above, the best average throughput increases when we select more parameters.

Note that using our definition of parameter importance, the top three selected parameters are *Journal Option*, *Device*, and *Block Size*. By setting the values of these three parameters, the best average throughput (triangle in Figure 7.3) only slightly lower than the global best average throughput achieved by fixing 3 parameter values (blue point). This is because our definition of parameter importance focuses on measuring the “impact” of parameter on performance, which can be either

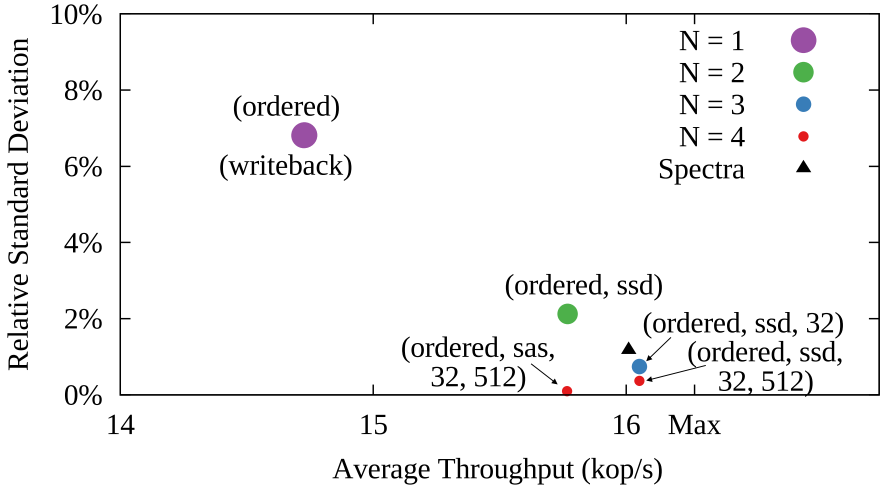


Figure 7.3: A zoom into the bottom-right part of Figure 7.2 (the "best quadrant"), with points for $N = 3, 4$ added. Plotted points show either the highest average throughput or the lowest relative standard deviation among all configurations gotten by fixing the values of N parameters. The labels around the dots show the corresponding fixed parameter values. The parameter values are ordered by (Journal Option, Device, Block Group, and Inode Size). The triangle marks the point achieved by fixing the values of parameters selected by Spectra.

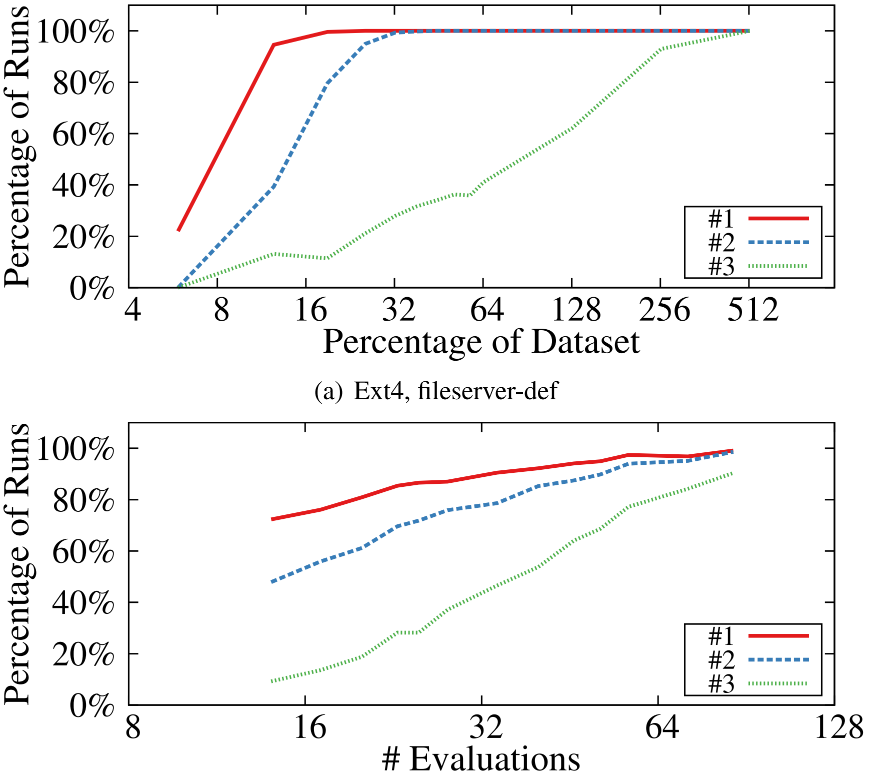
positive or negative. Still, the selected parameters come very close the global best. Moreover, our algorithm stops after selecting 3 parameters, as the RSD already drops below our 2% threshold at that point.

7.5.3 Spectra: Evaluation

All evaluations and analysis in § 7.5.1 and 7.5.2 were conducted on the complete dataset of all possible parameter configurations. However, collecting exhaustive datasets for storage parameters is usually impractical, given the challenges discussed in § 7.2.1. One design goal of Spectra is to select important parameters while evaluating only a small fraction of configurations. Spectra does so by utilizing Latin Hypercube Sampling (LHS), which has been effective in exploring system parameter spaces [80, 124].

Figure 7.4 presents the results of running Spectra on two different datasets, *Ext4, fileserver-def* and *Btrfs, fileserver-def*. The X axis shows the number of configurations that were evaluated and used by Spectra, and is in $\log_2$ scale. For each X, we repeated the same experiment for 1,000 runs, to measure Spectra's ability to select parameters. We used the important parameters selected using the whole dataset as the "ground truth." For *Ext4, fileserver-def*, the top 3 important parameters are *Journal Option*, *Device*, and *Block Size*. For *Btrfs, File-def*, they are *Special Option*, *Node Size*, and *Device*. The Y axis shows the fraction of runs that successfully found the same important parameters as the ground truth. The solid, dashed, and dotted lines show the results of finding the 1st, 2nd, and 3rd most important parameters, respectively.

Figure 7.4(a) shows that even with only 8 configurations evaluated (0.1% of dataset), Spectra has a 60% probability of correctly identifying the most important parameter. When using 32 (0.4%), Spectra was able to find the 1st and 2nd ranked parameter in 100% and 99.8% of the 1,000



(a) Ext4, fileserver-def

(b) Btrfs, fileserver-def

Figure 7.4: Spectra's ability to correctly find the top 3 important parameters within small portions of the dataset. The X axis (log_2 scale) shows the number of evaluations that was used. We ran Spectra on X sampled configurations for 1,000 runs. We used the PI calculated from the whole dataset as ground truth. The Y axis shows the percentage of runs that were able to correctly find the important parameters. The solid, dashed, and dotted lines show the results for finding the parameters ranked 1st, 2nd, and 3rd, respectively.

runs, respectively. Setting the values of the most important two parameters would already produce high average throughput (97% of the global optimum) with high stability (2% of RSD), as shown in Figure 7.3. The chance of correctly selecting the third important parameter increases with the percentage of the dataset used by Spectra. With 64 configurations (1% of the dataset), the probability of correctly finding the 3rd parameter is around 40%, while sampling 256 configurations (2.9%) successfully identifies the 3rd parameter with higher than 90% probability.

For Btrfs, shown in Figure 7.4(b), Spectra needed a larger fraction of the dataset to make correct selections. This is because Btrfs has only 288 configurations, compared with 8,832 for Ext4. Yet by evaluating only 45 of all configurations, Spectra found the 1st and 2nd parameters with greater than 70% and 80% probability respectively. Spectra identified the 3rd parameter in more than 80% of runs with 64 configurations sampled.

In sum, Spectra is effective in selecting parameters using only a few evaluations. In our experiments, Spectra found the top 2 important parameters with higher than 80% probability by evaluating fewer than 50 configurations. Fixing the values of the most important two parameters can already result in high and stable system throughput, as shown in § 7.5.2. Spectra can find the 3rd parameter with about 40% probability using only about 64 evaluations. Moreover, auto-tuning a storage system with an optimization algorithms often requires an initialization phase to explore the whole space [29, 124]. Spectra can utilize the data collected during the initialization phase to select parameters; in this case, no *extra* evaluation needs to be conducted. Integrating Spectra with auto-tuning algorithms is part of our future work.

7.6 Related Work

Parameter selection for computer systems There have been several attempts to select important parameters for various types of software systems. Aken *et al.* [198] applied Lasso to choose important knobs for databases. They converted categorical parameters into binary dummy features and included polynomial features to deal with parameter interactions. As discussed in § 7.2.2, Lasso does not scale well when the system has many categorical parameters. Plackett-Burman (P&B) design of experiments [155] has been applied to evaluate the impact of parameters in storage benchmarks [152] and databases [48]. However, P&B assumes that each parameter has only two possible values and the target variable is a monotonic function of the input parameters; neither holds for storage parameter spaces. Adaptive Sampling [54] and Probabilistic Reasoning [182] have been applied to evaluating the impact of database knobs. They either only work for continuous parameters, or have scalability issues in high-dimensional spaces. In comparison, Spectra applies variance-based metrics for storage parameter importance. To the best of our knowledge, we have conducted the first thorough quantitative study of storage-parameter importance by evaluating Spectra on datasets collected from various file systems and workloads. Spectra also provides insights into interaction between parameters.

Auto-tuning storage systems Several researchers have build systems made to automate storage-system tuning. Strunk *et al.* [181] applied Genetic Algorithms (GA) to automate storage system provisioning. Babak *et al.* [12] used GA to optimize the I/O performance of HDF5 applications. GA have also been applied for storage-recovery problems [96]. Deep Q-Networks have been successfully applied in optimizing performance for Lustre [117]. More recently, Cao *et al.* [29]

provided a comparative study of applying multiple optimization algorithms to auto-tune storage systems. However, many auto-tuning algorithms have scalability issues in high-dimensional spaces [174], which is one of our motivations. Selecting the subset of important parameters could reduce the space dramatically, which would then benefit either auto-tuning algorithms or manual tuning by experts.

Feature selection in general Many feature-selection techniques have been proposed in various disciplines. Li *et al.* [116] provide a thorough summary and comparison for most state-of-the-art feature-selection algorithms. Based on our arguments in § 7.2.2, we chose to use variance-based metrics for storage-parameter selection.

7.7 Conclusions

Modern storage systems come with many parameters that affect their behavior. Tuning parameter settings can bring significant performance gains, but both manual tuning by experts and automated tuning have difficulty dealing with the large number of parameters and configurations. In this paper, we propose Spectra, which addresses this problem with the following four contributions: (1) Spectra includes a variance-based metric for quantifying storage parameter importance, and a greedy yet efficient parameter-selection algorithm. (2) To the best of our knowledge, we provide the first thorough study of storage-parameter importance. We evaluated Spectra across multiple datasets (collected from more than 300,000 experimental runs) and showed that there is always a small subset of parameters that have the most impact on performance—but that the set of important parameters changes with different workloads, and that there are interactions between parameters. (3) We demonstrated Spectra's efficiency by testing it on small fractions of the configuration space. This efficiency gives Spectra the potential to be easily applied to new systems and environments and to identify important parameters with a small number of configuration evaluations.

In the future, we plan to extend Spectra to support other parameter-selection techniques, such as Group Lasso [38,100,214] and ANOVA [24,31,113,204]. We will evaluate and improve Spectra with more optimization objectives (e.g., reliability), and larger storage-parameter spaces. We also plan to integrate Spectra with auto-tuning algorithms.

Chapter 8

Graphs are not Enough: Using Interactive Visual Analytics in System Research

8.1 Introduction

Analyzing and understanding the behavior of computer systems has always been of interest to researchers and system administrators. Previous work has presented analysis of various aspects: performance [28,47,77], reliability [95], energy consumption [170], etc. In recent years, computer systems have grown more complex with the addition of new hardware, varied workloads, and increasing scale. This makes system analysis more important but also more challenging.

Existing analytic approaches used in system research can be broadly classified into two categories: *non-visual* and *visual* techniques. Non-visual methods include statistical measurements such as mean, standard deviation, and percentile(s) [47], plus machine-learning techniques including classification, clustering analysis, etc. [10,202] Visual approaches have included 2D techniques such as histograms [90], box plots [28], etc., and 3D versions such as surface plots [35, 188].

However, existing techniques are not enough for thorough understanding of system behavior, for three reasons. First, *computer systems are often impacted by many factors*. Modern computer systems can easily have hundreds of tunable parameters [29]. However, most commonly applied visualization techniques (e.g., line, histogram, scatter plots) can focus on one or few factors within one plot. To analyze the impact of all parameters, multiple graphs are needed. For example, during our previous study of just nine parameters in a typical storage system [29], we produced over 2,000 plots in an attempt to fully analyze the parameters' impact and dependencies. The problem is exacerbated because the running workload and underlying hardware can also affect system behavior [28, 29, 170]. Moreover, some system parameters have categorical values, while many plotting approaches (line, scatter, etc.) assume numerical axes. The standard regression technique of splitting categorical parameters into dummy binary values does not scale well, because it makes the configuration space grow exponentially [198]; thereafter, attempting linear regression between Boolean 0s and 1s—treating them as floating-point numbers—is often meaningless for categorical parameters.

Second, *some traditional approaches lack interpretability*. Systems researchers often want not only to explain the numbers, but also to understand the underlying implications at the system level. Many existing approaches project high-dimensional data into low-dimensional spaces; the newly constructed dimensions are usually linear or nonlinear combinations of the originals. Examples include Principal Component Analysis (PCA) [177], Independent Component Analysis [85], and visual techniques such as Multi-Dimensional Scaling (MDS) [107]. One major drawback of this kind of projection is that the physical meaning of each dimension is lost by the projection and by the nonlinear combination of many dimensions into fewer [116].

Third, *it is difficult to infuse domain knowledge*. It is important and beneficial to combine expert knowledge into system analysis procedures. For example, in our previous study we used our storage expertise to pick nine representative storage parameters and four common workloads [29]. Similarly, Basak *et al.* [10] pre-selected features manually when doing workload characterization. Due to the complexity of computer systems, there is no single master solution that can satisfy all requirements; often a combination of statistics, visualization, and human reasoning must be applied. However, current systems papers mostly use static, non-interactive 2D (occasionally 3D) plots, which make it inconvenient to exploit domain knowledge while analyzing.

To address the aforementioned limitations, we propose to apply another type of analytic technique in systems research: **interactive visual analytics**, which have been successfully applied to many real-world data sets [37, 87, 104]. Interactive visual analytics can often present high-dimensional spaces in a single 2D space, allowing researchers to explore interactions among multiple factors of the targeted system. They let users exploit their domain knowledge and intuition via visual interaction; this empowers users to take an active role in the analysis process, better understand the target system, and make sound decisions with high confidence.

To demonstrate the benefits of applying interactive visual analytics, we took storage-system performance analysis as an example. We conducted studies on our three-year dataset collected on a typical storage system; the dataset has 9 dimensions and 100k configurations (about 500k data points in total); many large installations often collect numerous similar telemetrics [138, 148, 168]. We prototyped a new tool, the *Interactive Configuration Explorer* (ICE), which uses an enhanced box plot whose form is well understood by systems researchers, with an embedded density plot for throughput distribution, to present the data to the user in a compact, easily interpretable form. We have found that ICE can help researchers explore the interaction among multiple system parameters (numeric, discrete, and categorical), and understand system performance, efficiency, stability, reliability, etc. We hope our study will lead to more use of interactive visual analytic approaches in systems research.

8.2 ICE: Interactive Configuration Explorer

A motivating example Maria is an analyst responsible for a large storage system. She has been working on a performance problem for weeks, without success. Fortunately she has collected metrics on her production systems, and also has a testbed that she can use for benchmarks, so she has lots of data about different configurations and workloads. But she needs to make sense of all those numbers, which is what our interactive visual analytic tool, the *Interactive Configuration Explorer* (ICE), is designed to do. Launching it, Maria first sees Figure 8.1. The performance that matters to her is throughput (Y axis—higher is better). Her system is currently used as a file server, so she decides to focus on that workload. When she clicks on it, the screen reconfigures to show just the file-server data (Figure 8.2, zoomed to show only the first two sections). The mean performance of each filesystem is shown by the black dots, and the range by the length of the bar.

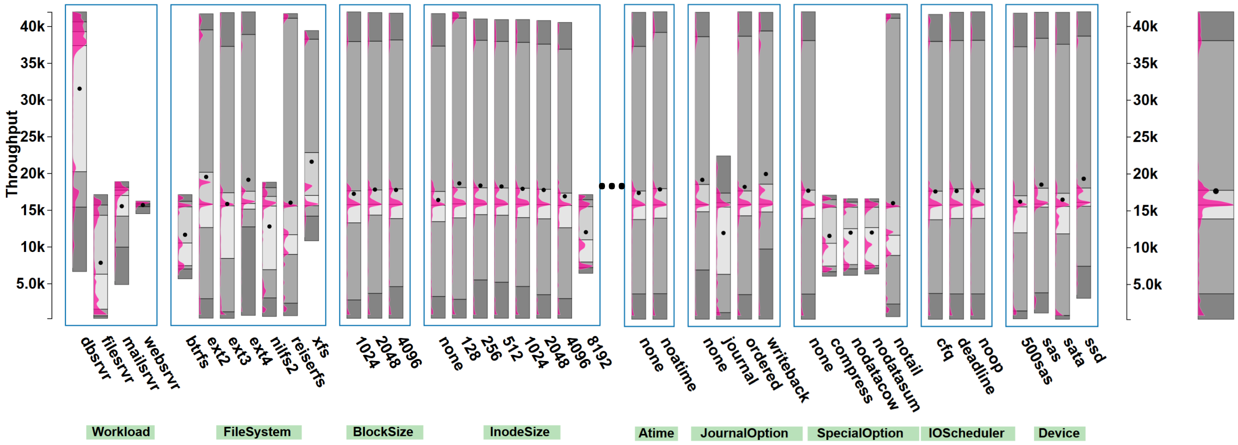


Figure 8.1: Screenshot of ICE. Block Group was cropped out, shown as "..." in the figure, to ensure the screen text is legible.

Maria sees that both *btrfs* and *xfs* have high throughput, but *xfs* has less variance. Nevertheless, she decides to look further into *btrfs* because of its snapshoting capabilities. Choosing that option produces Figure 8.3, where she chooses an 8KB inode size for its low variance, and sees that selecting *compress* for the "SpecialOp" will reduce variance further.

Maria knows that the system might later be used as an OLTP database server. Will *btrfs* still behave well? She backs out, selects *dbserver*, and sees Figure 8.4. It turns out that Btrfs is terrible for database workloads. But Ext4 seem to contain some promising configurations with high throughput (indicated by the peaks in the magenta regions inside the bar). She can then use ICE to select *ext4* and again explore different parameter selections for the new workload, just like she did for *fileserver*.

Both researchers and administrators commonly encounter this scenario: analyzing systems that are impacted by tunable parameters and other factors including workload, hardware, software, etc. As in the example, interactive visual analytics allow quick exploration of many configuration options. We now describe the design of ICE, and in Section 8.3 we will show examples of how we used it to understand system behavior.

Design of ICE ICE was designed to help users visualize, understand, and explore the impact of system parameters and workloads on system behavior. We tested it on experimental data collected on 7 different file system types and 4 representative workloads using Filebench [62, 190]. We also experimented with the parameters *block size*, *inode size*, *blocks per group*, *mount option*, *journal option*, *special option*, *I/O scheduler*, and 4 different storage devices. The total number of unique storage configurations is 24,888, and we collected more than 500,000 data points. (Real-world users are unlikely to run so many tests; ICE is designed to help analyze collected experimental data, which does not have to be exhaustive or enormous. The large dataset here made it possible for us to explore a design that could handle extreme situations.) ICE shows various configurations as bars, each of which is carefully designed to present rich information about the throughput distribution resulting from selected parameter values. Bars in ICE are a combination of stacked bar plots and *violin* plots [83], which are box plots superimposed with rotated kernel density plots. Figure 8.5

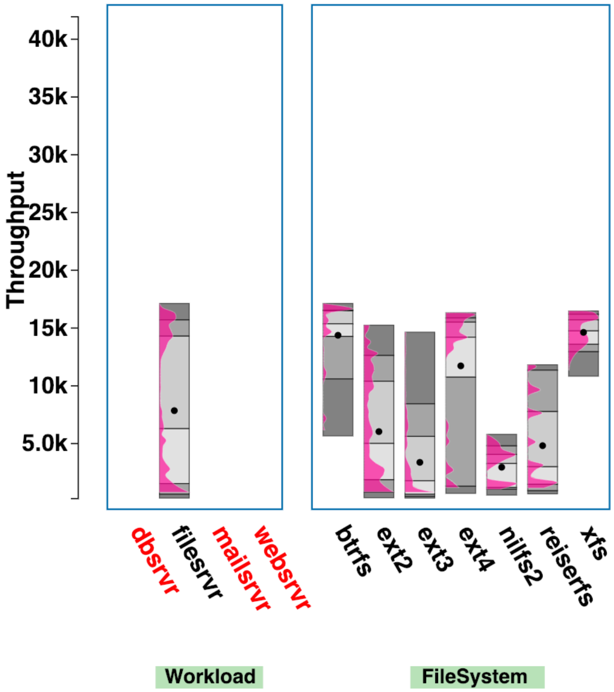


Figure 8.2: Partial screenshot of ICE after selecting the “fileserver” Workload.

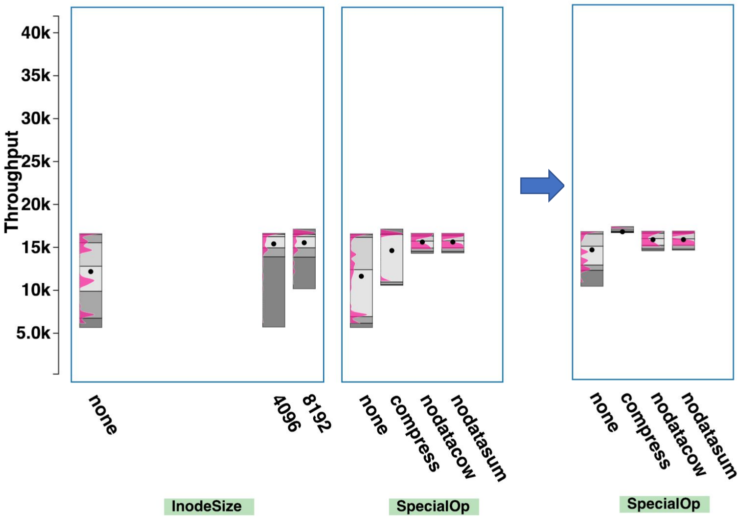


Figure 8.3: Using ICE to select parameter values for btrfs under the fileserver workload (partial screenshots).

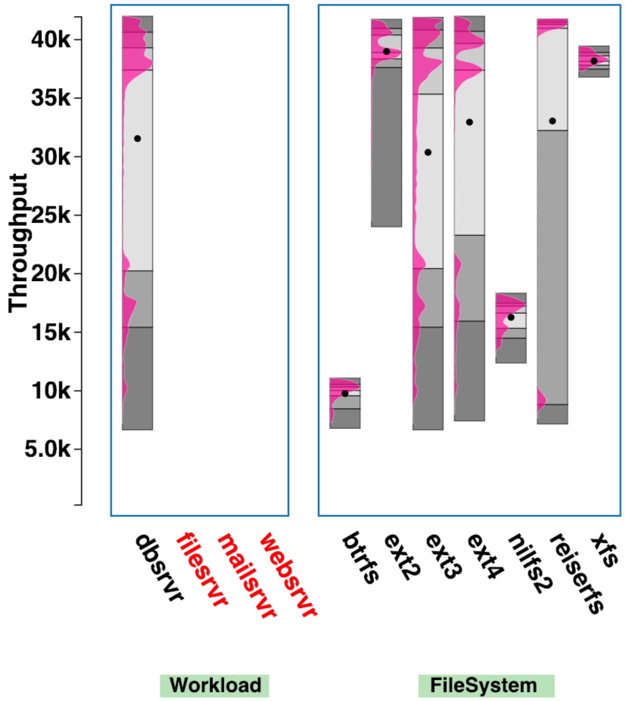


Figure 8.4: Partial screenshot of ICE after selecting the “dbserver” Workload.

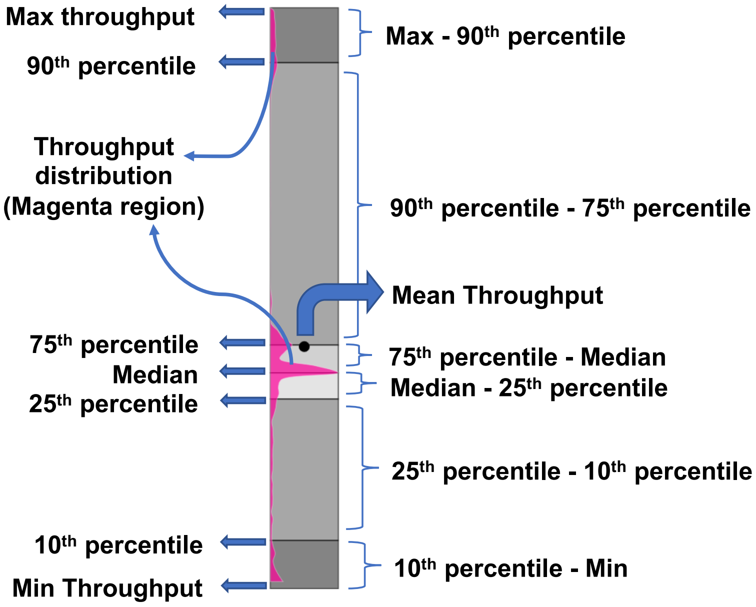


Figure 8.5: Annotated bar plot explaining how to read it.

shows an annotated example of one such bar. The shading distinguishes different percentiles of the throughput: the darker shades on the top and bottom represent the range from the maximum to the 90th percentile and from the minimum to the 10th percentile. Medium shades mark the ranges for the 90th to 75th and the 10th to 25th percentiles; the lightest shades in the middle mark the 75th to 25th percentiles. The black horizontal lines in each bar mark major percentile boundaries (90th, 75th, 50th (median), 25th, and 10th). The mean of the data is indicated by a solid black dot. In addition to the percentile shades, the distribution of the data is shown by the magenta-colored area(s) on each bar, giving more detail about exactly how the configurations represented by that bar are distributed in the throughput space. We chose all colors and shades carefully by using ColorBrewer [21,43], ensuring that they are visible on a variety of displays and to users who suffer from color-blindness.

Returning to Figure 8.1, ICE is designed based on *scented widgets* [206], which were originally proposed as graphical user interface controls enhanced with embedded visualizations that facilitate navigation in information spaces. We see that ICE displays multiple bars, each representing the throughput distribution of a subset of configurations in which one parameter is fixed to a given value. Since ICE is interactive, all of these bars change as the user explores the data. The parameters are grouped according to type (note that some parameter types have been omitted for space reasons; a full version of the display can be seen at `http://www.fsl.cs.stonybrook.edu/%7Ezhccao/ice`). A cumulative bar at the right-hand side of the figure, which also changes during exploration, shows the throughput distribution of the union of the chosen configurations. In this example, the initial display shows the distribution of all configurations for 7 file systems across 4 workloads.

Given an initial setting, the user can select any combination of workload, file system, and storage parameters, and the bars will be updated to show throughput distribution, as we saw in the example above. With this design, users can easily select parameters with different objectives. For example, Maria maximized throughput by selecting the bar with the highest solid black dot, but she could also reduce performance variance by focusing on shorter bars. Our case studies in Section 8.3 show how ICE can help users make such configuration decisions.

We designed ICE generically, so that it is easy to analyze new datasets collected on different systems. We plan to make ICE open-source to facilitate research on understanding system parameter spaces and optimizing large systems.

8.3 Case Studies

In Section 8.2 we showed one example of how ICE can be used to analyze system throughput and tune parameters to achieve high performance. In this section, we describe two more case studies to show how ICE can also help analyze performance stability and reliability. These studies are based on our real experience in analyzing and tuning storage systems [29,216].

8.3.1 Performance Stability

Now suppose that Maria wants to configure a system as an email server, for which she cares about performance stability. The *range* (difference between maximum and minimum) and *Inter-Quartile Range* (IQR) (difference between 75th quartile and 25th quartile) are often used to quantify stability in system performance [28]. ICE visually presents the range as the length of each bar, and IQR

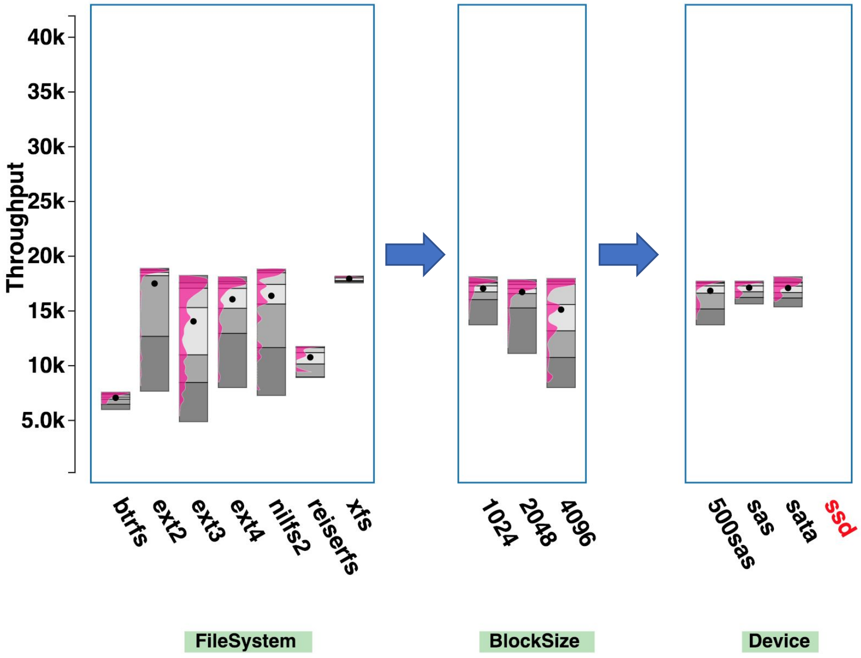


Figure 8.6: Using ICE to optimize a mail server (partial screenshots). We chose Workload = “mailserver”; FileSystem = “ext4”; and BlockSize = “1024”.

as the length of the lightest shade in the middle of each bar (see Figure 8.5). Maria starts her analysis with ICE and selects the *mailserver Workload*. The left part of Figure 8.6 shows a partial ICE screenshot after doing so. The bars for each parameter value present the updated throughput distribution if that value is chosen. For example, the bar above “btrfs” shows the throughput distribution of all Btrfs configurations under the mailserver workload. The updated bars guide Maria to select a value for another parameter, based on her objectives. Clearly, under *mailserver xfs* has by far the smallest throughput range, even though its highest throughput value is slightly lower than those of *nilfs* and *ext2*. Since stability is the primary concern, Maria thus decides to configure her server using XFS.

Unfortunately, Maria’s boss informs her that upper management has established a policy that all corporate computers have to use the Ext4 file system, regardless of application. She returns to ICE, selecting *ext4*. Now she continues configuring parameters for Ext4, because that file system shows a wide range of throughput, indicating unstable performance. As shown in the middle part of Figure 8.6, a value of *1024* for *BlockSize* gives the most stable result; Maria thus chooses this value.

The right part of Figure 8.6 shows the final step. Maria has three types of HDDs available: “sas” (a 146GB SAS HDD), “500sas” (a 500GB SAS HDD), and “sata” (a 250GB SATA HDD). She estimates that her email system will only need 100GB, so she can ignore the HDD capacity and focus solely on performance. The bar associated with *sas* appears the the shortest, which means the 146GB SAS HDD has the most stable performance. Therefore, Maria selects that HDD.

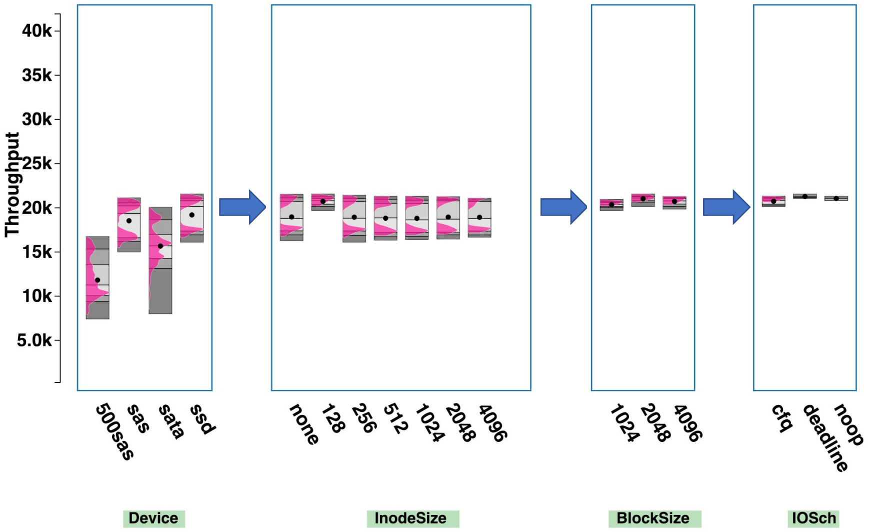


Figure 8.7: Using ICE to optimize multiple constraints (partial screenshots). We chose Workload="dbserver"; FileSystem = "ext4"; Device = "ssd"; InodeSize = "128"; BlockSize = "2048".

8.3.2 Constrained Tuning

When conducting system analysis, multiple objectives sometimes need to be considered at the same time. For example, system administrators may want to configure a *stable* system (i.e., low variability) and still achieve high *performance*. In this case certain constraints may be added to the analysis process. Here we show a case study of how ICE can be easily applied to reflect such multiple constraints and help the analysis.

This time Maria wants to configure her system as an OLTP database server that uses Ext4. However, she wants to ensure reliability for the file system; therefore, she sets the Ext4 journaling mode to *data=journal*. She then uses ICE to analyze the system and help her find the configuration that leads to the highest throughput under the current constraints, as shown in Figure 8.7. She has already tested four device types, 3 HDDs and 1 SSD. Unsurprisingly, the SSD shows the highest maximum throughput (top of bar) so she chooses that. ICE then updates the rest of the display to reflect that choice, and Maria then focuses on the *Inode Size*. A 128-byte inode is clearly preferable in this situation, so she selects that and moves to the *Block Size* where 2KB has better stability. Finally, she chooses *deadline* for the *I/O Scheduler*, as it improves stability more without hurting performance, as shown in the rightmost part of Figure 8.7.

It is important to note that Maria is not restricted to following the particular path given in this example. She could have chosen an I/O scheduler first, followed by selecting the best block and inode sizes, and waited to the end to chose a disk type. She also could have selected an inode size, observed how it interacted with the other parameters, and backed out so that should could pick a value later based on her choices for something else. One of the benefits of ICE is that the user can easily and quickly try different options to see how it affects the results, exploring the parameter space along the path that best suits her needs and research style.

8.4 Future Work

This position paper advocates the use of interactive visual analytics for computer systems analysis and research. We plan to continue this work. In particular, the following three enhancements to ICE are promising: (1) During interactive analysis, it is useful to track the progression of the analysis and how the current state compares to previous ones. We plan add a provenance scheme [134, 179] that would show previous results along a timeline, enabling us to see the "path" by which a particular analysis was reached. (2) ICE is already scalable in the number of presented *configurations* since it displays distributions, and is general because new datasets can be dynamically imported. We plan to further improve ICE to support even larger spaces, consisting of hundreds or thousands of parameters. Previous work has demonstrated that some parameters have greater impact than others [29, 198]. We plan to expand ICE to visualize and help analyze parameter importance based on measures of redundancy, uniqueness, coverage, or on metrics from information theory such as mutual information [29]. (3) When tuning system parameters, not all changes are equal in terms of the overhead. Some parameters can be dynamically changed without any additional cost, such as selecting the Linux I/O scheduler. However, some changes, such as installing hardware or reformatting a file system, would consume significant amounts of time or money. We are working to enhance ICE by assigning each parameter a "cost" category (e.g., reformatting is more costly than a reboot), allowing users to assign weights to each cost category, and then allowing ICE to explore the space under cost constraints. For example, users could opt to avoid any cost category above a certain threshold, or only avoid it unless the performance benefits exceed a certain desired metric.

We designed ICE to analyze computer system parameter spaces, where some previous techniques have not proven as useful as one might wish. Nevertheless, we are investigating ways to incorporate approaches such as Parallel Coordinates [87], Parallel Sets [104], and Data Context Maps [37] into ICE. We also would like to integrate machine-learning techniques [105, 187, 198] to help guide the analyst in exploring large parameter spaces.

8.5 Conclusions

The Interactive Configuration Explorer (ICE) is an *interactive visual analytics* tool that helps analyze and understand computer systems. It addresses the limitations of existing techniques, such as dealing with high-dimensional spaces and infusing domain knowledge, by making it easy for humans to understand and explore large parameter spaces. We described ICE and presented several exemplary case studies on a storage system to demonstrate how it can help analyze a system's performance, stability, etc. We believe that interactive visual analytics such as ICE, possibly in conjunction with other techniques (e.g., Parallel Coordinates [87] or Data Context Maps [37]), can greatly improve our ability to manage complex computer systems. ICE has the potential to pave the way for more applications of interactive visual analytics to systems research, leading to better understanding and more robust design of computer systems.

Chapter 9

A Practical Auto-Tuning Framework for Storage

9.1 Motivation

Despite some promising results in applying black-box optimization techniques for auto-tuning storage systems, we believe these techniques still lack several critical features to achieve practical auto-tuning. For example, as seen in Figure 5.2, after around 3.5 hours, GA already found a near-optimal configuration, but it spent a lot of additional rounds and resources, yet not improving overall performance much. Moreover, Figure 2.2 and Table 5.1 showed that storage evaluation results depend heavily on the hardware and running workloads. Our previous work reported similar observations [28, 170]. Therefore, our auto-tuning framework also needs to react to environment changes (e.g., hardware, workload). In Figure 5.2, SA got stuck in a configuration with throughput value of less than 18K IOps. Additional experiments we conducted suggest that the quality of initialization has a significant effect on the convergence time and final optimization results. Lastly, our preliminary experiments assume that all configurations have identical cost: that moving from one configuration to another has the same (fixed and low) cost. For many storage systems, however, it is not true: for example, changing the *block size* of a file system may require a costly and time-consuming reformat and data migration.

In this chapter we discuss our design for a more intelligent and practical auto-tuning framework, building on previous chapters and addressing the aforementioned limitations. We are exploring techniques that add vital missing features from existing optimization methods:

- A **workload modeler**, which can extract features from system-collected metrics and characterize the running workload based on them. This is useful in determining when to restart the auto-tuning process and how to "transfer" evaluation results from one workload to another.

- The key component of our auto-tuning framework will be the **optimizer**. In addition to the optimization algorithms evaluated in Chapter 5 and Spectra, and the parameter selection algorithm (Chapter 7), we further added several features.

  1. A criteria when the optimization algorithm should *stop searching*, having reached a "good enough" system configuration;

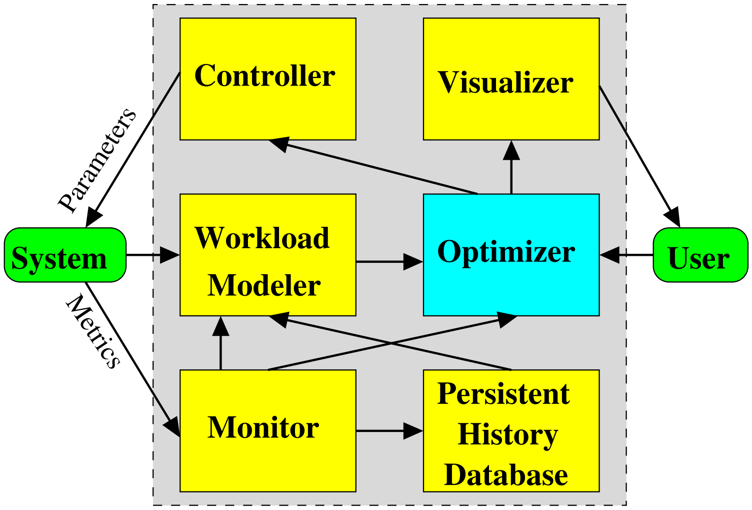


Figure 9.1: Auto-tuning Framework

2. A mechanism to *pick an initial set of search space locations*;
3. A mechanism to categorize each parameter based on its *changing cost* (e.g., a simple run-time changeable parameter vs. one that requires a system reboot and some downtime).

- The visual analytic tool discussed in Chapter 8, ICE, actually serves as an example of a **visualizer** in our framework. A visualizer help storage administrators understand storage parameter spaces and make design-tradeoff decisions.

The rest of this chapter is organized as follows. We describe the design of the auto-tuning framework and its components in Section 9.2. Section 9.3 describes how we apply distance metrics and classification techniques to quantify the similarity among workloads. We describe how to categorize parameters and deal with configuration change cost in Section 9.4. We compare various initialization and stopping methods in Section 9.5 and Section 9.6.

9.2 Auto-Tuning Framework

To address the limitations discussed in Section 9.1, we propose our enhanced auto-tuning framework, as shown in Figure 9.1. It consists of 6 components.

- **Monitor**, which collects and processes system metrics for other components' use.

- **Workload Modeler**, which utilizes metrics collected by the *Monitor*, to characterize the running workload.

- **Optimizer**, which includes the core auto-tuning algorithm with newly added features. It calculates the optimization objective for the current system configuration, based on metrics collected by the *Monitor*. Our framework is general enough to optimize for *any objective* that can be quantitatively measured. Examples are I/O throughput or latency, energy consumption, or even an economic cost function comprising multiple metrics [123, 181].

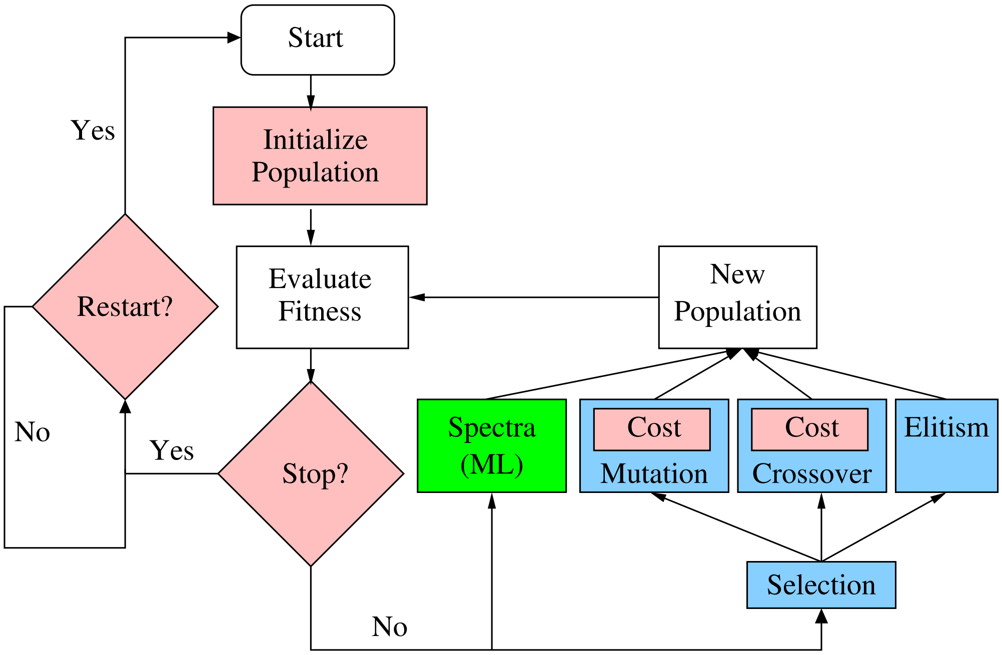


Figure 9.2: Workflow for an enhanced Optimizer (GA)

- **Controller**, which is responsible for changing the system settings based on the configuration picked by the *Optimizer*.
- **Persistent History Database**, which stores previous evaluation results persistently. The auto-tuning algorithm can use part (or all) of this history to direct the search or build predictive ML models. A practical implementation may also periodically purge older or less valuable database entries to reduce storage costs.
- **Visualizer**, which provides the user with interactive visualization and insights into complex n-D spaces. More details on *Visualizer* were discussed in Chapter 8.

Our *Optimizer* is designed to address the issues observed in our preliminary experiments. Here we use GA as a case study explaining how it works, but all the new components are applicable to other black-box optimization algorithms as well. As shown in Figure 9.2, white boxes represent GA's original optimization loop components and blue ones relate to GA's selection process; pink ones are new components in our hybrid optimization algorithm; and the green box represent Spectra, our parameter selection algorithm as detailed in Chapter 7.

9.3 Workload Modeler

In this section we use *parameters* to refer to system factors whose values can be manually set, and *features* or *metrics* for values that can only be measured. The workload modeler's role is to find a set of features that is sufficient to differentiate workloads and quantify their changes. It is useful when the *Optimizer* wants to re-utilize past evaluation results for a new workload—by finding the closest known workload for which we have a high-performing system configuration (see Section 9.5).

How to characterize a system workload remains an open problem. A few efforts were made in modeling database query workload [130, 198] and storage workloads [10, 26]. To model a storage workload, we investigated previous work on storage workloads [10, 26, 75, 150, 154, 160, 171], and came up with a list of 18 features that have been commonly applied in storage workload study. The complete list of features is as follows: *read&write LBA range*, *read&write LBA standard deviation*, *read&write LBA delta offset standard deviation*, *read inter-arrival average*, *read LBA range*, *read LBA standard deviation*, *read LBA delta offset standard deviation*, *read size average*, *write inter-arrival average*, *write LBA range*, *write LBA standard deviation*, *write LBA delta offset standard deviation*, *write size average*, *read/write ratio*, *read inter-arrival relative standard deviation*, *read size relative standard deviation*, *write inter-arrival relative standard deviation*, and *write size relative standard deviation*.

Designing a complete workload modeler goes beyond the scope of this thesis, due to the complexity of modern workloads, as well as the time-consuming process of collecting data. Nevertheless, as we already started working on this, we describe our preliminary results here. We collected block traces from multiple sources. We ran four macro workloads using Filebench (see Section 4) and their variants (e.g., different number of threads, different number of files, etc.). To collect block access information, we started *blktrace* as a background job at the same time with Filebench. We executed Filebench for 3 minutes. After each experiment, we parsed the block-access information using *blktrace*. Once we finished these processes, we transferred all these data to a separate server for offline post-processing. Our post-processing program reads in the output of *blktrace* and captures data into N-second vectors, and calculated the 18 aforementioned features and finally normalize every feature using z-score [215]. We trained a random forest (with 10 decision trees) with 20% of the Filebench traces, and tested it on the other 80% dataset. The predication accuracy is **94%**. Our preliminary results demonstrate the feasibility of workload prediction and modeling by utilizing the repeated workload patterns and carefully-designed features. We also collected traces from other sources. We ran MySQL and PostgreSQL using TPC-DS benchmark [195]. The FIU IODedup traces [102] and MSR Cambridge Traces [147] are also good sources of traces.

In the future, we plan to perform feature selection or clustering analysis on all extracted features and remove redundant ones. For the selected list of metrics, we will consider a distance function to quantify the similarities between workloads. Example distance functions include the earth-mover-distance (EMD) function [90, 194].

9.4 Parameter Categories and Cost Function

Many traditional optimization problems assume that moving from one configuration to another has the same constant cost. In practice, however, this is not always true. Imagine our optimizer finds a configuration with 10% better performance than the current one, but needs to change the format of the underlying file system—requiring a lengthy downtime to backup the data, reformat, then restore the data. Some users may not accept such a cost to gain 10% better performance—but other users might. Therefore, we propose to include the concept of *cost functions* into our auto-tuning framework. The cost of one parameter roughly correlates to how much downtime the system has to endure while changing the value of it.

We carefully studied common storage parameters, including *mkfs* and *mount* options for Ext4, XFS, Btrfs, and I/O related Linux kernel parameters, etc. We broadly categorize parameters into four categories based on their impact on downtime. Each category can be further subdivided into sub-categories, as shown in Table 9.1. We also label and categorize all parameters that we have experimented in *Storage V1 & V2* (Table 4.3), *Storage V3* (Table 4.4), and *Storage V4* (Table 4.5), in column "Example Parameters".

Category (Cost)	**Named Category**	**Description**	**Example Parameters (Table 4.3, 4.4 & 4.5)**
Category 0	dyn-kernel	Dynamically changeable os parameters, most of which can be tuned by *sysctl*.	I/O scheduler, dirty background ratio, dirty ratio
	dyn-app	Change app parameter without restarting the app.	(n/a)
Category 1	app-restart	Restart app.	(n/a)
	remount	*remount* file system.	atime option, inode readahead blocks (Ext4), log buffer count (XFS), log buffer size (XFS), allocation size (XFS), notail option (Reiserfs)
	umount-mount	*umount* file system and then *mount* it back.	journal option (Ext3, Ext4), compress (Btrfs), nodatacow (Btrfs), nodatasum (Btrfs), journal option (Nilfs2), journal option (Nilfs2)
Category 2	reboot	A system reboot is required.	(n/a)
	bios	BIOS change.	(n/a)
Category 3	reformat-restore	File system re-creation is required, implying a backup restore cycle.	block size (Ext2, Ext3, Ext4), inode size (Ext2, Ext3, Ext4), block group (Ext2, Ext3, Ext4), flex block group (Ext4), sector size (XFS), allocation group (XFS), node size (Btrfs), block size (Nilfs2), blocks per segment (Nilfs2), block size (Reiserfs)
	hardware-change	Hardware change.	storage devices

Table 9.1: Categories of Parameters.

Category 0 parameters are dynamically tunable, and thus come with very little to no cost. It mainly includes kernel parameters that can be dynamically changed or some application parameters that can be changed without restarting. *Category 1* parameters come with minor cost of just a few seconds of downtime, including application parameters that requires restarting the application, file system parameters that can be tuned through *remount*, or *umount* and *mount* the file system back. Parameters of *Category 2* are usually associated with a medium level of cost, which might require system downtime of several minutes. Parameters that require rebooting the system or some BIOS changes belong to this category. *Category 3* parameters come with much more cost than the other categories; these often require file system reformatting or some hardware changes. Changing *Category 3* parameters can bring downtime from hours to even several days, depending on the amount of data that needs to be migrated.

To identify the penalties associated with each system parameter, administrators have to categorize them. This needs to be done only once for each system parameter, and can then be disseminated publicly. We conducted a user study by categorizing all the parameters that we have experimented with. It took two graduate students with some familiarity of storage systems around two hours to categorize and test around 30 parameters. Users, however, may need to assign weights to the various cost categories in each environment (e.g., more conservative in production, more aggressive in experimental systems). The default and most simple cost function is to assign infinitely large weights to certain categories, which means never changing the values of these categories. For example, if the user does not want to reformat the system, our auto-tuning algorithm will just use tunable parameters in *Categories 0 & 1 & 2*. Our conversion with multiple storage experts, either from academia or industry, actually suggest that this could be the most common and acceptable cost functions.

Figure 9.3(a) shows experiments we conducted on GA using *Storage V3* under *mailserver* workload. Unlike previous experiments, here we do not change values of *Category 3* parameters, assuming that users cannot afford the higher cost of reformatting file system and data migration. As shown in the figure, GA can still gradually find better configurations and improve the system throughput, by only tuning *Category 0 & 1 & 2* parameters. The best throughput found is around 14.6k IOps, compared with 18k IOps if tuned all categories, as shown in Figure 9.3(b).

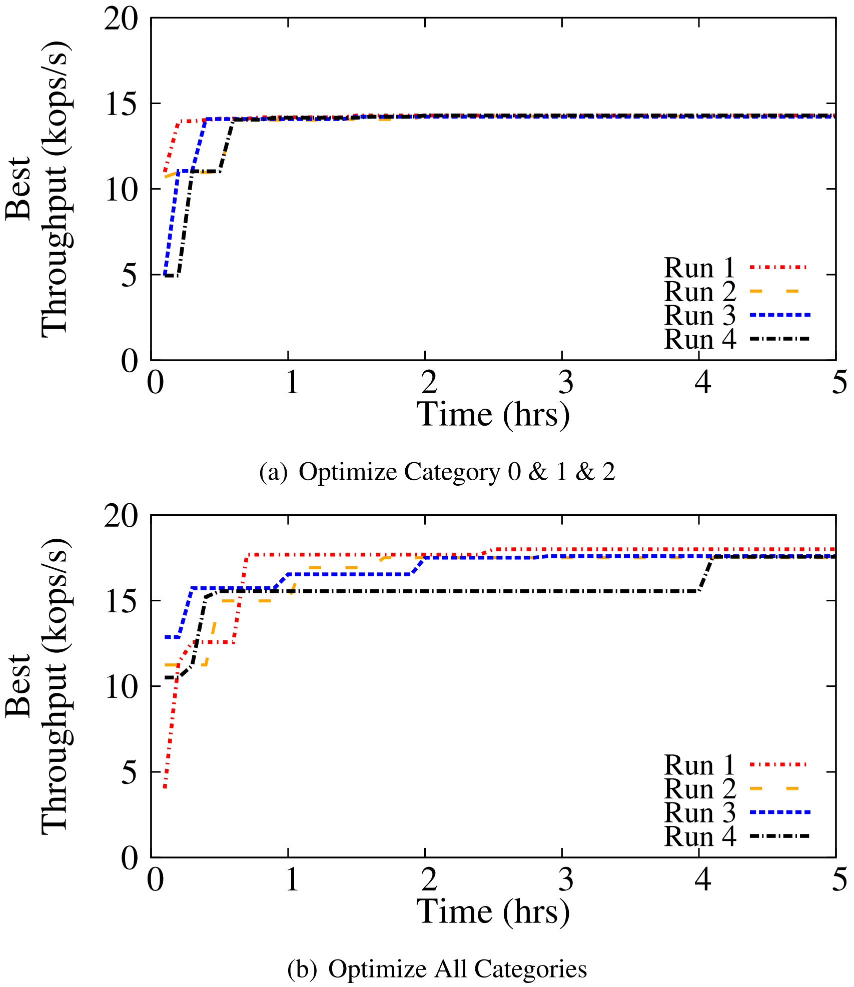
The other approach for dealing with parameter-change costs is to include the cost into the optimization objective function, and thus the objective becomes a complex cost formula (similar to our economics-based cost functions [123]), rather than a single system metric like I/O throughput.

9.5 Initialization

As discussed in Section 9.1, the quality of initialization has a large impact on the convergence time and final results of optimization. Much work has been done on proposing and analyzing different initialization methods for various optimization algorithms [66, 74, 174]. We are investigating the following initialization methods to design the best one for our needs:

1. **Simple Random Sampling**, where each configuration is chosen entirely by chance and has an equal chance of being included in the sample [31]. It is the default for many optimization techniques [66]. Although we expect it sometimes to be inefficient, it serves as a useful baseline for more intelligent methods.

2. **Stratified Random Sampling**, which divides the whole space into sub-spaces, and takes samples from each sub-space. It is quite useful when we expect the measurement of interest to vary among the different sub-spaces [31]. In case of optimizing for storage configurations, since parameters directly impact performance, an ideal initialization method should cover



(a) Optimize Category 0 & 1 & 2

(b) Optimize All Categories

Figure 9.3: GA results on Storage V3 under mailserver workload and un/restricted cost categories.

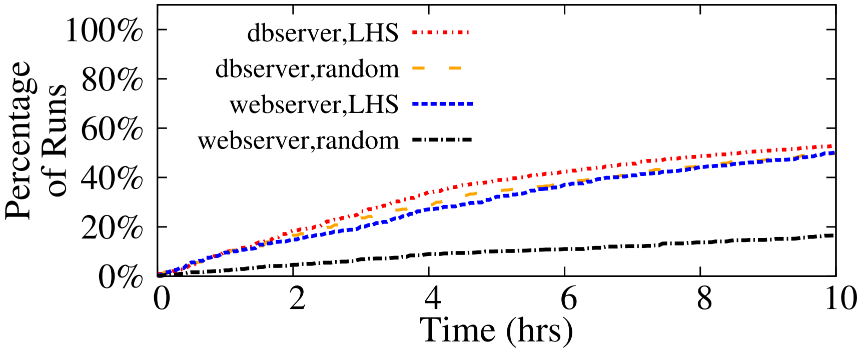


Figure 9.4: Comparison of LHS and random initialization on dbserver and webserver workloads.

each parameter value more uniformly. In fact, Latin Hypercube Sampling (LHS) [109, 135], which belongs to this type of sampling, are proved to be effective in black-box optimization [66, 157].

3. **Including domain knowledge**. Domain experts may already know some good configurations for certain workloads. Including them has been shown to increase the search efficacy [4, 19, 92, 205]. Another good example here is if experts know the impact of several common parameters on overall system performance, the initialization method could try to sample the preferred parameter values more frequently. Interestingly, our automated techniques can also be used to evaluate the accuracy of domain experts' actual recommendations.

In Figure 9.4 we compare the efficacy of two initialization methods: *Simple Random Sampling* and *LHS*. The experiments were run with GA and on *Storage V3*. Since exploration is one critical component of all optimization methods (see Section 2.4), We repeated the experiments on each initialization method 1,000 times. We compare different initialization methods for their probability of finding near-optimal configurations. Here we define a near-optimal configuration as one with throughput higher than **99%** of the global optimal value. The Y axis shows the percentage of total runs that found a near-optimal configuration within a certain time (X axis). Clearly LHS outperforms simple random sampling, with a higher chance to find near-optimal configurations and more quickly, for both *dbserver* and *webserver* workloads. We believe this is because different parameters have a different level of impact on performance (see Chapter 7), and GA's efficacy comes from assigning higher chances of survival to configurations with a combination of more effective parameter values. Initialization through stratified random sampling, such as LHS can also let GA find these effective parameter values earlier. Another interesting observation is that even though LHS outperforms random initialization in both workloads, the difference is much larger in *webserver* than *dbserver*. This is because the Ext4 inode size has to be smaller than its block size, which indicates that there could be more configurations with larger block size (4K, 2K) than a smaller block size (1K). In fact, *Storage V3* has 972 configurations with 1K block size, 1,458 with 2K block size, and 1,458 for 4K block size. With random initialization, the configurations within the first generation come with a higher chance of having block size values of 2K and 4K than 1K. For *webserver* workload, the near-optimal configurations all come with 1K block size, which explains why LHS outperforms random initialization a lot. Random initialization have a higher probability of spending more time exploring configurations with 2K and 4K block sizes. Although eventually GA can still find the near-optimal areas via mutation, it does take longer time. For *dbserver*, the difference between LHS and random initialization is not as significant as that of *webserver*, since the block size values of near-optimal configurations are mostly 2K.

When the *Workload Modeler* detects that the running workload has changed, the *Optimizer* needs to restart and thus re-initialize. Based on the similarity of the new workload with the previous one given by *Workload Modeler*, our re-initialization process can include some of the top configurations found with an old workload. This actually belongs to the third category of initialization method, of exploiting domain knowledge. The details of this technique are beyond the scope of this thesis and are left as future work.

9.6 Stopping Criteria

As shown in Section 9.1, some stopping criteria should be included in an auto-tuning algorithm, otherwise it can spend a lot of additional rounds and resources without improving overall performance much if at all. Commonly applied stopping criteria in black-box optimization algorithms include:

- *Time-based stopping criteria*, which let the optimization algorithm stop after a certain time or number of evaluations.
- *Sliding-window (weighted) average*, which stop the optimization algorithm if it fails to find a better configuration within a certain time window.
- *Algorithm-specific stopping criteria*, which use the *history* information stored by the optimization algorithm. For example, the diversity of genes within each generation can be used to determine when to stop [164].
- *User-specified stopping criteria*. Users may want to specify that they want to achieve X IOPS in terms of throughput. After finding a configuration that meets such requirements, the optimization algorithm can safely stop running.

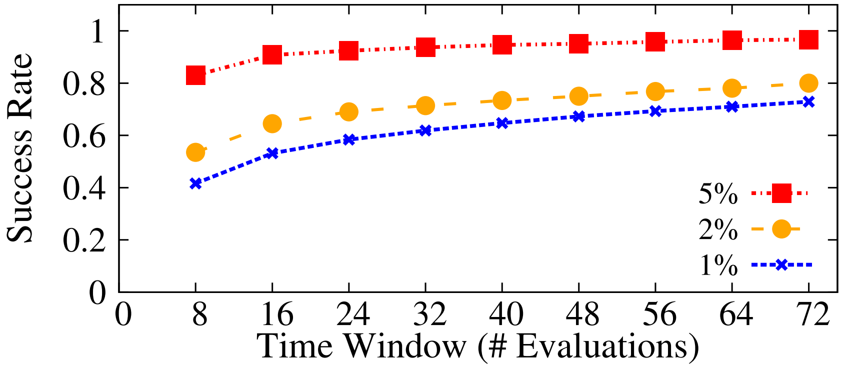


Figure 9.5: Time window based stopping criteria.

Figure 9.5 shows results using sliding-window based stopping criteria. We define a successful stop as one that stopped early and the best configuration found at that point has throughput that is at most K% lower than the best possible one. We again repeated the experiment for 1,000 runs, and the Y axis shows the percentage of runs with success stops. The X axis represents the sliding window size, which means that if the algorithm fails to find a better configuration within X consecutive evaluations, we just stop. We conducted three sets of experiments, with K percent values of 1%, 2%, and 5%. For example, $K = 1$ indicates that even if we stopped early, the best throughput found is higher than 99% of the best if terminated normally, (in our case, 120 generations). For $K = 1$ and window size of 8, around 40% of runs stopped early and successfully. Larger window sizes generally result in better success rates, with a window size of 72 evaluations reaching nearly 70% for $K = 1$ and 96% for $K = 5$.

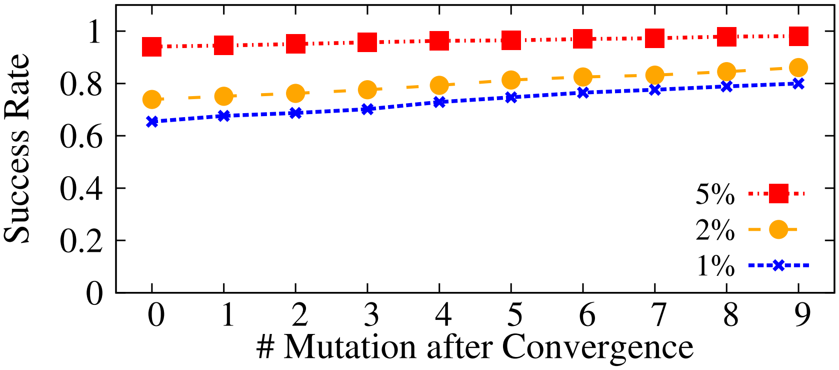


Figure 9.6: Diversity based stopping criteria.

History (see Section 2.4) can also be used to determine when to stop the algorithm. Here we use GA as an example. As discussed in Section 5.4, GA works by assigning a higher chance of survival to well-performing gene alleles; and thus after a certain number of generations, the diversity of alleles will decrease and eventually converge to a single allele. We say one parameter (gene) has converged if M% of the configuration within the current population share the same value for it. And we define the convergence of GA as N% of the parameters (genes) have converged. Figure 9.6 shows the results of this diversity-based stopping criteria, with $M = 80$ and $N = 80$. Similar to Figure 9.5, we repeated each experiment for 1,000 runs, and we used the same definition for a successful stop. The Y axis shows the percentage of runs with success stops. The X axis represents the number of mutations that occurred after convergence. We allow some mutations after convergence to avoid getting stuck in local optima. As we can see from Figure 9.6, diversity-based stopping criteria show better success rate than window-based ones. For $K = 5$, stopping right after convergence can give a success rate of 94%, while allowing 9 mutations can increase the rate to 98%. For $K = 1$, diversity-based stopping with 9 mutations allowed after convergence can still give 80% of success rate, compared with 73% given with window-based stopping (window size = 72). Another observation from Figure 9.6 is that allowing a number of mutations after convergence can slightly increase the success rate of any defined stopping criteria. We also tried other definitions of convergence, with different values of M and N. They produce similar results: increasing values of M and N will slightly increase the success rate, but of course it will take longer time to converge, and thus longer to stop the algorithm.

Despite some promising results, we believe more sophisticated criteria are needed to stop the algorithm more accurately and quickly. The key challenge would be to determine how close the current solution is to the global best, and whether the algorithm just got stuck in a local optima and simply needs more time for (random) exploration. For example, in our diversity-based stopping criteria, we allow a certain number of mutations after diversity converge to avoid getting stuck in local optima. However, the mutation here is purely random, not taking into account any previous evaluated alleles. A potential improvement is to design a more intelligent mutation, trying to assign higher probability to “unvisited areas” in the search space.

In addition to a stopping criteria for the whole optimization process, another type of early stopping criteria could also be defined within each evaluation of a configuration. Evaluating a single configuration for storage systems may take several minutes or even hours. If our optimizer can recognize early that the configuration under evaluation is operating worse compared to known ones, then the optimizer can stop evaluating the current run early. Moreover, when the *Workload Modeler* detects that the running workload has changed sufficiently, the algorithm might need to be restarted and re-optimize the system. Therefore, we also need to define certain *restarting criteria* for our auto-tuning framework. We leave the (early) stopping criteria for single evaluation and restarting criteria as our future work beyond this thesis.

Chapter 10

Conclusions

Optimizing storage systems can provide significant benefits especially in improving I/O performance. Alas, storage systems are getting more complex, contain many parameters and an immense number of possible configurations; manual tuning is therefore impractical. Worse, many of those parameters are non-linear or non-numeric; traditional linear-regression-based optimization techniques do not work well for such problems. Therefore, in this work, we propose to auto-tune storage system configurations.

We first performed a comparative study on various black-box optimization algorithms. **(1)** We evaluated *five* popular but different auto-tuning techniques, varied some of their hyper-parameters, and applied them to storage and file systems. **(2)** We show that the speed at which the techniques can find optimal or near-optimal configurations (in terms of throughput) depends on the hardware, software, and workload; this means that no single technique can "rule them all." **(3)** We explain why some techniques appear to work better than others.

In our auto-tuning experiments, we observed that repeated experiments in well-controlled, identical environments could produce results with high variations. Therefore, we then provided the first systematic study on performance variation in benchmarking a modern storage system. We showed that variation is common in storage systems, although its magnitude depends heavily on specific configurations and workloads. Our analysis revealed that block allocation is a major cause of performance variation in Ext4-HDD configurations. From a temporal perspective, the magnitude of throughput variation also depends on the window size and changes over time. The latency distribution for the same operation type could also vary even over repeated runs of the same experiment. We quantified the correlation between performance and latency variations using a novel approach.

Modern storage systems come with many parameters that affect their behavior. Tuning parameter settings can bring significant performance gains, but both manual tuning by experts and automated tuning have difficulty dealing with the large number of parameters and configurations. We propose Spectra, which includes a variance-based metric for quantifying storage parameter importance, and a greedy yet efficient parameter-selection algorithm. We evaluated Spectra across multiple datasets and showed that there is always a small subset of parameters that have the most impact on performance—but that the set of important parameters changes with different workloads, and that there are interactions between parameters. We demonstrated Spectra's efficiency by testing it on small fractions of the configuration space.

To help understand auto-tuning results and analyzing system behavior, we co-designed Interactive Configuration Explorer (ICE). ICE is an *interactive visual analytics* tool that helps analyze and understand computer systems. It addresses the limitations of existing techniques, such as dealing with high-dimensional spaces and infusing domain knowledge, by making it easy for humans to understand and explore large parameter spaces. We described ICE and presented several exemplary case studies on a storage system to demonstrate how it can help analyze a system's performance, stability, etc.

We believe traditional black-box optimization techniques still lack several critical features to achieve practical auto-tuning. We prototyped a workload modeler, which can extract features from system-collected metrics and characterize the running workload based on them. We categorize each parameter based on its *changing cost*, and showed how our auto-tuning framework will optimize storage systems within certain categories of parameters. We also compared the efficacy of multiple initialization methods and stopping criteria with our framework.

Another contribution of our project is that we are collecting a lot of data on evaluating different storage configurations on various workloads. For more than three years, we have collected a large data-set of over 500,000 data points. All our results were stored in a carefully designed database. We already released our current datasets and we will continue releasing as we collect more datasets, to facilitate research on understanding and optimizing storage performance.

Finally, it is our thesis that auto-tuning storage systems is important, promising, and feasible with a carefully designed framework to include missing yet critical features. We hope our auto-tuning framework can improve systems' performance efficiency, and save energy and human resources in the long term.

10.0.1 Future Work

Our auto-tuning framework can be extended further beyond the scope of this thesis. We see at least the following interesting and promising directions.

- Experiment with larger and more complex parameter spaces. Our current experiments were conducted on spaces consisting of 8 to 10 parameters. We are collecting data from a search space consisting of 12 parameters (expected to conclude around March 2019). We plan to extend our work with an even larger parameter space, where exhaustive search is impossible. We will test and improve the efficacy of our auto-tuning framework in real-time.

- We plan to extend Spectra to support other parameter-selection techniques, such as Group Lasso [38, 100, 214] and ANOVA [24, 31, 113, 204]. We will evaluate and improve Spectra with more optimization objectives (e.g., reliability), and larger storage-parameter spaces. We also plan to integrate Spectra with auto-tuning algorithms.

- We plan to investigate the possibility of applying more Machine Learning in our auto-tuning framework, including design hybrid algorithms that combine traditional optimization algorithms and ML.

- We plan to extend our work on workload characterization, collecting more traces from varied sources and test with more workload features.

- Details of future work for ICE were discussed in Section 8.4.

Bibliography

[1] Emile Aarts and Jan Korst. *Simulated annealing and Boltzmann machines*. New York, NY; John Wiley and Sons Inc., 1988.

[2] Abutalib Aghayev, Mansour Shafaei, and Peter Desnoyers. Skylight—a window on shingled disk operation. *Trans. Storage*, 11(4):16:1–16:28, October 2015.

[3] Abutalib Aghayev, Theodore Ts'o, Garth Gibson, and Peter Desnoyers. Evolving ext4 for shingled disks. In *Proceedings of the 15th USENIX Conference on File and Storage Technologies (FAST)*, pages 105–120, Santa Clara, CA, February-March 2017. USENIX Association.

[4] Ravindra K Ahuja and James B Orlin. Developing fitter genetic algorithms. *INFORMS Journal on Computing*, 9(3):251–253, 1997.

[5] Omid Alipourfard, Hongqiang Harry Liu, Jianshu Chen, Shivaram Venkataraman, Minlan Yu, and Ming Zhang. Cherrypick: Adaptively unearthing the best cloud configurations for big data analytics. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 469–482. USENIX Association, 2017.

[6] Guillermo A. Alvarez, Elizabeth Borowsky, Susie Go, Theodore H. Romer, Ralph Becker-Szendy, Richard Golding, Arif Merchant, Mirjana Spasojevic, Alistair Veitch, and John Wilkes. Minerva: An automated resource provisioning tool for large-scale storage systems. *ACM Trans. Comput. Syst.*, 19(4):483–518, November 2001.

[7] Terry Anderson. *The theory and practice of online learning*. Athabasca University Press, 2008.

[8] R. H. Arpaci-Dusseau, E. Anderson, N. T., D. E. Culler, J. M. Hellerstein, D. Patterson, and K. Yelick. Cluster I/O with river: making the fast case common. In *Workshop on Input/Output in Parallel and Distributed Systems*, pages 10–22, Atlanta, GA, May 1999.

[9] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. *Operating Systems: Three Easy Pieces*. Arpaci-Dusseau Books, 0.91 edition, May 2015.

[10] Jayanta Basak, Kushal Wadhwani, and Kaladhar Voruganti. Storage workload identification. *Trans. Storage*, 12(3):14:1–14:30, May 2016.

[11] Babak Behzad, Joey Huchette, Huong Luu, Ruth Aydt, Quincey Koziol, Mr Prabhat, Suren Byna, Mohamad Chaarawi, and Yushu Yao. Auto-tuning of parallel io parameters for hdf5 applications. In *Proceedings of the 2012 SC Companion: High Performance Computing, Networking Storage and Analysis*, SCC '12, pages 1430–, Washington, DC, USA, 2012. IEEE Computer Society.

[12] Babak Behzad, Huong Vu Thanh Luu, Joseph Huchette, Surendra Byna, Prabhat, Ruth Aydt, Quincey Koziol, and Marc Snir. Taming parallel i/o complexity with auto-tuning. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, pages 68:1–68:12, New York, NY, USA, 2013. ACM.

[13] Irwan Bello, Hieu Pham, Quoc V Le, Mohammad Norouzi, and Samy Bengio. Neural combinatorial optimization with reinforcement learning. *arXiv preprint arXiv:1611.09940*, 2016.

[14] James Bergstra and Yoshua Bengio. Random search for hyper-parameter optimization. *Journal of Machine Learning Research*, 13(Feb):281–305, 2012.

[15] James S. Bergstra, Rémi Bardenet, Yoshua Bengio, and Balázs Kégl. Algorithms for hyper-parameter optimization. In *Advances in Neural Information Processing Systems 24*, pages 2546–2554, 2011.

[16] MC Bhuvaneswari. *Application of Evolutionary Algorithms for Multi-objective Optimization in VLSI and Embedded Systems*. Springer, 2015.

[17] Christopher M Bishop. *Pattern Recognition and Machine Learning*, volume 1. Springer New York, 2006.

[18] D. Boutcher and A. Chandra. Does virtualization make disk scheduling passé? In *Proceedings of the 1st USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage '09)*, October 2009.

[19] Mark F Bramlette. Initialization, mutation and selection methods in genetic algorithms for function optimization. In *ICGA*, pages 100–107, 1991.

[20] Leo Breiman, Jerome Friedman, Charles J Stone, and Richard A Olshen. *Classification and regression trees*. CRC press, 1984.

[21] Cynthia A Brewer, Geoffrey W Hatchard, and Mark A Harrower. Colorbrewer in print: a catalog of color schemes for maps. *Cartography and geographic information science*, 30(1):5–32, 2003.

[22] Eric Brochu, Vlad M Cora, and Nando De Freitas. A tutorial on Bayesian optimization of expensive cost functions, with application to active user modeling and hierarchical reinforcement learning. *arXiv preprint arXiv:1012.2599*, 2010.

[23] Gavin Brown, Adam Pocock, Ming-Jie Zhao, and Mikel Luján. Conditional likelihood maximisation: a unifying framework for information theoretic feature selection. *Journal of machine learning research*, 13(Jan):27–66, 2012.

[24] Morton B Brown and Alan B Forsythe. Robust tests for the equality of variances. *Journal of the American Statistical Association*, 69(346):364–367, 1974.

[25] BTRFS. `http://btrfs.wiki.kernel.org/`.

[26] Axel Busch, Qais Noorshams, Samuel Kounev, Anne Koziolek, Ralf Reussner, and Erich Amrehn. Automated workload characterization for i/o performance analysis in virtualized environments. In *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering*, pages 265–276. ACM, 2015.

[27] M. Cao, T. Y. Ts'o, B. Pulavarty, S. Bhattacharya, A. Dilger, and A. Tomas. State of the art: Where we are with the Ext3 filesystem. In *Proceedings of the Linux Symposium*, Ottawa, ON, Canada, July 2005.

[28] Zhen Cao, Vasily Tarasov, Hari Raman, Dean Hildebrand, and Erez Zadok. On the performance variation in modern storage stacks. In *Proceedings of the 15th USENIX Conference on File and Storage Technologies (FAST)*, pages 329–343, Santa Clara, CA, February-March 2017. USENIX Association.

[29] Zhen Cao, Vasily Tarasov, Sachin Tiwari, and Erez Zadok. Towards better understanding of black-box auto-tuning: A comparative analysis for storage systems. In *Proceedings of the Annual USENIX Technical Conference*, Boston, MA, July 2018. USENIX Association. Data set at `http://download.filesystems.org/auto-tune/ATC-2018-auto-tune-data.sql.gz`.

[30] R. Card, T. Ts'o, and S. Tweedie. Design and implementation of the second extended filesystem. In *Proceedings to the First Dutch International Symposium on Linux*, Amsterdam, Netherlands, December 1994.

[31] George Casella and Roger L Berger. *Statistical Inference*, volume 2. Duxbury Pacific Grove, CA, 2002.

[32] Vladimír Černỳ. Thermodynamical approach to the traveling salesman problem: An efficient simulation algorithm. *Journal of optimization theory and applications*, 45(1):41–51, 1985.

[33] Kevin K. Chang, Abhijith Kashyap, Hasan Hassan, Saugata Ghose, Kevin Hsieh, Donghyuk Lee, Tianshi Li, Gennady Pekhimenko, Samira Khan, and Onur Mutlu. Understanding latency variation in modern DRAM chips: Experimental characterization, analysis, and optimization. In *Proceedings of the 2016 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Science*, SIGMETRICS'16, pages 323–336, New York, NY, USA, 2016. ACM.

[34] Min Chen, Shiwen Mao, and Yunhao Liu. Big data: A survey. *Mobile Networks and Applications*, 19(2):171–209, 2014.

[35] Ming Chen, Dean Hildebrand, Henry Nelson, Jasmit Saluja, Ashok Subramony, and Erez Zadok. vNFS: Maximizing NFS performance with compounds and vectorized I/O. In *Proceedings of the 15th USENIX Conference on File and Storage Technologies (FAST)*, pages 301–314, Santa Clara, CA, February-March 2017. USENIX Association.

[36] Y. Chen, M. Winslett, Y. Cho, and S. Kuo. Automatic parallel i/o performance optimization using genetic algorithms. In *Proceedings of the 7th IEEE International Symposium on High Performance Distributed Computing*, HPDC '98, pages 155–, Washington, DC, USA, 1998. IEEE Computer Society.

[37] Shenghui Cheng and Klaus Mueller. The data context map: Fusing data and attributes into a unified display. *IEEE transactions on visualization and computer graphics*, 22(1):121–130, 2016.

[38] Ch Chesneau and Mohamed Hebiri. Some theoretical results on the grouped variables lasso. *Mathematical Methods of Statistics*, 17(4):317–326, 2008.

[39] Liu Chu, Eduardo Souza De Cursi, Abdelkhalak El Hami, and Mohamed Eid. Reliability based optimization with metaheuristic algorithms and Latin hypercube sampling based surrogate models. *Applied and Computational Mathematics*, 4(6):462–468, 2015.

[40] Maurice Clerc. *Particle swarm optimization*, volume 93. John Wiley & Sons, 2010.

[41] Yvonne Coady, Russ Cox, John DeTreville, Peter Druschel, Joseph Hellerstein, Andrew Hume, Kimberly Keeton, Thu Nguyen, Christopher Small, Lex Stein, and Andrew Warfield. Falling off the cliff: When systems go nonlinear. In *Proceedings of the 10th Conference on Hot Topics in Operating Systems (HOTOS '05)*, 2005.

[42] James Cohoon, John Kairo, and Jens Lienig. Evolutionary algorithms for the physical design of vlsi circuits. In *Advances in evolutionary computing*, pages 683–711. Springer, 2003.

[43] Color Brewer 2.0. `http://colorbrewer2.org/`.

[44] Valentin Dalibard, Michael Schaarschmidt, and Eiko Yoneki. BOAT: Building auto-tuners with structured Bayesian optimization. In *Proceedings of the 26th International Conference on World Wide Web*, WWW '17, pages 479–488. International World Wide Web Conferences Steering Committee, 2017.

[45] Kenneth Alan De Jong. *Analysis of the behavior of a class of genetic adaptive systems*. PhD thesis, University of Michigan, Ann Arbor, Ann Arbor, MI, USA, 1975.

[46] Pablo de Oliveira Castro, Yuriy Kashnikov, Chadi Akel, Mihail Popov, and William Jalby. Fine-grained benchmark subsetting for system selection. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, page 132. ACM, 2014.

[47] Jeffrey Dean and Luiz André Barroso. The tail at scale. *Commun. ACM*, 56(2):74–80, February 2013.

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

[49] Peter Desnoyers. Empirical evaluation of nand flash memory performance. In *HotStorage '09: Proceedings of the 1st Workshop on Hot Topics in Storage*. ACM, 2009.

[50] Y. Diao, J. L. Hellerstein, A. J. Storm, M. Surendra, S. Lightstone, S. Parekh, and C. Garcia-Arellano. Using MIMO linear control for load balancing in computing systems. In *2004 American Control Conferences*, 2004.

[51] Marco Dorigo and Mauro Birattari. Ant colony optimization. In *Encyclopedia of machine learning*, pages 36–39. Springer, 2010.

[52] Marco Dorigo, Mauro Birattari, and Thomas Stützle. Ant colony optimization. *Computational Intelligence Magazine, IEEE*, 1(4):28–39, 2006.

[53] Fred Douglis, Deepti Bhardwaj, Hangwei Qian, and Philip Shilane. Content-aware load balancing for distributed backup. In *Large Installation System Administration Conference (LISA)*, 2011.

[54] Songyun Duan, Vamsidhar Thummala, and Shivnath Babu. Tuning database configuration parameters with ituned. *Proc. VLDB Endow.*, 2(1):1246–1257, August 2009.

[55] Katharina Eggensperger, Matthias Feurer, Frank Hutter, James Bergstra, Jasper Snoek, Holger Hoos, and Kevin Leyton-Brown. Towards an empirical foundation for assessing Bayesian optimization of hyperparameters. In *NIPS workshop on Bayesian Optimization in Theory and Practice*, volume 10, 2013.

[56] A.E. Eiben and C.A. Schippers. On evolutionary exploration and exploitation. *Fundam. Inf.*, 35(1-4):35–50, January 1998.

[57] Nosayba El-Sayed, Ioan A. Stefanovici, George Amvrosiadis, Andy A. Hwang, and Bianca Schroeder. Temperature management in data centers: Why some (might) like it hot. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS'12, pages 163–174, New York, NY, USA, 2012. ACM.

[58] Pablo A. Estévez, Michel Tesmer, Claudio A. Perez, and Jacek M. Zurada. Normalized mutual information feature selection. *IEEE Transactions on Neural Networks*, 20(2):189–201, 2009.

[59] Ext4. `http://ext4.wiki.kernel.org/`.

[60] Ext4 documentation. `https://www.kernel.org/doc/Documentation/filesystems/ext4.txt`.

[61] Linux/fs/ext4/ialloc.c. `http://lxr.free-electrons.com/source/fs/ext4/ialloc.c`.

[62] Filebench, 2016. `https://github.com/filebench/filebench/wiki`.

[63] Terry L Friesz, Hsun-Jung Cho, Nihal J Mehta, Roger L Tobin, and G Anandalingam. A simulated annealing approach to the network design problem with variational inequality constraints. *Transportation Science*, 26(1):18–26, 1992.

[64] RA Gallego, AB Alves, A Monticelli, and R Romero. Parallel simulated annealing applied to long term transmission network expansion planning. *Power Systems, IEEE Transactions on*, 12(1):181–188, 1997.

[65] Shravan Gaonkar, Kimberly Keeton, Arif Merchant, and William H. Sanders. Designing dependable storage solutions for shared application environments. *IEEE Trans. Dependable Secur. Comput.*, 7(4):366–380, October 2010.

[66] Michel Gendreau and Jean-Yves Potvin. *Handbook of metaheuristics*, volume 2. Springer, 2010.

[67] S. Ghemawat, H. Gobioff, and S. T. Leung. The Google file system. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)*, pages 29–43, Bolton Landing, NY, October 2003. ACM SIGOPS.

[68] F. Glover. Tabu Search – Part II. *ORSA Journal on Computing*, 2:4–32, 1990.

[69] Fred Glover. Tabu search: A tutorial. *Interfaces*, 20(4):74–94, 1990.

[70] Fred Glover and Manuel Laguna. *Tabu Search*. Springer, 2013.

[71] David E Goldberg and Robert Lingle. Alleles, loci, and the traveling salesman problem. In *Proceedings of the first international conference on genetic algorithms and their applications*, pages 154–159. Lawrence Erlbaum Associates, Publishers, 1985.

[72] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[73] John Grefenstette, Rajeev Gopal, Brian Rosmaita, and Dirk Van Gucht. Genetic algorithms for the traveling salesman problem. In *Proceedings of the first International Conference on Genetic Algorithms and their Applications*, pages 160–168. Lawrence Erlbaum, New Jersey (160-168), 1985.

[74] Lov K Grover. A new simulated annealing algorithm for standard cell placement. In *Proceedings of the International Conference on Computer-Aided Design*, pages 378–380, 1986.

[75] Ajay Gulati, Chethan Kumar, and Irfan Ahmad. Storage workload characterization and consolidation in virtualized environments. In *Workshop on Virtualization Performance: Analysis, Characterization, and Tools (VPACT)*, page 4, 2009.

[76] Isabelle Guyon and André Elisseeff. An introduction to variable and feature selection. *Journal of machine learning research*, 3(Mar):1157–1182, 2003.

[77] Mingzhe Hao, Gokul Soundararajan, Deepak Kenchammana-Hosekote, Andrew A Chien, and Haryadi S Gunawi. The tail at store: a revelation from millions of hours of disk and ssd deployments. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*, pages 263–276, 2016.

[78] Md E. Haque, Yong hun Eom, Yuxiong He, Sameh Elnikety, Ricardo Bianchini, and Kathryn S. McKinley. Few-to-many: Incremental parallelism for reducing tail latency in interactive services. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS'15, pages 161–175, New York, NY, USA, 2015. ACM.

[79] Georges R Harik and Fernando G Lobo. A parameter-less genetic algorithm. In *GECCO*, volume 99, pages 258–267, 1999.

[80] Jun He, Duy Nguyen, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Reducing file system tail latencies with Chopper. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies*, FAST'15, pages 119–133, Berkeley, CA, USA, 2015. USENIX Association.

[81] Weiping He and David H.C. Du. Smart: An approach to shingled magnetic recording translation. In *Proceedings of the 15th USENIX Conference on File and Storage Technologies (FAST)*, pages 121–134, Santa Clara, CA, February-March 2017. USENIX Association.

[82] J. L. Hellerstein, Y. Diao, S. Parekh, and D. M. Tibury. *Feedback Control of Computing Systems*. Wiley-IEEE Press, 2004.

[83] Jerry L Hintze and Ray D Nelson. Violin plots: a box plot-density trace synergism. *The American Statistician*, 52(2):181–184, 1998.

[84] J. H. Holland. *Adaptation in natural and artificial systems: An introductory analysis with applications to biology, control, and artificial intelligence.* U. Michigan Press, 1975.

[85] Aapo Hyvärinen and Erkki Oja. Independent component analysis: algorithms and applications. *Neural networks*, 13(4-5):411–430, 2000.

[86] Ronald L Iman, Jon C Helton, James E Campbell, et al. An approach to sensitivity analysis of computer models, part 1. introduction, input variable selection and preliminary variable assessment. *Journal of quality technology*, 13(3):174–183, 1981.

[87] Alfred Inselberg and Bernard Dimsdale. Parallel coordinates. In *Human-Machine Interactive Systems*, pages 199–233. Springer, 1991.

[88] Myeongjae Jeon, Saehoon Kim, Seung-won Hwang, Yuxiong He, Sameh Elnikety, Alan L. Cox, and Scott Rixner. Predictive parallelization: Taming tail latencies in web search. In *Proceedings of the 37th International ACM SIGIR Conference on Research & Development in Information Retrieval*, SIGIR'14, pages 253–262, New York, NY, USA, 2014. ACM.

[89] Young-Jae Jeon, Jae-Chul Kim, Jin-O Kim, Joong-Rin Shin, and Kwang Y Lee. An efficient simulated annealing algorithm for network reconfiguration in large-scale distribution systems. *Power Delivery, IEEE Transactions on*, 17(4):1070–1078, 2002.

[90] N. Joukov, A. Traeger, R. Iyer, C. P. Wright, and E. Zadok. Operating system profiling via latency analysis. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI 2006)*, pages 89–102, Seattle, WA, November 2006. ACM SIGOPS.

[91] Karthik Kambatla, Giorgos Kollias, Vipin Kumar, and Ananth Grama. Trends in big data analytics. *Journal of Parallel and Distributed Computing*, 74(7):2561–2573, 2014. Special Issue on Perspectives on Parallel and Distributed Processing.

[92] A Kapsalis, Vic J Rayward-Smith, and George D Smith. Solving the graphical steiner tree problem using genetic algorithms. *Journal of the Operational Research Society*, pages 397–406, 1993.

[93] M. Karlsson, C. Karamanolis, and X. Zhu. Triage: Performance differentiation for storage systems using adaptive control. *ACM Trans. Storage*, 1(4), 2005.

[94] Jalil Kazemitabar, Arash Amini, Adam Bloniarz, and Ameet S Talwalkar. Variable importance using decision trees. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems 30*, pages 426–435. Curran Associates, Inc., 2017.

[95] K. Keeton, C. Santos, D. Beyer, J. Chase, and J. Wilkes. Designing for disasters. In *Proceedings of the Third USENIX Conference on File and Storage Technologies (FAST 2004)*, pages 59–72, San Francisco, CA, March/April 2004.

[96] Kimberly Keeton, Dirk Beyer, Ernesto Brau, Arif Merchant, Cipriano Santos, and Alex Zhang. On the road to recovery: Restoring data after disasters. In *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006*, EuroSys'06, pages 235–248, New York, NY, USA, 2006. ACM.

[97] James Kennedy. Particle swarm optimization. In *Encyclopedia of Machine Learning*, pages 760–766. Springer, 2010.

[98] James Kennedy and Russell C. Eberhart. Particle swarm optimization. In *Proceedings of the IEEE International Conference on Neural Networks*, pages 1942–1948, 1995.

[99] H. Kim, S. Seshadri, C. L. Dickey, and L. Chiu. Evaluating phase change memory for enterprise storage systems: A study of caching and tiering approaches. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies*, pages 33–45, Berkeley, CA, 2014. USENIX.

[100] Seyoung Kim and Eric P Xing. Tree-guided group lasso for multi-task regression with structured sparsity. In *ICML*, pages 543–550, 2010.

[101] Scott Kirkpatrick, C Daniel Gelatt, and Mario P Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, 1983.

[102] Ricardo Koller and Raju Rangaswami. I/O deduplication: Utilizing content similarity to improve I/O performance. *Trans. Storage*, 6(3):13:1–13:26, September 2010.

[103] Ryusuke Konishi, Yoshiji Amagai, Koji Sato, Hisashi Hifumi, Seiji Kihara, and Satoshi Moriai. The Linux implementation of a log-structured file system. *ACM SIGOPS Operating Systems Review*, 40(3):102–107, 2006.

[104] Robert Kosara, Fabian Bendix, and Helwig Hauser. Parallel sets: Interactive exploration and visual analysis of categorical data. *IEEE Transactions on Visualization & Computer Graphics*, 12(4):558–568, 2006.

[105] Tim Kraska, Alex Beutel, Ed H. Chi, Jeffrey Dean, and Neoklis Polyzotis. The case for learned index structures. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, pages 489–504, 2018.

[106] Natalio Krasnogor and Jim Smith. A tutorial for competent memetic algorithms: model, taxonomy, and design issues. *Evolutionary Computation, IEEE Transactions on*, 9(5):474–488, 2005.

[107] J. B. Kruskal. Multidimensional scaling by optimizing goodness of fit to a nonmetric hypothesis. *Psychometrika*, 29(1):1–27, 1964.

[108] Pedro Larrañaga, Cindy M. H. Kuijpers, Roberto H. Murga, Inaki Inza, and Sejla Dizdarevic. Genetic algorithms for the travelling salesman problem: A review of representations and operators. *Artificial Intelligence Review*, 13(2):129–170, 1999.

[109] Latin hypercube sampling. `https://en.wikipedia.org/wiki/Latin_hypercube_sampling`.

[110] Changman Lee, Dongho Sim, Jooyoung Hwang, and Sangyeun Cho. F2FS: A new file system for flash storage. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST)*, pages 273–286, Santa Clara, CA, February 2015. USENIX Association.

[111] Ernest Bruce Lee and Lawrence Markus. Foundations of optimal control theory. Technical report, DTIC Document, 1967.

[112] H. D. Lee, Y. J. Nam, K. J. Jung, S. G. Jung, and C. Park. Regulating I/O performance of shared storage with a control theoretical approach. In *NASA/IEEE Conference on Mass Storage Systems and Technologies (MSST)*. IEEE Society Press, 2004.

[113] Howard Levene. Robust tests for equality of variances. *Contributions to probability and statistics. Essays in honor of Harold Hotelling*, pages 279–292, 1961.

[114] Cheng Li, Philip Shilane, Fred Douglis, Darren Sawyer, and Hyong Shim. Assert(!defined(sequential i/o)). In *6th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 14)*, Philadelphia, PA, 2014. USENIX Association.

[115] Jialin Li, Naveen Kr. Sharma, Dan R. K. Ports, and Steven D. Gribble. Tales of the tail: Hardware, os, and application-level sources of tail latency. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC'14, pages 9:1–9:14, New York, NY, USA, 2014. ACM.

[116] Jundong Li, Kewei Cheng, Suhang Wang, Fred Morstatter, Robert P Trevino, Jiliang Tang, and Huan Liu. Feature selection: A data perspective. *ACM Computing Surveys (CSUR)*, 50(6):94, 2017.

[117] Yan Li, Kenneth Chang, Oceane Bel, Ethan L. Miller, and Darrell D. E. Long. Capes: Unsupervised system performance tuning using neural network-based deep reinforcement learning. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '17, 2017.

[118] Yin Li, Hao Wang, Xuebin Zhang, Ning Zheng, Shafa Dahandeh, and Tong Zhang. Facilitating magnetic recording technology scaling for data center hard disk drives through filesystem-level transparent local erasure coding. In *Proceedings of the 15th USENIX Conference on File and Storage Technologies (FAST)*, pages 135–148, Santa Clara, CA, February-March 2017. USENIX Association.

[119] Yuxi Li. Deep reinforcement learning: An overview. *arXiv preprint arXiv:1701.07274*, 2017.

[120] Z. Li, M. Chen, A. Mukker, and E. Zadok. On the trade-offs among performance, energy, and endurance in a versatile hybrid drive. *ACM Transactions on Storage (TOS)*, 11(3), July 2015.

[121] Z. Li, K. M. Greenan, A. W. Leung, and E. Zadok. Power consumption in enterprise-scale backup storage systems. In *Proceedings of the Tenth USENIX Conference on File and Storage Technologies (FAST '12)*, San Jose, CA, February 2012. USENIX Association.

[122] Z. Li, R. Grosu, K. Muppalla, S. A. Smolka, S. D. Stoller, and E. Zadok. Model discovery for energy-aware computing systems: An experimental evaluation. In *Proceedings of the 1st Workshop on Energy Consumption and Reliability of Storage Systems (ERSS'11)*, Orlando, FL, July 2011.

[123] Z. Li, A. Mukker, and E. Zadok. On the importance of evaluating storage systems' $costs. In *Proceedings of the 6th USENIX Conference on Hot Topics in Storage and File Systems*, HotStorage'14, 2014.

[124] Zhao Lucis Li, Chieh-Jan Mike Liang, Wenjia He, Lianjie Zhu, Wenjun Dai, Jin Jiang, and Guangzhong Sun. Metis: robustly optimizing tail latencies of cloud systems. In *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference*, pages 981–992. USENIX Association, 2018.

[125] Chieh-Jan Mike Liang, Jie Liu, Liqian Luo, Andreas Terzis, and Feng Zhao. RACNet: A high-fidelity data center sensing network. In *Proceedings of the 7th ACM Conference on Embedded Networked Sensor Systems*, SenSys'09, pages 15–28, New York, NY, USA, 2009. ACM.

[126] Jens Lienig and James P Cohoon. Genetic algorithms applied to the physical design of vlsi circuits: A survey. In *Parallel Problem Solving from Nature—PPSN IV*, pages 839–848. Springer, 1996.

[127] Fernando G Lobo and David E Goldberg. The parameter-less genetic algorithm in practice. *Information Sciences*, 167(1):217–232, 2004.

[128] Christoffer Loffler, Christopher Mutschler, and Michael Philippsen. Evolutionary algorithms that use runtime migration of detector processes to reduce latency in event-based systems. In *Adaptive Hardware and Systems (AHS), 2013 NASA/ESA Conference on*, pages 31–38. IEEE, 2013.

[129] Yijuan Lu, Ira Cohen, Xiang Sean Zhou, and Qi Tian. Feature selection using principal feature analysis. In *Proceedings of the 15th ACM international conference on Multimedia*, pages 301–304. ACM, 2007.

[130] Lin Ma, Dana Van Aken, Ahmed Hefny, Gustavo Mezerhane, Andrew Pavlo, and Geoffrey J. Gordon. Query-based workload forecasting for self-driving database management systems. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, pages 631–645, New York, NY, USA, 2018. ACM.

[131] Pradipta De Vijay Mann and Umang Mittaly. Handling OS jitter on multicore multithreaded systems. In *Parallel & Distributed Processing Symposium (IPDPS), 2009 IEEE International*, IPDPS'09, pages 1–12. IEEE, 2009.

[132] Olivier Martin, Steve W Otto, and Edward W Felten. Large-step markov chains for the tsp incorporating local search heuristics. *Operations Research Letters*, 11(4):219–224, 1992.

[133] Pinaki Mazumder and Elizabeth M. Rudnick, editors. *Genetic Algorithms for VLSI Design, Layout &Amp; Test Automation*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1999.

[134] P. McDaniel, K. Butler, S. Mclaughlin, R. Sion, E. Zadok, and M. Winslett. Towards a secure and efficient system for end-to-end provenance. In *Proceedings of the second USENIX workshop on the Theory and Practice of Provenance (TAPP '10)*, San Jose, CA, February 2010. USENIX Association.

[135] M. D. McKay, R. J. Beckman, and W. J. Conover. A comparison of three methods for selecting values of input variables in the analysis of output from a computer code. *Technometrics*, 21(2):239–245, 1979.

[136] M. K. McKusick, W. N. Joy, S. J. Leffler, and R. S. Fabry. A fast file system for UNIX. *ACM Transactions on Computer Systems*, 2(3):181–197, August 1984.

[137] Peter Merz. Memetic algorithms for combinatorial optimization problems: Fitness landscapes and effective search strategies, 2001.

[138] Justin Meza, Qiang Wu, Sanjev Kumar, and Onur Mutlu. A large-scale study of flash memory failures in the field. In *Proceedings of the 2015 ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS 2015)*, pages 177–190, Portland, OR, June 2015. ACM.

[139] Sun Microsystems. Lustre file system: High-performance storage architecture and scalable cluster file system white paper. `www.sun.com/servers/hpc/docs/lustrefilesystem_wp.pdun.com/servers/hpc/docs/lustrefilesystem_wp.pdf`, December 2007.

[140] Sebastian Mika, Gunnar Ratsch, Jason Weston, Bernhard Scholkopf, and Klaus-Robert Mullers. Fisher discriminant analysis with kernels. In *Neural networks for signal processing IX, 1999. Proceedings of the 1999 IEEE signal processing society workshop.*, pages 41–48. IEEE, 1999.

[141] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[142] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015.

[143] Douglas C Montgomery. *Design and analysis of experiments*. John Wiley & Sons, 2017.

[144] Alessandro Morari, Roberto Gioiosa, Robert W Wisniewski, Francisco J Cazorla, and Mateo Valero. A quantitative analysis of OS noise. In *Parallel & Distributed Processing Symposium (IPDPS), 2011 IEEE International*, IPDPS'11, pages 852–863. IEEE, 2011.

[145] Heinz Muhlenbein. Evolution in time and space-the parallel genetic algorithm. In *Foundations of genetic algorithms*. Citeseer, 1991.

[146] Kevin P Murphy. *Machine learning: a probabilistic perspective*. MIT press, 2012.

[147] D. Narayanan, A. Donnelly, E. Thereska, S. Elnikety, and A. Rowstron. Everest: Scaling down peak loads through i/o off-loading. In *OSDI*, 2008.

[148] Iyswarya Narayanan, Di Wang, Myeongjae Jeon, Bikash Sharma, Laura Caulfield, Anand Sivasubramaniam, Ben Cutler, Jie Liu, Badriddine Khessib, and Kushagra Vaid. SSD failures in datacenters: What? when? and why? In *Proceedings of the Second ACM Israeli Experimental Systems Conference (SYSTOR '16)*, pages 7:1–7:11, Haifa, Israel, May 2016. ACM.

[149] E. B. Nightingale, K. Veeraraghavan, P. M. Chen, and J. Flinn. Rethink the sync. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI 2006)*, pages 1–14, Seattle, WA, November 2006. ACM SIGOPS.

[150] Qais Noorshams, Samuel Kounev, and Ralf Reussner. Experimental evaluation of the performance-influencing factors of virtualized storage systems. In *European Workshop on Performance Engineering*, pages 63–79. Springer, 2012.

[151] OpenStack Swift. `http://docs.openstack.org/developer/swift/`.

[152] Nohhyun Park, Weijun Xiao, Kyubaik Choi, and David J Lilja. A statistical evaluation of the impact of parameter selection on storage system benchmarks. In *Proceedings of the 7th IEEE International Workshop on Storage Network Architecture and Parallel I/Os (SNAPI)*, volume 6, 2011.

[153] Christian S. Perone. Pyevolve: A python open-source framework for genetic algorithms. *SIGEVOlution*, 4(1):12–20, November 2009.

[154] Pankaj Pipada, Achintya Kundu, K. Gopinath, Chiranjib Bhattacharyya, Sai Susarla, and P.C. Nagesh. Loadiq: Learning to identify workload phases from a live storage trace. In *Proceedings of the 4th USENIX Workshop on Hot Topics in Storage and File Systems*, HotStorage'12, Berkeley, CA, USA, 2012. USENIX Association.

[155] Robin L Plackett and J Peter Burman. The design of optimum multifactorial experiments. *Biometrika*, pages 305–325, 1946.

[156] Esteban Real, Sherry Moore, Andrew Selle, Saurabh Saxena, Yutaka Leon Suematsu, Quoc Le, and Alex Kurakin. Large-scale evolution of image classifiers. *arXiv preprint arXiv:1703.01041*, 2017.

[157] Jackie Rees and Gary J Koehler. An investigation of ga performance results for different cardinality alphabets. In *Evolutionary Algorithms*, pages 191–206. Springer, 1999.

[158] H. Reiser. ReiserFS v.3 whitepaper. `http://web.archive.org/web/20031015041320/http://namesys.com/`.

[159] Bernd Reisleben and Peter Merz. A genetic local search algorithm for solving symmetric and asymmetric traveling salesman problems. In *Evolutionary Computation, 1996., Proceedings of IEEE International Conference on*, pages 616–621. IEEE, 1996.

[160] Alma Riska and Erik Riedel. Disk drive level workload characterization. In *USENIX Annual Technical Conference*, volume 2006, pages 97–102, 2006.

[161] Ohad Rodeh, Josef Bacik, and Chris Mason. BTRFS: The Linux B-tree filesystem. *Trans. Storage*, 9(3):9:1–9:32, August 2013.

[162] Richard P Runyon, Kay A Coleman, and David J Pittenger. *Fundamentals of behavioral statistics* . McGraw-Hill, 2000.

[163] Anooshiravan Saboori, Guofei Jiang, and Haifeng Chen. Autotuning configurations in distributed systems for performance improvements using evolutionary strategies. In *Proceedings of the 2008 The 28th International Conference on Distributed Computing Systems*, ICDCS '08, pages 769–776, Washington, DC, USA, 2008. IEEE Computer Society.

[164] Martín Safe, Jessica Carballido, Ignacio Ponzoni, and Nélida Brignole. On stopping criteria for genetic algorithms. In *Advances in Artificial Intelligence–SBIA 2004*, pages 405–413. Springer, 2004.

[165] Sadiq M Sait, Mahmood R Minhas, Junhaid Khan, et al. Performance and low power driven vlsi standard cell placement using tabu search. In *Evolutionary Computation, 2002. CEC'02. Proceedings of the 2002 Congress on*, volume 1, pages 372–377. IEEE, 2002.

[166] Ricardo Santana, Raju Rangaswami, Vasily Tarasov, and Dean Hildebrand. A fast and slippery slope for file systems. In *Proceedings of the 3rd Workshop on Interactions of NVM/FLASH with Operating Systems and Workloads*, INFLOW '15, pages 5:1–5:8, New York, NY, USA, 2015. ACM.

[167] F. Schmuck and R. Haskin. GPFS: A shared-disk file system for large computing clusters. In *Proceedings of the First USENIX Conference on File and Storage Technologies (FAST '02)*, pages 231–244, Monterey, CA, January 2002. USENIX Association.

[168] Bianca Schroeder, Raghav Lagisetty, and Arif Merchant. Flash reliability in production: The expected and the unexpected. In *Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST)*, pages 67–80, Santa Clara, CA, February 2016. USENIX Association.

[169] Carl Sechen. *VLSI placement and global routing using simulated annealing*, volume 54. Springer Science & Business Media, 2012.

[170] P. Sehgal, V. Tarasov, and E. Zadok. Evaluating performance and energy in file system server workloads. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, pages 253–266, San Jose, CA, February 2010. USENIX Association.

[171] Bumjoon Seo, Sooyong Kang, Jongmoo Choi, Jaehyuk Cha, Youjip Won, and Sungroh Yoon. Io workload characterization revisited: A data-mining approach. *IEEE Transactions on Computers*, 63(12):3026–3038, 2014.

[172] Burr Settles. *Active Learning*. Morgan & Claypool Publishers, 2012.

[173] SGI. XFS filesystem structure. `http://oss.sgi.com/projects/xfs/papers/xfs_filesystem_structure.pdf`.

[174] Bobak Shahriari, Kevin Swersky, Ziyu Wang, Ryan P Adams, and Nando de Freitas. Taking the human out of the loop: A review of Bayesian optimization. *Proceedings of the IEEE*, 104(1):148–175, 2016.

[175] Shai Shalev-Shwartz. Online learning and online convex optimization. *Foundations and Trends in Machine Learning*, 4(2):107–194, 2011.

[176] Kai Shen, Ming Zhong, and Chuanpeng Li. I/o system performance debugging using model-driven anomaly characterization. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, San Francisco, CA, December 2005. USENIX Association.

[177] Jonathon Shlens. A tutorial on principal component analysis. *arXiv preprint arXiv:1404.1100*, 2014.

[178] Scikit-Optimize. `https://scikit-optimize.github.io/`.

[179] R. Spillane, R. Sears, C. Yalamanchili, S. Gaikwad, M. Chinni, and E. Zadok. Story Book: An efficient extensible provenance framework. In *Proceedings of the first USENIX workshop on the Theory and Practice of Provenance (TAPP '09)*, San Francisco, CA, February 2009. USENIX Association.

[180] T Starkweather, S Mcdaniel, D Whitley, K Mathias, D Whitley, et al. A comparison of genetic sequencing operators. In *Proceedings of the fourth International Conference on Genetic Algorithms*, 1991.

[181] John D. Strunk, Eno Thereska, Christos Faloutsos, and Gregory R. Ganger. Using utility to provision storage systems. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies*, FAST'08, pages 313–328, Berkeley, CA, USA, 2008. USENIX Association.

[182] David G Sullivan, Margo I Seltzer, and Avi Pfeffer. *Using probabilistic reasoning to automate software tuning*, volume 32. ACM, 2004.

[183] Lalith Suresh, Marco Canini, Stefan Schmid, and Anja Feldmann. C3: Cutting tail latency in cloud data stores via adaptive replica selection. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation*, NSDI'15, pages 513–527, Berkeley, CA, USA, 2015. USENIX Association.

[184] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press Cambridge, 1998.

[185] A. Sweeney, D. Doucette, W. Hu, C. Anderson, M. Nishimoto, and G. Peck. Scalability in the XFS file system. In *Proceedings of the Annual USENIX Technical Conference*, pages 1–14, San Diego, CA, January 1996.

[186] *sync(8) - Linux manual page*. `https://linux.die.net/man/8/sync`.

[187] Gary KL Tam, Vivek Kothari, and Min Chen. An analysis of machine-and human-analytics in classification. *IEEE Transactions on Visualization and Computer Graphics*, 2017.

[188] V. Tarasov, S. Bhanage, E. Zadok, and M. Seltzer. Benchmarking file system benchmarking: It *is* rocket science. In *Proceedings of HotOS XIII:The 13th USENIX Workshop on Hot Topics in Operating Systems*, Napa, CA, May 2011.

[189] Vasily Tarasov, Zhen Cao, Ming Chen, and Erez Zadok. The dos and don'ts of file system benchmarking. *FreeBSD Journal*, January/February, 2016.

[190] Vasily Tarasov, Erez Zadok, and Spencer Shepler. Filebench: A flexible framework for file system benchmarking. *;login: The USENIX Magazine*, 41(1):6–12, March 2016.

[191] TensorFlow. `https://www.tensorflow.org/`.

[192] Olivier Thas. *Comparing distributions*. Springer, 2010.

[193] Robert Tibshirani. Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society. Series B (Methodological)*, pages 267–288, 1996.

[194] A. Traeger, I. Deras, and E. Zadok. DARC: Dynamic analysis of root causes of latency distributions. In *Proceedings of the 2008 International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS 2008)*, pages 277–288, Annapolis, MD, June 2008. ACM.

[195] Transaction Processing Performance Council. TPC benchmark DS (decision support). `http://www.tpc.org/tpcds/`, 2006.

[196] Stephen Tweedie. Ext3, journaling filesystem. In *Ottawa Linux Symposium*, July 2000. `http://olstrans.sourceforge.net/release/OLS2000-ext3/OLS2000-ext3.html`.

[197] Balajee Vamanan, Hamza Bin Sohail, Jahangir Hasan, and T. N. Vijaykumar. TimeTrader: Exploiting latency tail to save datacenter energy for online search. In *Proceedings of the 48th International Symposium on Microarchitecture*, MICRO'48, pages 585–597, New York, NY, USA, 2015. ACM.

[198] Dana Van Aken, Andrew Pavlo, Geoffrey J. Gordon, and Bohan Zhang. Automatic database management system tuning through large-scale machine learning. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, pages 1009–1024, 2017.

[199] Peter J Van Laarhoven and Emile H Aarts. *Simulated annealing: theory and applications*, volume 37. Springer Science & Business Media, 1987.

[200] Matej Črepinšek, Shih-Hsi Liu, and Marjan Mernik. Exploration and exploitation in evolutionary algorithms: A survey. *ACM Comput. Surv.*, 45(3):35:1–35:33, July 2013.

[201] Mengzhi Wang, Kinman Au, Anastassia Ailamaki, Anthony Brockwell, Christos Faloutsos, and Gregory R. Ganger. Storage device performance prediction with cart models. In *The IEEE Computer Society's 12th Annual International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems. (MASCOTS)*, pages 588–595, 2004.

[202] Mengzhi Wang, Kinman Au, Anastassia Ailamaki, Anthony Brockwell, Christos Faloutsos, and Gregory R. Ganger. Storage device performance prediction with cart models. In *Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '04/Performance '04, pages 412–413, New York, NY, USA, 2004. ACM.

[203] S. Weil, S. Brandt, E. Miller, D. Long, and C. Maltzahn. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI 2006)*, pages 307–320, Seattle, WA, November 2006. ACM SIGOPS.

[204] Bernard Lewis Welch. On the comparison of several mean values: an alternative approach. *Biometrika*, 38(3/4):330–336, 1951.

[205] Darrell Whitley, Keith Mathias, and Patrick Fitzhorn. Delta coding: An iterative search strategy for genetic algorithms. In *ICGA*, volume 91, pages 77–84. Citeseer, 1991.

[206] Wesley Willett, Jeffrey Heer, and Maneesh Agrawala. Scented widgets: Improving navigation cues with embedded visualizations. *IEEE Transactions on Visualization and Computer Graphics*, 13(6):1129–1136, 2007.

[207] DF Wong, Hon Wai Leong, and HW Liu. *Simulated annealing for VLSI design*, volume 42. Springer Science & Business Media, 2012.

[208] H-S Philip Wong, Simone Raoux, SangBum Kim, Jiale Liang, John P Reifenberg, Bipin Rajendran, Mehdi Asheghi, and Kenneth E Goodson. Phase change memory. *Proceedings of the IEEE*, 98(12):2201–2227, Dec 2010.

[209] Bowei Xi, Zhen Liu, Mukund Raghavachari, Cathy H. Xia, and Li Zhang. A smart hill-climbing algorithm for application server configuration. In *Proceedings of the 13th International Conference on World Wide Web*, WWW '04, pages 287–296, New York, NY, USA, 2004. ACM.

[210] Yunjing Xu, Zachary Musgrave, Brian Noble, and Michael Bailey. Bobtail: Avoiding long tails in the cloud. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, NSDI'13, pages 329–342, Berkeley, CA, USA, 2013. USENIX Association.

[211] Ji Xue, Feng Yan, A. Riska, and E. Smirni. Proactive management of systems via hybrid analytic techniques. In *Cloud and Autonomic Computing (ICCAC), 2015 International Conference on*, pages 137–148, Sept 2015.

[212] Ji Xue, Feng Yan, Alma Riska, and Evgenia Smirni. Storage workload isolation via tier warming: How models can help. In *11th International Conference on Autonomic Computing (ICAC 14)*, pages 1–11, Philadelphia, PA, June 2014. USENIX Association.

[213] Yang Yu, Hong Qian, and Yi-Qi Hu. Derivative-free optimization via classification. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*, AAAI'16, pages 2286–2292. AAAI Press, 2016.

[214] Ming Yuan and Yi Lin. Model selection and estimation in regression with grouped variables. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 68(1):49–67, 2006.

[215] Standard Score. `https://en.wikipedia.org/wiki/Standard_score`.

[216] Erez Zadok, Aashray Arora, Zhen Cao, Akhilesh Chaganti, Arvind Chaudhary, and Sonam Mandal. Parametric optimization of storage systems. In *HotStorage '15: Proceedings of the 7th USENIX Workshop on Hot Topics in Storage*, Santa Clara, CA, July 2015. USENIX, USENIX.