CONFIGURATION MODELING AND DIAGNOSIS IN DATA CENTERS

A Dissertation
Submitted to
the Temple University Graduate Board

In Partial Fulfillment
of the Requirements for the Degree of
DOCTOR OF PHILOSOPHY

---

by
Sanjeev Sondur
August 2020

ABSTRACT

The behavior of all cyber-systems in a data center or an enterprise system largely depends on their configuration which describes the resource allocations to achieve the desired goal under certain constraints. Poorly configured systems become a bottleneck for satisfying the desired goal and add to unnecessary overheads such as under-utilization, loss of functionality, poor performance, economic burden, energy consumption, etc. Ill-effects related to system misconfiguration are well documented with quantifiable metrics showing their impact on the economy, security incidents, service recovery time, loss of confidence, social impact, etc. However, configuration modeling and diagnosis of data center systems is challenging because of the complexities of subsystem interactions and the many (known and unknown) parameters that influence the behavior of the system. Further, a configuration is not a static object - but a dynamically evolving entity that requires changes (either automatically or manually) to address the evolving state of the system. We believe that a well-defined approach for configuration modeling is important as it paves a path to keep the systems functioning properly in spite of the dynamic changes to configurations.

Proper configuration of large systems is difficult because interdependencies between various configuration parameters and their impact on performance or other attributes of the system are generally poorly understood. Consequently, properly configuring a system or a subsystem/device within it is largely dependent on expert knowledge developed over time. In this work, we attempt to formalize some approaches to configuration management, particularly in the area of network devices and Cloud/Edge storage solutions. In particular, we address the following aspects in this study: (i) impact of resource allocation on the energy-performance trade-off, with a network topology as an example, (ii) prediction of performance of a complex IT system such as Cloud Storage Gateway (CSG) or an Edge Storage Infrastructure (ESI) under given conditions, (iii) development of a data-driven method to efficiently configure (allocate resources) a CSG/ESI to satisfy required QoS levels (e.g. performance) under given conditions (e.g. minimal costs), and (iv) a model to express configuration health as a quantifiable metric.

With increasing stress on data center networks and correspondingly increasing energy consumption, we propose a method to simultaneously configure routing and energy management related parameters to ensure that the network can both avoid congestion and maximize opportunities for putting network ports in lower power mode.

We also study the problem of choosing hardware and resource settings to minimize cost and achieve a given level of performance. Because of the complexity of the problem, we explored machine learning (ML) based techniques. For concreteness, we studied the problem in the context of configuring a Cloud Storage Gateway (CSG) that involves parameters such as core speed and number of CPU cores, memory size and bandwidth, IO size and bandwidth, storage data and metadata cache size, etc. It turns out that it is very difficult to obtain a reliable ML model for this, and instead our approach is to use a model for the opposite problem (predicting optimal cost or performance for a given configuration) along with meta-heuristic such as genetic algorithm or simulated annealing. We show that an intelligent grouping of configuration parameters based on expected relationships between parameters and relative importance of the groups substantially outperforms the standard meta-heuristic based exploration of the state space.

Our work in the configuration space revealed a dominant void. We noticed the absence of common vocabulary or quantifiable metric to clearly and unambiguously express the quality of the configuration. In our diagnosis work, we designed a model to define a simple, reproducible, and verifiable metric that allows users to express the quality of

device configuration as a health score. Our configuration diagnosis model expresses the strength (or weakness) of a configuration as a Health Index, a vector of dimensions like performance, availability, and security. This health index will help users/administrators to identify the weak configuration objects and take remedial actions to rectify the configurations.

Our work on *Configuration Modeling and Diagnosis* addresses an important topic in this vast chaotic space. Using industry-driven problems and empirical data, we bring in some meaning to this complex problem. Though our research and experiments involved specific devices (network topology, Cloud Gateway, Edge Storage, network routers, etc.) - we show that the proposed solution is generic and can be adequately applied to other domains. We hope that this work will encourage other communities to explore new configuration challenges in a rapidly changing IT landscape.

Dedicated to my family and my guru.

ACKNOWLEDGEMENTS

I am grateful to my committee for their guidance and support during this program. I distinctly remember the conversation in 2014 about ”1+1 $\neq$ 2“ with Dr.Shi. It took me the next few years to understand the true meaning behind the discussion (in context to computer science). His initial guidance encouraged me to enroll in this program. I found Dr.Kant as a mentor, teacher, and advisor - who had the tough job of chipping out the rough edges of industry arguments to bring out the academic (thinker) in me. Under his guidance, I can see myself come a long way from the days of SDDs (misspelled acronym: solid state devices) to SSDs (Single Shot Multi-Box Detector) and solidify my foundation in any fronts. For his patience and efforts, I hope that I will evolve to be worthy of calling myself his student. Dr.Slobodan enlightened me into the world of machine language and the simple concept of looking at data before anything else. I had several impromptu discussions with him with each discussion stirring my eagerness to learn more. On reading several of Dr.He's papers, it confirmed the complexity of the problem, and small piece of puzzle I'm trying to address in the vast configuration space.

After a long internal search, I found Dr.Gross at Oracle, and he has encouraged me by substantiating my work by relating it to the real world problems. Girisha Shankar from Cisco (India) has been my silent behind-the-scenes cheerleader, supplementing my thoughts with brain-storming discussions and help me bridge the academic and industrial worlds. Dr.Charles Wang encouraged me to pick a research topic related to industry problems. Under his guidance, my first survey paper on Software Defined Network for

Beginners[1] was well received on ResearchGate. Dr.Bo Ji gave me an appreciation into algorithms and I remember enthusiastically dashing into his office claiming that I could solve the set-coverage problem! I had a lot of cooperation and guidance from Dr.Zheng, Dr.Korsh and other faculty members and colleagues.

I had wonderful discussions, stimulating arguments and help from former lab members: Dr.Anis Alazzawe, Dr.Mrs.Madhurima Ray. Dr.Dusan Ramljak, Dr.Ibrahim El-Shekeil, and Dr. Amitangshu Pal. I wish godspeed to my fellow lab members - Joyanta Biswas, Ms.Tanaya Roy, and Ms.Pavana Pradeep. Their lively atmosphere cheered me to handle the program one-day-at-a-time and make steady progress.

Finally, I am grateful for my family for their patience, cold days and long nights of sacrifice, encouragement, and support to help fulfill my life's long-pending ambition.

[1] [See Sondur (2014)]

TABLE OF CONTENTS

ABSTRACT iii

ACKNOWLEDGEMENTS ii

LIST OF TABLES viii

LIST OF FIGURES ix

1 INTRODUCTION 1

1.1 Contributions 3

1.2 Outline 5

2 CONFIGURING DATA CENTER NETWORK FOR ENERGY EFFICIENT OPERATION 7

2.1 Introduction 7

2.2 Related Work 9

2.3 Energy Management Techniques 10

2.3.1 Energy Management Basics 10

2.3.2 Energy Management in Network Links 12

2.4 Enhanced Energy Model Framework 15

2.4.1 Implementation of Traffic Consolidation 17

2.5 Simulation Results 19

2.5.1 Experimental Setup 19

2.5.2 Experimental Results and Discussions 20

2.6 Conclusion 25

3 AUTOMATED CONFIGURATION OF CLOUD STORAGE GATEWAYS 27

3.1 Introduction 27

3.1.1 Motivation and Challenges 29

3.2 Current State of the Art 30

3.3 Cloud Storage Gateway (CSG) 32

3.3.1 Brief Overview of Object Store 33

3.3.2 Characterizing the Behavior of a CSG 35

3.3.3 Complexities in Configuration Control 37

3.4 Configuration Problem Formulation 39

3.4.1 Research Questions 41

3.5 Solution Approach 42

3.5.1 Feature Vector 43

3.5.2 Research Hypothesis 44

3.5.3 Classification Problem 45

3.6 Implementation Details 45

3.6.1 Test Environment 45

3.6.2 Workload Execution 46

3.6.3 Metrics Collected 47

3.6.4 Data pre-processing and Classification 48

3.7 Predicting Performance of a Configuration 50

3.8 Evaluation Results 50

3.8.1 Influence of Chosen Feature Sets on Performance 50

3.8.2 Confusion Matrix 51

3.8.3 Boundaries of Performance Prediction 53

3.8.4 Comparing on Prediction Accuracy 53

3.9 Conclusion 54

4 EFFICIENT CONFIGURATION OF COMPLEX CYBER-SYSTEMS 55

4.1 Introduction 55

4.2 State of Current Art 58

4.3 Configuration Management of Edge Storage (ESI) 60

4.3.1 Overview of Edge Storage Infrastructure 60

4.3.2 Data Flow in an ESI 61

4.3.3 Importance of Meta-Data 62

4.3.4 Configuration Modeling for ESI 62

4.4 Solution Design 65

4.4.1 Modified Genetic Algorithm (**mGA**) 67

4.4.2 Modified Simulated Annealing (**mSA**) 68

4.5 Evaluation Results 71

4.5.1 Implementation Details 71

4.5.2 Hyper-parameters of the Stochastic Process 72

4.5.3 Extracting Feature Importance 73

4.5.4 Recommending a Configuration using mGA 74

4.5.5 Recommending a Configuration using mSA 77

4.5.6 Pareto “like” Boundary 79

4.5.7 Comparing execution time 80

4.6 Conclusion 81

5 A CONFIGURATION HEALTH SCORING SYSTEM AND ITS APPLICATION TO NETWORK DEVICES 82

5.1 Introduction 82

5.2 Current State of Art and Challenges 83

5.3 Formulating a Configuration Health Index 85

5.4 Estimating the Health Index from a Configuration 87

5.4.1 Configuration Files and Objects 88

5.4.2 Object Representation 91

5.4.3 Quantifying the attributes 92

5.5 CHeSS Framework 92

5.5.1 Configuration File and its Canonical Form 93

5.5.2 Ontologies, OpenConfig and CHeSS 94

5.5.3 Learning Model 96

5.5.4 Enhancing the Reliability Of Weights 97

5.5.5 Visual Representation of Health Index (HI) 99

5.6 Evaluation Results 100

5.6.1 Implementation Details 101

5.6.2 Scalability of the Framework 102

5.6.3 Applicability and Feasibility of Framework 103

5.7 Conclusion 103

6 CONCLUSION **105**

6.1 Opportunities and Future Work 107

BIBLIOGRAPHY **109**

LIST OF TABLES

2.1 Simulation Configuration (parameters used in experiment) 22

3.1 Subset of Feature Vector Supporting the Problem. 43

3.2 Sample Workload Type and Applications. 47

3.3 Test Termination Condition based on Data Volume and Accuracy. 48

3.4 Sample Classification of Design Variables. 49

4.1 Sample Classification of Design Variables. 63

4.2 Grouping Design Variables 70

4.3 Very Fast Simulated Annealing Functions 72

5.1 Health Index of Sample Configuration Files. 102

LIST OF FIGURES

2.1 Network Energy as part of Data Center Energy Consumed 8

2.2 Device Reactive Voltage Management under Busy/Idle States 11

2.3 TRS packet transmit model over-laid with inter-device communication . . 14

2.4 A Fat-tree Network and NS3 Energy Model 15

2.5 Enhanced NS-3 Energy Management State Models 16

2.6 Power Consumption and Average Delay Compared to Baseline at 25% Utilization 20

2.7 Power Consumption and Average Delay Increase % in TOS and TRS . . 20

2.8 Power Saving and Increase in Average Delay under TRS (with and without switch backplane power management) 23

2.9 Comparison of power saving and average delay with LC and GC working together under TRS power management 23

2.10 Effect of upgrading the size of fat tree(k) on power saving and on average delay 24

3.1 Cloud Storage Gateway Architecture. 27

3.2 Object Storage Infrastructure 34

3.3 Cloud Storage Gateway - Test Environment. 46

3.4 Sample of Data Collected. 49

3.5 Accuracy of Various Predictions 52

3.6 Sample Confusion Matrix (performance prediction) 52

4.1 Workflow/ Data Path in Edge Storage Infrastructure. 61

4.2 Design of Algorithms. 65

4.3 Illustration of Research Questions. 66

4.4 PCA and Feature Importance. 73

4.5 gGA and mGA (PCA+Grouping) Test Results (Iterations and Cost) 75

4.6 Comparing gSA Test Results (Iterations and Cost) with/without Design Attr. Grouping 78

4.7 Comparing mSA Test Results (Iterations and Cost) with/without Design Attr. Grouping 79

4.8 Pareto-like Boundary and Execution Times 80

5.1 A Framework to express the Health Index of a Configuration File 86

5.2 Sample Configuration File with Associated attributes and weights 88

5.3 CHeSS Framework with Knowledge Repository and Learning Model . . 93

5.4 Parsing the Hierarchical Object and Assigning Weights (P,S,A) 95

5.5 Learning w_i from Samples and Clustering for a Given CO_i 98

5.6 3D representation of HI of Sample Configuration File 99

5.7 Sample of a Complex Hierarchical Object CO_i. 102

CHAPTER 1

INTRODUCTION

In an ever-expanding digital world, the behavior of an IT unit/device (i.e a storage unit, a compute server, a network device, an enterprise system, etc.) is of significant importance for multiple reasons, such as security, energy efficiency, user experience, reliability, economical operation, etc. Device behavior has an important influence on different user communities: (i) system administrators for efficient operation and ease of use, (ii) end-users for better experience, (iii) operators for economical reasons, and so on. However, *behavior* is an abstract word, that can take a different meaning depending on the context or user-community. For example, it could refer to high performance or least energy consumption or highly secure and so-forth. Further, behavior can be an attribute on the local device or globally across the full system (end-to-end service or a full topology). The behavior of an IT device or a system largely depends on its *configuration*, generally represented as an ASCII file (txt, json, xml, etc.) or stored in a repository (local or central). Configurations define resource allocation, transaction processing, security controls, etc. A good knowledge and understanding of the device configuration is important for the efficient operation of the device and the system as a whole. Negative effects of poor or misconfigured system are well documented as: unavailability [Newman (2017)], financial burden [Elliot (2017)], security breach [Chirgwin (2017)], etc. Yet, configuration modeling is rarely studied, though it impacts every part of the operation (and our daily life).

The concept of *configuration modeling* is itself subject to scrutiny, because of the complexities of subsystem interactions and the many ways in which performance metrics can be defined [Eckart et al. (2009)]. It is very difficult to characterize the influence of many tune-able parameters, that are set by the vendor or the administrator. These dependencies and complex behavior make analytical modeling difficult. Besides, most device configurations have to work under given constraints, such as costs, or power cap, etc. Throwing additional resources or allocating extra capability (compute, storage, bandwidth, etc.) will not overcome the system bottleneck or satisfy the required constraints.

After a device or a system is configured properly and resource allocated optimally, communicating the *quality* of the configuration is very difficult. Often, the configuration of the device or the system is communicated in *ambiguous* terms between user groups for various reasons (engineering design, troubleshooting, etc.). Expressing the device configuration using common vocabulary and with quantifiable metrics is essential to eliminate ambiguity and to help users communicate their needs. Our extensive analysis and literature search on this subject uncovered a huge gap and a need to design a quantifying metric that can help to express the configuration in easily understandable terms.

Our research focuses on *configuration modeling and diagnosis* in data centers and to understand the relationship between the configuration parameters and device behavior, to recommend an optimal configuration for the desired behavior. Since configurations are the key to a device, our work looks at designing a metric to quantify the configuration *health* of a device. We believe that a focused approach to configuration modeling will help in efficient allocation of resources and deriving desired behavior under given conditions. A metric to express the health of the configuration (both locally to a device and globally at the system level) could help users communicate in unambiguous terms and proactively resolve any potential issues.

The vast space of Configuration Modeling and Diagnosis in Data Centers (or in an enterprise system) is far-reaching to be explored in a limited time. Our work focused on a subset of problems on network and storage domains, that have a compelling need in the industry. With a focus on the above challenges and having worked with real-world industry data set, we present our work and findings and highlight the contributions below.

1.1 Contributions

Building on the well-established concept of power saving *c-states* in CPU cores, our first work focuses on *re*-configuring and resource allocation of the data center networks under low utilization to save energy. The problem addressed allocating 'restricted' resources (devices, paths, bandwidth, energy) and analyzing the performance vs. energy trade-off. Using our enhancements [Sondur et al. (2017)] to a well-accepted NS-3 network simulation tool, we modeled the network device energy and performance trade-off and studied the various parameters that influence device behavior under energy constraints. Our simulation-based study [Ray et al. (2018)] revealed that with suitable controls and resource configuration, energy conservation over 40% is possible under low utilization (below 25%).

To address the complex configuration problem, our work comprises configuration studies in emerging technologies like Cloud Storage Gateway (CSG) [Prahlad et al. (2012)]. This technology combines the power of the Edge Computing with the Cloud and involves characterizing (i) the local behavior of the system plus (ii) the multiple domains involved in the system (unpredictable network plus storage and compute). Edge Storage Infrastructure (ESI) is architecturally close to CSG, with constraints defined by the deployment nature of the Edge Controller. In addition to imposing stringent QoS needs of the new generation of edge applications, ESI is deployed in constraint conditions that place limits on power consumption, cooling needs, size, etc. Predicting the performance (i.e. throughput) of CSG/ESI under a given configuration is challenging because of the difficulties involved in describing the system. The various unknown interactions between different layers (network, storage cache, etc.) make modeling the multiple subsystems of CSG/ESI difficult. In the absence of a well-known model to characterize such a complex system, we used a data-driven model from a commercial CSG system executing industry given workloads. With this empirical data, we applied a machine learning model augmented with domain knowledge to predict the performance under given configuration and workload conditions [Sondur and Kant (2019)].

The converse of the above discussion is very hard since it involves an abstract question. "Is it possible to recommend a configuration (resource allocation) to satisfy a given condition?". This inevitably needs providing an *adequate set of parameters* (i.e. CPU, memory, bandwidth, disk area, etc.) that satisfy the user requirements (e.g. performance, heat dissipation, power limit, etc.). On this front, our research work includes recommending a configuration for a CSG/ESI to satisfy a given workload/ performance demand under given constraints. That is, given two parameters (workload, performance), our work was directed towards suggesting an optimal configuration (server, resource, etc.) that satisfies given constraints (cost, size, power, cooling). Our work addresses the main operational challenges of a data center for optimal utilization of resources, wherein ill-advised configurations are the main cause for several predicaments such as loss of brand name, customer experience, excessive operational costs, poor utilization, security issues, etc.

Our proposal for recommending an optimal configuration is based on meta-heuristics based stochastic optimization enriched with problem specific domain knowledge. Our approach involves adding domain knowledge to a meta-heuristic stochastic process such as Genetic Algorithm (GA) and Simulated Annealing (SA), and we show that such an approach provides much smoother evolution of the objective function and thus can be terminated much earlier than an uninformed GA or SA [Sondur et al. (2020)].

Having worked on the configuration issues on the above research, it was evident that there is no common vocabulary to express the quality of the device configuration. Describing a configuration in abstract terms as good, secure, reliable causes ambiguity among the user population and further complicates the diagnosis process. System designers and administrators need an 'a priori' qualitative metric to express the health of the device configuration. Our work proposed such a health index targeting multiple metrics including performance, security, and availability. We designed a generic framework to compute such a health index and specialized them for network devices such as routers and switches [Sondur et al. (2020)].

Our dissertation outlines the above work alongside current state of art, challenges involved, a proposed solution with empirical data, results, and conclusions. Though the solution presented is specific to a device or system (network router, Cloud Storage Gateway, Edge Storage), the proposed approach is generic and the methods are broad enough to be applicable to other domains for configuration or behavior prediction and system diagnosis. We discuss the adaption of the generic approach to other domains at the conclusion of this dissertation.

1.2 Outline

The outline of this document is as follows. Chapter 2 studies the configuration problem with respect to resource allocation and performance-energy trade-off for energy efficient operation of data center networks. Using two different configuration approaches, this work shows that the correct allocation of bandwidth, device energy controls can save energy consumption up to 40% under low utilization conditions (traffic less than 25%). Our work on Cloud Storage Gateway (CSG) configuration and performance prediction alongside a machine learning based prediction model is explained in chapter 3. Based on extensive testing with real world customer workloads, we show that it is possible to achieve excellent prediction accuracy (about 97%) while ensuring that the model does not suffer from under-fit or over-fit. In chapter 4, we study the challenges involved in recommending an *optimal* *configuration* for an Edge Storage Infrastructure (ESI), which is architecturally similar to CSG. Using empirical data from earlier work, we present two stochastic optimization approaches enriched with relevant domain knowledge to improve the quality of the solution. In the first case, we enhance Genetic Algorithm (GA) with principal component analysis to recommend near-optimal configurations for a CSG/ESI system. The modified Genetic Algorithm (mGA) computes a set of solutions (population of chromosomes) that satisfy user given performance/ workload under given constraints. In the second case, we present a modified Simulated Annealing Algorithm (mSA) by grouping domain attributes for faster convergence of a solution. We show that our modified algorithms converge faster compared to an uninformed generic algorithm. Section 5 presents CHeSS, a framework to quantify the health index of a configuration, using network router as a specific example. CHeSS expresses the device configuration as a quantify-able metric enabling the users to communicate the quality of a configuration in unambiguous terms. We conclude the work done in section 6 showing the general applicability of the above solutions to other domains.

CHAPTER 2

CONFIGURING DATA CENTER NETWORK FOR ENERGY EFFICIENT OPERATION

2.1 Introduction

With increasing stress on data center networks and correspondingly increasing energy consumption, we propose a method to simultaneously configure routing and energy management related parameters to ensure that the network can both avoid congestion and maximize opportunities for putting network ports in lower power mode. We explored the problem of efficient resource allocation (i.e. routing, energy management, etc.) for network nodes in data center networks with a goal to save energy. In this chapter, we addressed allocating "restricted" resources (devices, paths, bandwidth, etc.) and analyzing the performance vs. energy trade-off. A common idiom is: *Nothing comes for free*, meaning that to achieve a goal (e.g. conserve energy) another "related entity" has to be sacrificed or traded-off (e.g. performance). Using network device as an example, we study such a trade-off between performance and energy consumption. The generic concepts presented here are applicable to other domains like storage, or CPUs etc. We start by presenting the basic concepts of energy management in an electronic device and apply it to a network router as an example to demonstrate our work.

Energy consumption in large data centers continues to rise and is becoming a substantial part of their operational costs [Connor (2015); Bertoldi et al. (2017)]. Traditionally, the CPU has been the dominant power consumer in a server, and hence collectively in the data center. However, the CPU speed and energy efficiency has improved quite rapidly in last few decades – faster than networking and much faster than storage. Yet, data movement needs continue to go up rapidly. As a result the energy consumption of data movement relative to computation has risen rapidly and requires increasing attention in terms of energy efficiency at all levels from on-chip wires to on-chip interconnects to data center networks. In fact, in high performance computing (HPC) data centers, the energy consumption of the data center network can rival with that of the CPUs, and techniques to manage energy without adverse impact on performance become crucial [Murugan et al. (2012)].

The US Dept. of Energy reported that in 2016 data centers consumed 2% of the electricity generated in the USA and an average mid-size data center consuming about 360MW of electric power [Sondur et al. (2018)]. The latest energy consumption report [Masanet et al. (2020)] illustrates that it is really hard to pin down the network component numbers, because of the difficulty in associating the energy to relative components executing the workload. For example, it is becoming more difficult to separate entities such as a TCP packet processing or virus check. Should the related power consumption at the endpoint be part of the network or compute energy? Are IDS/IPS and DMZ virus checkers part of the network or not? If network only means NICs and switches/routers, then it will be very low and largely independent of data rates, and will even go down further if servers become fast

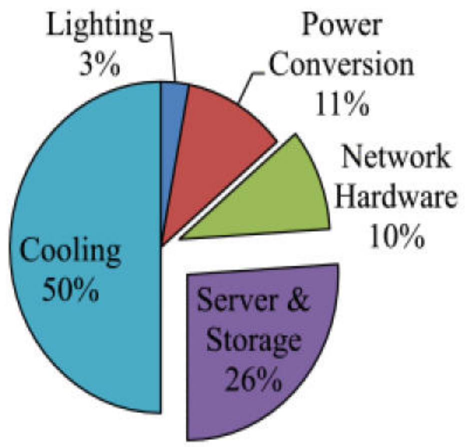


FIGURE 2.1: Network Energy as part of Data Center Energy Consumed

enough to do the soft switch/router functionality. Considering the dynamic nature of these entities and workloads, we can associate about 5 to 10% of the above data (Fig. 2.1) to network hardware. This work relates to conserving the network energy under low utilization conditions, there by saving energy costs related to network devices.

2.2 Related Work

The existing work on energy modeling in NS3 is quite limited. Hu [Wu et al. (2011)] presented the first framework for incorporating the energy model in NS3. Their study computes topology specific energy consumption via a framework shown in Fig. 2.4(b). It does not link the network device to the energy model for capturing the work done by the device, i.e. the actual work of packet transmission. But per packet transmission energy is not realistic for synchronous links. Mostowfi [Mostowfi et al. (2015)] presented a study of LPI enabled devices with Fast Wake-Deep Sleep states. This study requires simulation of device energy directly linked to the device packet transmission states. Our design presents an enhanced framework that links the modeling framework directly to the network device(s) to capture real metrics related to packet transmission. Tapparello [Tapparello et al. (2014)] used the same model from Hu to design an energy harvesting framework. Our model is built over these basic frameworks while enhancing the energy metrics captured and granularity of data collected. We believe that these granular data and enhanced energy metrics will be of interest for future research studies and aid in understanding the underlying mechanism of device energy consumption. Our enhancement can incorporate models for multi-state power management of communication links presented by Kant [Kant (2011)]. Kant showed energy management through link speed control, link width control or link power state control.

Any transition in and out of a sleep mode involves delay, which affects network performance. It is crucial to represent this aspect in the simulation model, but has not been typically addressed by the high level energy models. We do so via this *enhanced energy model* that keeps track of energy consumption, latency and average queue length for every transition.

2.3 Energy Management Techniques

2.3.1 Energy Management Basics

Energy consumption of electronic devices can be reduced either by doing idle or static power management or active power management. In idle power management, we take advantage of low power sleep mode to reduce the idle power. By active power management, we run a device slower than its maximum voltage and frequency (DVFS), but it hurts performance (because of lower frequency). Energy consumption of electronic devices can be reduced by taking advantage of idle periods and/or low device utilization. When a device is idle, it still consumes “idle power” resulting from leakage current in the transistors. The idle power can be reduced by placing the device in a sleep mode (low power mode), and the corresponding techniques are known as static power management. If the device is operating at a low utilization level, it is often possible to run it slower and still keep the utilization below the desired threshold. The slower speed may allow for operation at a lower voltage and thus saves energy. This is known as dynamic (or active) power management and the best known example of this is the DVFS (dynamic voltage frequency switching), whereby the frequency of operation and voltage can be changed suitably.

Let V denote the Source-Drain voltage and I_L denote the leakage current. Then the idle power consumption $P_{idle} = V \times I_L$. Suppose that the device operates at frequency f and the effective circuit capacitance is C. Then the dynamic (or active) power is given as: $P_{dynamic} = 1/2 \times C \times V^2 \times f \times U$ where U denotes the utilization level of the device. The total power is then simply $P_{idle} + P_{dynamic}$. It is seen that to the extent the voltage can also be reduced along with frequency reduction, it reduces both idle and dynamic power. A

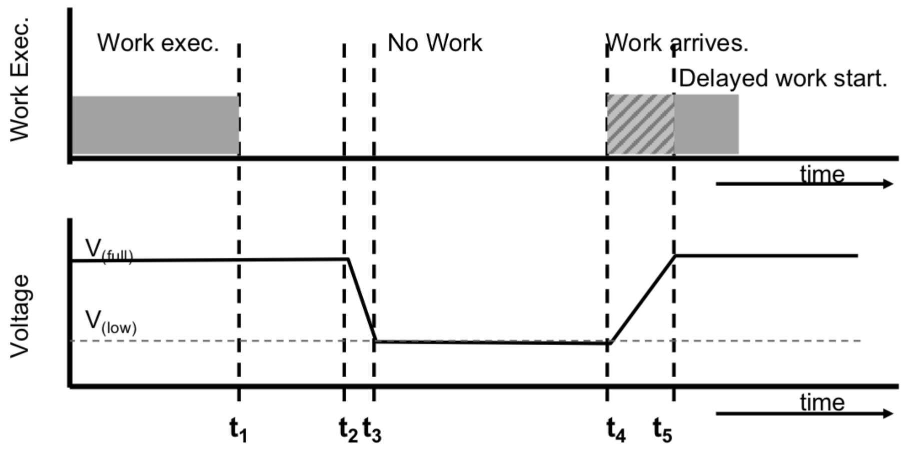


FIGURE 2.2: Device Reactive Voltage Management under Busy/Idle States

device could have multiple sleep states with different amounts of power consumption, the trade-off being between the power consumed and the latency of entering/exiting the sleep mode. The device can also have multiple active power states, each defined by a suitable frequency-voltage combination. The latency of state change could vary widely depending on the type of device.

Fig.2.2 illustrates the above concept with a simple diagram. Let's assume that the device is operating at full power (hence full voltage $V_{(full)}$ till time period t_1 in busy state doing some active work. After period t_1, there is no further work for an unknown duration (till the next work arrives t_4). We take advantage of the non-busy period to reduce the device voltage and conserve power. At the end of busy period, the device would wait for a predefined period called run-way time (t_1 to t_2), and if no further work arrives within this period, the device is put to a low-voltage state $V_{(low)}$. The transition from busy to idle state is not instantaneous, but takes some predetermined time duration t_2 to t_3. This transition period is a constant and depends on the internal device electronics. Energy is conserved when the device is under low power state, and the duration is referred as *sleep time* (t_3 to t_4). Longer the sleep time (i.e. long interval between two consecutive units of work) higher the saved energy. However, when the first unit of work arrives, the device has to wake up from idle state to full power state. This period is referred as *wake up time* and shown as t_4 to t_5, and this wake-up latency adds to the delay in work done. Note the delay in figure referring to the late start of the work. Latency measured as the delay introduced by the state-transition is experienced only by the first request that wakes up the device. Latency itself is not important, its impact on performance is. The performance impact on the work vs. the energy saved due to state-transition is an important trade-off, that needs to be carefully analyzed using the right metrics. We studied this reactive power control management in the network devices.

2.3.2 Energy Management in Network Links

We apply the above concept and discuss the power consumption characteristics of network links. The current high speed networking, irrespective of its ultimate personality (e.g., PCI-E, Ethernet, IBA, FC, etc.) is based on serial links with differential switching technology with multiple "lanes" used to increase the bandwidth. Such links are typically synchronous and continuously transmit some symbols. This characteristic both increases the contribution of network to energy consumption and offers better opportunities for energy management as the network technology increases in speed. The main reason for this is that the data center network links carry very little traffic most of the time, and high bandwidth is required only sporadically. For example, the power consumption of a 10 Gb/sec Ethernet can be anywhere between 2-10 times the power consumption of 1 Gb/sec Ethernet, depending on the number of ports and the technology used [Sohan et al. (2010)]. Also, the power consumption goes up with the number of ports whether they are used or not. Thus a network speed upgrade will usually result in increased energy consumption even though the average network traffic carried is unlikely to increase much. Thus, the increased energy consumption makes network energy management crucial. Fortunately, the decreased average utilization level also makes it easier to exploit the network low power modes.

The basic serial link PHY supports two sleep states (low power), called L0s and L1, respectively which can be used for idle power management. The L0s power state is unidirectional, in that the transmitter for each direction of the link can independently decide to go into sleep mode when it has nothing to transmit, whereas the receiver side remains active. The L1 power state involves a handshake between transmitter and receiver, and thus allows both of them to go into sleep state when there is nothing to transmit. The L1 state can reduce the idle power quite substantially because it stops the transmission of synchronization frames as well. However, the downside is that it requires re-synchronization and hence substantial delay. The L0s and L1 are supported by PCI-E and L0s could potentially be supported for any link at the PHY level in hardware since it does not affect link synchronization. IEEE has defined standards towards managing network devices [IEEE 802.3ba, 802.3bj, 802.3az et al] to achieve energy efficiency. The Low Power Idle (LPI) [EEE (2016)] is defined in 802.3az for the Energy Efficient Ethernet (EEE) initiative. LPI can be thought of as an improved and link level version of L1. In my research work implementation, We developed code to support essentially L0s and LPI like mechanisms, which can be customized further as needed. There are new emerging energy management standards for 40 & 100 Gb/s Ethernet links such as 802.3bj [Mostowfi et al. (2016)] (deep sleep mode) and 802.3bm (shallow sleep with fast wakeup). Our implementation can accommodate these standards by suitable change to parameters.

The IEEE standard 802.3az-2010 [Christensen et al. (2010)] defines the LPI mechanism as opposed to the continuous IDLE signal, in order to stop the transmission when there is no actual data to send and resume quickly upon arrival of new packets. LPI sends periodic refresh signal to maintain the synchronization, while consuming only about 10% of the active power. Wake up from LPI takes a significant exit latency, since the transmitter needs to wake up the receiver, before transmitting anything. The two relevant parameters in this regard are *sleep time* (T_s) and *wake-up time* (T_w). For a 10 Gb/s link, with 1500 bytes packet size T_s and T_w will be 2.88 μs and 4.48 μs respectively. This amounts to transmission time of several packets and thus the mechanism is useful when the traffic

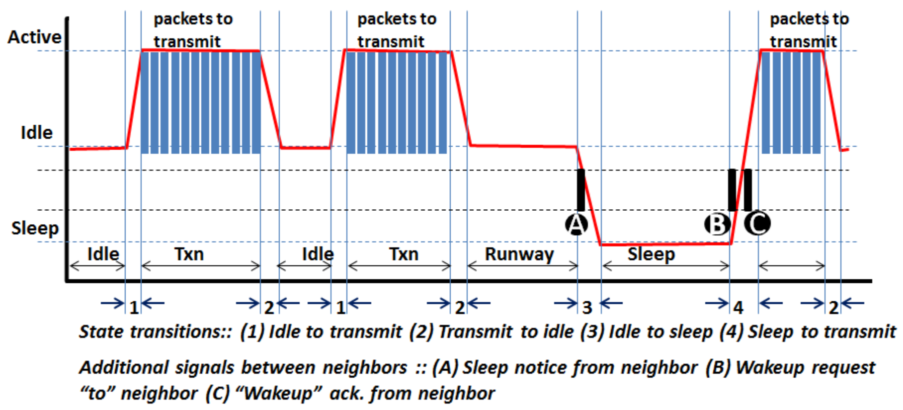
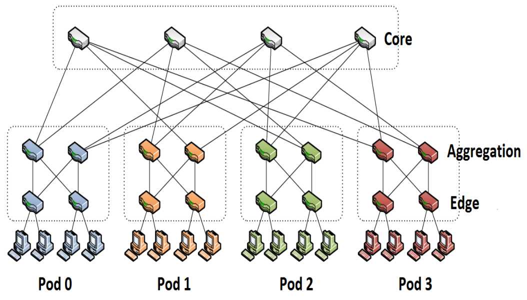

Active
Idle
Sleep
packets to transmit
packets to transmit
packets to transmit
Idle
Txn
Idle
Txn
Runway
Sleep
1
2
1
2
3
4
2
A
B
C
State transitions:: (1) Idle to transmit (2) Transmit to idle (3) Idle to sleep (4) Sleep to transmit
Additional signals between neighbors :: (A) Sleep notice from neighbor (B) Wakeup request "to" neighbor (C) "Wakeup" ack. from neighbor

FIGURE 2.3: TRS packet transmit model over-laid with inter-device communication

shows significant gaps between packet bursts. So, LPI can be useful only at very low utilization levels. For our research, we designed the above system to support the energy management techniques at fine to medium time granularity. For most of the applications, the energy management should not affect application behavior. However if there are some extremely latency sensitive applications present, then those should be isolated and the energy management can not be used for them.

As for the DVFS, most links have the capability to operate at a set of predetermined speeds, however, this capability is normally designed for configuration, rather than dynamic change. Thus the change is rather slow. Ethernet supports auto speed negotiation to drop down a link to lower standard speeds, but even a rapid speed change mechanism – known as RPS (Rapid PHY Selection) is too slow. Thus DVFS is not useful for outside-the-box links, but could useful for others such as core interconnect.

The TOS model is hardware driven and allows the transmitter to sleep independently when the gap between the traffic is small. The basic approach is to transition to L0s if the idle period exceeds some specified amount called "runway"[Kant (2011)]. The exit happens on arrival of the next packet. For simplicity, a fixed runway duration is used, although utilization dependent runway could be easily implemented [Kant (2011)]. The TRS model is software driven, birectional and it is an implemetation of LPI. In TRS, the transmitter sends sleep packet to the neighboring receiver before going to sleep. Both the



(a) An illustration of fat-tree network

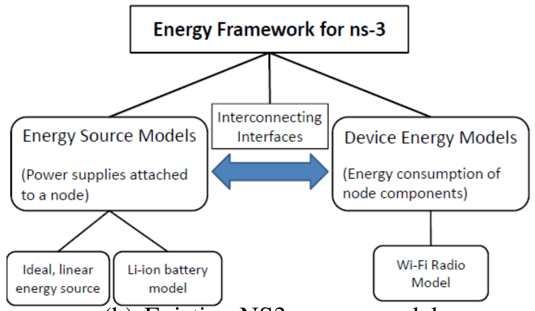

Energy Framework for ns-3
Energy Source Models
(Power supplies attached to a node)
Interconnecting Interfaces
Device Energy Models
(Energy consumption of node components)
Ideal, linear energy source
Li-ion battery model
Wi-Fi Radio Model
(b) Existing NS3 energy model

FIGURE 2.4: A Fat-tree Network and NS3 Energy Model

transmiter and receiver have to come to an agreement before the transmitter can move to sleep mode. So it has extra overhead. As the transmitter wakes up, it has to send wake up packet to neighboring receiver before sending any traffic intended for it, as shown in Fig. 2.3. So, it has a higher exit latency. Thus TRS uses larger runway to ensure the transmitter and receiver can only move into the sleep mode when the gap between traffic is large.

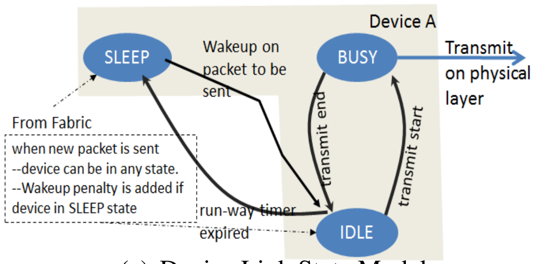
2.4 Enhanced Energy Model Framework

Existing network simulators are not designed to study the granular effects of energy consumption and workload performance or would be easy to extend. We found that none of them to be workable as the power models used and power management techniques implemented are very rudimentary and inadequate. We choose to extend NS3 [NS3doc (2015)] network simulator because of its flexible design and widespread familiarity in the community.

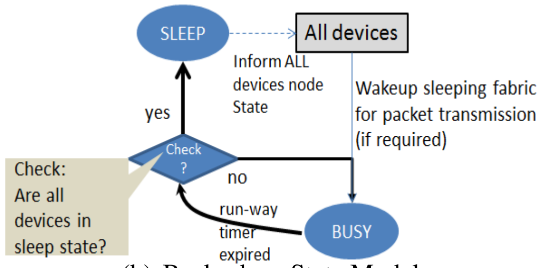
Since, NS3 has little to offer in terms of energy management; it merely defines some parameters on energy consumption. To remove this deficiency, we enhanced and implemented an energy model for NS3 based on currently available network device energy management features in both inside-the-box fabrics (e.g., PCI-E) and outside-the-box fabrics (e.g., Ethernet). we extended NS3 in four directions with respect to energy management:

(a) individual port level power management; (b) node fabric or backplane power management; (c) traffic consolidation performed at port (link) level; and (d) global controller that helps to consolidate traffic further. The first two opportunistically use sleep modes to save power, whereas the last two attempt to enhance the opportunities for sleep. The sleep modes in CPU cores are referred as c-states, the higher the c-state number the deeper the sleep mode. That is, state C6 saves more power but with a higher wake-up penalty compared to C4 state.

An idle port may go into sleep mode (with or without informing its neighbors) and wake up in a reactive manner upon new traffic arrival Fig. 2.5(a). Furthermore, the backplane of a switch could consume a significant amount of energy, but it is not realistic to put it in sleep mode unless all ports attached to it are sleeping. We assume that the CPU of the backplane can go to a relatively deep sleep mode, such as C6 or higher, along with memory in self-refresh mode. The exit latency in this case could be in 100μs range, which could cause significant delay to the incoming packet of a flow arriving at a switch/router, when the backplane is asleep. To reduce the impact of exit latency from sleep mode, we use a timer (called *runway*) both at port level and at the backplane level. Whenever a port goes idle, it waits for its respective runway time and thereafter moves to sleep mode. Likewise, a switch with all sleeping ports initiates its runway timer, and moves to sleep as the timer expires Fig. 2.5(b). However, it is crucial to ensure that no packet will be lost.


Device A
SLEEP
Wakeup on packet to be sent
BUSY
Transmit on physical layer
From Fabric
when new packet is sent
--device can be in any state.
--Wakeup penalty is added if device in SLEEP state
transmit end
transmit start
run-way timer expired
IDLE

(a) Device Link State Model


SLEEP
All devices
Inform ALL devices node State
yes
Wakeup sleeping fabric for packet transmission (if required)
Check ?
no
Check: Are all devices in sleep state?
run-way timer expired
BUSY

(b) Back-plane State Model

FIGURE 2.5: Enhanced NS-3 Energy Management State Models

Another major tasks is to select the time constants properly which conforms to the device state (L0 vs L1 or C6 vs C7). The runway is a parameter of the algorithm and can be set to a fixed value or adjusted dynamically based on the device utilization. Additionally, the transition into and out of sleep state takes some time. Usually the entry delay is small, but the exit delay is larger and directly contributes to the delay of packets that arrive when the device is in sleep mode. For simplicity, we bundle both of them under a single delay that is charged on exit. It is reasonable to assume that the power consumption during entry to and exit from sleep mode is the same as idle power. In addition to the reactive exit used here, it is possible to implement more sophisticated proactive exit, that predicts the arrival time of the next packet and attempts to exit sleep mode before the actual arrival. This could reduce the impact of exit latency; however, it is unlikely to work very well for individual ports. For backplane, it may be possible to use more sophisticated mechanism - such extension can be added easily.

As a configuration control technique, we changed the routing logic to have the traffic consolidated over the fewest number of ports (links) i.e. port level traffic consolidation; whereas the backplane power management is done opportunistically depending on network utilization. Thus, by limiting the number of heavily utilized ports, we can use the sleep modes more aggressively for moderate to low utilized ports. In our routing scheme, all the nodes choose the highest utilized outgoing link among all available links (outgoing links connect to immediate neighbors) towards the destination, to forward an incoming packet. We have coined this as *local consolidation (LC)*, since the nodes consolidate the traffic, solely based on local information. The nodes (switch or router) play the primary role in *local consolidation* mechanism.

2.4.1 Implementation of Traffic Consolidation

Additionally, we have implemented a *global controller (GC)* (somewhat like SDN controller). The *GC* has monitoring capability, that regularly exchanges information with both the end points (hosts) and nodes (switch or routers) . In return, *GC* can provide hints to the nodes (local consolidation) for better traffic consolidation. It can also pass hints to the end point controller for energy efficient flow placement. Since the *GC* has global visibility, one possible type of hint is the network segments that may be congested. This capability can be exploited to ease traffic bottlenecks during flow placement or traffic consolidation. This coordinated approach helps us to save even more energy without raising network delay significantly as shown in Figs. 2.9(a) and 2.9(b).

The basic NS3 routing provides ECMP (Equal Cost Multi Path) where the routing table generates all possible minimal cost paths for every source and destination pair. To forward a packet to the next hop, a node selects one out of these paths according to the uniform random distribution. In our implementation, we derive the ECMP routing table from NS3 only once at the beginning of our extended routing mechanism. Then we selectively choose subset of the available paths to make better routing decisions.

NS3 has its own notion of flow, which is used to track the packets exchanged by the nodes and to compute the flow level statistics. NS3 defines a flow as a 5-tuple which includes upper layer protocol, source IP & port, and destination IP & port. In our implementation, we generate an explicit ID for each flow, which is carried by every packet of the flow. This allows us to divide the packet stream between a source- destination pair into multiple flows, each suitably tagged by the flow id. Such a mechanism is crucial for implementing *local consolidation* (Section 2.4). The packets with the same *flow id* are forwarded in one outgoing net device, and this information is stored in an extended routing table. Alongside the packet routing information (source, destination, next-hop), this extended routing table also stores the flow id. In contrast with the default NS3 routing table, the routing entry is dynamic in nature and is modified upon flow start/end. When the first packet of a flow comes to any node and finds no entry for that flow in the extended routing table, it picks one path from the default routing table and stores it in the extended table. When the flow ends (we have a notion of *last packet* attribute to identify this), the flow entry is removed. If any incoming packets of a corresponding flow finds its entry, then it simply follows the path that has been set by the first packet of that flow.

To consolidate the traffic, we always focus on the subset of available net devices (links). If any flow can reach its destination by reusing certain highly-utilized outgoing net device, then we put a extended routing table entry for that corresponding net device and flow id. We also define the *required bandwidth* attribute for the first packets of each flow, which we use to implement better admission control policy. We consolidate the flows to any outgoing net device up to certain utilization threshold. If the required bandwidth of the new incoming flows exceeds the threshold, it is forwarded to another outgoing net device. If no net device has the available bandwidth, then the flow is dropped.

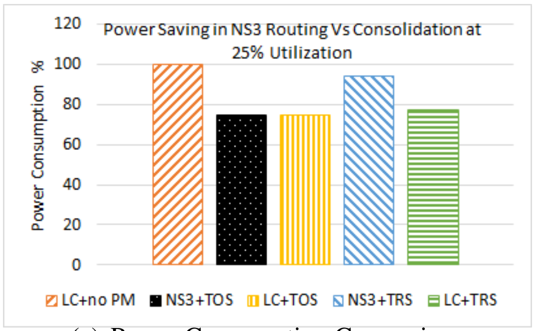
2.5 Simulation Results

We illustrate the energy model and effect of configuration (e.g. TOS vs. TRS, workload, traffic consolidation, etc.) by presenting some results for a fat-tree based data center network.

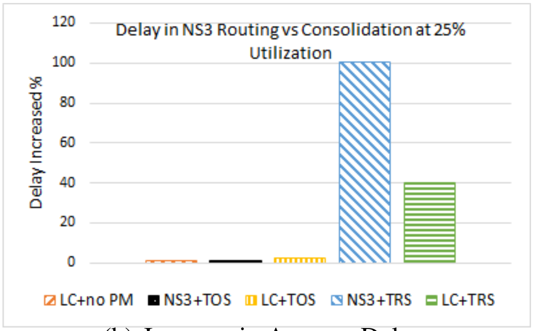
2.5.1 Experimental Setup

We choose the fat-tree topology here because of its near universal implementation in commercial data centers. Our fat-tree consists of 16 nodes ($k = 4$) as shown in Fig. 2.4(a) with bandwidth of the links in ratio 1:2:4 i.e. edge bandwidth : aggregate bandwidth : core bandwidth. We use the traffic consolidation threshold of 90% of the total link bandwidth. The parameters used in experiments are listed at Table 2.1.

The performance illustration in this paper uses analytically generated traffic for flexibility in emulating various traffic characteristics. The flow duration follows the Pareto distribution with shape equal to 3.5 and mean of 36 ms, ranging from 22ms to 375ms. The bandwidth of each flow follows the uniform distribution with mean 100 Mbps and with a range of 50 Mbps to 150Mbps. To support various levels of traffic burstiness, we implement two state Markov Modulated Renewal Process (MMRP) as traffic model. In this model, the two states have different flow generation rates based on the given burstiness

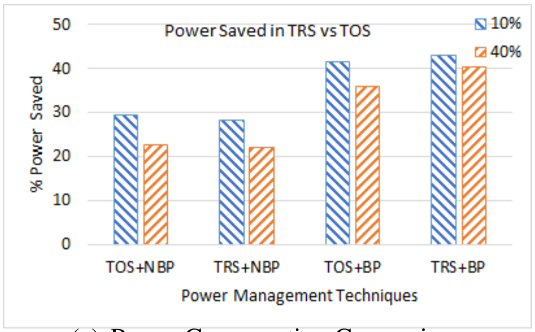


(a) Power Consumption Comparison

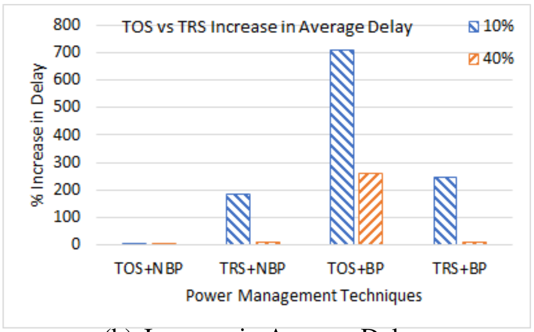


(b) Increase in Average Delay

FIGURE 2.6: Power Consumption and Average Delay Compared to Baseline at 25% Utilization



(a) Power Consumption Comparison



(b) Increase in Average Delay

FIGURE 2.7: Power Consumption and Average Delay Increase % in TOS and TRS

and average rate parameters. The residence time distribution in each state is exponential and is calibrated using the average residence time in each state.

2.5.2 *Experimental Results and Discussions*

We use four different power saving techniques;

(a) No power management at all,

(b) Transmit only sleep, henceforth called TOS for each port, which is basically a hardware based L0s implementation, i.e., the transmitter sleeps whenever it has no packets to transmit without any receiver coordination. Link synchronization is maintained.

(c) Transmit-Receive sleep, henceforth called TRS for each port, which is essentially an implementation of LPI mechanism and allows the transmitter and receiver to coordinate their sleep, and

(d) Additional switch backplane power saving mechanism with port based TOS and TRS discussed earlier.

We conducted experiments to compare the power-performance characteristics under different situations. First, we compare *local consolidation (LC)* against the *NS3 default ECMP* routing, mentioned in section 2.4.1. We then compare TOS and TRS under local consolidation (LC). We also examine the impact of backplane power savings, and the effect of burstiness of the traffic. In all cases, we also examine the change in delay faced by the flows compared to their respected baseline. We show the results for network utilization levels of 5%, 10%, 25% and 40% respectively. Very high utilizations are not interesting because they would not offer much opportunity to save energy. Finally, we compare the power-performance characteristic of TRS mechanism with LC only and with LC together with the GC. While most of the simulations were run with $k = 4$, we also show how the savings and delays vary with higher k.

Fig. 2.6 compares the power consumption and increase in delay (as percentage of the baseline i.e. with no power saving) under both NS3 default routing and LC. It is found that in all the cases the power consumption with LC is no more than that for the NS3 routing mechanism. Since the TOS mechanism works at a fine time granularity, its power consumption is almost identical under both LC and ECMP. With TRS, at 25% utilization, LC consumes about 77% power and increases the delay by 39%, whereas NS3 default routing consumes approximately 94% power and increases the delay by 100%. TRS has a high exit latency and this delay is due to premature exit from the deep sleep state which happens more frequently with ECMP than with consolidation. This same result was obtained in a large number of other cases that we tested, but not reported here. This establishes the superiority of LC, and we use it in all further experiments shown here.

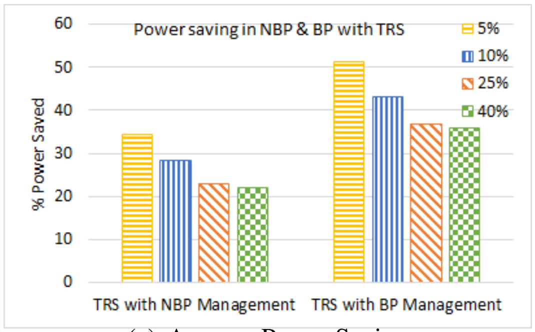
Table 2.1: Simulation Configuration (parameters used in experiment)

Active Power	4.95W
Idle State Power	4.95W (Sync Link)
TOS Sleep State Power	1.98W
TOS Sleep to Active Penalty	$0.1\mu s$
TOS Runway	$5\mu s$
TRS Sleep State Power	0.495W
TRS Sleep to Active Penalty	$73.6\mu s$
TRS Runway	$150\mu s$
Backplane Active Power	20W
Backplane Sleep Power	4W
Backplane Sleep to Active Penalty	$200\mu s$
Backplane Runway	$500\mu s$

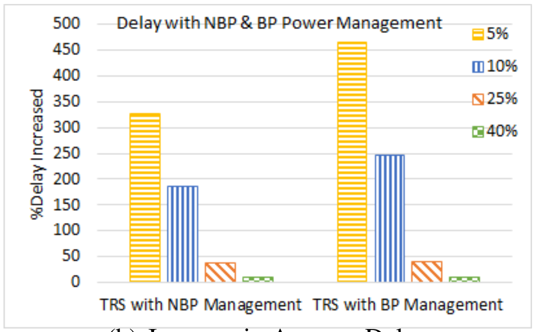
Fig. 2.7 compares TOS and TRS mechanisms at 10% and 40% utilization. The cases shown are both without and with backplane (NBP and BP) power management. *Without* the BP power management in place, TOS performs slightly better than TRS in terms of power saving. However at 10% utilization TRS incur more delay due to premature exit from sleep as compared to TOS. Furthermore, *with* the BP power management the scenario changes in favor of TRS. With TRS and BP power management, it saves 2-5% more power and reduces the delay compared to TOS. Because of the large latency associated with BP low power exit, the flows experience a significant additional network delay with TOS. The cost becomes more for lower utilization and also for the TOS as compared to a network using TRS under a moderate utilization. This is due to the fact that TRS uses a larger runway to go to sleep mode, and thus the BP also goes into sleep mode when the gaps are really wide, which is very rare under moderate to high utilization. Hence the average latency impact is less with TRS in 40% compared to 10%. Overall, the TRS delay is 140-200% lower than delay in TOS with BP power management.

Based on this result, we henceforth only investigate TRS. TRS is also more relevant for Ethernet, since TOS capability may not be exposed.

Fig. 2.8 shows the power savings and additional delay with LC under TRS *with* and

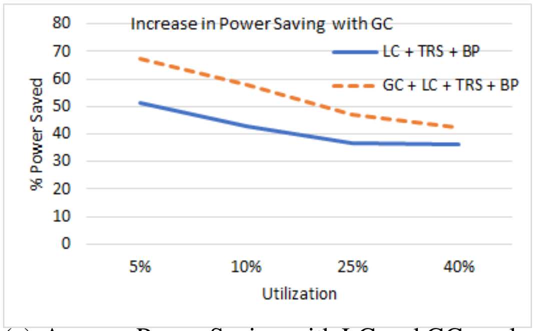


(a) Average Power Saving

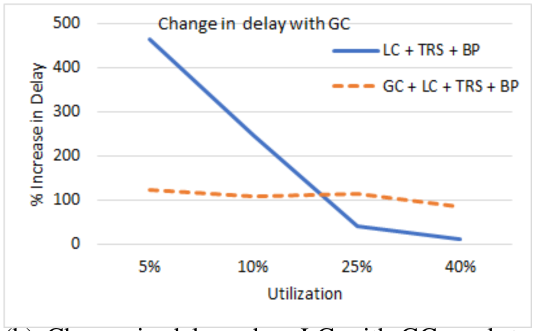


(b) Increase in Average Delay

FIGURE 2.8: Power Saving and Increase in Average Delay under TRS (with and without switch backplane power management)



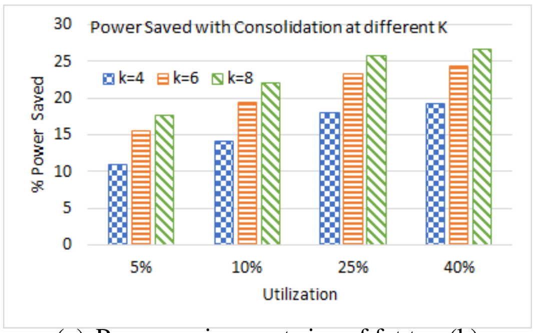
(a) Average Power Saving with LC and GC working together under TRS power management



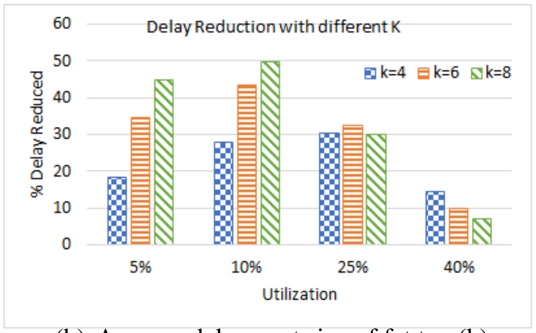
(b) Change in delay when LC with GC work together under TRS

FIGURE 2.9: Comparison of power saving and average delay with LC and GC working together under TRS power management

without BP power management. We save nearly 22-34% power with only TRS and NBP and incur delay due to exit latency in the range 10% to 300%. It is seen that *with* BP power management, we have 14%-17% more power savings compared to *without* BP at the cost of further increasing the delay. We show the results for 5%, 10%, 25% and 40% utilization. Lower utilization provides more power savings along with higher delay. *With* BP power management, the overall power savings fall in the range 36-51%, whereas the delay can increase very substantially to up to 10-460%. It is also interesting to notice that both 25% and 40% utilization under BP and NBP reveal same type of power saving behavior with the 25% having more delay than 40%. This delay is due to TRS exit latency and not the delay due to consolidation. It may be possible to reduce this delay with a more intelligent BP power management mechanism, but this is beyond the scope of this paper.



(a) Power saving w.r.t size of fat tree(k)



(b) Average delay w.r.t size of fat tree(k)

FIGURE 2.10: Effect of upgrading the size of fat tree(k) on power saving and on average delay

Next, Figs. 2.9(a) and 2.9(b) show the experiment conducted in presence of GC. We have done both port-level (TRS) and backplane (BP) power management here. We compare the performance vs power characteristics of TRS with BP under local consolidation (LC) both in the presence and absence of global controller (GC). We have used the same four utilization levels here. GC provides the hints to the LCs for energy efficient flow placement and routing, which can be seen from the curve Fig. 2.9(a). Without GC it saves around 51% power at 10% utilization and 36% at 40% utilization. Whereas, with the GC added, we save nearly 16% and 6% more power at 10% and 40% utilization respectively. Higher the utilization, less will be scope for power savings. Under low utilization, GC also helps to reduce the delay by helping LC do better consolidation and avoid premature wakeup. However, under moderate to high utilization a better consolidation by GC means higher queuing delay which can be seen from the Fig. 2.9(b) where beyond 20% (approximately) utilization level GC has experienced more delay.

Finally, Figs. 2.10(a) and 2.10(b) show the scalability of the mechanism as a function of k, which controls the network size. Unfortunately, the simulation time increases rapidly with k, and so we only go up to $k = 8$, which corresponds to 128 racks. A fat-tree requires more bandwidth at higher levels as k increases, and for the purposes of this study we simply scale up the bandwidth proportional to $k/2$ at higher levels. This is somewhat unrealistic in reality since Ethernet links provide only certain speeds (e.g., 10, 40 &

100Gb/s); however, a systematic scaling allows for observing trends that would be masked otherwise.

Fig. 2.10(b) shows power savings delays for load consolidation (LC) mechanism as compared with simple TRS power saving with NS3's ECMP mechanism. This is done for utilization levels of 5%, 10%, 25% and 40%. It is seen that the higher k provides additional power savings and also a delay reduction except at high utilizations.

There are two main reasons for delay reduction here: (a) the higher link speed for the higher level links, (b) avoidance of premature link wakeup due to traffic consolidation. A larger k provides more options for the traffic ($k/2$ links) and there is greater chance to consolidate traffic on 1 or 2 links which allows others to sleep. In contrast, in ECMP all $k/2$ may carry some traffic. The faster links at higher levels also tend to create larger gaps. However, as the utilization increase, these advantages diminish.

2.6 Conclusion

Using our enhancements to the popular NS3 network simulation tool, we presented the effect of configuring various parameters on the behavior of a data center network. We believe that this work substantially enhances NS3 in an area where it is currently very weak, i.e. network energy management. Although we have only implemented some basic energy management capabilities, they provide a launching pad for more sophisticated capabilities such as multiple sleep states and more sophisticated algorithms for deciding when to enter or exit a sleep state. Although our energy models have mechanisms such as those found in PCI-E and the standardized LPI mechanism for Ethernet, it is possible to extend these to other networks such as Infiniband and even optical networks. We have also implemented LC, GC and some simple coordination mechanisms between LC and GC. In future we would like to examine more sophisticated coordination mechanisms between them. Another possible extension would be to develop mechanisms for dynamically deciding when to move workloads (e.g. storage chunks) or redirect the requests (to other active copies) to alleviate congestion during high traffic episodes and to enable traffic consolidation (and hence network energy savings) during low traffic periods. We have released our implementation on a public research site (ResearchGate & github) to encourage new areas of research.

CHAPTER 3

AUTOMATED CONFIGURATION OF CLOUD STORAGE GATEWAYS

3.1 Introduction

A Cloud Storage Gateway [1] is an emerging concept in Cloud Storage Solutions; wherein the application is installed on-premise (or in close proximity to the data) and translates cloud storage object-store APIs such as SOAP or REST to the block I/O-based storage protocols such as SCSI, Fibre Channel, NFS or SMB.

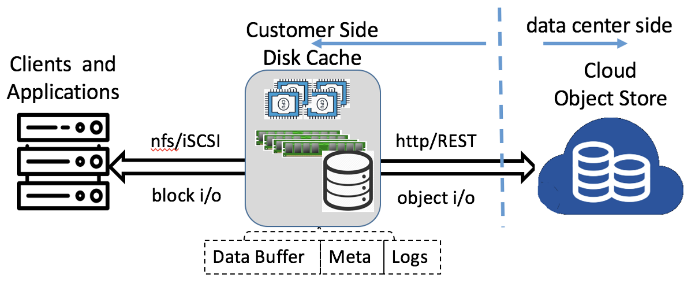


FIGURE 3.1: Cloud Storage Gateway Architecture.

Cloud Storage Gateway (CSG) concept was pioneered by Google [Prahlad et al. (2010)], and subsequently offered by many leading industry vendors as a Cloud Storage solution. As shown in Fig. 3.1, CSG appliance connects the client applications running locally to

[1] Recently the industry has substituted the word ”gateway” with the word ”controller” to emphasize the idea that their gateway products do more than just serve as a bridge

an object store hosted in a remote cloud data center. Although the remote storage could be block based, it is almost universally object based due to many advantages of the cloud model. The advantage of CSG is that while the user data resides on the cloud storage devices, it makes the accesses appear locally going to a SCSI device.

Customers deploy CSG to expand storage capabilities of their local computing infrastructure. For example, a large video animation customer like Disney could work with hundreds of graphics files of size 1GB or more, a financial company may store a large number of fiscal records in medium size files (say 1-10MB text files). The locally running business workloads would typically persist or retrieve a large amount of such data through the CSG. Since a single CSG may be used by many different business applications with different persist/retrieve patterns, a proper configuration of CSG is a very challenging problem.

Data center operators have a huge amount of operational data collected over time that can be exploited to understand the system configuration parameters and their influence on the operational behavior. We keep the discussion focused by studying the configuration challenges as pertaining to the customer side CSG system (i.e. on premise), and not the backend data center hosted cloud object store system in Fig. 3.1. We use a commercially available CSG available from a prominent market vendor. The main goals of our study are as follows:

- Design experiments to collect performance and configuration data for a large number of configurations of this CSG.
- Explore the use of suitable machine learning techniques to build models for solving the forward problem (predicting performance for given configuration parameters), and reverse problem (predicting certain configuration parameters based on target performance).
- Explore how the domain knowledge can be exploited to reduce the configuration

space and enhance the accuracy of the predictions.

The key contribution of our work is to demonstrate that we can build robust models for relating user settable system and hardware parameters to the performance of cloud storage gateway and thereby *debugging* the configurations. Even in cases where such automated analysis fails to provide the optimal result, it is expected to yield configurations that are close to optimal and thus can be tuned further with far less effort and time than the prevalent manual approaches whose success entirely depends on the experience of the administrators. To the best of our knowledge, the prior work has predominantly considered performance as a function of workload parameters rather than the user tunable system parameters.

3.1.1 Motivation and Challenges

Proper configuration management of complex cyber-systems is a very challenging problem in the real-world, and yet very much under-appreciated in the research community. Misconfigurations in large enterprises often account for up to 80% of the malfunctions and attack vulnerabilities, and routinely consume days of engineer's time to diagnose and fix [Xu and Zhou (2015)]. Configuration management of data center storage systems can be particularly complex and labor intensive task [Klimovic et al. (2018)], and CSG is no exception. In addition, CSG configurations combine the complexities inherent in storage system configuration, storage cache configuration, unpredictable network traffic and the complexities of back-end cloud systems. Similar to other cyber-systems, CSG has many configuration parameters or ”knobs” with little clarity on how to set them or what precise impact they have on the output end.

While working with the commercial vendor of a CSG product, we noticed that the most common problems were related to customer complaints about poor performance or I/O time-out errors. We invariably found that on further investigation that most of these complaints were a result of poor understanding of the workload (i.e. request streams) and the configuration parameters of the CSG. The main source of difficulties in configuration management are the numerous parameters with complex inter-dependencies that are mostly unknown or poorly understood with respect to their impact on the overall performance, availability or user experience [Yin et al. (2011)].

This prompted our research into understanding the relationship between various parameters in the Cloud Storage Gateway environment. For example, there are few *uncontrollable* variables such as eviction rate, cloud storage response, internet throughput, etc. and some parameters under the user control such as: workload, hardware characteristics, storage cache configuration, etc.

When the number of parameters that we wish to vary is small and known a priori, one could study their impact on performance via direct measurement, simulation modeling or even analytic (e.g., queuing) modeling. However, the configuration management problem is quite different from this in that we have a large number of parameters with intricate inter-dependencies between their settings and general lack of understanding of their relative importance from the output perspective. It is well known that the storage system performance depends on the workload characteristics, deployed optimizations, and their specific configuration [Klimovic et al. (2018)]. Configurations also "are often difficult and knowledge-intensive to develop, brittle to various environment and systems changes, and limited in capacity to deal with non-steady-state phenomena [Tesauro et al. (2005)]."

Cloud storage gateway (CSG) is a relatively new paradigm in cloud storage solutions, and effective methods for its configuration management is largely unexplored.

3.2 Current State of the Art

Klimovic and Costa [Costa and Ripeanu (2010); Klimovic et al. (2018)] support our complexity problem involved in analysing the workload data streams and a wide choice of configuration space to be explored in a cloud storage system. In designing Selecta, Klimovic address the storage configuration for Data Analytics workload using TPC traces on the block storage devices on data center side, while our work studies the object storage gateway configuration on the customer side using corporate workloads. Costa [Costa and Ripeanu (2010)] state that configuring a storage system for desired deduplication performance is extremely complex and difficult to characterize. Rao [Rao et al. (2009)] show that a traditional control theoretic framework is inadequate to capture the complexities of resource allocation for VMs. Ofer [Ofer et al. (2018)] use deep learning techniques in object storage systems to recommend the best strategy for cache eviction and refreshing data. Their study is the closest that relates to our work both in terms of application of machine learning and working with cloud based object storage systems. While their study applies deep learning to cache eviction/refresh techniques in object store, we explore the configuration management of object store based CSG.

Hsu designed Inside-Out [Hsu et al. (2016)] to predict performance in a distributed storage system. They study low-level system metrics (e.g., CPU usage, RAM usage and network I/O) as a proxy for measuring high-level performance. Cao [Cao et al. (2018)] evaluated few popular black box auto-tuning techniques for storage using macro-workloads generated by Filebench. Their comparative study supports our research in that optimal configurations depend by hardware, software, and workloads and that no one technique is superior to all others.

Almseidin [Almseidin et al. (2017)] use empirical methods to evaluate best-fit algorithms for their intrusion detection system. Authors in [Esposito et al. (2016)] apply fuzzy logic and game theory for storage service selection. They choose the optimal storage service to satisfy the constraints of price, QoS, etc. Ularu [Ularu et al. (2013)] use decision trees to configure an application and highlight the use of decision trees on solving a configuration problem because of the wide solution space to be explored.

3.3 Cloud Storage Gateway (CSG)

Unlike the cloud storage services which they complement, CSGs use standard network protocols that integrate with existing applications and can also serve as an intermediary to multiple cloud storage providers. Increasingly, CSGs also provide many sophisticated services such as backup and recovery, caching, compression, encryption, storage de-duplication and provisioning. A CSG will typically serve multiple clients using a set of local storage devices (possibly a RAID but not necessarily) that is seen by the clients as a local block storage. All clients assigned to this local storage share the storage, although there might be some internal fine-grain storage allocation policies that are not revealed to the client. Each client will be allocated space for its data, metadata, and log files. The CSG can be viewed as two I/O layers (see Fig. 3.1): (i) front-end for local user I/O and (ii) the back-end for cloud storage I/O operations. At a minimum, the CSG will provide the ability of intelligently partitioning the space into data, metadata, and log files, and a suitable caching mechanism for each so that data transfers from the backend can be properly handled.

A CSG should bridge the IO gap between the on-premise SCSI based disk operations (high IO rate, low latency, almost zero errors) and the backend Cloud object storage system (low IO rate, high latency, retry on timeouts/errors). The server capabilities, resource allocation, and configuration of the Edge Controller becomes an important factor that defines the latency/performance experienced by the users. The complexities involved in the Cloud-based object-store workload patterns are different from the traditional local block IO workloads. We will examine (i) these complex factors and their effect on the behavior of CSG, (ii) performance/ latency experienced by the user workloads, and (iii) the Cloud upload conditions (or failures). Beyond delivering user performance, Edge Controllers have to satisfy constraints such as cost/ space/ cooling, etc. We can quickly infer the challenges in 'right-sizing' the server, allocation of resource(s) and satisfying the user/

application demands.

3.3.1 Brief Overview of Object Store

In object store, data is represented as an ”object”, which refers to a piece of data described (and pointed to) by the appropriate metadata. The metadata resides separately in a metadata server (MDS). The objects are stored on ”object storage devices” (OSDs) that natively manage the mapping of the objects to the underlying device structure such as sectors or blocks. The metadata server together with the OSDs also implements access control to the objects so that it is not possible to directly access an object from OSDs. Instead, a query to the metadata server generates a capability that must be presented to the OSD for access to the data. A single metadata server typically serves multiple OSDs. To access an object, an application first contacts the metadata server, and then directly accesses the relevant OSD to retrieve the data using the capability provided by the metadata server. This makes the object store model quite scalable since accesses to multiple OSDs can proceed in parallel. This structure is shown in Fig. 3.2[Flash Memory Submit (2018)].

An object could represent any type of entity including entire file, fragment of a file, a contiguous set of database rows, a directory, etc. Object size is often limited, so that a very large file may have to be split into multiple objects. Most common types of data stored in the object store are unstructured data (but rich with metadata) such as images, audio, and video clips. Objects are typically identified with 64 bit Object ID and grouped within partitions with 64 bit partition ID. This gives each object a unique 128 bit namespace [Flash Memory Submit (2018)].

Rich metadata and flexibility of objects enable the user to find data based upon regular expressions or search in large data-sets on metadata properties. This allows users to treat the Cloud as a large database of objects. As the size of the Cloud grows, so does the ability to find data based on required object-properties (e.g Films created before 1980).

Metadata processing is essential to suitably access an object and small sized metadata makes it easier to cache. Thus, dedicating adequate compute resources to process metadata is crucial for good storage retrieval performance.

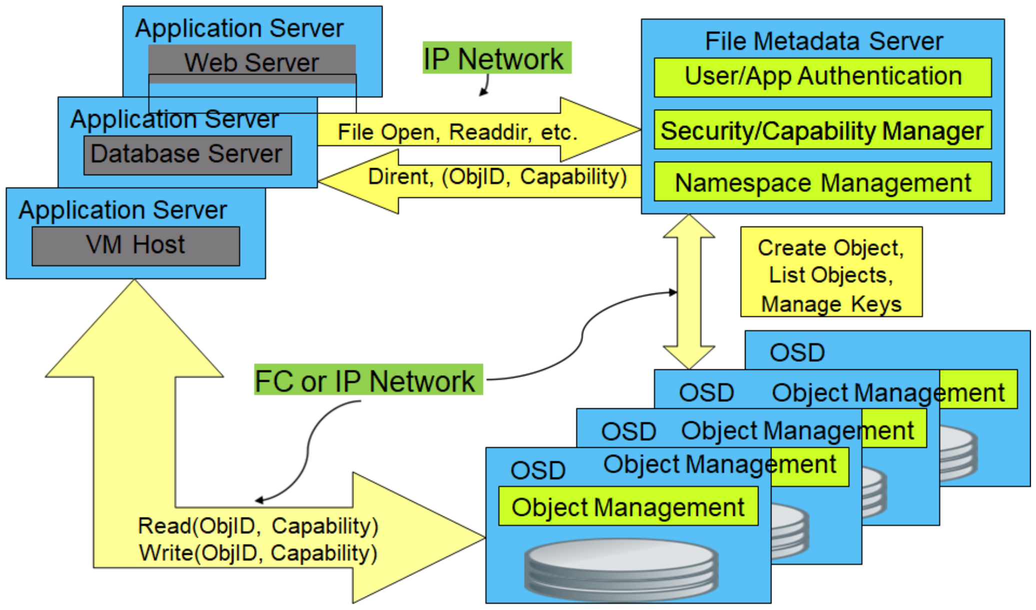


FIGURE 3.2: Object Storage Infrastructure

In traditional block file systems, block allocation and data transfer are very expensive. In block IO file systems API structure forces users to make more frequent API access (open, write, seek, close). OSDs grants (or denies) access to individual objects and fetch (or write) the objects using high level API calls that encapsulates low level details. Hiding more functionality in OSD APIs allow users to make less calls, thereby allow high level API optimization to get higher data throughput and performance. High level OSD API schematic allow for user access, authentication and object namespace control.

Because of these advantages, Object storage forms an ideal platform for data storage on the Cloud. Every object can be accessed directly with a unique ID and direct http/REST API, making data access faster. For these reasons, all commercial vendors of CSG devices use Object store as a Cloud Storage platform.

Security and Metadata

Security and metadata management in CSG system is a complex task. File access by a legacy application over SCSI should be translated to a OSD object operation. This involves translating the low level SCSI APIs such as file open/close and data write, to high level OSD APIs including metadata operations. Further, file attributes such as permissions, directory structure, etc. have to be transferred to OSD operations like Object ID, object path, metadata attributes. These metadata overheads involve considerable space and time, and consume both CPU and memory resources. Faster CPU speeds can translate the file-attribute (permission, sub-directory, name) to object metadata (ownership, object path, Object ID) faster. Thus, size of metadata operations and allocated resource (CPU speeds, metadata space) play an important factor in determining the user experience in an CSG system. All Object Storage Service (OSS) operations have to consult metadata for *relevant information* about the object. Caching metadata can improve performance and potentially avoid consulting the Cloud based OSD frequently. In this sense, resource allocation for metadata becomes an important factor in space available to pre-fetch metadata. Better user experience and higher performance depend on both data and metadata operations. Hence, compute (CPU) and storage (space) resource allocation to both data and metadata play an important part in determining the performance experienced by the users of an CSG system.

3.3.2 Characterizing the Behavior of a CSG

The performance and behavior of a CSG depends on the hardware platform architecture h (CPU, memory, storage, network, local I/O rate, etc.), workload characteristics k (incoming request rate/distribution, data writes/sec, data reads/sec, metadata reads/sec, metadata size, etc.) and application goals p. Application goal is predominately expressed as I/O performance (MBytes per sec). Generally, the application goals are achieved by : (a) most reads are satisfied locally (which is essential to match the higher I/O inject rate to the slower back-end rate), and (b) maintain and batch writes locally so as to make the writeback more efficient. These functions along with the management of meta-data files, rotating log files, garbage collection etc. are normal storage system attributes that affect the behavior of CSG. Furthermore, workload characteristics such as burstiness are also important. Workload and performance may be specified either directly in terms of resource requirements, or in more abstract terms such as priority, latency, or resource combination (e.g., a "small" vs. "medium" vs. "large" configuration). In any case, these are ultimately translated to individual system parameters, either explicitly or via policies.

CSG combines the complexities inherent in a storage system, storage cache allocation[2], IO demands and the unpredictable nature of back end Cloud systems [Tanimura and Koie (2015)]. It is often very difficult to pinpoint how a change in system configuration can affect the overall performance of the system. Even the concept of performance itself can be subject to scrutiny when considering the complexities of subsystem interactions and the many ways in which performance metrics can be defined [Eckart et al. (2009)]. Similar to other cyber-systems, CSG too has many configuration parameters or "knobs" with little clarity on how to set them or what precise impact they have on the output end.

It is easy to see that if the workload characteristics k, system configuration h and CSG configuration r are not matched, the end users would likely experience undesired performance p which is usually defined as the read and write rates supported with certain maximum latency and without any I/O timeouts. Read/write operations beyond the acceptable range or complete rejection is considered I/O failure. As workloads change over time (i.e. as i^{th} workload k_i deviates from initial assumed pattern k_i), the initial user-defined configuration r_i may no longer support the demands of new workload(s) and cause undesired end-user experiences.

A major issue in properly configuring the CSG is that *the vendors invariably do not*

[2] In the rest of this work, cache refers to "storage data cache", and this cache has nothing to do with processor cache.

reveal most of the internal details, and instead expose a limited set of administrator controllable configuration parameters to tune the system. Often, these administrator controllable parameters are not even the actual configuration parameters, but merely some sizing controls or decision variables that affect multiple internal parameters. In other words, the knobs visible to the enterprise are rather fuzzy with little knowledge of what exactly they do. Of course, this is partly done to simplify the job of the administrator; a vendor willing to expose all raw knobs would invariably make them unusable. Thus the phenomenon of fuzzy knobs with little visibility is an essential characteristic of real systems and cannot be wished away! It alone precludes simple analytic models for characterizing performance of a component like CSG or fine-tuning it for very specific workload or hardware. With absence of any quantifiable, well-defined correlation or closed loop representation, configuration management is more an art than science.

3.3.3 Complexities in Configuration Control

Generally, as the size of the local storage (henceforth referred to as ”storage cache”) increases, we expect the CSG throughput to increase because more I/O can be handled locally. The benefit is entirely dependent on how well the caching mechanism keeps and prefetches ”hot” data. Since the overall space available for caching is *shared* by multiple clients, there is interaction across clients. For example, giving more space for some clients hurts others, and the net effect is very complex to predict. This caching mechanism is hidden by the vendor and not controllable by the end-user, adding to the complexity of the configuration control. These comments apply to both data and metadata but with different effects. The metadata needs to be consulted for every I/O regardless of whether the corresponding data is in the storage cache or not. Depending on the workload and the granularity of access, metadata caching becomes more dominant than data caching. At the same time, metadata is generally much smaller than data, and thus it is much easier to provide generous amounts of storage for metadata. We explore the storage cache vs.

metadata size in our research through varied workload and configurations studies. The log size should have no influence on performance except that writebacks of the log would take up some backend I/O bandwidth. This initial description paves the way for understanding the complexity of configuring the CSG system.

Incorrect configurations could result in significant competition between the following three activities:

1. Eviction of modified pages requiring writeback to the backend object storage which is likely to experience high latency, limited I/O bandwidth due to network issues, and perhaps a significant write amplification due to the need to write the entire object.

2. Cache misses from client requests thereby requiring reading of backend object storage, which experiences similar issues as writebacks (e.g., significant read amplification and latency due to transfer of entire objects, for which the CSG need to make adequate room).

3. Local read/writes performed by the clients which are expected to be much more frequent and expect a low latency.

Note that the cache eviction is not complete until the write confirmation is received from the cloud storage, and the data must be kept in the storage cache until then. Consequently, a more aggressive eviction would only result in fewer entries in the storage cache to handle new data requests that must be fetched from the cloud. The CSG may also need to retry the *entire object* operation if unsuccessful. In addition to the whole object transfers on the backend, the transfer normally uses HTTP which adds considerable overhead. The varying size of the object could interfere with the SLA guaranteed to the user about upload/inject rates. We quantify SLAs based on local SCSI update traffic generated by the client, and any SLA violation is seen by the client as I/O timeout or errors. Note that increasing the storage cache size does not solve the writeback problem; in fact, it could even make it worse.

3.4 Configuration Problem Formulation

The CSG system in Fig. 3.1 serves as a 'storage cache' to buffer the incoming user request and match it with the cloud storage uploads (or downloads). User requests come over a SCSI bus at a high arrival rate λ_{in} (high throughput, high bandwidth, low latency) and the backend cloud storage presents a low eviction rate system λ_c (high latency / low bandwidth / higher error retries). Resource allocation for the storage cache needs to match the k_n incoming request streams and the service rates. We model CSG Controller of Fig. 3.1 as a queuing system. The incoming request stream k can be represented as:

$$k = f_1(ar, rs, rm) \tag{3.1}$$

where: k the request stream is a function of: ar the request arrival rate, rs the size of the request, and rm the metadata size. These request streams are characterized by the real world customer workloads as given in Table 3.2. Each of these requests consumes hardware and storage cache resource for servicing. Note in Fig. 3.1 that all n request streams share the same resource of CSG controller. There is no known functional relationship to analyze the queue behavior, and the controller has no user controllable factors to allocate resources per request stream. The relationship between the input stream, requested resource, queue and the storage cache is unknown. This leads us to conclude that there is no clear optimization or queuing technique that can model the serviceability of the incoming streams. For example, if the cache size is small and the eviction rate to backend cloud storage is high, then the system should be able to handle high requests. The converse means the requests will be dropped.

To add to this complex analysis, the CSG runs on a hardware platform characterized by core speed, number of cores, memory capacity, disk I/O capacity and network I/O throughput. Here, network I/O throughput represents the measurable I/O throughput to upload an object from the client system to the cloud storage. The hardware characteristics of the CSG platform is represented as:

$$h = f_2(cs, nc, me, m_{bw}, di, th) \tag{3.2}$$

where h the hardware characteristics is a function of: cs core speed, nc number of cores, me memory capacity, m_{bw} memory bandwidth, di disk I/O rate and th I/O throughput measured between the on-premise CSG and cloud object store. For example, one disk partition d_j on three independent disks has a better I/O rate compared to three individual partitions on one SCSI disk. Similarly a network card capacity of 10 Gbps would have better eviction rate than a network card capacity of 1 Gbps. Another evident characteristic is the performance boost from using a SSD versus a HDD disk. Again modeling the complex interactions between these different limited parameters and their relative effect on the *CSG storage cache performance* is unknown.

The disk cache is split into three distinct partitions: data storage buffer db, metadata md and log space ls. (Section 3.3.3 & Fig. 3.1). These are bound by the total resource disk space available ds_{max}, such that:

$$ds_{max} \geqslant db + md + ls \tag{3.3}$$

Each of these parameters db, md, ls coupled with the request stream k influences the behavior of the system as explained earlier.

Finally, if the configuration r of the CSG is optimized on a given hardware h, the input stream/ workload k will experience a performance or service rate of p. We denote this performance as:

$$p = f_3(h, k, r) \tag{3.4}$$

denoting that the performance (or service rate) p depends on hardware characteristics h, service rate (or workload) k and CSG configuration r. Note that Eq. 3.1, Eq. 3.2 and Eq. 3.4 is a multidimensional vector. Changing any one of the above parameters will affect the performance output p.

3.4.1 Research Questions

We define our research problem using a specific example relating to configuration challenge faced by a system administrator. Suggest a CSG configuration that satisfies the required performance p (10MBps), given a specific hardware architecture h (2 x 1.2GBps cores, 64GB RAM, 1GBps NIC card, 0.25GBps network speed) and a workload w (5 concurrent users, 10 files, avg 5GB size, upload time: 24hrs). In general, we define the following:

1. $P = \{p_1, p_2, \cdots p_n\}$: performance constraints for each of the n applications, predominantly defined by expected I/O rate, e.g.: 100MBps min for both I/O write and reads.

2. $K = \{k_1, k_2, \cdots k_n\}$: workload characteristics for each of the n applications. We used real-world workload patterns observed from end customers of a commercial industry vendor (See Table 3.2), predominantly defined by average file size, number of files, users, sub-directory hierarchy etc.

3. $H = \{h_1, h_2, \cdots h_m\}$: m hardware characteristics, i.e. the machines running the CSG application, e.g.: core speeds, memory, local storage disk characteristics, network I/O bandwidth, etc.

4. $R = \{r_1, r_2, \cdots r_l\}$: l cloud gateway configurations each specified in terms of system configurations e.g.: 100GB storage cache, 25GB log space, 50GB metadata space, 10 concurrent threads etc.

We can now ask two key questions:

Q.1 What should be the *storage cache configuration* to satisfy the 'k' request streams.

Q.2 What is the *recommended configuration* that satisfies the user required "performance" for given workload (i.e. give the right allocation of resources: $nc, cs, mc, m_{bw}, \cdots$)

for a given $\{k, p\}$.

We are interested in a mechanism that draws a relationship between the workload characteristics K, system architecture H, configuration R, performance P and can *answer* the following.

1. Verify Configuration - given the system architecture, workload characteristics, and CSG configuration, determine if the performance constraints are met with a high probability.

   $$[H_{new}, K_{new}, R_{new}] \Rightarrow P_{new} \text{ is satisfied} \tag{3.5}$$

   In most cases, only a few parameters are new; however, because of the dependencies and nonlinear interactions, it may or may not be possible to exploit the unchanged parameters. This represents Q.2 of our discussion.

2. Configure - given the system architecture, workload characteristics, application goals and performance constraint, propose a configuration R_{new} that satisfies all the constraints. That is, given H_{new}, K_{new}, and P_{new} find R_{new}.

   $$[H_{new}, K_{new}, P_{new}] \Rightarrow R_{new} \tag{3.6}$$

   This problem is the reverse of the first problem and is substantially harder. As in the last problem, only a few parameters may be new but it may or may not be possible to exploit the unchanged parameters.

3. Predict - Based on a time series of performance data for a given configuration, with changing workload, predict if the current configuration is likely to fail.

   $$[H_{old}, R_{old}] and [K_i, P_i], i = 1, 2, ...n \Rightarrow \text{ Failure} \tag{3.7}$$

3.5 Solution Approach

As stated earlier, the complexity of the relationships between the user settable CSG parameters and the performance precludes a modeling or simulation based characterization.

Therefore, we turn to machine learning based methods to learn various relationships along with our domain knowledge into the functioning of the CSG. Machine learning (ML) is, of course, no panacea; it often requires a significant amount of training data and the learned model may be ”over-fitted”, and thus may be unable to accurately predict behavior when the inputs (workload or configuration parameter values) are sufficiently different from those for the training data. We will address this aspect carefully in our analysis.

3.5.1 Feature Vector

The feature vector used to support our research is built from the above equations Eq.(3.1, 3.2, and 3.3) and represented as below. We have included throughput (a.k.a performance in bytes/sec) in the feature vector. As explained earlier in section 3.4, our work defines QoS and SLA in terms of latency. Any latency exceeding the limit is experienced as I/O time-out. We do not explicitly include latency in the feature vector since a configuration that leads to time-outs will be rejected right away and is not relevant for performance analysis. We discuss some tests that violated SLA and unacceptable latency in Section 3.6.3; these were attributed to wrong configuration choice for specific workload constraints. Learning configurations that cause time-outs (and thereby avoid them) is a reasonable goal, but much harder and beyond the scope of this work. Therefore, all our tests used in the analysis are for valid SLA conditions.

ar	request arrival rate	rs	size of the request
rm	request metadata size	cs	core speed
nc	number of cores	me	memory capacity
mbw	memory bus bandwidth	di	disk I/O rate
th	network I/O throughput	db	data buffer
md	metadata	ls	log size
p	system performance (measured as throughput)		

Table 3.1: Subset of Feature Vector Supporting the Problem.

There are many parameters relevant to CSG operation that could potentially affect the performance; however, it is neither possible, nor practical to consider them all. There is no escape from applying the domain knowledge to consider only those parameters that are likely to be controllable or relevant. One such example is the ubiquitous use of NFS for users to mount the remote storage device (but still within the local data center boundaries). NFS has many mount options (rsize, wsize, etc.) that can be chosen for individual mounts. However, these parameters are invariably set at default values and unlikely to be changed. Similarly, although the storage interconnect speeds (PCI bus speed, SATA/SAS interface speed, etc.) are potentially important, their selection happens at a much more basic level (i.e., when deploying the storage device/system) rather than for performance optimization. Therefore, consideration of these aspects is beyond the scope of this work. We will present the effect of choosing the right design variables (a.k.a feature set) on the accuracy of performance prediction in results section.

3.5.2 Research Hypothesis

We use statistical machine learning, classification, and optimization mechanism to $learn$ these relationships. Our prediction model is expressed as a function of the above feature vector. Let $\vec{x} = \{x_1, x_2, \ldots\ x_k\}$ denote the vector configuration parameter values. Let $\phi(\vec{x})$ denote the *hypothesized functional relationship* to be learned and $\gamma(\vec{x})$ is the true observed output for given values of the input $\vec{x}$.

We now have the basics to answer our research question Q.1 and Q.2 by using the above features and to accurately design our hypothesis $\phi(x)$ and output γ.

For a given workload, a service rate k_i, a set of constraints on hardware h_i and CSG configuration r_i, we $predict$ the maximum performance p_i by the following hypothesis (i represents i^{th} variation).

Hypothesis:

$$\phi(ar, rs, rm, cs, nc, mc, mbw, di, db, md)$$

$$\text{Output:} \quad \gamma() = p \tag{3.8}$$

Similarly, rearranging the features, and re-writing the hypothesis, we $predict$ the required configuration h, r required to achieve a given performance p for a given workload (service rate) k.

Hypothesis:

$$\phi(cs, nc, mc, mbw, di, db, md)$$

$$\text{Output:} \quad \gamma() = (ar, rs, rm, p) \tag{3.9}$$

3.5.3 Classification Problem

In Eq.3.8, we compute a single parameter p (performance) for a given set of constraints, and in Eq.3.9, we compute multiple parameters $nc, cs, mc, mbw, db, \cdots$ (cores, core speed, memory, $\cdots$). Eq.3.8 is called single label classification [Sorower (2010)]. Computing a single label/parameter is relatively easier than computing multiple inter-related labels/parameters. For our end results, we predict performance, storage cache size etc. as multiple classes (e.g. performance = {class 1, class 2, ...} or storage cache size = { class 1, class 2, ... }). We will quantify this during our discussion on workload design and results in Section [3.6.2 & 3.8].

3.6 Implementation Details

We implemented the algorithms in Python using *scikit-learn* [Pedregosa et al. (2011)] library for Machine Learning components such as Principal Component Analysis (PCA), Classifiers, ML metrics (e.g. accuracy, precision etc.), Feature Importance etc. For machine learning metrics, PCA, feature importance, etc. we refer readers to available materials (see Almseidin et al. [Almseidin et al. (2017)]).

3.6.1 Test Environment

Our test environment (Fig. 3.3) is comprised of (i) Dell PowerEdge R320 with 4 cores @ 1.8GHz, 16GiB memory, 3 ATA Disks- each of 500GB and one 1GB Ethernet interface (ii) Dell PowerEdge R730xd with 8 cores @ 2.1GHz, 32GiB memory, one SCSI disk of

5495GB and one 1GB Ethernet interface. Both servers have Ubuntu 14.04 with required tools and connected to local network. We used different hardware configurations to study the influence of cores, core speeds, disk, and memory configurations. On each of these servers, we partitioned the disk for several storage cache configurations. The server is connected to the HDD volumes on a remote cloud object store service, as NFS mounts. We used C++ and Python scripting tools for executing the workload and collecting metrics.

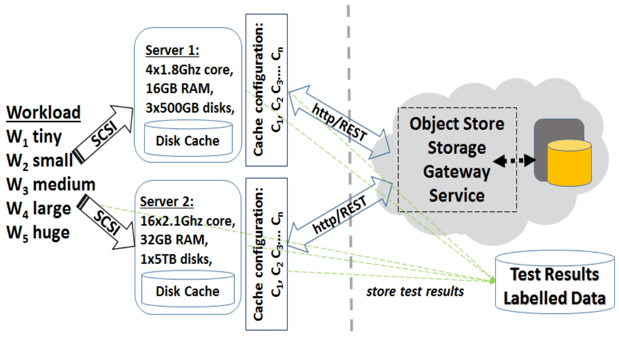


FIGURE 3.3: Cloud Storage Gateway - Test Environment.

Once the test scripts were ready, the evaluation setup for each experiment involved partitioning the disks for various configurations, allocate storage cache/ metadata size (see Table 3.4), connecting the newly configured server to data center object store and running the workloads and of course collecting the metrics.

3.6.2 Workload Execution

In absence of publicly available CSG data patterns or workload streams or trace dumps, we used a set of vendor provided workload patterns (shown in Table 3.2), that are reflective of real-world CSG user population. Meta-data shown in the table refer to attributes that influence both the meta-data operations and the performance. These are shown in the meta-data column as ownership (O), sub-directory depth (F=Flat,D=Deep), and object-permission (P). These workloads are equivalent to studies from Yi [Yi et al. (2015)], Varma [Varma (2008)], and YCSB [Cooper et al. (2010)]. Since CSG is responsible for buffering and uploading the data generated locally by edge applications, the workload characteristics are dominated by write requests of varying size (given in Table 3.2). The workload in

Request Id	**Object Operations, (# users, objects, object size)**	**Meta-Data**	**Sample Applications. [Varma (2008); Yi et al. (2015)]**
W1	25 x 10,000 x 4 KB	(none)	Health Monitors
W2	25 x 10,000 x 4 KB	O,D,F,P	Health Monitors
W3	25 x 10,000 x 256 KB	(none)	MRI/ CT Scans/ Traffic Images
W4	25 x 10,000 x 256 KB	O,D,F,P	MRI/ CT Scans/ Traffic Images
W5	5 x 10,000 x 1 MB	(none)	DICOM Visible Light
W6	5 x 10,000 x 1 MB	O,D,F,P	DICOM Visible Light
W7	5 x 1,000 x 10 MB	(none)	Mammography/ Street View(1 min.)
W8	5 x 1,000 x 10 MB	O,D,F,P	Mammography/ Street View(1 min.)
W9	2 x 200 x 1 GB	(none)	Pathology
W10	2 x 200 x 1 GB	O,D,F,P	Pathology

Table 3.2: Sample Workload Type and Applications.

the table represents daily activities in the environment being monitored (e.g., a hospital, road traffic, manufacturing plant, etc.) which often have daily, weekly, and even seasonal patterns. The system design would generally be based on the traffic on a typical but busy day, and that's what we consider as our representative workload. It is certainly possible to occasionally encounter workloads that are substantially different in nature from the norm, and the performance or configuration prediction for those could deviate significantly from reality.

For example, workload patterns for a smart-health monitoring system is characterized as "W1" defined by image size of 4KB, about 10,000 images/24 hrs, with associated meta-data on date, ownership, location etc. Another workload pattern for health-care (e.g. Pathology) is characterized as "W5" with image size 1GB, about 200 images/24hrs, with meta-data about patient ID, hospital ID, etc.

3.6.3 Metrics Collected

We executed workloads on different servers and various configurations, and collected metrics on execution time, meta-data time (e.g. to create sub-directories, open and close files etc.), throughput in bit/sec. Alongside the workloads we captured the configuration information about the server (e.g. cores, core speed, memory, disk capacity etc.) and CSG

Raw Data Set (% of total)	**Train Data (67% of total)**	**Test Data (33% of total)**	**Prediction Accuracy**
247 (25%)	160	87	88.9%
594 (50%)	321	174	93.1%
792 (75%)	514	278	94.2%
991 (100%)	664	347	95.4%

Table 3.3: Test Termination Condition based on Data Volume and Accuracy.

cache configurations (i.e. storage cache area, meta-data, log size). We captured the available network throughput independent of the CSG, using a special RESTAPI tool set. We attempted a few workloads that would result in I/O timeouts to study the boundary conditions. Since the performance or throughput metrics at boundary conditions were meaningless (i.e. zero throughput or IO timeouts), we discounted these metrics from our labeled data-set. We ran all the workloads on both servers for different combination of storage cache/metadata size configurations. The different instances were tested both for the homogeneous case (all workloads identical) and the heterogeneous case (workloads with different file sizes and read/write ratios). To collect a wide range of samples (test-data), we ran over 100 combinations of workload and configurations, executed about 990 of test cases, i.e., approximately 9 to 10 different configurations per workload type. During the course of our experiments and data collection, we persisted over 8 million objects and populated over 5.5TB of cloud object store.

A sample of the data collected is shown in Fig. 3.4, with horizontal axis showing various server types and workload executed, and the vertical axis showing the performance classification of various observed throughput metric (across these tests). We terminated the test after getting satisfying prediction accuracy based on the volume of data collected (see Table 3.3).

3.6.4 Data pre-processing and Classification

For each experiment, we created the 'sample' using workload, configuration, compute servers characteristics, observed performance into the feature vectors as {2 cores, 1.2 GHz

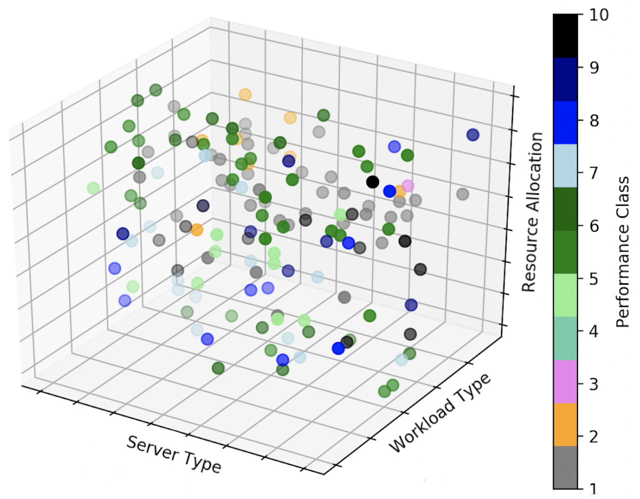


FIGURE 3.4: Sample of Data Collected.

Design Variables	**No. of Classes**	**Example of Buckets or Enumeration**
Core Speed (GHz)	5	1.2, 1.8, 2.4 $\cdots$
Memory Capacity (GB)	5	16, 32, 64 $\cdots$
Storage data cache size (GB)	7	25, 50, 100, 200, 500, 1000, $>$ 1000
Metadata size (GB)	5	25, 50, 100, 200 & 500
Observed Performance	10	Uniform distribution (100Kbps to 350Mbps)

Table 3.4: Sample Classification of Design Variables.

speed, 16GB RAM, 100GB cache, 50GB metadata, 347Mbps network I/O, 108Mbps performance etc.}. We classified the discrete numbers into buckets to give us meaningful insight into the behavior of the gateway servers and configuration. For observed metrics, a discrete throughput data of 488175 bits/s and 21392622 bits/s was classified into bucket sets as throughput $class\ 1$ or throughput $class\ 4$. Similarly a configuration of storage cache or meta-data size of 450GB and 200GB was classified as $class\ 5,\ 4$ and so on. Via such discretization, we classified the data into buckets as shown in Table 3.4. In terms of machine learning, this bucketization means that the regression problem is transformed into a classification problem..

3.7 Predicting Performance of a Configuration

There are well proven algorithms in the field of machine learning and their applicability is specific to the data characteristics and domain [Sorower (2010)]. Further, the efficiency and accuracy metrics of the algorithms depend on *both* the problem domain and associated parameters like learning rate, regularization parameter, etc. We explored a wide range of ML algorithms to find the best fit for our problem domain. We found that Decision Trees fitted the relationship between the performance and configuration parameters with higher accuracy. Using statistical machine learning methods, Decision Trees tries to infer a split of the training data based on the values of the available features to produce a good generalization. The algorithm can handle both binary or multi-class classification problems. The leaf nodes can refer to either of the K classes concerned. It is basically an approximation function working on a multi-dimensional Cartesian space using piece-wise continuous functions. Decision trees have been used in other storage metrics predictions such as [Wang et al. (2004)] where authors exploit the trees for response time prediction of a single disk across different workload parameters.

3.8 Evaluation Results

We present the evaluation results individually for research question about predicting performance (Q.1). We will present the solution design and results on research questions Q.2 in chapter 4.

3.8.1 Influence of Chosen Feature Sets on Performance

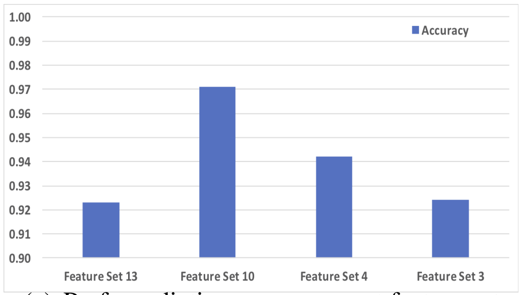
As stated earlier, including the configuration parameters (or feature set) without a proper consideration of their relevance not only makes the model more complex but also interferes with the accuracy of the model. We demonstrate this in the following by studying the performance p as a function of configuration parameters (k, h, r). Fig. 3.5(a) shows the prediction accuracy results for various choice of attributes. In Fig. 3.5(a), Feature Set 3 includes high level attributes $\{k, h, r\}$ (Eq.3.4) and Feature Set 4 includes additional attribute by expanding the resource $\{k, h, ds, md\}$. Performance prediction accuracy using these two limited feature-set is about 93%. Feature Set 10 comprise of $\{cs, nc, mc, bw, di, ar, rs, ms, ds, md\}$, this results in a higher prediction accuracy of 97%. We verified the results by expanding the feature set with additional attributes.

A blind inclusion of more attributes is labelled as Feature Set 13, which includes additional attributes of network bandwidth (nw) and logfile size (ls). These additional attributes add undesired noise in the data and results in poorer predication accuracy (down to 91%). Based on our extensive experience and domain knowldge with Edge Storage, we know that this noise is the result of adding unpredictable network bandwidth (nw) and logfile size (ls), both of which do not contribute to the CSG performance. These results reinforce our earlier comments regarding the selection parameters in Eq. 3.2 and Eq. 3.1.

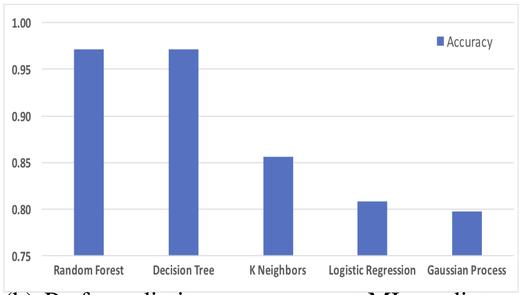
There is no auto-solution for improving efficiency of ML algorithms, as they depend on the application domain, careful selection of attributes (feature set), and hyper-parameters like regularization parameters, learning rate, etc [Sorower (2010)]. Therefore, we tried several types of models and ultimately settled on Decision Tree (DT) for Q.1, as it consistently performed the best (see Fig.3.5(b)). Building a performance prediction model for Q.1 based on Decision Trees yielded an accuracy around 97% for various test-train data combinations (k-fold validation, k=5). An extensive analysis of the model ensured that it does not suffer from under-fit or over-fit.

3.8.2 Confusion Matrix

Another measure for validating our solution efficiency is the confusion matrix. The confusion matrix is a visual representation of statistical accuracy and error of the algorithm. The confusion matrix from one of the test executions in our k-fold validation is shown in Fig.3.6. It gives the metrics with *normalized* numbers (true positive, false positive, false



(a) Perf. prediction accuracy w.r.t feature set.



(b) Perf. prediction accuracy w.r.t ML predictors.

FIGURE 3.5: Accuracy of Various Predictions

negative etc.) for each performance class. While reading the confusion matrix, the diagonal element c_{ij} for i=j should be closer to 1, indicating true values of label (y-axis) i closely match predicted values (x-axis). Incorrect predictions for label $class(i)$ is shown in cells c_{ij}, for $i \neq j$. The figure shows at high accuracy rate in many performance classes, and some mis-classifications are shown for $i \neq j$. Naturally, getting a better efficiency on prediction requires more samples/training data set.

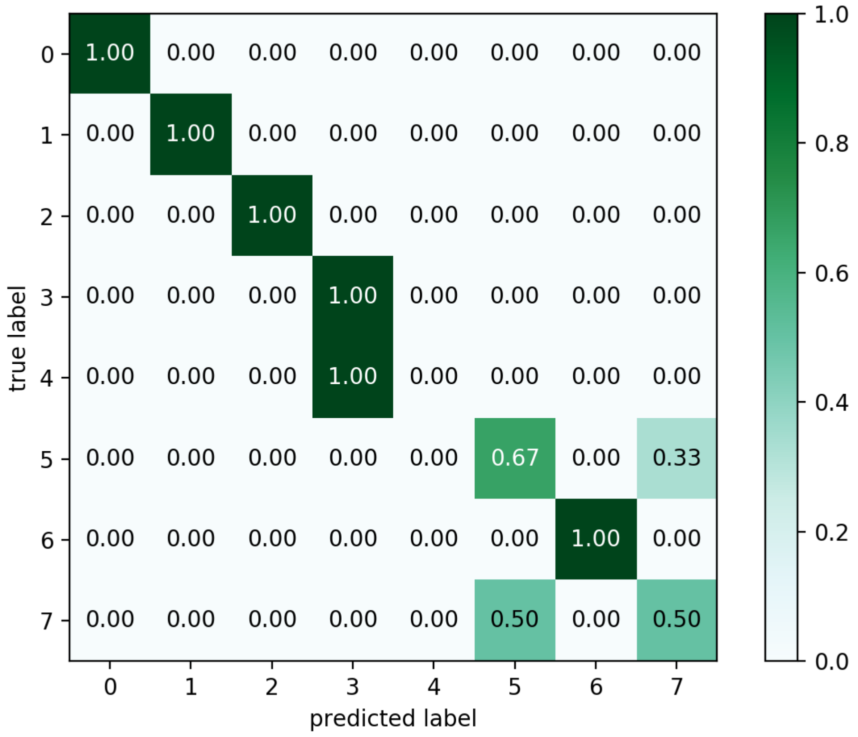


FIGURE 3.6: Sample Confusion Matrix (performance prediction)

3.8.3 Boundaries of Performance Prediction

Prediction accuracy in ML models is largely limited by the diversity of the sample data. ML models are better in finding the answers through interpolation than extrapolation. That is, ML accuracy suffers when the prediction questions are outside the boundaries of the data that the model is trained on. Hence, it is necessary to attach a "certainty factor" alongside the predicted results. Data metrics collected during the experiments are influenced by the hidden factors inside the OSD devices of a data center (box D in Fig. 3.1). One such example is disk rebuilding stress in the storage racks, wherein the RAID based disks have to be reconstructed due to corruption or multiple disk failures. The delays caused by failures and recovery (e.g., rebuilding RAIDs) can impact the performance of the backend OSD systems adversely [Sondur and Gross (2020)] and add to additional IO latency. This will eventually show up as an abnormally low upload performance metric in the CSG. The anomaly caused by such incidents (though rare) has to be filtered and excluded from the data-set and prediction model.

The range of data (workload, server, configuration etc.) considered in our experiments are large and diverse as shown in Fig 3.4. It covers a large range of data-set commonly observed in an Edge scenario. Abnormality on the edge side caused by anomalies (given above) are not under user control and care should be taken to filter these extreme points. We believe that the prediction applies to normal operations, not to perturbations by unexpected/abnormal IO events.

3.8.4 Comparing on Prediction Accuracy

As stated earlier, we are not aware of other public studies on characterizing the relationship between configuration parameters and performance of CSG or other systems, and thus a direct comparison against prior results is not possible directly. However, we compare our results against results from similar techniques used in a different storage context (e.g., performance vs. workload parameters). In particular, the study by Wang [Wang et al. (2004)] using their CART-based models show a relative error between 17% and 38% for response time prediction. Using Inside-Out [Hsu et al. (2016)], Hsu reports a performance prediction error around 9%.

3.9 Conclusion

In this chapter, we present a methodology for the configuration and performance prediction of cloud storage gateway (CSG), which is an emerging system of crucial importance in providing scalable access to remote storage. Because of the large number of configuration parameters and inter-dependencies among them, modeling the influence of configuration parameters on the performance is a challenging problem. We show that machine learning techniques suitably aided by the use of domain knowledge can provide robust models which can be used for predicting performance from the configuration parameters.

We show that the performance prediction Decision Tree model with the right choice of design variables can provide performance prediction accuracies in the range of 5% without requiring large amounts of data. One future extension is to update the configuration incrementally based on the performance data using control theoretic means. Commercial vendors may provide similar gateway solution using a VM-image hosted at their data center. In future work, we will focus on designing of the algorithms and extrapolation studies to recommend a near-optimal configuration for user given workload and performance. Lessons learned from this research can be expanded to the auto-tuning of the hosted gateway solution and back end cloud based object store configurations.

CHAPTER 4

EFFICIENT CONFIGURATION OF COMPLEX CYBER-SYSTEMS

4.1 Introduction

The behavior of all cyber-systems in a data center or an enterprise system largely depends on their *configuration* which describe the resource allocation to achieve the desired goal under given constraints. The ill-effects of misconfiguration (or poor resource allocation) has been widely articulated as unavailability [Newman (2017)], financial burden [Elliot (2017)], security breach [Chirgwin (2017)], etc. However, configuration settings cannot be classified as simply "correct" or "incorrect"; instead, the overall behavior of a device and that of the entire system depends on their setting and interactions between them. In particular, the interactions between various parameter settings and their nonlinear and often nonmonotonic impact on performance rules out simple approaches, such as setting up a convex optimization problem with explicit objective function and constraints, and solving using traditional techniques such as hill climbing. Instead, we exploit machine learning and meta-heuristics that exploits the domain knowledge concerning the problem at hand to determine good configuration settings that satisfy the given objectives. It is also worth noting here that while traditional optimization methods provide a single solution to the problem, multiple solutions are often necessary in practice so that the administrator can choose from them based on considerations that are difficult to formalize.

While the methods explored in this chapter are general and can be applied to almost any configuration problem, the domain knowledge is necessarily problem specific and is often crucial in obtaining sensible solutions. Therefore, we ground the analysis in this chapter by considering the problem of configuring a local storage system backed up by a large remote storage system. The prime example of this is a Cloud Storage Gateway (CSG) [Prahlad et al. (2012)] that couples an amount of locally available storage in a data center with a large but remote Cloud storage to create the impression of essentially unlimited storage. Doing so requires a proper setting of many configuration parameters of the local system as discussed later. Similar situations arise in many other storage contexts such as Network Attached Storage (NAS), Containers (VM image, Dockers), Cloudlets [Satyanarayanan (2017)], Data Storage Service for Edge and Fog [Monga et al. (2019)], etc. In particular, an Edge storage has to contend with many constraints (e.g., space, power, cooling, etc.) in addition to the strict QoS requirements, and thus forms an ideal context for studying the configuration of Cloud backed local storage.

We used a commercial CSG product for conducting our experiments and collecting the data for our configuration study. The details of the experimental data collection and its analysis are presented in chapter 3 [Sondur and Kant (2019)]. This work showed that while it is possible to build an accurate statistical machine learning model (SML) for the problem of predicting performance for a given configuration (hereafter referred as the ”forward problem”), it is very difficult to achieve acceptable accuracy from such a model for predicting configuration for a given performance/cost level (hereafter referred as the ”backward problem”). Furthermore, any such model will realistically apply to the configuration of only a small set of parameters. Thus the approach that we explore in this chapter is to use the forward model as an oracle that is intended to be used sparingly along with metaheuristics to determine multiple ”good” configurations. Our observational study [Sondur et al. (2019)] from the experiments gave us insight to apply relevant domain knowledge to solve the “backward problem.” The meta-heuristics is guided by the domain knowledge so that it does not make entirely random choices and thus can converge substantially faster and yield better results than an unguided meta-heuristics.

In chapter 3, we showed that predicting performance of a complex cyber-system such as a CSG is a challenging task because of complex interaction of a large number of parameters. However, it is possible to use machine learning techniques along with the domain knowledge to learn these relationships well enough to generate accurate performance predictions for given configuration settings. Unfortunately, the backwards problem of recommending suitable configuration for given performance or cost targets remains unsolved, since the typical machine learning techniques are unable to achieve good accuracy and in any case a direct machine learning model would be specific to a particular combination of configuration parameters. This chapter presents an efficient methodology to address the configuration question using an Edge storage system as an example. To the best of our knowledge, prior-art has addressed issues relating to resource provisioning and resource management, while we address the configuration problem as finding (or recommending) the required compute plus storage resource to satisfy a given condition (workload, performance, size, energy, etc.). We also exploit the domain knowledge into meta-heuristics algorithm to reduce the (configuration) search space and converge at the required solution. In designing an approach to find a suitable configuration for a cyber-system, *CyberCon*, we use problem specific domain knowledge to enhance stochastic processes in order to converge at a solution (i.e. configuration state). In CyberCon, we explore both Genetic Algorithm (GA) and Simulated Annealing (SA) based meta-heuristics, and find that SA is generally substantially faster than GA, and can be improved further by using specific annealing functions.

The results from our modified Genetic Algorithm (**mGA**) and modified Simulated Annealing (**mSA**) algorithms show that they reach the maximum fitness (i.e. required configuration) about 22 to 30% faster than the generic versions of the algorithm, henceforth denoted as *gGA* (Generic Genetic Algorithm) and *gSA* (Generic Simulated Annealing) respectively.

4.2 State of Current Art

In a recent survey, Zhang [Zhang and Xu (2020)] categories various techniques in resource management in Cloud and Edge computing as latency optimization, shorter task completion time, container placement, data replication, and Edge caching strategies. Wang [Wang et al. (2017)] presents a survey on the impact of Edge caching capacity, delay, and energy efficiency on system performance in a mobile Edge servers. Meta-heuristics based work has been proposed to solve Cloud resource provisioning problems by allocating applications among the virtual machines to satisfy user QoS [Kumar et al. (2020)]. In CHOPPER [Gill et al. (2018)], authors present a resource scheduling and provisioning scheme based on QoS metrics (execution time, execution cost, energy consumption, and waiting time). Task offloading, workload scheduling, application placement, and migration schemes used in Cloud/ Edge Computing largely pertain to network and computational resource allocation. This flexibility is not available for ESIs as they have to fulfill the workload request "locally and immediately". Our work focuses on *choosing a set of configuration parameters* that satisfy user workload/performance demands under given constraints.

Klimovic and Costa [Klimovic et al. (2018); Costa and Ripeanu (2010)] confirm our observations regarding the difficulty of proper configuration in Cloud storage systems. In designing Selecta, Klimovic address the storage configuration for data analytics workload using TPC traces on block storage devices inside data centers, while our work studies the Object-store based ESI configuration on the Edge side using vendor provided workloads. Costa [Costa and Ripeanu (2010)] state that configuring a storage system for desired deduplication performance is extremely complex and difficult to characterize. Rao [Rao et al.

(2009)] show that a traditional control theoretic framework is inadequate to capture the complexities of resource allocation for VMs. Ofer [Ofer et al. (2018)] study comes close to our work, but their study applies deep learning to object cache eviction/refresh techniques in Object-store, rather than setting configuration parameters.

For the prediction of performance vs. workload parameters of a storage system, Wang [Wang et al. (2004)] used a Classification Regression Trees (CART)-based model and showed a relative error between 17% and 38% for response time prediction. Hsu designed Inside-Out [Hsu et al. (2016)] to predict performance in a distributed storage system by studying low-level system metrics (e.g., CPU usage, RAM usage and network I/O) as a proxy for measuring high-level performance. Compared to Inside-Out performance prediction accuracy of 91%, our pSML model achieves an accuracy above 95%. Cao [Cao et al. (2018)] evaluated few popular black box auto-tuning techniques for storage using Filebench. Both Hsu and Cao comparative study also shows that optimal configurations depend on hardware, software, and workloads and that no one technique is superior to all others. Ularu [Ularu et al. (2013)] use Decision Trees to configure an application and highlight the use of Decision Trees for solving the configuration problem because of the wide solution space to be explored.

In ElfStore, Monga [Monga et al. (2019)] study a resilient Storage service for Edge/Fog computing using similar workload with IO Block size (1 to 10MB) and show the importance of meta-data operations. Their metadata operations have latency around 120ms, and data operations have read/write latency of 1.4 to 6.5sec. Their work does not advance into finding a suitable configuration or resource allocation for satisfying a QoS. Cachier [Drolia et al. (2017)] is a caching model designed to minimize latency between the Edge and the Cloud. Cachier finds that an increase in the storage cache size need not lower the latency, but in fact it increases latency after a certain threshold. They show that storage cache size is an effective "tuning" knob used to minimize the latency of requests in image recognition applications (both supporting our work). Their research does not include configuring the Edge resources needed for such work. Similar to our work, authors in [Satyanarayanan (2017)] show that managing dispersed Cloudlet infrastructure is very challenging because of the many unknowns pertaining to the software mechanisms and controls. The survey by Sitton [Sittón-Candanedo et al. (2019)] shows that the need for real time response and very low latency in the Edge is challenged by constraints such as limited storage, interconnected protocols, network latency, high power consumption, etc.

4.3 Configuration Management of Edge Storage (ESI)

Given the importance of exploiting the domain knowledge in addressing the configuration management problem, it is important to understand some architectural details of the cloud storage gateway (CSG) that we experimented with and analyzed extensively for this work in the context of edge computing (henceforth, referred as Edge Storage).

4.3.1 Overview of Edge Storage Infrastructure

Edge Storage Infrastructure (ESI) shown in Fig. 4.1. It provides a *local storage buffer* to bridge the gap between the high throughput demands of the latency sensitive edge applications and the low/unpredictable network connectivity to the Cloud. ESI connects the edge-applications to an Object-store on the Cloud because of the inherent advantages for Object-store in the Cloud model. Edge client applications operate using small blocks of data (of 4KB or 16KB size) at SCSI speeds, shown as (A) in Fig. 4.1. On the Cloud side, marked as (D), ESI has to interface with Cloud Object-store over an unpredictable network (C) (low throughput, high latency) with *varied object sizes* that are largely dependent on the applications. In addition, ESI should address reliable communication such as IO block acknowledgements or retry an error-ed IO block on the SCSI side and acknowledge or re-fetching the entire object on the Cloud side. To address this imbalance, ESI has to satisfy key requirements such as (i) protocol translation from/to SCSI to/from http/REST based services, (ii) map 4KB/16KB block IO requests to Object-store APIs for objects of varied

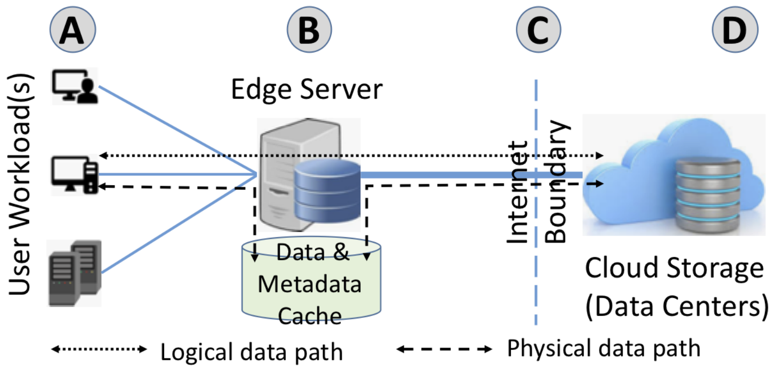


FIGURE 4.1: Workflow/ Data Path in Edge Storage Infrastructure.

size (or vis-versa), (iii) satisfy high throughput/low latency edge-client request over a low throughput/high latency connection to the Object-store Cloud service, (iv) manage storage overheads like security, meta-data management, reliability, rotating log file, garbage collection, etc.

4.3.2 Data Flow in an ESI

For a read operation, depending on the state of local storage data cache, ESI can either service the data locally or fetch the data from the Cloud Storage. ESI can achieve high read performance by pre-fetching data and associated metadata, but at the cost of occupying storage data-cache space. A write request from the edge application has to be persisted successfully to the Cloud Storage, first by persisting it locally on the storage data cache, and then transferring it successfully over to the Cloud. Both such read and write operations to the Cloud are limited by the amount of cache space (data cache plus meta-data area) and the unpredictable network bandwidth, and overhead operations such as garbage collection, cache-eviction, meta-data operations, etc. However, the edge clients are transparent to these internal-details and view the ESI as a *virtual extension of the Cloud*. This virtualization is shown as the “logical data path” in Fig. 4.1.

4.3.3 Importance of Meta-Data

Although the Cloud storage could be block based, it is almost universally object based due to many advantages of Object Storage in the Cloud model [Gracia-Tinedo et al. (2017)]. In an Object-store system, every object is associated with corresponding metadata that is maintained by a metadata server. When an object is initially requested by ESI, this metadata also must be brought in from the Cloud. It is generally desirable to keep the metadata longer than the data so that if the object is evicted and then re-requested, the ESI can avoid small IOs associated with metadata accesses. However, a proper balance must be maintained between the space allocated to the data and metadata for optimal performance. Thus, both the workload access pattern and metadata management determine the performance experienced by the user. As part of our study in ESI configuration recommendation and resource allocation, we will explore the space allocated for data cache and metadata cache.

4.3.4 Configuration Modeling for ESI

ESI performance is defined as the throughput experienced by the edge-clients, and is generally expressed as MB/s [Oracle (2010)]. ESI architecture involves the complexities inherent in storage systems, storage data cache allocation, satisfying IO demands, and unpredictable network bandwidth [Tanimura and Koie (2015)]. Mathematical modeling of the behavior of an ESI is difficult because of the complex inter-dependencies between the numerous parameters and the poor understanding of these relationship and their impact on the overall performance.

ESI consumes computational resources for protocol translation, client authentication and capability based storage access management, meta-data operations, workload interfaces (block IO, Object-store APIs), etc. Thus, platform parameters such as core speed, number of cores, memory size, and memory bandwidth all become crucial. Satisfying

Attribute	**No. of Classes**	**Example of Buckets or Enumeration**
Core Speed (GHz)	5	1.2, 1.8, 2.4 $\cdots$
Memory Capacity (GB)	5	16, 32, 64 $\cdots$
Storage data cache size (GB)	7	25, 50, 100, 200, 500, 1000, $>$ 1000
Metadata size (GB)	5	25, 50, 100, 200 & 500
Observed Performance	10	Uniform distribution (100Kbps,350Mbps)

Table 4.1: Sample Classification of Design Variables.

latency sensitive edge-client IO request needs data buffering, intelligent cache operations, etc. that demand cache resources (both storage and memory). The limited storage resource in an ESI has to be efficiently partitioned for storage data-cache, meta-data, and other operational overheads (e.g. swap space, log files). Unpredictable network connectivity to the back-end Cloud raises additional challenges in cache eviction, refresh vs. prefetch, efficient bundling of object requests, exploiting workload patterns, etc. The inter-dependencies and complex behavior of these parameters (e.g. CPU, memory, storage cache-space, etc.) make accurate and tractable analytic modeling of performance very difficult.

The throughput (p) experienced by the edge-clients depends largely on the workload (k), ESI hardware (h) and the resources (r) allocated to the compute and storage layers of ESI. An improper choice of these parameters will result in a poor experience by the end user such as IO timeouts (rejected requests) or poor throughput (low performance) or large unacceptable latency. To address the difficulty in the detailed analytic characterization of the above parameters, we formulate the above parameters as a classification problem. Although, in theory, most parameters can take a large range of values, practical limitations, and sensitivity considerations usually confine the feasible values to a small set of discrete values. Table 4.1 provides an illustrative example in this regard.

Let nc denote the number of cores, cs the core speed, mc the memory size, bw the memory bandwidth, and di the disk IO rate. Also denote ar as the request arrival rate, rs

the request size, and ms the metadata size. We then propose the following functions to represent the classification of hardware h and workload k:

$$h = f_1(nc, cs, mc, bw, di) \tag{4.1}$$

$$k = f_2(ar, rs, ms) \tag{4.2}$$

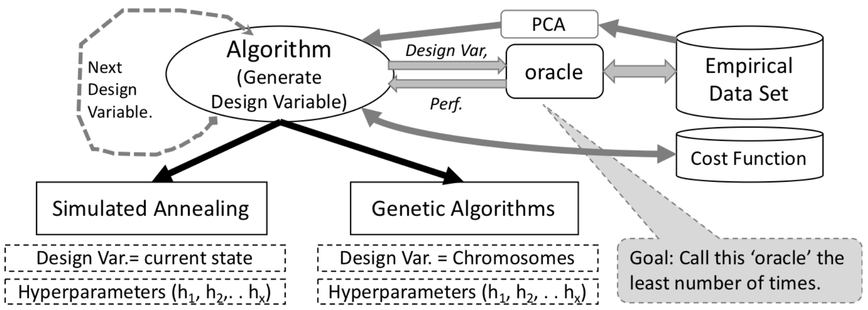
Note that in postulating these functions, we have included only a subset of the parameters that could potentially be relevant. This again is based on domain knowledge, since simply throwing in arbitrary platform parameters may actually dilute the model and lead to worse results, as presented in section 3.8.1. In particular, we did not include in h other architectural details such as size/speed of L1, L2, L3 caches, since practically the choice would be limited to certain models of hardware platform from a given vendor. Also, while some parameters (e.g., the DRAM speed) could be selectable (with in an appropriate range), their level of influence does not warrant their consideration. This aspect necessarily involves the use of domain knowledge. Simply throwing in as many parameters into the model as possible can be self-defeating both in terms of data requirements for training the model and in diluting the model with weak dependencies that are difficult to characterize.

In addition to the hardware and workload characteristics, the performance achieved by a workload class also depends on the storage resources allocated to it. In particular, the total space r allocated to a workload class is simply the summation of storage data-cache size db, meta-data size md, and log size ls. (Obviously, r should be less than the total space available). Since the log size ls does not play a significant role in performance, we will ignore it here.

We can now express throughput p in terms of workload class w, ESI hardware class h and resource allocation class r as:

$$p = f_3(h, k, r) \tag{4.3}$$

Research Questions: The forward and backward problems introduced earlier could now be concretely defined as follows:


Next Design Variable.
Algorithm (Generate Design Variable)
Design Var,
Perf.
PCA
oracle
Empirical Data Set
Cost Function
Simulated Annealing
Genetic Algorithms
Design Var.= current state
Hyperparameters (h1, h2,. . hx)
Design Var. = Chromosomes
Hyperparameters (h1, h2, . . hx)
Goal: Call this 'oracle' the least number of times.
FIGURE 4.2: Design of Algorithms.

RQ.1 Predict the performance p under given workload k, hardware h and resource allocation r.

RQ.2 Recommend an optimal configuration Ψ, i.e. hardware h **and** resource allocation r to satisfy the given workload k & performance p **and** user defined constraints.

4.4 Solution Design

The complex relationship between various user settable parameters (compute capacity, storage resource etc.) and vendor provided latent parameters (cache eviction rate, data replacement logic, meta-data operations, etc.) makes analytic models intractable. Therefore, we used machine learning techniques to learn the various relationships that influence the outcome (performance). We provide detailed analysis for RQ.1 and the proposed solution in the chapter 3. The performance prediction model in section 3.7 is used as an *oracle* to solve the configuration question in RQ.2. We explain the solution design using the illustration in Fig. 4.2. Given a set of design variables, the oracle can predict the performance based on the machine learning model explained in section 3.7. The oracle is trained on the empirical data collected from various experiments. Additional domain knowledge through Principal Component Analysis (PCA) and cost function is provided as supporting functions to the algorithms. We explain the two stochastic processes, the methods to generate design variables, and embedding the domain knowledge approach below. The hyper-parameters of the algorithms are explained in section 4.5.2.

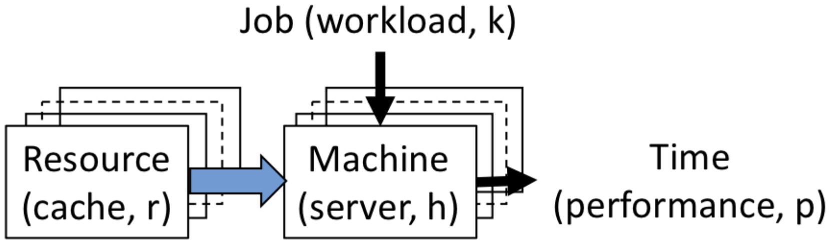


FIGURE 4.3: Illustration of Research Questions.

Predicting a satisfying configuration parameters for RQ.2, based on user workload and target performance is difficult since it involves determining a large set of complex inter-dependent variables that satisfy the given condition. Research question RQ.2 can be explained using the Fig. 4.3, and raising a question on “which combination of machine h and resource r would give satisfy the user criteria p, given the workload k?”.

Besides, there could be more than one solution that satisfy the required condition. That is, there could be various combinations of hardware (CPU, memory, etc) and resource (storage data-cache size, meta-data size) that satisfy the user given workload/ performance under given constraints (e.g. minimum heat dissipation, size).

Our solution (i.e selected configurations state Ψ) should satisfy the user given condition (i.e. performance p_{user}) at a minimum cost possible. We define a constraint function as:

$$p \geqslant p_{user} \tag{4.4}$$

where p is the expected performance from configuration Ψ. The objective to find such a configuration Ψ at a minimum cost.

$$\min(cost(\Psi)) \tag{4.5}$$

The objective can be expressed as the deployment cost, power consumption, cooling requirements, etc. The cost function is represented as the *normalized* cost of a configuration Ψ based on the design variables. For example, k^{th} configuration Ψ_k for some choice of hardware h_i and resource r_j, is represented as $\Psi_k = \{cs_i, nc_i, bw_i, \cdots db_j, md_j, ..\}$ and has a cost $cost(\Psi_k)$. Data for cost function can be derived from vendor specification for hardware server and allocated resources (disk capacity). We approached the above question RQ.1 and RQ.2 in two phases: (i) a statistical machine learning (p-SML) model to predict the behavior (e.g. performance) of the system, and (ii) a stochastic process guided by the domain knowledge to recommend a suitable configuration that satisfies given goals and constraints for user given workload/performance.

*4.4.1 Modified Genetic Algorithm (**mGA**)*

For brevity, we refer readers to available literature on GA [Ferentinos et al. (2002); Kerr and Mullen (2019); Rashid et al. (2013)] and highlight our modified Genetic Algorithm approach. The generic pseudocode [Rashid et al. (2013)] for GA is presented in Algorithm 1. The principles of GA is based on evolutionary biology with the goal of optimizing a user given fitness function. The GA allows a population of inputs (line 1) to evolve over time (line 3) to maximize this fitness function (line 4). Stop condition is represented as number of epochs or iterations. The fitness corresponds to how well an individual performs; i.e. inputs with the highest fitness are desired. GA uses a fitness function to choose the best candidates to generate the next breed of population. The best candidates generate the next generation of children by random mutation and cross-over. Each element, or gene, of every chromosome is mutated with a probability defined by the mutation rate (line 5). The design variables in Eq.4.1 and Eq.4.2 form the genes in the chromosome in GA. Efficiency of the algorithm can be perceived in multiple ways, for example as the quality of the output chromosomes (defined by the fitness function) or how fast the optimal solution is obtained (defined by number of epochs to reach the solution).

Our modified Genetic Algorithm (mGA) aims at narrowing down the large search space and output multiple satisfying solutions which can then be filtered for the user defined objective. GA algorithm defines the design variables (Eq.4.1 & Eq.4.2) as a population, that is evaluated for fitness and then undergo random cross-over and mutated to derive at a new state. Each population is represented by a chromosome that maps to a

Algorithm 1: Generic pseudocode of a Genetic Algorithm [Rashid et al. (2013)]

```
1 initialize the population;
2 evaluate population;
3 while (!stopCondition) do
4     select the best-fit individuals for reproduction;
5     breed new individuals through crossover and mutation operations (*);
6     evaluate the individual fitness of new individuals;
7     replace least-fit population with new individuals;
```

design state, (i.e. a set of design variables). Our solution uses the performance prediction model from RQ.1 as the fitness function to determine if the current state (i.e. chromosome) satisfies the user required performance (Eq.4.4). *p-SML* performance prediction oracle is consulted to predict the performance of each chromosome (configuration state or design variable).

Adding Domain Knowledge to mGA: To achieve higher efficiency, a good solution would result in a smaller number of calls to such an oracle. Instead of default random mutation in GA (at line 5), our approach aims to intelligently mutate the chromosome population to jump to the new state in a controlled manner. We used additional insight from principal component analysis (PCA) objects derived from the p-SML model to control the gene mutation probability (marked as **(*)** in line 5 in Algorithm 1). We explain the metrics from PCA objects, probability factors, selection of principle design variables based on features importance in evaluation section 4.5.4. Our mGA solution with an "informed" approach can converge to the required configuration Ψ at least 22% faster than the baseline GA.

*4.4.2 Modified Simulated Annealing (**mSA**)*

SA can be used to approximate the global minimum for a function with many variables. We defer to available literature for detailed discussions on SA [Ben-Ameur (2004); Ingber (1989); Zhao et al. (1996)]. The pseudocode [Ferentinos et al. (2002)] for SA as given in Algorithm 2. In SA, a state refers to a set of design variables and a neighboring state refers to a set of values “relatively” closer to current design variables. In SA, energy is represented as the cost function that has to be minimized [Ben-Ameur (2004)]. An acceptable “state” is a solution to the problem that is being solved.

Algorithm 2: Pseudocode for Simulated Annealing [Ferentinos et al. (2002)]

```
1  initialize(temperature T, random starting point);
2  while (coolIteration $\leqslant$ maxIterations) do
3      coolIteration = coolIteration + 1;
4      select a new point from the neighborhood (*) ;
5      compute currentCost(at this point);
6      $\delta$ = currentCost - previousCost;
7      if $\delta \leqslant 0$ then
8          accept neighbor;
9      else
10         accept with probability exp(-$\delta$/T) ;
11     T = $\beta$ * T (*) ;
```

The SA method has been widely used since the cost function can be easy to put into practice [Ferentinos et al. (2002)]. Similar to GA, our SA algorithm uses design variables from Eq.4.1 and Eq.4.2 to represent the state. The entropy of the system is defined as the cost of the current state (i.e cost of the configuration $cost(\Psi)$). The SA steps in Algorithm 2 can be summarized as: (i) we first start with an initial annealing temperature (T_0) and a random design state (line 1), (ii) we search for the next state depending on annealing temperature T_k and a random distribution (line 4), (iii) we compute the difference in entropy (δ) between the current state and past state (line 5,6), and probabilistically accepting the current state depending on Boltzmann probability factor (line $7 \cdots 10$). The annealing scheme is defined as the temperature gradient (line 11). The algorithm stops after reaching a cooling temperature (line 2).

Our solution is based on very fast simulated annealing (VFSA) presented by Xu [Xu et al. (2018)], that enhances both the annealing temperature (line 11) and the perturbation model (line 4). Zhao [Zhao et al. (1996)], Ingber [Ingber (1989)], Lee [Lee (2015)] and others have discussed VFSA in detail and show the advantages over standard SA. To speed

Group	**Design Attribute Pairs**
Group G1	Number of Cores, Memory capacity
Group G2	Core speed, Memory bandwidth
Independently varied	Storage Data Cache, Disk IO rate

Table 4.2: Grouping Design Variables

up the convergence rate of SA, VFSA uses Cauchy distribution function [Lee (2015); Zhao et al. (1996)] as the perturbation. This perturbation model is able to realize a narrower search as the iterative solution approaches an optimum solution, which accelerates the convergence speed [Xu et al. (2018)]. Our modifications to SA is shown as **(*)** at line 4 & line 10 in Algorithm 2. We discuss the supporting functions of SA & mSA in Table 4.3 using the following notations: k is the current iteration, n is the number of design variables, T_0 is the initial annealing temperature, α is the damping coefficient ($0 < \alpha < 1$), μ is a uniform random variable between 0 and 1, $(B_j - A_j)$ is the range of j^{th} design variable $(0 \leqslant j \leqslant n{-}1)$, Ψ_i is the configuration (design variables) at i^{th} state, c_i is the configuration cost at i^{th} state, p_i is the predicted performance of configuration Ψ_i at i^{th} state, and p_{user} is user given performance.

mSA uses the same annealing scheme and Cauchy distribution perturbation model from SA (Table. 4.3, entity T_k & ζ). For acceptance probability ρ, mSA makes a slight modification to accommodate the case where the next solution has the same performance but lower cost. If the acceptance probability for the current state is 1, a new random state is chosen (to avoid getting stuck in local minima) else a state in the neighborhood is chosen.

Adding Domain Knowledge to mSA: We incorporate domain knowledge in the algorithm by dividing the design variables into groups based on their level of interdependencies. That is, the design variables within a group show strong interdependence and thus should be set collectively, where the settings across groups can be done independently. In theory, such grouping can be done purely using clustering techniques, but this would require a large amount of data and may still result in some unexpected groups. However, by applying the domain knowledge, the grouping can be either done entirely manually, or by coercing the clustering algorithm to prefer certain groupings over others. For example, in the context of a computer system, it is well understood that a faster CPU should be paired with a faster DRAM, else the CPU will simply stall waiting for the memory. A faster disk is also important, but much less so, since the IOs involve a context switch whereas a memory access does not. Similarly, more CPU cores doing independent work will likely need more memory, and for workloads involving remote IO, both network and IO speeds must increase in tandem. *Grouping* of configuration variables based on insights avoids exploration of states that are unlikely to useful and thus is expected to both speed up the convergence and lead to better solutions within a given number of iterations.

Note that the entropy in a given state s_i is calculated as the cost of the configuration in state s_i, i.e cost(Ψ_i). In the generic SA algorithm, acceptance of a solution is defined by Boltzmann probability factor (line $7 \cdots 9$), and represented as a (negative) exponential function of the entropy change δ. Since the negative exponential function tends to zero very rapidly as the entropy increases, it explores only a very small neighborhood in the vicinity of the current solution. We thus choose a function with a longer tail, and found that the square function works quite well, i.e., better than cubic or linear function (Table. 4.3, entity δ). The evaluation results from the changes to the entities in generic SA and mSA is discussed in Fig. 4.6 and Fig. 4.7.

4.5 Evaluation Results

4.5.1 Implementation Details

Details of the experiments is given in the previous chapter (section 3.6). We implemented the algorithms in Python using *scikit-learn* [Pedregosa et al. (2011)] library for Machine Learning components such as Principal Component Analysis, Classifiers, ML metrics (e.g. accuracy, precision, etc.), Feature Importance etc. For stochastic algorithms, we used *NSGA-II* [Deb et al. (2002)] Genetic Algorithm from Platypus library [D (2019)].

Entity	**Generic SA**	**Modified SA (mSA)**
Annealing temp T_k	$T_0 * exp(-\alpha(1-k)^{1/n})$	
Entropy change δ	$exp(c_i - c_{i-1})$	$(c_i - c_{i-1})^2$
Acceptance Prob. ρ	1, if $p_i \geqslant p_{user}$ $\frac{-(\delta)}{T_k}$,otherwise	1, if $p_i \geqslant p_{user}$ & $c_i \leqslant c_{prev}$ $\frac{-\delta}{T_k}$,otherwise
Perturbation Model ζ_j	$T_k(\mu - 0.5)\Big[\Big(1+\frac{1}{T_k}\Big)^{\lvert 2\mu-1\rvert} - 1\Big](B_j - A_j)$	
Select neighbor (state s_{i+1})	$\begin{cases} random\_new\_state(), & \text{if } \rho = 1 \\ \zeta_j, & \text{otherwise} \end{cases}$	
Design Variables	Individually varied	Varied as a group
Valuation Results	Fig 4.6	Fig 4.7

Table 4.3: Very Fast Simulated Annealing Functions

NSGA-II algorithm (discussed by Deb et al. in [Deb et al. (2002)]) gives the flexibility to define fitness function, define objectives and constraints, variable bounds, chromosome construction, crossover and mutation, solution-set, etc. Simulated Annealing with supporting functions for neighborhood search, annealing scheme, Cauchy distribution, etc. was implemented in Python.

4.5.2 Hyper-parameters of the Stochastic Process

In a stochastic process like GA or SA, there is no way of knowing a priori which hyper-parameters will secure an optimal solution search [Kerr and Mullen (2019)]. For example, hyper-parameters like the size of the initial population, size of fit candidates, mutation probability, etc. describe a GA, and initial annealing temperature T_0, and damping coefficient α describe a SA process. For GA & mGA, we set the initial population of 110 with the tournament selector to choose the top 8 "fit candidates" for next generation mutation. For SA & mSA, our initial annealing temperature T_0 is set at 500 and damping coefficient α at 0.23. To enable meaningful comparison of different algorithms, we choose to count the number of calls to *performance prediction oracle*, and as explained earlier a good solution would result in a smaller number of calls to such an oracle. The count of calls to the oracle by different algorithms is shown as normalized iterations (y-axis) in the results.

In this section, we discuss experimental results for both the forward and backward prediction problems discussed earlier. For the latter, we present the two independent results based on the two domain knowledge enhanced meta-heuristics processes. mGA and mSA are compared with their respective baseline algorithms, i.e generic GA and standard SA. Results for Simulated Annealing in section 4.5.5 are further divided based on enhancements shown in Table 4.3. In the results diagrams below, we present a subset (of 10 results) from the different test cases executed. Finally, we discuss the execution times observed from these experiments in section 4.5.7.

4.5.3 Extracting Feature Importance

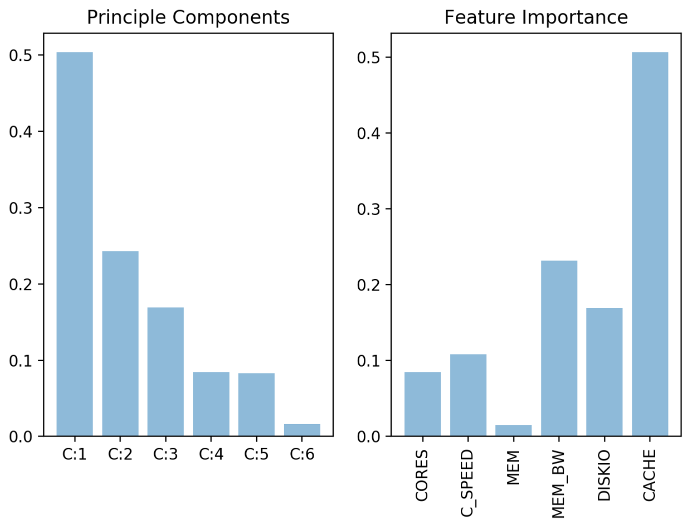


FIGURE 4.4: PCA and Feature Importance.

Principal Component Analysis (PCA) is a dimensionality reduction technique that projects the data from its original p-dimensional space to a smaller k-dimensional subspace. PCA maximizes the variance accounted by the first k components and thereby attempts to include those components that have the most influence on the output. The k-dimensional subspace considered by PCA involves components that are linear combinations of the original variables; therefore, we still need to identify the most relevant original variables. In PCA terminology, the contribution of each variable to each principal component is described by *Loadings* [Legendre and Legendre (2012)], which can be easily extracted. Large loadings (positive or negative) indicate that a particular variable has a strong relationship with a particular principal component. The sign of a loading indicates whether a variable and a principal component are positively or negatively correlated.

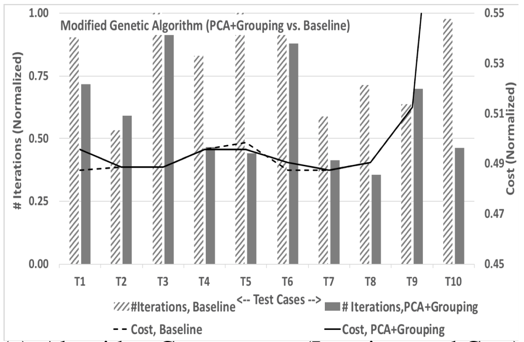
Feature ablation is a technique for calculating feature importance (FI) that works for all machine learning models. A feature with a high importance has a greater impact on the target variable. We compared both FI from the DT model and PCA & Loadings from the PCA objects to gain confidence in ranking the predominant attributes that contribute to ESI performance. The *scree plot* of PCA and FI for our data-set is given in Fig. 4.4. In the figure, the left sub-graph shows PCA values for different orthogonal components ($C1 \cdots C6$) on x-axis, and the right sub-graph shows FI values for the design variables (on x-axis). Based on the above metrics, PCA & FI provided a reasonable metric to understand the variance of a parameter and its relative contribution towards the performance. Instead of randomly mutating the set of genes to generate a new population set (i.e. new configuration state), we focused on a deterministic way to control the cross-over and the mutation process. We used the above metric from PCA and FI to probabilistically mutate the genes in the modified (PCA+GA) mGA approach and generate a 'controlled' new state. The results of our mGA solution is given below.

Next, we discuss the solution for the backward problem (BP), relate to finding the near-optimal configuration Ψ that satisfies a given workload/performance criteria (Eq. 4.4) at minimal possible cost (Eq. 4.5). We compare the functions against baseline and validate how quickly an algorithm converges at such a required configuration state.

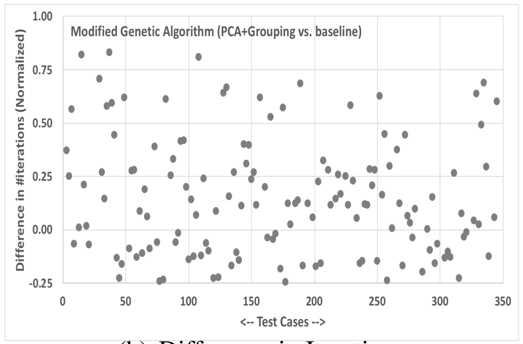
4.5.4 Recommending a Configuration using mGA

The design variables (CPU, memory, IO bandwidth, etc.) plus the workload properties (file size, meta-data, no. of files, etc.) forms the chromosome in the gene pool that represents a population. Note that during mutation, we do not vary the workload variables (*ar,rs,rm*) as these are user given properties for predicting the required configuration. The fitness function (FF) defined by the Decision Tree from section 3.8.1 selects a sub-set of the population (design variables) that satisfy the user defined performance. To ensure efficiency, this performance predicting oracle has to be consulted sparingly for rapid convergence.

In gGA, the design variables (i.e. gene pool) are randomly mutated to get to a new state (i.e. new population set). The population set is continuously evaluated for fitness and the *best fit population* is selected as a suitable solution (i.e. population with predicted performance equal to user defined performance). An uncontrolled mutation may result in design variables being randomly selected from a wide range and this may result in finding a suitable solution after a considerable time (measured as the number of calls to oracle). Our goal is to enhance the gGA algorithm to intelligently mutate the gene pool such that the desired solution (i.e. fitness function) is reached faster (i.e. less number of iterations).



(a) Algorithm Convergence (Iterations and Cost)



(b) Difference in Iterations

FIGURE 4.5: gGA and mGA (PCA+Grouping) Test Results (Iterations and Cost)

We extracted additional data from ML objects, PCA model, and feature importance (FI) as explained in section 4.5.3. A high FI metric relates to a high relevance of the variable towards the output. For example, Fig 4.4 shows that data-cache value of 0.506 has the highest relevance to the final ESI performance. We used this *data relevance* to probabilistically mutate different genes. Using data from Fig. 4.4, gene representing data-cache undergo mutation with 0.506 probability, and the gene representing core speed undergo mutation with 0.108 probability and so on. This disciplined mutation allows the mGA process to move to a new state (i.e new population set) in a controlled fashion. Design variables with lesser influence tend to settle down quickly and the influencing variable (data cache, memory bandwidth) span 'within a limited' range searching for a satisfying solution (i.e. user desired performance). This intelligent control of gene-mutation results in reaching the solution-set faster (i.e less number of iterations).

Genetic algorithms results in a 'multiple solution-sets' that satisfy the fitness function, which can be further refined or filtered for desired results. In our approach, the solution set should satisfy the user given constraints (Eq.4.4), normally at a minimum cost (Eq.4.5). In our implementation, we keep track of the number of iterations required to "find" a satisfying configuration such a minimal cost. The algorithm "records" the number of calls to the oracle needed to obtain the most satisfying configuration state (chromosome in solution set) at a "minimum" possible cost and time. Fig. 4.5(a) shows the normalized values of such convergence (iterations) for both mGA and gGA algorithms for various test cases (T1$\cdots$T10) along with the cost function for the solution (Ψ). This figure show a subset of 10 test cases (x-axis) from large number test cases we executed. For example, test case T1 is a query to suggest an optimal configuration for: Workload W5, class: Medium with small meta-data, Perf class:1 (i.e. Large Workload: 10,000 files of 1 MB size, 5 users, Required Perf.: 150MBps). The mGA based approach converges to a solution after 225 iterations with a solution: 2 cores x 3.2GHz, 16 GB Mem, 3.2GB Mem bus, DiskIO = 10K IOPS, Normalized Cost = 0.4875. The same query to a gGA takes about 333 iterations to find a minimum cost solution. The test cases T1...T10 in the Fig. 4.5(a) refers to the first ten dots in Fig. 4.5(b), each dot representing a test case pair (difference between generic GA vs modified GA).

The test scenarios are sequential test cases starting with workload W5 to W10, with perf class P1 to P10. Let "i" represent 'any' random dot (test case) and "i+1" refers to the adjacent dot (test case) in the figure. For example, test case T_i refers to workload W6 perf class P2, next test case T_{i+1} is for the same workload W6 and next perf class P3., and so on till we reach last test case T_n for workload W10, perf class P10. There will be some test cases with 'no solution found' referring that no configuration satisfies a desired perf. class P_j for workload k_l, e.g It is not possible to achieve a perf class P7 for workload W6.

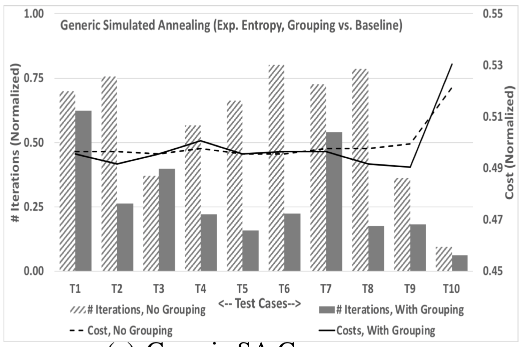
Fig. 4.5(a) shows the test cases executed on the x-axis and the normalized number of iterations and costs on the y-axis. (The normalization is wrt 500, which is the maximum number of iterations.) The lines for the costs refer to the minimal cost of solution (Eq.4.5) obtained by both mGA and gGA. As seen in Fig. 4.5(a), with a controlled gene mutation in mGA algorithm, the design variables find the satisfying fitness function (performance) faster at the same minimum cost in less number of iterations. The cost of the configuration from mGA algorithm matches the minimal cost obtained in the gGA algorithm.

Fig. 4.5(b) shows a more comprehensive comparison between gGA and mGA. The metric on y-axis is [#iterations(gGA) - #iterations(mGA)]/500, and the x-axis is simply the 400 cases that were run with different parameters. Note that the dots above the x-axis (positive values) show that mGA performs better than gGA, whereas negative values show the opposite. It is clear that mGA outpaces gGA in almost 90% of the cases. Please note that because of the inherent randomness in the state space search in GA and SA, there is no expectation that any technique will perform better all the time. Furthermore, in the cases where mGA performs better than gGA, it takes 22% fewer iterations than gGA on the average.

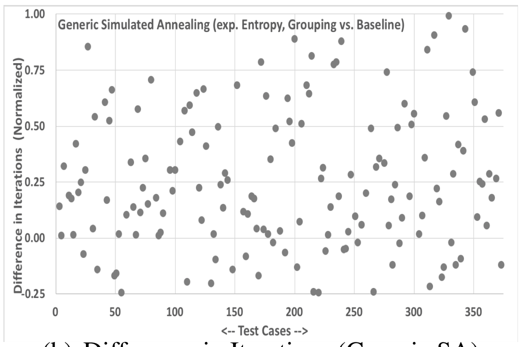
4.5.5 Recommending a Configuration using mSA

Similar to GA, the design variables (CPU, memory, IO bandwidth, etc.) and the workload (file size, meta-data, etc.) formed the "state" of the system in SA. Obviously, the workload variables (*ar,rs,rm*) are not changed. Initially, we start with a random design variables to represent a configuration state Ψ, and consult the DT from section 3.8.1 to verify if the current configuration state satisfies the user defined performance (Eq. 4.4), and the cost of configuration state is computed.

In our mSA approach, we have two variations of choosing the design variables either by domain based grouping or individual perturbation (Table. 4.2) and two variations of annealing functions (Table. 4.3). We use gSA as a *baseline*, and present the effect of grouping design variables in Fig. 4.6. We used the same test cases T1$\cdots$T10 shown in GA above, and similarly the y-axis represents the normalized iterations (calls to oracle) required to reach the minimum cost satisfying configuration (Ψ). Fig. 4.6(a) shows that a solution with grouping design variables discovers the satisfying configuration faster than a standard approach. The minimal cost of the discovered configuration validates that grouping design variable algorithm (in most cases) matches (or better) the minimal cost compared to gGA (without design variable grouping). In Fig. 4.6(b), we show more comprehensive results, with each dot referring a test case with different parameter. The figure clearly shows that mSA with grouping outperforms the gSA in about 90% of the test cases. Each dot on the positive side refers to the case where the mSA algorithm reaches the solution faster than gSA.



(a) Generic SA Convergence

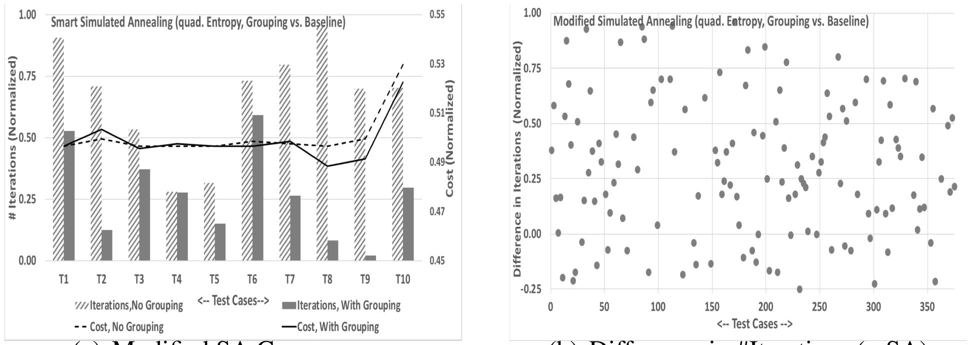


(b) Difference in Iterations (Generic SA)

FIGURE 4.6: Comparing gSA Test Results (Iterations and Cost) with/without Design Attr. Grouping

We now focus on mSA algorithm with modifications as explained in section 4.4.2. While the baseline gSA computes the delta entropy δ based on exponential function, mSA algorithm computes the same entropy as a quadratic function. Again, mSA is compared based on two criteria, (i) with individual perturbation of each design variable, and (ii) perturbation of design variables as groups (Table.4.2). The evaluation results of mSA is shown in Fig. 4.7 for same test cases T1 · · · T10 and y-axis shows iterations as a normalized metric. In each of these test cases, mSA with grouping of the design variables performs better compared to non-grouping, and reaches the desired configuration faster (see Fig. 4.7(a)), and simultaneously matching or outperforming the minimal cost function.

Similar to GA and SA cases, we summarize the results from several other tests in Fig. 4.7(a), with each dot representing a test case on the x-axis and the normalized difference in #iterations on the y-axis. As seen from the figure, mSA outperforms gSA in 90% of the cases. For test cases where mSA performs better than gSA, it takes 32% fewer iterations than gSA on the average, i.e. on an average, mSA reaches the required solution about 32% faster than gSA.

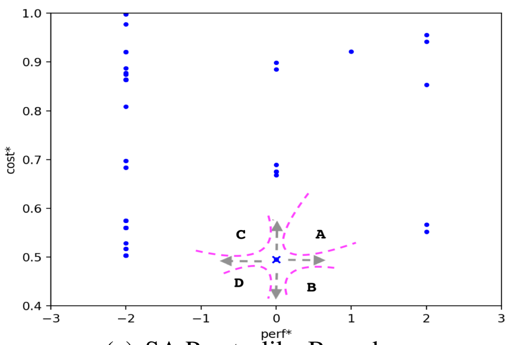


(a) Modified SA Convergence (b) Difference in #Iterations (mSA)

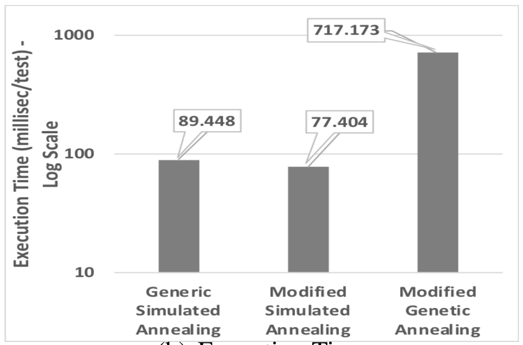
FIGURE 4.7: Comparing mSA Test Results (Iterations and Cost) with/without Design Attr. Grouping

4.5.6 *Pareto "like" Boundary*

Our interest in this work is to satisfy a user constraint on given performance, rather than trying to solve for two objectives (i.e. a maximum performance and minimal cost). A design is considered Pareto optimal [Chinchuluun and Pardalos (2007)] if there does not exist any other design which improves the value of any of its objective criteria without deteriorating at least one other criterion. Using *similar concept*, we present a Pareto "like" boundary to show that the end results satisfy constraints (Eq. 4.4) and the objective (Eq. 4.5).



(a) SA Pareto-like Boundary



(b) Execution Time

FIGURE 4.8: Pareto-like Boundary and Execution Times

Pareto "like" boundary from one of our test cases is shown in Fig. 4.8(a), with performance class on the x-axis and cost on the y-axis. Each point in the graph refers to a configuration point from the mSA algorithm. Our optimal point is shown as "x" in the figure, with *A,B,C,D* showing other possible solution "areas". Any point towards A would refer to the required performance (or higher), but at higher costs - hence not a desired solution. Any points towards C or D is undesired since it refers to lesser than desired performance. *If the algorithm found* any points in B (other than x), it would be desirable since it refers to both satisfying performance and lesser costs. However, a rightful solution like mSA would find the most optimal solution **at** point x, and no points in B area.

4.5.7 Comparing execution time

We executed all the algorithms for several test cases and captured the execution time. All metrics were captured under identical conditions, i.e. the tests running on the same server with minimal overhead from unwanted OS processes. The average execution time per test case is shown in Fig. 4.8(b). The data shows that mSA is about 10% faster than SA, and mGA is considerably slow. Similar findings is reported by Keer [Kerr and Mullen (2019)] and Ferentinos [Ferentinos et al. (2002)], who found that for each iteration of genetic algorithm is considerably more expensive than simulated annealing.

4.6 Conclusion

In this chapter, we present *CyberCon*, an efficient goal oriented methodology to recommend an optimal configuration for a complex cyber-system. Though our proposal is generic enough to be applicable for other domains, we demonstrate our technique using an ESI, which is of crucial importance in efficiently supporting Edge services in the highly resource constrained environment. Assessing the impact of various parameters on an Edge computing system is very complex because of the unknown influence of the parameters or their complex inter-dependencies. We proposed a meta-heuristics based approach aided by domain knowledge (in this case Genetic Algorithm and Simulated Annealing) and machine learning techniques (Decision Tree). We have shown that the proposed approach can robustly identify suitable values of configuration parameters that satisfy the target performance and deployment constraints. In designing our *CyberCon* methodology, we embed problem specific domain knowledge to quickly reduce the search space. Such an approach can intelligently jump to a new design state by exploiting the domain knowledge regarding the behavior of performance as a function of various parameters (e.g., monotonicity or unimodal behavior). The results show that the approach yields results closer to the optimal value about 22 to 30% faster than the standard uninformed algorithms.

CHAPTER 5

A CONFIGURATION HEALTH SCORING SYSTEM AND ITS APPLICATION TO NETWORK DEVICES

5.1 Introduction

It is well recognized that inconsistent changes to configuration parameters and IT production environments are the predominant causes of system outages or performance problems [Barroso et al. (2013); Yin et al. (2011)]. Past studies have indicated that up to 80% of down-times of mission-critical applications are caused by mistakes, miscommunications or misunderstanding related to configuration changes [Connolly (2014)] and up to 85% of performance incidents can be traced to such changes [Cappelli (2015)].

An inappropriate setting of the configuration parameters of a module not only affects that module but also others as well because of the complex dependencies. Furthermore, configuration parameters usually cannot be classified into simple "correct" or "incorrect" categories; instead, the overall behavior of a module and that of the entire system depends on their setting and interactions with settings of other parameters. It is clear that with a large number of components and increasingly complex computing infrastructures, *assessing the health* of the system becomes extremely difficult. Thus the focus of this chapter is to see how a health index for a system can be defined and related to the configuration parameters. Although much of our discussion is generic, we will primarily focus on network devices and evaluate our approach using data from some commercial routers.

5.2 Current State of Art and Challenges

As stated earlier, CVSS concept is closest to our HI proposal. CVSS attempts to assign severity scores to security vulnerabilities, which allows the security experts to prioritize development and application of suitable security patches and workarounds. Human intervention is a critical part of CVSS process since CVSS relies on human assigned scores or weights to express the severity of the problem. A domain expert examines the discovered problem and assigns weights for different CVSS parameters. Overtime, CVSS results in a large repository of vulnerabilities scored appropriately to aid end users in identifying and fixing vulnerable components. CVSS repository can be further used to refine the accuracy of the scores, classify vulnerabilities and predict exploits and situational awareness [Bozorgi et al. (2010); Edkrantz and Said (2015)].

Many scholarly articles [Yin et al. (2011); Xu and Zhou (2015)] address novel methods in localizing, trouble-shooting and detecting configuration problems. There is also rich literature on misconfiguration detection [Herodotou and et.al (2014)]. Most of the related works address configuration problems after any ill-effect are detected. Metrics like CVSS [Ross et al. (2017)] measure the security vulnerability of the device or system, but do not address the configuration itself. Cao [Cao et al. (2017)] has empirically expressed the vastness of the configuration space to be explored, using NFS configuration as an example. The network device configurations exhibit similar complexity.

Benson [Benson et al. (2009)] only concerns routing. It assigns a complexity score to a configuration file based on the number of lines in the file and a description of the functionality based on interviews of administrators. Our approach attempts to automate the process by a more fine-grain consideration of the impact or importance of the configuration. Their work on router configuration 'complexity' is the closest to our work, and the authors present a 'qualitative' metric that is limited to analysing the routing and access rules of network devices. Similar to CHeSS, they use human inputs to evaluate the metric based the contents of a static configuration file. Their work captures the difficulty of adding new functionality such as interfaces, updating existing functionality etc. Our work computes the HI attributes for web-services, network devices, applications, storage clusters, etc. Benson's work is specific to routing and does not address the configuration issue in general.

Ontology related work has been used to translate multi-vendor configurations into a common format. With a restricted goal, Ngoupé [Ngoupé et al. (2015)] used ontologies to design a generic model to express the configuration of a Cisco router. Ontologies have been used [Martinez et al. (2015)] to extract the semantics of a device configurations. Wong [Wong et al. (2005)] attempted to express Cisco and Nokia configurations with an ontological mapping. These ontology related works highlight our concern and the need to translate multi-vendor configurations into a common canonical format.

Recently, OpenConfig [OpenConfig working group (2016)], a forum supported by many networking industry vendors, has come up with a set of vendor-neutral data models for configuration of parameters that are supported natively on networking platforms. OpenConfig uses YANG, a Data Modeling Language for the Network Configuration Protocol (NETCONF), defined in RFC 6020. Though each vendor is likely to have their own device models for the parameters, and they support OpenConfig model by providing an appropriate function to translate the device specific configuration model to neutral OpenConfig data model and vice-versa. For CHeSS, OpenConfig provides a readily available solution and a well-accepted canonical data model for representing network device configurations. An industry forum like OpenConfig lays the foundation for other domains to represent a vendor neutral configuration model, although currently there is no effort to extend it to other entities.

There is no existing metric or methodology to elicit Health Index (HI) in the field of computer networking. Often, the relationship between two or more interrelated configurations needs to be explored. One restriction is the limitation of the manufacturer's documentation spelling out every combination and the effects therein. Most of this knowledge is human intensive, and varies widely with human skill and experience levels.

Network Management Systems (NMS) [Cisco Systems, Inc (2019)] housed in Network Operation Centers (NOC) are used for configuring and monitoring the nodes in a managed network. However, they focus on tracking the operational state of the network nodes following the occurrence of network events rather than quantify the health of configuration parameters. Our approach focuses on apriori analysis of the configuration and provides a framework to compute the HI of a device configuration.

5.3 Formulating a Configuration Health Index

Configurations are generally represented as a name-value tuple [p,v] with dependencies across parameters (and hence their values), but the dependencies are rarely expressed explicitly. These dependencies may be implied by the naming convention or semantics of the parameter, or explained in documents intended for administrators or subject matter experts, and very difficult to deduce automatically. In this chapter, we introduce a novel concept called **CHeSS**: a ”Configuration Health Scoring System” that discovers, measures and quantifies the Health Index of a device. CHeSS can quantify the health of the device configuration (usually specified through a configuration file) to aid the administrators, engineers or decision makers (collectively referred to as users) to understand the strength or weakness of the configuration and take necessary actions. The goal of our research is to define a framework to quantify the ”health-index” of a device based on the configured values of various parameters. With additional work, HI can be extended to relate to the ground truth and build capabilities to flag potential execution-time anomalies.

CHeSS was inspired by the well-known Common Vulnerability Scoring System or CVSS [Forum of Incident Response and Security Teams (2017)], which is a free and open industry standard for assessing the severity of security vulnerabilities. CHeSS takes a fundamentally different approach than CVSS, and is not merely a multi-dimensional extension of the CVSS concept. The key difference is that, at its root, CHeSS is *not* driven by reported real or potential anomalies in the behavior of the device, but rather by a model of the configuration parameters and how they affect the health attributes. This makes the scope of CHeSS very broad (i.e., semi-automatically relating configuration settings to health scores) and yet specific to help users to evaluate the impact of the configuration.

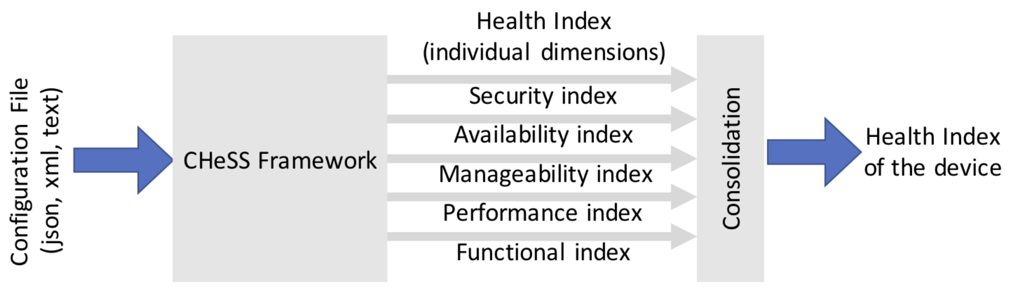


FIGURE 5.1: A Framework to express the Health Index of a Configuration File

The Health Index (HI) metric of the configuration file is expressed as a vector of impacted attributes such as: security (S), availability (A), manageability (M), performance (P), and functionality (F). That is,

$$\vec{HI} = \{S, A, M, P, F, \cdots\} \tag{5.1}$$

The computation of HI requires analysis of various configuration parameters on the different attributes. CHeSS framework shown in Fig. 5.1 takes the configuration file as input, converts it to a canonical form, analyses the configuration parameters and quantifies the HI metric. The elements of $\vec{HI}$ metric are the key representation of the device behavior under the given configurations. A framework such as CHeSS can be applied to any IT asset including computer servers, VM images, storage, switches, etc.; however, in this chapter we focus largely on network devices.

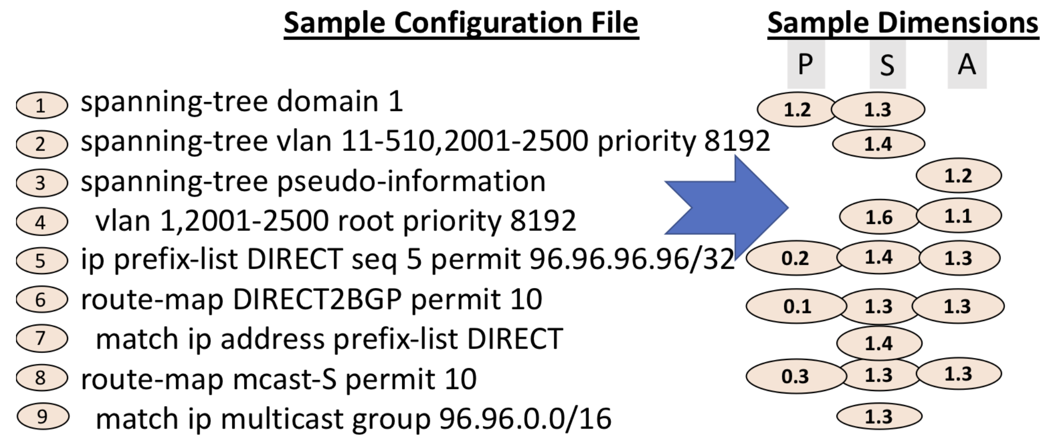
Each attribute above can be regarded as a vector of suitable *measures*. For example, the performance measures of interest will typically include one or more measures of throughput, latency, and loss. For example, in the networking context, we may be interested in bytes/sec and packets/sec (throughput measure), send and receive latencies, and packet drop fraction. Given the precise meaning of these measures, it is most meaningful to characterize them first and then suitably define what we want to call as an overall performance metric. This added granularity allows users to peel-back to more specific issue(s) if any problem is observed (e.g., poor security can be peeled back to weak encryption).

Using practical production level configuration files, we explain our design, computation principles and compute the health index HI. We show that our approach allows users to confidently express the behavior of the device as a metric (HI). With CHeSS, we lay a foundation that can be extended to encompass other attributes such as energy consumption. We validate our approach by applying the domain knowledge and verify its applicability from domain experts in the industry. In the past, there has been considerable interest in the industry in developing such a metric; however, we have not found any prior art in this area to give a comparative analysis.

HI can be used in a large data center or an enterprise to filter or group devices for user defined criteria. For example, administrators can use HI to query configurations with security below acceptable threshold. Network designers can identify the weakest performance link in the path using HI. HI can potentially be used in isolating boundaries (e.g. Virtual Machines, Containers or other sandboxes) and to investigate if these boundaries are being breached. In this chapter, we discuss a range of application and feasibility of HI.

5.4 Estimating the Health Index from a Configuration

This chapter provides a quantitative framework to analyse the configuration of a network device and predict its dynamic behaviour. We try to bring some explainability and simplicity to the vast configuration space explained earlier. We present our framework 'CHeSS' in general terms, and explain its applicability by exploring the HI and configuration of a network router.



```
1  spanning-tree domain 1
2  spanning-tree vlan 11-510,2001-2500 priority 8192
3  spanning-tree pseudo-information
4    vlan 1,2001-2500 root priority 8192
5  ip prefix-list DIRECT seq 5 permit 96.96.96.96/32
6  route-map DIRECT2BGP permit 10
7    match ip address prefix-list DIRECT
8  route-map mcast-S permit 10
9    match ip multicast group 96.96.0.0/16
```

FIGURE 5.2: Sample Configuration File with Associated attributes and weights

5.4.1 *Configuration Files and Objects*

In the following we discuss this formulation with a running example where the HI consists of three attributes, namely performance P, security S and availability A. We also key our discussion to Fig. 5.2 that shows the fragment of configuration file for a Cisco router. The left side circled numbers in Fig. 5.2 are statements from a sample configuration file, and the right side weights indicate whether the configuration statement contribute to the particular attribute. In the following discussions and depending on the context, i represents the i^{th} statement, or i^{th} object, or i^{th} weight, etc.

Obviously, the HI vector consists of three elements, i.e., $\vec{HI} = \{P, S, A\}$ in our example. We regard a properly canonicalized configuration file as a sequence of *Configuration Objects*, with i^{th} configuration object (denoted CO_i) represented as a (name, value) pair; i.e., $CO_i = \{p_i, v_i\}$. Each configuration object can contribute to one or more attributes of the device. For example, a configuration statement *"passwd myEncrYptedPaSSw0rd encrypted"* could contribute to functionality, security and performance. While this configuration statement could make the device secure, it also adds to performance penalty as the transaction has to undergo a encryption/ decryption task. Thus, we need to assign the relative contribution of a configuration object on each attribute. That is, for configuration object CO_i, we can define a weight vector $\vec{W}_i = [w_{i1}, ..., w_{ik}]$ where k is the number of attributes (e.g., $k = 3$ in our example of performance, security and availability). Each weight w_{ij} can be thought of as the correlation coefficient between the j^{th} attribute and the i^{th} configuration object.

w_{ij} is constrained by an upper bound (in our example, $+2$) and a lower bound (a small weight $\varepsilon > 0$), and a mid-point (say 1) to denote default behavior. That is, $w_{ij} \in [\varepsilon, +2-\varepsilon]$ where $w_{ij} = 1$ means that the attribute j is not impacted by CO_i, and increasing magnitude of w_{ij} represents increasing positive correlation. Configuration objects that are not explicitly defined would take the manufacturer defined internal default value and result in a default weight $w_{ij} = 1$. A configuration object i that results in the maximum possible value of the attribute j would take the maximum weight, i.e for some k^{th} object, $CO_k = \{p_k, v_k\}$, $w_{kj} = 2 - \varepsilon$. The weights are scaled version of the attribute value, and the scaling function is dependent on the configuration object.

In general, any measure w_{ij} would depend not only on the configuration of the relevant entity but also on the overall environment that it operates in. Thus, for example, the actual and maximum throughput of two web-servers that identically configured, but part of different cyber infrastructures, could be different. In other words, both the default performance and the maximum performance could be different. Since the HI concerns only the configuration of the device in question, we would like to keep such differences out of HI estimation. That is, regardless of the actual performance in our web-server example, we will assign $w_{ij} = 1$ to default performance and $w_{ij} = 2 - \varepsilon$ to the maximum performance.

There could be a large number of configurable objects for a given device, and the exact number depends on the manufacturer. To ease administration tasks, maintainability and ease of configuration, manufacturers provide an implicit default value that could result in an optimal operation. Administrators generally provide explicit values for a limited set of configuration objects (say about 10 to 20$\%$), with remaining objects taking default manufacturer values. Hence, only the administrator modified objects contribute to the weight $\vec{W_i}$ and the remaining non-defined objects take the manufacturer provided default value (i.e. $w_{ij} = 1$). The computed $\vec{HI}$ would be relative to the '*bare-bones* (i.e. out of the box) configuration comprising of the default values, and represent either the strength or weakness of the configuration.

The key challenge is to estimate values of w_{ij}'s, and given the complexity, it would necessarily be based, at least partly, on manual input provided by knowledgeable administrators. A manual input is to be expected for a qualitative scoring system, just as it is in CVSS. However, a significant challenge is to minimize the manual input, and yet enable a consistent and accurate assignment of the weights. We discuss some avenues for this later but largely as topics for further exploration on the subject. This necessarily requires associating each attribute with a suitable set of concrete measures, as discussed in section 5.1.

A sample of weights of the configuration objects is shown on the right side of Fig. 5.2. The circled markers on the left indicate the configuration objects (or statements). Each row is a vector $\vec{w_i}$ representing the *contribution* of a single configuration object CO_i, and each column represents an element in the attribute vector $\vec{HI_i}$ (a vector with element corresponding to different attributes).

Using the above notations, we define a framework for computing the health index of a configuration file. Applying in-depth systems knowledge and domain expertise from network administrators and manufacturers, we study the HI of a network router (i.e. configuration file of a router).

5.4.2 Object Representation

This is the first attempt to define the health index of a configuration file. We used and applied publicly available manufacturer's specification, network domain knowledge and system administration expertise. Keeping a flexible design, we ensured that our approach can be refined to include any newly acquired knowledge. At a later date, if needed, the HI attributes can be expanded into other factors (e.g. risk factor, energy consumption, etc).

We start with the following preconditions:

1. Each configuration statement is considered an individual, independent object CO_i.
2. Each configuration object CO_i contributes to one or more attribute in $\vec{HI}$. The contribution of a configuration object CO_i on the attributes is bound by a lower and upper limit, as explained earlier.
3. An object CO_i may have a hierarchy of k additional sub-statement or sub-configurations ($CO_i = CO_{i1} \cdots CO_{ik}$). The sub-statements and hierarchy of values of CO_i should be recursively considered to derive the final HI of the parent object CO_i.

The first case treats each configuration statement (e.g. ip prefix-list $\cdots$; line 5 in Fig. 5.2) as an independent object. We address inter-dependency between config. objects as follows. If one or more statements (i.e CO_i & CO_j) contribute the same attribute (e.g. security), then the weights (say: w_i, w_j) given to each statement are relative to each other (e.g: security weight of: CO_i = 1.3; CO_j = 0.7). Such relative weights will positively compensate if CO_i and CO_j add the overall attribute or negatively compensate to bring down the effective weights. The system is designed to gain knowledge from several such examples (explained in section 5.5.3). For the last case, if multiple sub-statements of a configuration file define a configuration object, then we compute the contribution of individual sub-statement and roll up the computed values hierarchically to the parent object.

For $line$ 1 in Fig. 5.2, spanning-tree object definition is defined by a total of four sub-statements (lines $1, 2, 3, 4$); we assign the weights individually to each sub-statement and roll up the final value to the parent object. The hierarchical representation 'spanning-tree' statement(s) of the sample configuration file in Fig. 5.2 is shown as a canonical tree structure in Fig. 5.4. The parent *node* spanning tree object CO_i has several layers of child sub-objects below it. Each path from the root node to the leaf node represents a unique statement in the configuration file. The final weight of the object CO_i is the rolled up weights from its immediate child sub-objects and so on. An example of a complex hierarchical object from one of our production configuration file is shown in Fig. 5.7.

5.4.3 Quantifying the attributes

We have established that each top-level configuration object CO_i has an associated weight $\vec{w_i}$ representing the object's contribution to the attributes $\vec{HI_i}$. The overall metric (or quality of each attribute in HI) is represented as the geometric mean of the all the weights $\vec{w_i}$ of its child objects CO_i. The HI of the root object (i.e. entire configuration file) for attribute j is given as (for all attributes j):

$$HI_j = \sqrt[n]{\Big(\prod_{i=1}^{n}(w_{ij})\Big)} \tag{5.2}$$

This equation is recursively applied to each object and sub-object (if it includes a hierarchy), and the final computed value is applied to the root. The attribute values of the root node defines that of the configuration file (and in turn the network device).

5.5 CHeSS Framework

The design methodology and the framework used to compute the HI is illustrated in Fig. 5.3. The configuration file is transformed into a canonical format in step (1) and presented to the *CHeSS* framework. CHeSS computes the HI based on the data in its

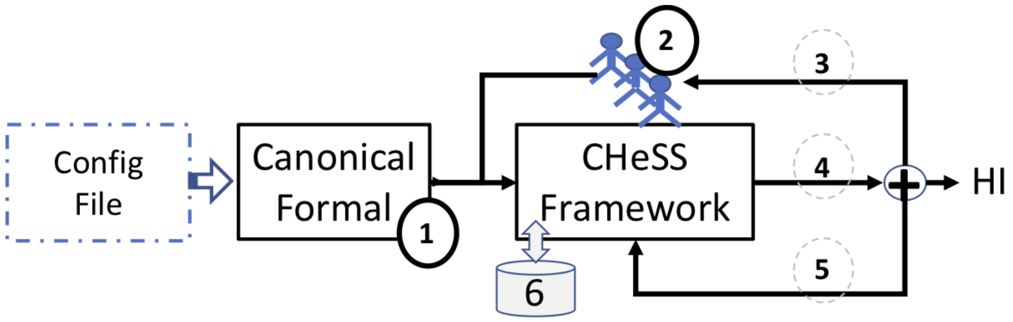


FIGURE 5.3: CHeSS Framework with Knowledge Repository and Learning Model

repository (box 6) or inputs from the experts (shown as 2). The computed HI (step 4) is fed back to the CHeSS framework through feedback (step 5) and stored in its repository for later re-use. It also serves as an input to experts for ratification (step 3).

5.5.1 Configuration File and its Canonical Form

A configuration file is manufacturer and device (model, version) specific and may be represented in various formats, such as json, xml, text or binary file. The contents of the configuration file are the configuration statements, typically expressed as parameter-value pairs. A configuration file would contain several configuration statements, which individually and collectively impact the overall HI of the device. In a typical deployment, many of the configurations remain set to their default values. This default configuration helps the users to deploy the network fast and also ensures baseline performance, security and availability. Thus, only a subset of configurations need to be customised to deployment topology. This smaller subset of configurations require manual input from human experts for weight assignment, thus reducing the burden and dependency on human-input for entire set of configurations.

Administrators do not handle all the configuration objects, and only modify a sub-set of the objects that are related to their tasks. A configuration statement may contribute to one of the metrics with a gain (positive effect) but negatively contribute to another metric (penalty). The configuration parameters (or rather the objects represented by the parameters) are often hierarchical with sub-configurations up to arbitrary depth. Some of the complications related to configuration files include configuration statements spread over multiple lines, not explicitly configuring default values, using vendor specific parameter names or specification syntax, etc. Sometimes configuration files may be incomplete, have errors, such as specification of nonexistent parameters. In such cases, configuration files are deemed invalid. Furthermore, configuration files are volatile and continue to evolve. It is even possible that the administrator makes some changes to the resource configuration manually but this change is not reflected in the configuration file.

In order to keep our mechanisms general, we assume that the original configuration file is converted to a canonical form that removes these idiosyncrasies. This canonical form has the following properties: (a) all parameter names and values are specified in a standard way, (b) each configuration statement is contained in exactly one line, and (c) the entries in the file correctly represent the real configuration.

5.5.2 Ontologies, OpenConfig and CHeSS

Clearly, translating the original vendor file into a canonical file is nontrivial, as it would require a clear schema definition of each vendor file and a mechanism to translate it to the canonical form. As stated earlier, OpenConfig [OpenConfig working group (2016)] provides an attractive solution for the networking domain and can be used directly by CHeSS. Shaikh et al. [Shaikh et al. (2016)] use OpenConfig model in their work on vendor neutral SDN network representations for transport. Though, at present, OpenConfig models represent only a subset of the device models, the industry consortium is actively working to represent most of the device models most effectively. For resources other than network, the application of OpenConfig will require significant extensions to the YANG language and is outside the scope of this chapter.

Device specific configuration file is first translated to a canonical model that represents all the configuration objects. In our example of a canonical model, each path from the root

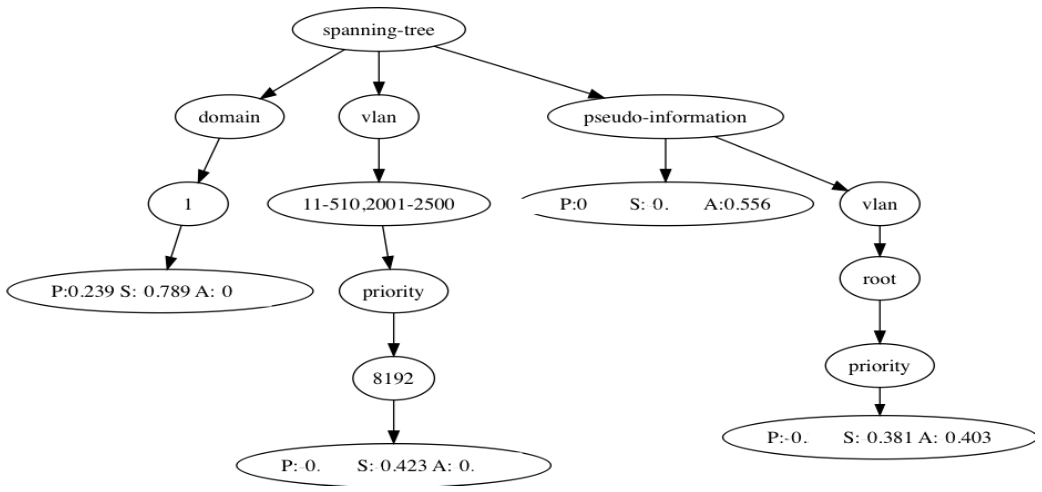


FIGURE 5.4: Parsing the Hierarchical Object and Assigning Weights (P,S,A)

node to the leaf node represents a unique statement in the configuration file. Configuration statements $\{1 \cdots 4\}$ of Fig. 5.2 are transformed to a tree structure shown in Fig. 5.4, with configuration object CO_i *spanning-tree* as the root, and all the sub-objects as its children. Each unique path from root to leaf node represents a configuration statement. The hierarchical configuration statements (if any) are represented as additional paths and sub-layers. An example of weights w_{ij} for the sample hierarchical statement CO_i = *spanning-tree*, along with its child objects and unique paths is shown in Fig. 5.4.

If a given top-level CO_i, contains any hierarchical sub-objects (as shown in Fig. 5.4), the weights of all the sub-objects are recursively computed to derive the $\vec{HI}$ of the topmost object CO_i as given in section 5.4.2. In this example, the parent *spanning-tree* node $\vec{w}_i$ is recursively computed from the values in its child nodes (and so on).

It is possible that during translation to a canonical model, CHeSS may encounter some configuration objects that lack a canonical representation. Such limitations can be overcome by consulting a human for a relevant canonical model. Human-in-the-loop (HIL) machine learning approach [Zhang et al. (2019)] can be used alongside ontology representation to translate the device configuration file to a canonical format. HIL can be designed to parse a file based on machine learning approach and probabilistically consult a human operator to learn new formats, parameters, and parsing logic.

5.5.3 Learning Model

As mentioned earlier, the assignment of weights w_{ij}'s is both challenging and crucial for proper interpretation of health index. We assume that these weights are assigned by domain experts, who validate the name/value pair of each CO_i and quantify its contribution to the health index. This is shown as step (2) of CHeSS framework in Fig. 5.3. To reduce the dependency on human assigned weights, we propose a two step approach: (i) a transfer function to map the observed operational data (a.k.a oper-data) measured to the device HI and (ii) to probabilistically consult human experts to learn new weights (explained in next section).

For the first approach, we propose to identify a set of operational parameters that best describe the health index identified for the device. These operational parameters can be measured in Network Operation Centers (NOC) and quantitatively assigned to the devices. For example, oper-data like CPU occupancy and throughput can describe the performance HI. Similarly, security HI parameters can be described by number of security incidents and availability HI parameters can use observed metrics like number of system failures, duration of service unavailability, MTBF (Mean Time Between Failures) etc. For each such operational parameter, we calculate the HI estimates based on the assigned weights of configuration attribute. The measured value is then compared against this estimate and the difference is fed back to calculation of HI with readjusted weights.

For example, observed oper-data Ψ_i on a device d_i in an environment ξ_k may indicate four security incidents, 620MBps performance throughput and 99.99% availability. This oper-data is transformed to $\vec{HI}$ and represented as:

$$\vec{HI} = \{\text{P,A,S}\} = \{1.3,\ 1.2,\ 1.9\}$$

Note that $\vec{HI}$ is relative to default value of +1, with increasing numbers denoting the strength and decreasing numbers showing weakness.

Over time, as HI is computed for various configuration files, CHeSS builds up a large repository of various CO_i and relative weights $\vec{W}_i$. This knowledge repository is shown as (box 6) in Fig. 5.3. For any objects CO_i missing in the repository, CHeSS consults the human expert (box 2) to manually assign the relative weight $\vec{w}_i$ for the newly encountered object. After computing HI, the feedback loop (step 5) feeds the CHeSS repository with more data for future consultation.

This learned data (learned from human experts or transfer function) is stored in a knowledge repository shown as box 6 in the figure. As the collection of configuration objects (CO_i) increases, the CHeSS framework can auto-assign the weights ($\vec{w}_i$) from its knowledge repository. As knowledge grows, the delta error (step 5) between human assigned values (step 3) and auto-generated values (step 4) should diminish. We can adapt an unsupervised learning model wherein clustering can be used to improve the reliability of the weights. One such approach is outlined in the following section.

5.5.4 Enhancing the Reliability Of Weights

Learning the relative weights of different objects reliably requires a large number of configuration files. This is difficult to do manually; furthermore, only a small number of configuration files may be available within an organization for specific type and model of device. To address this, we propose collecting configuration files for a larger set of parameter combinations from domain experts and populate the knowledge repository. Assuming this is feasible, CHeSS framework can, over time, learn weights of each CO_i and its relative contribution to the HI with greater confidence. As the data points increase, we can build additional intelligence to learn the dependencies between objects, identify any anomalies, etc.

However, there are many difficult challenges in such an endeavor. First, in spite of the harmonization of the configuration file syntax through the canonical representation, the differences between devices from different vendors or those with different models could still pose challenges in estimating weights. Second, there may be relationships between the weights of different objects, but the perception of these relationships across domain experts could vary. Third, assigning definitive values to the weights can itself be challenging, particularly for attributes such as security, and the expert opinions on the quantification and its fuzziness may vary. These limitations exist whenever a human input is solicited, as opinions may differ. Benson [Benson et al. (2009)] has addressed this hurdle through 'operator-interviews' and using 1000s of device configurations available in enterprise data centers.

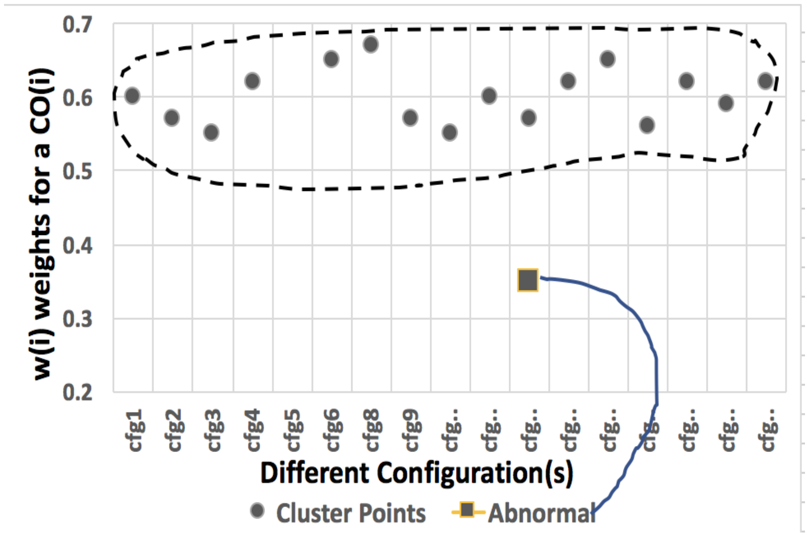


FIGURE 5.5: Learning w_i from Samples and Clustering for a Given CO_i

One simple approach for sanity check of the weights provided by the experts is to use clustering methods and corresponding outlier detection (Fig.5.5). Clustering algorithms can be used to group the weights w_i's of a CO_i for further analysis.

As the collection of configuration objects (CO_i) increases, the CHeSS framework can train a neural net N to learn the relative weights. The details of neural net N is out of scope of this chapter, though we envisage that a regression based machine learning model can address this easily. While processing a new CO_i for a new configuration file, the CHeSS framework can auto-assign the weight (w_i) from the neural net N based on the knowledge

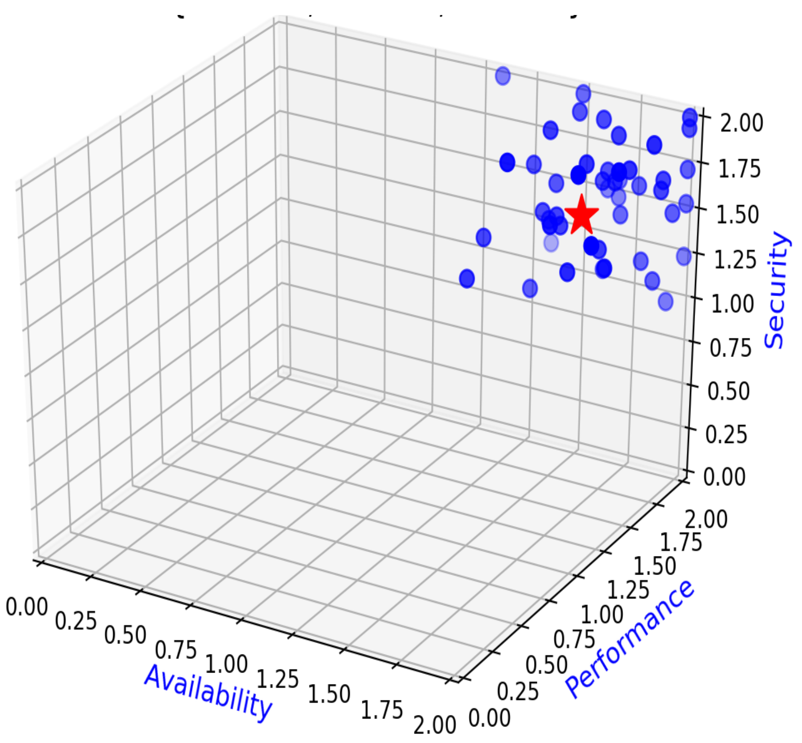


FIGURE 5.6: 3D representation of HI of Sample Configuration File

repository. The framework can consult a human expert (step 2) to learn about new weights w_k for a newly encountered configuration object CO_k. As knowledge repository grows, CHeSS tries to consult the human expert *sparingly* and attempts to decrease the error between human assigned values and auto-generated values. This interaction between human and machine learning approach can enhance the accuracy of prediction and enable two way learning. In such cases, HIL will help human experts learn about new deployment models (or new configuration types).

5.5.5 Visual Representation of Health Index (HI)

We calculate the metric $\vec{HI}$ in Eq. 5.2 of the topmost layer: the root object. As said earlier, this final value defines the overall HI of the configuration file. The configuration health index can now be represented as a point in a n attribute space; with each axis representing a device characteristics. The visual representation of the computed HI values for a sample file is shown in Fig. 5.6. Visually, this HI indicates that it is marginal on performance, nominal on availability and security. HI of our sample configuration file is represented as:

$$\text{HI} = \{\text{P,A,S}\} = \{+1.665, +1.594, +1.518\}$$

On closer inspection of this sample configuration file, domain experts (the system administrators) confirmed that these metrics represented the health/ quality of the sample configuration.

Individual small $round$ dots represent the w_i health index for all top-level configuration object CO_i (i.e. immediate children of root object). These are scattered around the 3D space, indicating their relative contribution on the health index, i.e. these points show how each CO_i affect the three attributes (P,A,S). A clustering of the small dots (CO_i) can show the reasons for the HI biased towards a metric (say biased towards poor performance). Similarly, clustering of the individual $\vec{HI}_i$ in a given quadrant would indicate potential tilting or overarching impact of the configuration parameters in one direction. Users can see the relative position of the HI marker and quantify/ express the HI in simple human understandable terms (high performance, weak security etc.). Based on the relative position of the individual blue dots, these CO_i are the primary candidates for closer inspection and configuration modification.

Obviously users need a health index that indicates high performance, high security and high availability. These values may be bound by some upper limits and counter each other. An optimal value may be bounded much below this. Finding the upper bounds and optimal value for $\vec{HI}$ is a topic for future research.

5.6 Evaluation Results

We briefly describe our implementation and present our evaluation results using few real world configuration files.

5.6.1 Implementation Details

Our Python based tool: (i) parses the 'sample' configuration file, (ii) translates the configuration statements into a canonical format (iii) collects individual objects CO_i, (iv) constructs a tree representing the canonical form of the configuration file (See Fig. 5.4), and (v) recursively computes individual HI_i and the final HI of the root level object. Initially the weights $\vec{w_i}$ given by domain expert is used to compute the overall HI of the configuration file. The gathered *domain* knowledge of CO_i and $\vec{w_i}$ is stored in the knowledge repository for later use. While parsing the new configuration file, the CHeSS framework consults the knowledge repository for any prior knowledge of the CO_i. If a match is found for a given CO_i, the weight $\vec{w_i}$ is assigned to the object, else CHeSS will update the new value w_i in the repository (box 6 of Fig. 5.3). If the weights in the knowledge base and domain expert differ beyond a threshold - CHeSS will flag this as an anomaly. Domain experts can take action to verify the anomaly and re-adjust the weights in the knowledge repository as explained earlier. Knowledge repository in our initial phase is a simple JSON file representing the tree graph and corresponding weights. Additional aspects of knowledge repository (representation, storage, security, global or local share etc.) is a beyond the scope of this chapter.

As an initial study, we used few run-time configuration files from a data center network topology. We processed the configuration as explained earlier and computed their health index. These files contained between 8000 to 22000 lines of configuration data. Discounting for any blank lines and comment lines (which were less than 100), this is still a significant number of CO_i objects to process. As explained earlier, most of the configuration objects had hierarchical sub-objects and sub-sub-objects. In some cases, the hierarchical objects can be around several layers deep; as is expected in a complex network configuration file (see Fig.5.7. HI of a few configuration files used in our evaluation is given in Table 5.1. Space and privacy limitations restrict us from sharing the sample (production level) configuration files used in the table.

File Name	**File Size**	**Health Index**
sample_config_file_1.txt	8936	P = 1.665, A = 1.594, S = 1.518.
sample_config_file_2.txt	22035	P = 1.664, A = 1.615, S = 1.652.
sample_config_file_3.txt	5425	P = 1.607, A = 1.720, S = 1.727.

Table 5.1: Health Index of Sample Configuration Files.

The visual representation of HI of one of the sample file is shown in Fig 5.6. The x, y, & z axes represent the availability, performance and security attributes of HI. Individual blue dots represent the individual HI_i of all top-layer objects CO_i. These high level configurations affect the overall HI and their relative position in different quadrants *reveal* the state of the configuration. For example, a dense set of blue dots in any quadrant can indicate that these configurations are either penalizing or aiding the overall HI. The root level HI of the configuration is printed on the top of the 3D graph. This HI represents a fairly high degree of performance, security and availability. Domain experts agree that this is the case - as this sample file represents a production level configurations, and as expected should have a high value for each of these attributes.

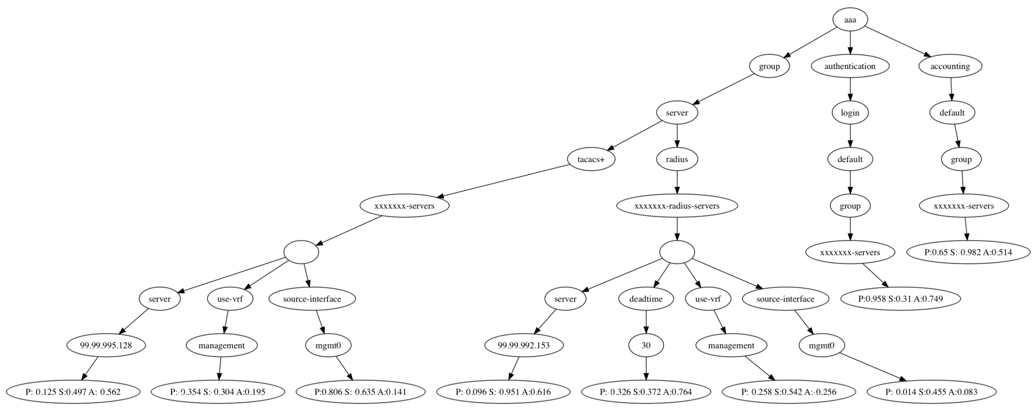


FIGURE 5.7: Sample of a Complex Hierarchical Object CO_i.

5.6.2 *Scalability of the Framework*

Large data centers have 100s of 1000s of nodes; that are grouped into a manageable 'config-types' or 'config-classes' to ease deployment and manageability. Though CHeSS can be run over individual devices, for scalability reasons, CHeSS can be run on these few config-types in the master config repository to discover any anomalies. Such a apriori information is helpful for the network operators to take informed actions or applying them globally.

5.6.3 Applicability and Feasibility of Framework

The applicability of HI can be explored in several areas. Users can query the network topology (or collection of device configurations) to locate devices/resources with specific HI. For example, users can locate devices with performance $P \leqslant 1.75$, avoid devices with weak security (e.g., $S \leqslant 1.25$), or identify the weakest availability node (e.g., $A \leqslant 0.75$) in a given path for diagnostic purposes. It can be helpful in correlating an incident with diagnosis. Results from the lessons learnt about the configuration during incident diagnosis can be used to improve the weights. The HI can be used along with CVSS scores to comprehensively evaluate the security of the network device.

Administrators can use CHeSS to periodically monitor the data-center/enterprise network for HI (without undue traffic or load overhead) metrics. It can assist the administrator in predicting/ diagnosing a configuration error and catch the ill-effects of undue changes early-on. Similar approach can be used to design and monitor HI of web-services, applications, storage clusters, etc.

5.7 Conclusion

In this chapter, we introduce *CHeSS*: a framework to discover, measure and quantify the health index (HI) of a resource or a device. The oper-data and inputs from domain expertise are used to compute the $\vec{HI}$. CHeSS computes the HI based on each configuration object (or hierarchy of objects) of the configuration file. The computed HI can be visualized in a multi-dimensional space to aid users into a clear and common understanding of the HI of the device/resource. Using human-in-the-loop learning, we present a model that can improve the prediction accuracy of the health index over time.

This is the first step in a promising but unexplored field that could benefit from further exploration. The most critical aspect of the proposed HI is the assignment of weights, since the weights directly govern the accuracy and usefulness of the HI. We proposed a two step learning model based on reverse learning from oper-data and learning weight of objects through HIL. The learning of weights could potentially exploit transfer learning from similar device from different vendors used in similar ways. Finally, we proposed that recent vendor neutral device model by OpenConfig can be used to translate vendor specific files to a canonical model.

In addition to HI of individual devices, it is useful to estimate the HI of a combined network of resources, devices or service. Combining the HI's together to estimate the network HI must consider both the topology of the network and the operational dependencies across the devices/resources. These dependencies are likely to be service (or workload) dependent, which makes the dependency characterisation itself quite challenging. These new challenges open further avenues for future research ideas.

CHAPTER 6

CONCLUSION

Configuration modeling and diagnosis in a data center is a complex area of study due to numerous interdependencies that are hard to discover, analyze, and model. During this research work, we proposed a few solutions that include:

(i) effective allocation of resources for a data center network with a goal to conserve energy and limit the latency overheads within an acceptable range.

(ii) application of statistical machine learning techniques to predict the performance of a Cloud Storage Gateway (CSG).

(iii) embedding of domain knowledge in stochastic processes (Genetic Algorithms & Simulated Annealing) to recommend a suitable configuration (i.e. allocate hardware and storage resource to CSG) that satisfy both the required QoS levels (e.g. performance) and the given constraints (e.g. costs), and

(iv) formulation of a health index of a resource or a device.

Using real-world data-sets, we demonstrate the effectiveness of the above solution in the Edge/Cloud computing environment and show its general applicability to address the similar issues facing the data-center operators.

Energy management of cyber-systems and devices is important to curtail the ever increasing energy consumption, waste heat dissipation, and operational costs in data center devices. In our work on energy management techniques, we first developed an energy management extension to the popular network simulation tool (NS3) to study the performance-energy trade-offs in data center networks. Using this extension, we proposed a model to configure the energy management related parameters of a network node and route the traffic to ensure that the network can both avoid congestion and maximize energy saving opportunities. Using a fat-tree network topology, our work showed that effective routing, intelligent consolidation of data flows, plus energy management techniques can reduce power consumption up to 45% under low utilization conditions. Our work on energy management extension to the NS3 tool opens up new avenues to study energy management techniques in data centers.

In the course of this research work, we addressed challenges involved in effective configuration and resource allocation in a complex cyber-systems that involve compute, storage and network domains. Using a commercially available CSG product, we extensively analyze the impact of various configuration parameters on CSG performance. Given the complexities of behavioral characterization through direct analytic means, we explore the applicability of machine learning to characterize this relationship to predict the performance from the configuration parameters. Based on extensive testing with real world industry workloads, we show that with adequate training data, the performance prediction can achieve an accuracy of around 95%. This work provides a practical solution to a very difficult and increasingly important problem in configuring a compute/storage system.

The *inverse of the above problem* is very demanding to formalize and answer. As numerous configuration parameters affect the desired metric of the solution (e.g., performance, cost, availability, etc.), finding suitable values of configuration parameters for a given goal is very difficult. Using an Edge Computing/Storage (ESI) paradigm as an example, we propose an approach that (i) combines a machine learning based model of the desired metric, domain knowledge based grouping and ordering of configuration parameters, and (ii) applies well-designed meta-heuristics to efficiently determine suitable configuration settings for the system. The method inherently can provide multiple solutions, which is crucial in practice to consider aspects that are not easily formalizable. We develop our technique in the context of Edge storage (ESI) that is needed in multiple contexts, but the techniques are applicable generally. An extensive evaluation using real-world vendor provided workloads, shows that the proposed intelligent meta-heuristics reduces the number of iterations by 22% (for modified Genetic Algorithms) and 32% (for modified Simulated Annealing) in 90% of the cases while yielding a solution with very similar cost.

Our work established that the behavior of all cyber systems is dependent on a set of configuration parameters, which if set improperly could result in sub-optimal performance, security vulnerabilities, functional issues, reduced availability, etc. Currently, there are no effective metrics to express the quality of a configuration of a device. With a goal to aid unambiguous description of a device configuration, we propose “CHeSS”, a framework to capture the ”health” of a subsystem or component in terms of multiple dimensions such as performance, security, availability, etc. We demonstrate the feasibility and application of the Health Index framework using examples of real world configuration files. We extend our approach to use a knowledge repository to better refine the health index, which improves with an increase in available data.

6.1 Opportunities and Future Work

Following the domain knowledge ingrained solution from our research, we envision that future work based on similar approach can help resource allocation in other paradigms, such as configuration of Data Storage Service for Edge and Fog, Storage as a Service (SaaS), Containers (VMimages, Dockers), Edge solutions, etc. The behavior (performance) prediction model can expand to include environmental factors[1] such as ambient

[1] See: Sondur and Gross (2020); Vaidyanathan et al. (2015)

temperature, vibration induced degradation, etc., and access their impact on the final outcome. The model can further benefit from an enriched feature set and operational data available to the data center operators to improve the efficiency of the performance prediction oracle and the configuration recommending algorithms (faster convergence). Lessons learned from this research can be applied to auto-tune the configuration as the conditions change (e.g. workload or expected performance) or new cost function (e.g. new limits on energy consumed). The recommended configuration output from the algorithms can be used in solutions to “pack” or get an optimal ratio of virtual resource to physical resource; for example, consolidate the optimal number of VMImages (or dockers) on a physical server.

In CHeSS, we took the first step in a promising but unexplored field that could benefit from further exploration. The CHeSS framework is built on assignment of weights which determine the actual value of HI and hence its accuracy and effectiveness. The weights need not necessarily be coming from human experts. We can consider reverse learning of these weights based on the real-time analysis of the operational data collected from network devices. The learning of weights could potentially exploit transfer learning from similar devices from different vendors used in similar ways. It is useful to estimate the HI of a combined network of resources, devices or service, and doing so must consider both the topology of the network and the operational dependencies across the devices/resources. The CHeSS framework currently makes use of canonical model for representation of device configurations. In future, we can explore the vendor neutral framework like OpenConfig models to represent the configurations which will help in analysing the HI across vendor devices. Industry experts have acknowledged that CHeSS approach of the Health Index is a concept way overdue and sorely needed by the industry.

We hope that this dissertation and the challenges highlighted here will open new avenues for future research ideas.

BIBLIOGRAPHY

Almseidin, M., Alzubi, M., Kovacs, S., and Alkasassbeh, M. (2017), “Evaluation of machine learning algorithms for intrusion detection system,” in *Intelligent Systems and Informatics (SISY), 2017 IEEE 15th International Symposium on*, pp. 000277–000282, IEEE.

Barroso, L. A., Clidaras, J., and Hölzle, U. (2013), “The datacenter as a computer: An introduction to the design of warehouse-scale machines,” *Synthesis lectures on computer architecture*, 8, 1–154.

Ben-Ameur, W. (2004), “Computing the initial temperature of simulated annealing,” *Computational Optimization and Applications*, 29, 369–385.

Benson, T., Akella, A., and Maltz, D. A. (2009), “Unraveling the Complexity of Network Management.” in *NSDI*.

Bertoldi, P., Avgerinou, M., and Castellazzi, L. (2017), “Trends in data centre energy consumption under the European Code of Conduct for Data Centre Energy Efficiency,” .

Bozorgi, M., Saul, L. K., Savage, S., and Voelker, G. M. (2010), “Beyond heuristics: learning to classify vulnerabilities and predict exploits,” in *Proceedings of the 16th ACM SIGKDD international conference on Knowledge discovery and data mining*, pp. 105–114, ACM.

Cao, Z., Tarasov, V., Raman, H. P., Hildebrand, D., and Zadok, E. (2017), “On the performance variation in modern storage stacks,” in *15th Conference on File and Storage Technologies ({FAST} 17)*.

Cao, Z., Tarasov, V., Tiwari, S., and Zadok, E. (2018), “Towards better understanding of black-box auto-tuning: a comparative analysis for storage systems,” in *2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18)*, pp. 893–907.

Cappelli, W. (2015), “Causal analysis makes availability and performance data actionable,” *Gartner Report*.

Chinchuluun, A. and Pardalos, P. M. (2007), “A survey of recent developments in multi-objective optimization,” *Annals of Operations Research*, 154, 29–50.

Chirgwin, R. (2017), "Suspicious BGP event routed big traffic sites through Russia," .

Christensen et al. (2010), "IEEE 802.3 az: the road to energy efficient ethernet," *IEEE Communications Magazine*, 48.

Cisco Systems, Inc (2019), "Cisco Network Management," `https://www.cisco.com/c/en/us/products/cloud-systems-management/index.html`.

Connolly, F. (2014), "Production operations – The last mile of a DevOps strategy," *LMC Report*.

Connor, R. (2015), *The United Nations world water development report 2015: water for a sustainable world*, vol. 1, UNESCO Publishing.

Cooper, B. F., Silberstein, A., Tam, E., Ramakrishnan, R., and Sears, R. (2010), "Benchmarking cloud serving systems with YCSB," in *Proceedings of the 1st ACM symposium on Cloud computing*, pp. 143–154, ACM.

Costa, L. B. and Ripeanu, M. (2010), "Towards automating the configuration of a distributed storage system," in *2010 11th IEEE/ACM International Conference on Grid Computing*, pp. 201–208.

D, H. (2019), "Platypus - Multiobjective Optimization in Python," .

Deb, K., Pratap, A., Agarwal, S., and Meyarivan, T. (2002), "A fast and elitist multiobjective genetic algorithm: NSGA-II," *IEEE Transactions on Evolutionary Computation*, 6, 182–197.

Drolia, U., Guo, K., Tan, J., Gandhi, R., and Narasimhan, P. (2017), "Cachier: Edge-Caching for Recognition Applications," in *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, pp. 276–286.

Eckart, B., He, X., Ong, H., and Scott, S. L. (2009), "An extensible I/O performance analysis framework for distributed environments," in *European Conference on Parallel Processing*, pp. 57–68, Springer.

Edkrantz, M. and Said, A. (2015), "Predicting Cyber Vulnerability Exploits with Machine Learning." in *SCAI*.

EEE (2016), "IEEE standards to enable energy-efficient operation of Ethernet," .

Elliot, S. (2017), "Amazon. com goes down, loses $66,240 per minute," .

Esposito, C., Ficco, M., Palmieri, F., and Castiglione, A. (2016), "Smart Cloud Storage Service Selection Based on Fuzzy Logic, Theory of Evidence and Game Theory," *IEEE Transactions on Computers*, 65, 2348–2362.

Ferentinos, K. P., Arvanitis, K. G., and Sigrimis, N. (2002), "Heuristic optimization methods for motion planning of autonomous agricultural vehicles," *Journal of Global Optimization*, 23, 155–170.

Flash Memory Submit (2018), "SNIA Object Store: Tutorial: Everything You wanted to Know About Storage," `https://www.snia.org`.

Forum of Incident Response and Security Teams (2017), "Common Vulnerability Scoring System," .

Gill, S. S., Chana, I., Singh, M., and Buyya, R. (2018), "CHOPPER: an intelligent QoS-aware autonomic resource management approach for cloud computing," *Cluster Computing*, 21, 1203–1241.

Gracia-Tinedo, R., Sampé, J., Zamora, E., Sánchez-Artigas, M., García-López, P., Moatti, Y., and Rom, E. (2017), "Crystal: Software-defined storage for multi-tenant object stores," in *Proceedings of the 15th Usenix Conference on File and Storage Technologies*, pp. 243–256, USENIX Association.

Herodotou, H. and et.al (2014), "Scalable near real-time failure localization of data center networks," in *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, ACM.

Hsu, C.-J., Panta, R. K., Ra, M.-R., and Freeh, V. W. (2016), "Inside-out: Reliable performance prediction for distributed storage systems in the cloud," in *2016 IEEE 35th Symposium on Reliable Distributed Systems (SRDS)*, pp. 127–136, IEEE.

Ingber, L. (1989), "Very fast simulated re-annealing," *Mathematical and Computer Modelling*, 12, 967 – 973.

Kant, K. (2011), "Multistate power management of communications links," in *Proc. of COMSNET*.

Kerr, A. and Mullen, K. (2019), "A comparison of genetic algorithms and simulated annealing in maximizing the thermal conductance of harmonic lattices," *Computational Materials Science*, 157, 31–36.

Klimovic, A., Litz, H., and Kozyrakis, C. (2018), "Selecta: heterogeneous cloud storage configuration for data analytics," in *2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18)*, pp. 759–773.

Kumar, M., Sharma, S., Goel, S., Mishra, S. K., and Husain, A. (2020), "Autonomic cloud resource provisioning and scheduling using meta-heuristic algorithm," *Neural Computing and Applications*.

Lee, C.-Y. (2015), “Fast simulated annealing with a multivariate Cauchy distribution and the configuration’s initial temperature,” *Journal of the Korean Physical Society*, 66, 1457–1466.

Legendre, P. and Legendre, L. F. (2012), *Numerical ecology*, Elsevier.

Martinez, A., Yannuzzi, M., de Vergara, J. L., Serral-Gracià, R., and Ramírez, W. (2015), “An ontology-based information extraction system for bridging the configuration gap in hybrid SDN environments,” in *2015 IFIP/IEEE International Symposium on Integrated Network Management (IM)*, pp. 441–449, IEEE.

Masanet, E., Shehabi, A., Lei, N., Smith, S., and Koomey, J. (2020), “Recalibrating global data center energy-use estimates,” *Science*, 367, 984–986.

Monga, S. K., Ramachandra, S. K., and Simmhan, Y. (2019), “ElfStore: A resilient data storage service for federated edge and fog resources,” in *2019 IEEE International Conference on Web Services (ICWS)*, pp. 336–345, IEEE.

Mostowfi et al. (2015), “A simulation study of energy-efficient ethernet with two modes of low-power operation,” *IEEE Communications Letters*, 19, 1702–1705.

Mostowfi et al. (2016), “An analytical model for the power consumption of Dual-Mode EEE,” *Electronics Letters*, 52, 1308–1310.

Murugan, M. et al. (2012), “On the Interconnect Energy Efficiency of High End Computing Systems,” *Sustainable Computing: Informatics and Systems*.

Newman, L. H. (2017), “How a tiny error shut off the internet for parts of the US,” .

Ngoupé, É. L., Stoesel, S., Parisot, C., Hallé, S., Valtchev, P., Cherkaoui, O., and Boucher, P. (2015), “A data model for management of network device configuration heterogeneity,” in *2015 IFIP/IEEE International Symposium on Integrated Network Management (IM)*, IEEE.

NS3doc (2015), “Network Simulator (ns3),” .

Ofer, E., Epstein, A., Sadeh, D., and Harnik, D. (2018), “Applying Deep Learning to Object Store Caching,” in *Proceedings of the 11th ACM International Systems and Storage Conference*, SYSTOR ’18, pp. 126–126, New York, NY, USA, ACM.

OpenConfig working group (2016), “Open Config Data Model,” .

Oracle (2010), “Performance Evaluation of Storage and Retrieval of DICOM Image Content ... ,” .

Pedregosa, F., Varoquaux, G., and Gramfort, A. e. (2011), “Scikit-learn: Machine Learning in Python,” *Journal of Machine Learning Research*, 12, 2825–2830.

Prahlad, A., Muller, M. S., and Kottomtharayil, R. e. (2010), "Cloud gateway system for managing data storage to cloud storage sites," US Patent App. 12/751,953.

Prahlad, A., Muller, M. S., and Kottomtharayil, R. e. (2012), "Data object store and server for a cloud storage environment, including data deduplication and data management across multiple cloud storage sites," US Patent 8,285,681.

Rao, J., Bu, X., Xu, C.-Z., Wang, L., and Yin, G. (2009), "VCONF: A Reinforcement Learning Approach to Virtual Machines Auto-configuration," in *Proceedings of the 6th International Conference on Autonomic Computing*, ICAC '09, pp. 137–146, New York, NY, USA, ACM.

Rashid, M., Newton, M. A. H., Hoque, M., and Sattar, A. (2013), "Mixing Energy Models in Genetic Algorithms for On-Lattice Protein Structure Prediction," *BioMed research international*, 2013, 924137.

Ray, M., Sondur, S., Biswas, J., Pal, A., and Kant, K. (2018), "Opportunistic Power Savings with Coordinated Control in Data Center Networks," in *Proceedings of the 19th International Conference on Distributed Computing and Networking*, ICDCN '18, New York, NY, USA, Association for Computing Machinery.

Ross, D. M., Wollaber, A. B., and Trepagnier, P. C. (2017), "Latent feature vulnerability ranking of CVSS vectors," in *Proceedings of the Summer Simulation Multi-Conference*, Society for Computer Simulation International.

Satyanarayanan, M. (2017), "The Emergence of Edge Computing," *Computer*, 50, 30–39.

Shaikh, A., Hofmeister, T., Dangui, V., and Vusirikala, V. (2016), "Vendor-neutral network representations for transport SDN," in *2016 Optical Fiber Communications Conference and Exhibition (OFC)*, pp. 1–3.

Sittón-Candanedo, I., Alonso, R. S., Corchado, J. M., Rodríguez-González, S., and Casado-Vara, R. (2019), "A review of edge computing reference architectures and a new global edge proposal," *Future Generation Computer Systems*, 99, 278–294.

Sohan et al. (2010), "Characterizing 10 Gbps network interface energy consumption," in *IEEE LCN*, pp. 268–271.

Sondur, S. (2014), "Software Defined Networking for Beginners," .

Sondur, S. and Gross, Kenny Kant, K. (2020), "Thermo-Mechanical Coupling Induced Performance Degradation in Storage Systems," in *2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, IEEE.

Sondur, S. and Kant, K. (2019), "Towards Automated Configuration of Cloud Storage Gateways: A Data Driven Approach," in *International Conference on Cloud Computing*, pp. 192–207, Springer.

Sondur, S., Ray, M., Biswas, J., and Kant, K. (2017), "Implementing data center network energy management capabilities in NS3," in *2017 Eighth International Green and Sustainable Computing Conference (IGSC)*, pp. 1–8.

Sondur, S., Gross, K., and Li, M. (2018), "Data Center Cooling System Integrated with Low-Temperature Desalination and Intelligent Energy-Aware Control," in *2018 Ninth International Green and Sustainable Computing Conference (IGSC)*, pp. 1–6.

Sondur, S., Kant, K., Vucetic, S., and Byers, B. (2019), "Storage on the Edge: Evaluating Cloud Backed Edge Storage in Cyberphysical Systems," in *2019 IEEE 16th International Conference on Mobile Ad Hoc and Sensor Systems (MASS)*, pp. 362–370.

Sondur, S., Shankar, G., and Kant, K. (2020), "CHeSS: A Configuration Health Scoring System and Its Application to Network Devices," in *2020 23rd Conference on Innovation in Clouds, Internet and Networks and Workshops (ICIN)*, pp. 250–257.

Sondur, S., Alazzawe, A., and Krishna, K. (2020), "CyberCon: Efficient Goal Oriented Configuration of Complex Cyber-Systems," *Submitted for publication*.

Sorower, M. S. (2010), "A literature survey on algorithms for multi-label learning," *Oregon State University, Corvallis*, 18.

Tanimura, Y. and Koie, H. (2015), "Operation-Level Performance Control in the Object Store for Distributed Storage Systems," in *2015 IEEE International Conference on Data Science and Data Intensive Systems*, pp. 111–112, IEEE.

Tapparello, C., Ayatollahi, H., and Heinzelman, W. B. (2014), "Energy harvesting framework for network simulator 3," in *ENSsys@SenSys*.

Tesauro, G. et al. (2005), "Online resource allocation using decompositional reinforcement learning," in *AAAI*, vol. 5, pp. 886–891.

Ularu, E. G., Puican, F. C., Suciu, G., Vulpe, A., and Todoran, G. (2013), "Mobile Computing and Cloud maturity-Introducing Machine Learning for ERP Configuration Automation." *Informatica Economica*, 17.

Vaidyanathan, K., Gross, K., and Sondur, S. (2015), "Ambient temperature optimization for enterprise servers: Key to large-scale energy savings," in *Energy Efficient Electronic Systems (E3S), 2015 Fourth Berkeley Symposium on*, pp. 1–3, IEEE.

Varma, R. (2008), "Storage media for computers in radiology," *The Indian journal of radiology and imaging*, 18, 287–9.

Wang, M., Au, K., Ailamaki, A., Brockwell, A., Faloutsos, C., and Ganger, G. R. (2004), "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, 2004.(MASCOTS 2004). Proceedings.*, pp. 588–595, IEEE.

Wang, S., Zhang, X., Zhang, Y., Wang, L., Yang, J., and Wang, W. (2017), "A survey on mobile edge networks: Convergence of computing, caching and communications," *IEEE Access*, 5, 6757–6779.

Wong, A. K. Y., Ray, P., Parameswaran, N., and Strassner, J. (2005), "Ontology mapping for the interoperability problem in network management," *IEEE Journal on Selected Areas in Communications*, 23.

Wu, H., Nabar, S., and Poovendran, R. (2011), "An energy framework for the network simulator 3," in *SimuTools*.

Xu, T. and Zhou, Y. (2015), "Systems approaches to tackling configuration errors: A survey," *ACM Computing Surveys (CSUR)*, 47, 70.

Xu, Y., Ye, Q., and Meng, G. (2018), "Hybrid phase retrieval algorithm based on modified very fast simulated annealing," *International Journal of Microwave and Wireless Technologies*, 10, 1072–1080.

Yi, S., Li, C., and Li, Q. (2015), "A Survey of Fog Computing: Concepts, Applications and Issues," in *Proceedings of the 2015 Workshop on Mobile Big Data*, Mobidata '15, pp. 37–42, New York, NY, USA, ACM.

Yin, Z., Ma, X., Zheng, J., Zhou, Y., Bairavasundaram, L. N., and Pasupathy, S. (2011), "An empirical study on configuration errors in commercial and open source systems," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pp. 159–172, ACM.

Zhang, S., He, L., Dragut, E., and Vucetic, S. (2019), "How to Invest my Time: Lessons from Human-in-the-Loop Entity Extraction," in *The 25th ACM SIGKDD International Conference*, pp. 2305–2313.

Zhang, Y. and Xu, K. (2020), "A Survey of Resource Management in Cloud and Edge Computing," in *Network Management in Cloud and Edge Computing*, pp. 15–32, Springer.

Zhao, L.-S., Sen, M. K., Stoffa, P., and Frohlich, C. (1996), "Application of very fast simulated annealing to the determination of the crustal structure beneath Tibet," *Geophysical Journal International*, 125, 355–370.