2018

Improving efficiency and resilience in large-scale computing systems through analytics and data-driven management

BOSTON UNIVERSITY

COLLEGE OF ENGINEERING

Dissertation

IMPROVING EFFICIENCY AND RESILIENCE IN LARGE-SCALE COMPUTING SYSTEMS THROUGH ANALYTICS AND DATA-DRIVEN MANAGEMENT

by

OZAN TUNCER

B.S., Middle East Technical University, 2012

Submitted in partial fulfillment of the

requirements for the degree of

Doctor of Philosophy

2018

Approved by

First Reader

Ayse K. Coskun, Ph.D.
Associate Professor of Electrical and Computer Engineering

Second Reader

Manuel Egele, Ph.D.
Assistant Professor of Electrical and Computer Engineering

Third Reader

Orran Krieger, Ph.D.
Professor of the Practice of Electrical and Computer Engineering

Fourth Reader

Vitus J. Leung, Ph.D.
Principal Member of Technical Staff
Sandia National Laboratories

Fifth Reader

Nilton Bila, Ph.D.
Research Staff Member
IBM T.J. Watson Research Center

Research is what I'm doing when I don't know what I'm doing.

Wernher von Braun

Acknowledgments

First, I would like to thank my advisor, Professor Ayse K. Coskun, for her invaluable support and constant encouragement throughout my Ph.D. studies. I appreciate all her contributions, time, and ideas that have made my PhD experience productive.

In addition to my advisor, I am also grateful to Professor Manuel Egele and Professor Orran Krieger for their valuable advice and feedback on my research. I would like to thank Dr. Vitus Leung for his inspirational guidance and for the summer internship opportunity at Sandia National Laboratories. I would also like to thank Dr. Nilton Bila and Dr. Canturk Isci for their invaluable advice and support on my research and during my internship at IBM T. J. Watson Research Center.

I am grateful to our collaborators and co-authors: Emre Ates, Yijia Zhang, and Dr. Ata Turk at Boston University, Jim Brandt at Sandia National Laboratories, Dr. Sastry Duri at IBM Research, Dr. Marina Zapater at EPFL, Prof. José Ayala at Complutense University of Madrid, Prof. José Moya at Universidad Politécnica de Madrid, Dr. Kalyan Vaidyanathan at BAE Systems, and Dr. Kenny Gross at Oracle Corp. for their productive collaboration and all the stimulating discussions.

I thank my fellow lab mates and friends for their encouragement and support which greatly helped me to overcome the struggles of the Ph.D. life. I would like to especially acknowledge Dr. Fulya Kaplan, Dr. Tiansheng Zhang, Tolga Bolukbasi, Onur Sahin, Emre Ates, Onur Zungur, Cali Stephens, and Laura Cunningham.

Finally, I would like to give special thanks to my family for their encouragement, understanding, and unconditional support.

The research that forms the basis of this dissertation has been partially funded by Sandia National Laboratories, by IBM T. J. Watson Research Center, and by Oracle Corp.

The contents of Chapter 3 are in part reprints of the material from the paper Ozan

Tuncer, Emre Ates, Yijia Zhang, Ata Turk, Jim Brandt, Vitus J. Leung, Manuel Egele, Ayse K. Coskun, "Diagnosing Performance Variations in HPC Applications Using Machine Learning", in *Proceedings of the International Supercomputing Conference (ISC-HPC)*.

Chapter 4 contains reprints of the material from the papers Ozan Tuncer, Kalyan Vaidyanathan, Kenny Gross, and Ayse K. Coskun, "CoolBudget: Data Center Power Budgeting with Workload and Cooling Asymmetry Awareness", in *IEEE International Conference on Computer Design (ICCD)*, 2014; Ozan Tuncer, Vitus J. Leung, and Ayse K. Coskun, "PaCMap: Topology Mapping of Unstructured Communication Patterns onto Non-contiguous Allocations", in *Proceedings of the ACM on International Conference on Supercomputing (ICS)*, 2015; Marina Zapater, Ozan Tuncer, José L. Ayala, José M. Moya, Kalyan Vaidyanathan, Kenny Gross, and Ayse K. Coskun, "Leakage-Aware Cooling Management for Improving Server Energy Efficiency", in *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 2015; and Ozan Tuncer, Yijia Zhang, Vitus J. Leung, Ayse K. Coskun, "Task Mapping on a Dragonfly Supercomputer", in *IEEE High Performance Extreme Computing Conference (HPEC)*, 2017.

IMPROVING EFFICIENCY AND RESILIENCE IN LARGE-SCALE COMPUTING SYSTEMS THROUGH ANALYTICS AND DATA-DRIVEN MANAGEMENT

OZAN TUNCER

Boston University, College of Engineering, 2018

Major Professor: Ayse K. Coskun, Ph.D.
Associate Professor of Electrical and Computer Engineering

ABSTRACT

Applications running in large-scale computing systems such as high performance computing (HPC) or cloud data centers are essential to many aspects of modern society, from weather forecasting to financial services. As the number and size of data centers increase with the growing computing demand, scalable and efficient management becomes crucial. However, data center management is a challenging task due to the complex interactions between applications, middleware, and hardware layers such as processors, network, and cooling units.

This thesis claims that to improve robustness and efficiency of large-scale computing systems, significantly higher levels of automated support than what is available in today's systems are needed, and this automation should leverage the data continuously collected from various system layers. Towards this claim, we propose novel methodologies to automatically diagnose the root causes of performance and configuration problems and to improve efficiency through data-driven system management.

We first propose a framework to diagnose software and hardware anomalies that

cause undesired performance variations in large-scale computing systems. We show that by training machine learning models on resource usage and performance data collected from servers, our approach successfully diagnoses 98% of the injected anomalies at runtime in real-world HPC clusters with negligible computational overhead.

We then introduce an analytics framework to address another major source of performance anomalies in cloud data centers: software misconfigurations. Our framework discovers and extracts configuration information from cloud instances such as containers or virtual machines. This is the first framework to provide comprehensive visibility into software configurations in multi-tenant cloud platforms, enabling systematic analysis for validating the correctness of software configurations.

This thesis also contributes to the design of robust and efficient system management methods that leverage continuously monitored resource usage data. To improve performance under power constraints, we propose a workload- and cooling-aware power budgeting algorithm that distributes the available power among servers and cooling units in a data center, achieving up to 21% improvement in throughput per Watt compared to the state-of-the-art. Additionally, we design a network- and communication-aware HPC workload placement policy that reduces communication overhead by up to 30% in terms of *hop-bytes* compared to existing policies.

Contents

List of Tables

List of Figures

List of Abbreviations

BIPS	.....	Billions of Instructions Per Second
CDF	.....	Cumulative Distribution Function
CoP	.....	Coefficient of Performance
CPU	.....	Central Processing Unit
CRAC	.....	Computer Room Air Conditioning unit
CSTH	.....	Continuous System Telemetry Harness
DRAM	.....	Dynamic Random Access Memory
DVFS	.....	Dynamic Voltage-Frequency Scaling
FDR	.....	False Discovery Rate
GPU	.....	Graphics Processing Unit
HPC	.....	High Performance Computing
ICA	.....	Independent Component Analysis
kNN	.....	K Nearest Neighbors
KS	.....	Kolmogorov-Smirnov
LDMS	.....	Light-weight Distributed Metric Service
MPI	.....	Message Passing Interface
NPB	.....	The NAS Parallel Benchmark Suite
RPM	.....	Revolutions Per Minute
QoS	.....	Quality of Service
SVM	.....	Support Vector Machine
VM	.....	Virtual Machine

Chapter 1

Introduction

The number and size of large-scale computing systems (e.g., high performance computing (HPC) or cloud data centers) have been increasing rapidly in recent years following the explosive demand in digital services such as social networking, web search, and weather forecasting. The number of data centers[1] in 2017 has been predicted as 8.6 million (IDC, 2014), and today's largest data centers are using nearly 20 million CPU cores (Dongarra et al., 2017). As a result, data centers are among the largest electricity consumers in the world. In the upcoming years, the global data center market is expected to have an annual growth rate of 5% (Technavio, 2017), further increasing the economical, environmental, and performance impact of data center energy and resource efficiency.

The expansion of data centers leads to an increasingly complex management problem, where there are many levels of interactions among numerous subsystems such as networking, storage, cooling, and processors. Due to the scale, heterogeneity, and complexity of data centers, managing these systems using heuristics, manual control, or methods that are designed for specific conditions is quickly becoming insufficient to achieve robust and efficient operation. Moreover, in these complex large-scale systems, a great variety of performance or resilience problems arise from different sources including hardware failures, firmware bugs, shared resource contention, and configura-

[1] In this thesis, a data center broadly refers to a large-scale computing system that runs HPC or cloud workloads. The applicability of our methodologies to different types of data centers are discussed in individual chapters.

tion errors. These problems are becoming parts of normal operating conditions rather than rare events that must be avoided (Barroso et al., 2013). As a result, approximately 70% of data center spending is on management and administration (Villars et al., 2012).

The efficiency and resilience of large-scale computing systems are typically ensured via monitoring infrastructures that continuously collect data from various system layers regarding resource utilization, application performance, software log outputs, configurations, and component temperatures (Agelastos et al., 2014; Aceto et al., 2013). This thesis builds upon the observation that existing data center management frameworks do not fully utilize the data that is collected from various system layers. We claim that significantly higher levels of automated support are needed to improve robustness and efficiency of large-scale computing systems, and this automation should be based on intelligent frameworks that leverage the continuously collected data using statistical system modeling and learning-based approaches. To this end, this thesis provides frameworks to automatically diagnose performance and software configuration problems, and data-driven dynamic system management policies to improve the energy efficiency of large-scale computing systems.

1.1 Problem Diagnosis in Large-scale Computing Systems

In large-scale computing systems, each application may require hundreds or thousands of individual servers as well as their corresponding storage and networking subsystems, and power distribution and cooling infrastructures to work together seamlessly. Component faults, misbehavior, or resource contention that occurs in a single processor or a network link can affect the performance of the entire application. Especially in HPC clusters, where parallel applications run for hours or even days on hundreds of compute nodes, a slowdown in a single node would lead to waste of compute resources

in all other nodes that are used in parallel with the degraded node. Cloud systems, on the other hand, are typically designed to be resilient against failures of individual components (Verma et al., 2015). However, cloud applications can still suffer from increased tail latency upon component misbehavior (Dean and Barroso, 2013).

Studies have reported that HPC applications suffer by more than 100% performance degradation due to hardware- and software-related *anomalies* (Skinner and Kramer, 2005; Bhatelé et al., 2013; Wang et al., 2014). The anomalies that cause performance variations include orphan processes left over from previous jobs (Brandt et al., 2010), firmware bugs (Cisco, 2017), memory leaks (Agelastos et al., 2015), CPU throttling for thermal control (Brandt et al., 2015), reduced CPU frequency due to hardware problems (Snir et al., 2014), and resource contention (Bhatelé et al., 2013; Dorier et al., 2014).

A key enabler for detecting anomalies is analyzing and understanding the healthy behavior of subsystems across various software, middleware, and hardware layers. However, analyzing system behavior is often a difficult task given the vast amount of noisy and high-dimensional resource usage and performance data being collected via system monitoring infrastructures (Agelastos et al., 2014). While a number of anomaly detection techniques have been proposed (Klinkenberg et al., 2017; Yu and Lan, 2016), these techniques still largely rely on human operators to discover the root causes of the anomalies, leading to delayed mitigation and wasted compute resources. An effective way of decreasing the delay between anomalies and remedies is to automate the diagnosis of anomalies, paving the way for automated mitigation.

In addition to performance and resource usage data that is collected from various subsystems, collection and analysis of application-specific data (e.g., logs or configurations) can help in the diagnosis of anomalies. Specifically, software configurations have been reported as a major source of disruptions and outages in cloud services (Rabkin

and Katz, 2013; Yin et al., 2011; Barroso et al., 2013).

To detect software misconfigurations, researchers have developed various tools to automatically check for errors in software configurations (Attariyan et al., 2012; Behrang et al., 2015). Among such tools, statistical and learning-based techniques have gained popularity as low overhead configuration checkers that can be applied in an application-agnostic manner (Wang et al., 2004; Zhang et al., 2014; Santolucito et al., 2017). Statistical configuration checkers train on a corpus of configurations and learn common patterns, and then identify configurations that deviate from the norm as potential errors. To perform a reliable analysis, these techniques need to be trained with configuration data collected from a large number of working systems.

Cloud environments, where a large number of users deploy their customized applications, provide a unique opportunity for configuration analytics. However, to analyze software configurations in the cloud, one first needs to extract configuration information from cloud system instances (i.e., images, VMs, and containers) without losing any information that is crucial for detecting errors. This is challenging because software configurations are typically stored in loosely-structured text files where each software has its own custom configuration syntax. Furthermore, as cloud instance contents are largely unlabeled, one needs to discover which files are configuration files and also figure out to which applications these files belong.

This thesis proposes novel frameworks to enable automated diagnosis of performance anomalies in large-scale computing systems and software configuration analytics in the cloud. The specific contributions are as follows:

- We propose an application-agnostic online anomaly detection framework that enables automatic diagnosis of anomalies that contribute to performance variations (Tuncer et al., 2017a). Our framework leverages historical resource usage and performance data collected from compute nodes to extract signatures of

previously-observed anomalies using machine learning. Using experiments on a Cray XC30m supercomputer, we demonstrate that our approach consistently outperforms state-of-the-art techniques on diagnosing anomalies.

- This thesis, for the first time, introduces a framework to enable the analysis of text-based software configurations in multi-tenant cloud platforms. Our framework, *ConfEx*, discovers text-based configuration files in cloud instances with unlabeled content, and extracts the information in these files in a way that enables automated comparison and analysis of configurations among thousands of cloud instances.

1.2 Data-driven System Management

The growth in number and size of large-scale computing systems not only makes resilient operation a challenge but also makes the energy footprint of data centers a significant burden on the world's energy resources. The worst-case projection shows that the global data center electricity usage will reach 13% of the world's electricity usage by 2030, whereas this percentage is as low as 3.3% in the best-case projection (Andrae and Edler, 2015). Enabling higher efficiency is a key factor that determines where future data centers will stand between these best- and worst-case scenarios.

A common goal of efficient data center management is to achieve high performance while avoiding the waste of compute resources. Data center management has various areas including server and virtual machine allocation, scheduling, controlling the deployment and maintenance of applications, pricing, and managing storage, power, or network. Improving efficiency in any of these areas leads to better utilization of available resources, improving the overall data center energy efficiency.

Management in today's data centers is a challenging task due to their complexity, size, and heterogeneity. While automated anomaly diagnosis can provide a better un-

derstanding of system behavior, achieving high efficiency requires a holistic awareness of the diversity among applications, performance requirements, physical restrictions such as thermal thresholds, and interactions between applications and hardware components such as processors and cooling units.

In this thesis, we specifically focus on two aspects of system management: (1) power management, which has a direct impact on the energy efficiency of data centers, and (2) workload management for highly-parallel HPC applications, which has high impact on application performance and throughput in HPC clusters.

Data Center Power Budgeting: The total available power in a data center is limited due to the capacity constraints of power infrastructures (Barroso et al., 2013). Furthermore, limiting the power consumption of a data center can lead to substantial cost savings by avoiding peak power demand charges as well as by enabling smart-grid demand-side regulation programs (Chen et al., 2014a) and renewable energy use (Bianchini, 2012). To maximize efficiency under such power constraints, the available power should be dynamically distributed across computing resources while taking application performance requirements into account. A good strategy should intelligently react to changes in workloads and environment, and find the best management decisions to minimize the energy footprint without sacrificing performance.

Workload Placement in HPC Systems: In addition to power management, workload management has a significant impact on performance and efficiency. Especially for highly-parallel HPC applications, poor workload management can increase running times by 100%, wasting valuable compute cycles (Bhatelé et al., 2013). An effective way of reducing application communication overhead is topology mapping, which is the placement of HPC applications to available compute nodes. By being aware of application communication patterns and the system's network topology,

topology mapping can reduce application communication overhead and improve the overall application performance. However, topology mapping based on application communication patterns is an NP-hard problem (Hoefler and Snir, 2011).

We propose data-driven methodologies to improve the efficiency of large-scale computing systems through cooling-aware cluster- and server-level power management as well as HPC workload management. Our specific contributions are as follows:

- In order to improve data center performance under power constraints, we introduce a novel data center power budgeting policy, *CoolBudget* (Tuncer et al., 2014). *CoolBudget* distributes the given power budget among the cooling units and servers to maximize the *fair speedup* of the running applications based on accurate power, performance, and temperature models that are validated on a real enterprise server.

- To improve the efficiency of servers, we propose a leakage-aware proactive server fan control policy based on empirical power models that estimate the static, dynamic, and thermally-induced power consumption in a server (Zapater et al., 2015a). Our policy reduces the sum of server leakage power and cooling power by up to 6% compared to other policies, and proactively avoids thermal violations.

- For efficient placement of HPC applications, we propose a topology mapping technique, *PaCMap* (Tuncer et al., 2015). *PaCMap* uses a holistic view on HPC job placement by mapping individual application tasks directly onto the processors of the available compute nodes as opposed to first selecting compute nodes and then mapping tasks to the processors. Our policy decreases *hop-bytes* (i.e., the weighted sum of the traversed network distance for all messages, weighted by the message size) by up to 30% compared to the state-of-the-art.

- Using large-scale experiments on a state-of-the-art HPC supercomputer, we

demonstrate that efficient topology mapping reduces communication time by up to 47% even when using novel network topologies that are designed to minimize the communication overhead (Tuncer et al., 2017b).

1.3 Organization

The remainder of this thesis starts with a review of the background and related work on automated detection of performance anomalies and software misconfigurations as well as data-driven management of large-scale computing systems in Chapter 2. Chapter 3 introduces our learning-based framework for diagnosing performance anomalies and *ConfEx*, which is our configuration analytics framework. In Chapter 4, we present our work on data-driven cluster- and server-level power management and HPC workload management for improved efficiency. Chapter 5 concludes this thesis and discusses future research directions.

Chapter 2

Background and Related Work

This thesis proposes data-driven techniques for automated problem diagnosis and efficient system management in large-scale computing systems such as HPC clusters and cloud data centers. In this chapter, we first present a detailed overview of the state-of-the-art on detecting performance anomalies, and continue with a discussion on software configuration analytics. We then describe existing work on data-driven management of large-scale systems with a focus on workload management in HPC systems and power management. At the end of each section, we highlight the novel aspects of our work compared to existing work.

2.1 Detection of Performance Anomalies

Performance anomalies (such as thermally-induced CPU throttling or orphan processes) lead to performance variation (Snir et al., 2014), waste of compute resources (Bhatelé et al., 2013), and poor scheduling (Tsafrir et al., 2007) without necessarily causing failures. In the last decade, there has been growing interest in building automatic performance anomaly detection tools for large-scale computing systems (Ibidunmoye et al., 2015). Existing anomaly detection approaches can be divided into three categories: rule-based methods, statistical time series analysis, and machine learning based solutions.

Rule-based anomaly detection methods are commonly deployed in large scale systems. These methods use threshold-based rules on the monitored resource usage

metrics, where the rules are set by domain experts based on the performance and resource usage constraints, application characteristics, and the target system properties (Ahad et al., 2015; Jayathilaka et al., 2017). Such rules significantly depend on the target system infrastructure and are not generalizable to other systems.

Time-series analysis methods for anomaly diagnosis build a time-series model and make predictions based on the observed values of the collected metrics. These methods raise an anomaly alert whenever the prediction does not match the observed value. Previous research has employed multiple time-series models including support vector regression (Jin et al., 2016), auto-regressive integrated moving average (Laptev et al., 2015), and Holt-Winters forecasting (Ibidunmoye et al., 2016). While such time-series prediction models successfully detect anomalous behavior, they are not designed to identify the types of anomalies (i.e., diagnose). Moreover, these models can lead to high computational overhead when the collected set of metrics is large.

A number of approaches based on machine learning have been proposed to detect anomalies on cloud and HPC systems. These approaches utilize unsupervised learning algorithms such as affinity propagation clustering (Nair et al., 2015), DBSCAN (Zhang et al., 2016), isolation forest (Adhianto et al., 2010), and hierarchical clustering (Gurumdimma et al., 2016), as well as supervised learning algorithms such as support vector machines (SVM) (Dalmazo et al., 2016), k-nearest-neighbors (kNN) (Lan et al., 2010; Jin et al., 2016), random forest (Arzani and Outhred, 2016), and Bayesian classifier (Wang et al., 2016). These techniques still rely on human operators to discover the root causes of the anomalies, leading to delayed mitigation and wasted compute resources. An effective way of decreasing the delay between anomalies and remedies is to automate the *diagnosis* of anomalies (Bodik et al., 2010), which paves the way for automated mitigation.

In machine learning based anomaly detection, feature extraction is an essential

step to avoid unacceptable computation overhead and to improve the detection accuracy. Besides using common statistical features such as mean and variance (Klinkenberg et al., 2017), previous works on anomaly detection have also explored features such as correlation coefficients (Chen et al., 2011) and mutual information gain (Gurumdimma et al., 2016). Researchers also explored advanced feature extraction methods including principal component analysis (Lan et al., 2010; Yu and Lan, 2016), independent component analysis (ICA) (Lan et al., 2010; Wang et al., 2016), and wavelet-transformation (Guan et al., 2013; O'Shea et al., 2016). However, the features selection using statistical techniques such as ICA can disregard the features that are useful for anomaly detection (Tuncer et al., 2017a).

Failure detection and diagnosis on large scale computing systems is related to performance anomaly detection as these two fields share some common interests and technologies (Snir et al., 2014). Nguyen et al. proposed a method to pinpoint faulty components by analyzing the fault propagation through components (Nguyen et al., 2013). Similarly, a diagnostic tool called PerfAugur has been developed to help trace the cause of an anomaly by finding common attributes that predicate an anomaly (Roy et al., 2015).

In this thesis, we design and implement a machine learning based framework for diagnosing performance anomalies (Tuncer et al., 2017a) (Section 3.1). In contrast to existing techniques, our approach does not rely on expert knowledge, is not limited to a specific anomaly type, and identifies the type of anomalies (i.e., conducts diagnosis) in contrast to solely detecting them. As we demonstrate in Section 3.1.3, our technique consistently outperforms the state-of-the-art methods in diagnosing performance anomalies in HPC systems.

2.2 Software Configuration Analytics in the Cloud

The configurations of commonly-used cloud applications such as Apache HTTP server and MySQL are becoming increasingly complex for improved application versatility, performance, and security. To function correctly, securely, and with high performance, cloud applications often depend on precise tuning of hundreds of configuration parameters (Xu et al., 2015). In typical cloud services that consist of multi-tiered software stacks, ensuring the desired operation may require correctly configuring thousands of parameters (Ramachandran et al., 2009).

While configurations are traditionally validated by applications during startup, recent work has shown that, depending on the software, 14-93% of configuration parameters in today's cloud software do not have any special code for checking their correctness (Xu et al., 2016). Hence, configuration errors (i.e., misconfigurations) have become one of the major causes of service disruptions and outages in the cloud (Yin et al., 2011). This problem is exacerbated by the prevalence of open-source software where developers can easily deploy services using third-party applications without mastering the configurations of these applications. Hence, automated DevOps support has become important to reduce engineering costs to develop efficient, robust, and secure services in the cloud.

Cloud automation tools such as Chef (Taylor and Vargo, 2014) and Ansible (Hochstein, 2014) focus on deployment and scaling of cloud services, and provide centralized management support for software configurations. While these tools help developers manage hundreds of cloud instances (i.e., images, VMs, and containers), they provide limited support for validating configurations beyond what is needed for deployment and scaling. The developers who have not mastered the configurations of the third-party software they use can easily instruct cloud automation tools to deploy erroneous configurations. As a result, automated configuration validation has

received the attention of cloud researchers and engineers.

Automatically verifying the correctness of software configurations are challenging due to the large number of parameters, parameter dependencies, and the error-prone nature of human-editable text files. Furthermore, rapidly evolving cloud software complicate the design and maintenance of configuration validation tools (Zhang and Ernst, 2014; Xu et al., 2015). A promising approach to address the problem of misconfigurations is automated error detection without using application-specific expert knowledge. Such techniques can be applied in a wide set of applications to help cloud users find and prevent misconfigurations while requiring minimal modification for each application.

Application-agnostic execution trace analysis through binary instrumentation have been shown to provide insight on the root causes of configuration errors (Attariyan et al., 2012; Zhang and Ernst, 2014). However, such approaches are often impractical on production workloads due to their intrusiveness. Other techniques target validating configurations before deployment to avoid service disruptions and outages. Source code analysis (Xu et al., 2013; Zhou et al., 2017; Nadi et al., 2014) and natural language processing on software documentations (Potharaju et al., 2015; Jin et al., 2014) have been used to infer configuration constraints. Configuration entries that do not comply with these constraints are then marked as errors.

Statistical and learning-based techniques such as PeerPressure (Wang et al., 2004) and EnCore (Zhang et al., 2014) have gained popularity as low overhead configuration checkers that can be applied in an application-agnostic manner. Training learning-based models and using them for validation of configurations requires *discovery* and *extraction* of configurations across large populations of installed applications. However, configurations in cloud instances (i.e., image, VM, or container) are often stored in non-standard locations in the file system, which complicates the task of configu-

ration discovery. Moreover, configuration parameters are often embedded in human readable text files that require application-specific domain knowledge to parse reliably. To apply existing analysis techniques, the information extracted from these files needs to be in the form of key-value pairs, where a key represents a specific configuration parameter consistently across different files and cloud instances.

In prior studies, the extraction of configuration key-value pairs have performed using several methods: Parsing files that reside in known paths with custom scripts (e.g., (Potharaju et al., 2015)), crawling erroneous files from mailing lists and technical forums (e.g., (Xu et al., 2016)), parsing files located in default paths using configuration parsing libraries (e.g., (Zhang et al., 2014)), and using standardized configuration stores such as Windows registry (e.g., (Wang et al., 2004)). Existing tools for handling configurations focus on centralized management rather than extracting key-value pairs. CFEngine (Burgess and Ralston, 1997) can parse standard file formats such as XML and JSON, but not application-specific files formats such as in httpd and Nginx configurations. Several tools, including Puppet (Loope, 2011) and bcfg2 (Desai, 2005), can edit application-specific files by leveraging Augeas library (Lutterkort, 2008). In Section 3.2, we demonstrate that using the Augeas library alone is not sufficient for reliable corpus-based analysis.

In this thesis, we introduce our configuration analytics framework, *ConfEx* (Section 3.2), which is the first framework to enable discovery and extraction of consistent configuration data and robust configuration analysis in image repositories and multi-tenant cloud platforms. In contrast to existing techniques, our framework is not limited by configuration files in known file system paths and *discovers* text-based configuration files in cloud instances with over 98% precision and recall. In addition, our framework enables resolving the inconsistencies in the outputs of existing configuration parsers to improve the robustness of configuration analysis.

2.3 Management of Large-scale Computing Systems

The increasing size and complexity of large-scale computing systems has led researchers to develop heuristics and data-driven approaches for efficient management. The areas of research for efficient data center management include hardware fault tolerance (Snir et al., 2014), networking (Kreutz et al., 2015), interconnects (Kachris and Tomkos, 2012), checkpointing (Egwutuoha et al., 2013), storage (Chen et al., 2014b), dynamic pricing (Al-Roomi et al., 2013), power management (Kong and Liu, 2015), and workload management (Hoefler and Snir, 2011). Among these aspects, power management has a direct impact on the energy efficiency of data centers and can help reducing data centers' burden on the world's energy resources. In addition, workload management can significantly reduce wasted compute resources in HPC systems by decreasing application running times by up to 50% even in state-of-the-art HPC clusters (Tuncer et al., 2017b). Hence, this thesis focuses on power management and HPC workload management.

2.3.1 Power Management

Data center power management is a widely studied area. Researchers have attacked the problem using different control knobs including scheduling, job allocation, and server power limiting. As cooling constitutes up to 50% of the total data center power consumption (Dayarathna et al., 2016), power management should also account for the cooling of servers. We divide the related work into two groups: cluster-level power management and server-level cooling management.

Cluster-level Power Management

Moore et al. (Moore et al., 2005) use thermally-aware job allocation to minimize cooling costs. The performance-aware approaches proposed by Oxley et al. (Oxley

et al., 2018) and Kumar et al. (Kumar and Raghunathan, 2016) explore the tradeoffs among workload co-location, server heterogeneity, and the data center cooling and computational costs.

Other works focus on reducing energy costs through power provisioning to increase the overall utilization (Barroso et al., 2013), smart grid integration (Chen et al., 2014a), and renewable energy usage (Depoorter et al., 2015). These techniques focus on energy costs rather than controlling data center power consumption, and require the data centers to have the ability to limit their total power consumption.

Efficiently limiting data center power consumption implies maximizing performance under a total power limit through *power budgeting*. For power budgeting, Liu et al. considers renewable energy use in the data centers, and limits the computational power according to the renewable power availability (Liu et al., 2012). The technique proposed by Lim et al. aims to maximize the computational performance under a given power cap for virtualized systems (Lim et al., 2011). These techniques, however, do not budget the available power among cooling and computing, but solely focus on power distribution among servers.

There are some techniques that jointly address cooling and power budgeting. The unified workload, power, and cooling management framework introduced by Wang et al. adjusts server fan speeds to increase cooling efficiency (Wang et al., 2010). Their technique, however, is not aware of heat recirculation, which can significantly impact data center cooling costs (Chaudhry et al., 2015). The self-consistent power budgeting technique proposed by Zhan et al. (Zhan and Reda, 2015) maximizes the data center throughput using a multi-choice knapsack algorithm. Their solution allocates sufficient power for cooling and selects dynamic voltage-frequency scaling (DVFS) settings for each server to maximize throughput. Their policy depends on computationally expensive computational fluid dynamics simulations, which may not be

feasible for runtime optimization.

In today's data centers, many servers are over-cooled because the inlet temperature recommendations of manufacturers, which are based on the worst-case conditions (utilization, altitude, etc.), leave a large thermal headroom margin between the server internal temperatures and critical temperatures. Reducing this thermal headroom is challenging as it can lead to frequent violations of critical temperature thresholds.

CoolBudget, our cooling- and application-aware power budgeting technique (Tuncer et al., 2014) (Section 4.1), distinguishes from prior work as follows: CoolBudget is the first to exploit the thermal headroom between the server internal temperatures and the critical thresholds using accurate power, performance, and temperature models that we validate based on measurements on a real enterprise server. By reducing this thermal headroom, CoolBudget increases the data center air temperatures and decreases cooling power consumption. Moreover, unlike prior work, CoolBudget uses *fair speedup* as an optimization goal to achieve both high performance and fairness among workloads. By using *fair speedup*, CoolBudget sacrifices the performance of individual servers without decreasing fairness to improve the overall data center throughput.

Server-level Cooling Management

Several works have used fan control to reduce cooling costs in data centers. Han et al. (Han and Joshi, 2012) propose a runtime fan controller based on offline thermal modeling validated via simulation. Chan et al. (Chan et al., 2012) approach the fan control problem both from the energy minimization and fan-induced vibration perspective. These works do not consider leakage-cooling trade-offs.

Policies such as TAPO-server, proposed by Huang et al. (Huang et al., 2011), indirectly vary fan speed by controlling the processor thermal threshold at runtime, to search for the optimum fan speed in a reactive way. Recent work by Pradelle

et al. (Pradelle et al., 2014) uses a hill-climbing optimization technique that relies on utilization as a proxy variable for the estimation of heat dissipation which is not sufficient to select the optimum cooling for an arbitrary workload. These policies reactively control cooling and can lead to thermal overshoots.

Our leakage-aware fan control policy (Zapater et al., 2015a) (Section 4.2) is the first leakage-aware proactive cooling management strategy that is robust against arbitrary workloads. Using empirical models that we develop and validate on a real enterprise server, our policy minimizes the temperature-dependent portion of the server power consumption and proactively avoids thermal violations.

2.3.2 Workload Management in HPC Systems

The workload management in HPC systems has two main stages: (1) scheduling, which is the decision on when to run which job, and (2) topology mapping, which is the decision on where (i.e, on which nodes) to run a job.

Scheduling of parallel jobs in an HPC system has been extensively studied in the last decades (Feitelson et al., 1997). Various optimization techniques have been proposed using multiple job queues, where each queue serves a different job type (Lawson and Smirni, 2002), using soft job deadlines (He et al., 2004), and using backfilling based on job running times estimated by users (Talby and Feitelson, 1999) or statistical predictors (Tsafrir et al., 2007).

Topology mapping is the placement of parallel HPC application tasks (e.g., MPI ranks) onto the available compute nodes. HPC applications typically run many times on the same system with different input data and perform operations such as complex scientific simulations and financial forecasting. Most of these applications have a specific communication pattern, which can be extracted in the form of weighted graphs via source code instrumentation (Preissl et al., 2010) or via analysis of historical communication traffic (Zhai et al., 2009). Hence, topology mapping can be expressed

as mapping an applications' communication graph onto machines network graph, which is an NP-hard problem (Hoefler and Snir, 2011).

Topology mapping can be used to reduce the running time of HPC applications, which can spend over 50% of their execution time for communication, by placing highly-communicating tasks close to each other (Deveci et al., 2014). In today's HPC systems, topology mapping is performed in two steps: First, the system software *allocates* available compute nodes to an incoming job without having visibility into the job's communication graph, and then, the application software *maps* the individual application tasks onto the allocated compute nodes.

Bhatelé et al. proposed task mapping techniques such as step embedding and folding to map 2D stencil applications into 2D and 3D mesh machines (Bhatelé et al., 2010), as well as traversal and affine mapping algorithms to map irregular graphs into meshes (Bhatelé and Kalé, 2011). These techniques focus on contiguous allocation, where each application is assigned to a contiguous block of machine nodes. Although contiguous allocation increases application locality, it decreases the machine utilization (Subramani et al., 2002). Furthermore, non-contiguous allocation is used in many recent HPC systems such as Cray XC and XK series[1].

For machines with non-contiguous allocation, Albing et al. proposed node allocation techniques using different curves to remove the restrictions on the network topology and machine dimensions imposed by Hilbert curves (Albing et al., 2011). These solutions limit the allocation performance by ignoring network links outside the given curve. Other researchers have introduced clustered allocation schemes to overcome such limitations. Bender et al. proposed the MC1x1 algorithm, which is a $(2-1/2d)$ approximation to the optimal solution for minimizing the average pairwise L1 distance of tasks in a d-dimensional mesh when allocating k processors (Bender et al., 2008). Their techniques are applicable only on mesh and torus topologies, and

[1]Cray: `http://www.cray.com`

have primarily focused on stencil communication patterns (Balzuweit et al., 2016). Deveci et al. considered multi-level partitioning of both the machine and application geometries (Deveci et al., 2014).

With the emergence of a novel network topologies such as dragonfly (Kim et al., 2008), researchers proposed node allocation strategies specifically for the new network topologies. Jain et al. (Jain et al., 2014) compared the performance of six dragonfly-specific job allocation policies. They observed that random allocation is generally beneficial in spreading network traffic and reducing communication hot spots. Budiardja et al. (Budiardja et al., 2013) showed that spreading jobs to all groups during allocation distributes the network traffic, and thus, reduces congestion.

Our job placement policy for HPC systems, PaCMap (Tuncer et al., 2015) (Section 4.3), is the first policy that proposes using a holistic view on topology mapping by combining the system-driven node allocation and the application-driven task mapping algorithms. As we demonstrate in Section 4.3.2, this holistic view leads to a more efficient job placement compared to existing techniques, reducing application communication volume by up to 30% in terms of *hop-bytes*. Additionally, PaCMap is not limited to stencil applications and is applicable on machines that supports non-contiguous allocation.

Chapter 3

Automated Problem Diagnosis in Large-scale Computing Systems

Detection and diagnosis of misbehavior and failure in large-scale computing systems (e.g., HPC clusters and cloud platforms) have traditionally relied on the experience and expertise of human operators. By continuously monitoring and analyzing system logs, performance counters, and application resource usage patterns, system operators can assess system health and identify failures and anomalies. As system size and complexity grows, such manual processing becomes increasingly time-consuming and error-prone. Hence, automated problem diagnosis is essential for the efficient operation of future large-scale computing systems.

The majority of existing tools focus on anomaly detection rather than diagnosis (i.e., identifying the type of the anomaly). These tools largely rely on human operators to discover the root causes of anomalies, leading to delayed mitigation. Diagnosis is a substantially harder problem than detection as it requires deep understanding of healthy and anomalous behavior of various system components. Begin able to pinpoint the root cause of an anomaly becomes more challenging given the vast amount of noisy and high-dimensional data collected from different system layers.

In this chapter, we propose two novel frameworks for automated problem diagnosis. First, we introduce our performance anomaly diagnosis framework (Tuncer et al., 2017a), which identifies anomalies that do not necessarily cause failures. This framework leverages machine learning algorithms to learn and identify the signatures

of previously-observed anomalies based on compute node resource usage patterns. Second, we focus on enabling the use of software-specific data for automated problem diagnosis. We specifically focus on cloud software configurations, which are among the leading causes of service disruptions and outages in the cloud (Yin et al., 2011). We design and implement *ConfEx*, a configuration analytics framework for multi-tenant cloud environments. *ConfEx* enables the discovery and extraction of configuration information from unlabeled cloud system instances such as VMs and containers for the detection of misconfigurations via intelligent analytics.

3.1 Diagnosis of Performance Anomalies

Performance anomalies (such as thermally-induced CPU throttling) lead to performance variation, waste of compute resources (Bhatelé et al., 2013), and poor scheduling (Tsafrir et al., 2007) without necessarily causing failures. The impact of such anomalies is more prominent in HPC clusters where applications run for hours or even days on hundreds of compute nodes in parallel. These anomalies manifest themselves in the resource usage patterns of applications. However, identifying anomalies is challenging given the vast amount of noisy and high-dimensional resource usage data being collected via system monitoring infrastructures (Agelastos et al., 2014).

We design a novel framework to automatically detect compute nodes suffering from previously observed anomalies at runtime and identify the type of the anomaly independent of the applications running on the compute nodes. To detect and diagnose anomalies, we first extract easy-to-compute statistical features from sliding time series windows of performance metrics (e.g., number of network packets received) and resource usage metrics (e.g., CPU utilization), which are already being collected in lare-scale computing systems. Then, using human-interpretable machine learning algorithms, we identify the subset of features required for detecting the target anomalies

and generate concise signatures of these anomalies. At runtime, we compute only the necessary statistical features and compare them against the generated anomaly signatures to detect and diagnose anomalies. With a diverse training set that represents the expected application characteristics, our framework enables anomaly detection even when running applications that are not observed during the training.

We evaluate our framework on a Cray XC30m supercomputer and on a virtualized HPC cluster using multiple benchmark suites and demonstrate that our approach successfully diagnoses anomalies even when running applications that are not used during training. We demonstrate that our approach effectively identifies 98% of synthetic anomalies while leading to only 0.08% false anomaly alarms with an *F-score* over 0.99 where the *F-score*s of other state-of-the-art techniques remain below 0.94.

3.1.1 Using Machine Learning for Online Anomaly Diagnosis

Our goal is to quickly and accurately detect whether a compute node is *anomalous* (i.e., experiencing anomalous behavior) and classify the type of the anomaly (e.g., network contention, orphan process, etc.) at runtime, independent of the application that is running on the compute node. We target anomalies that are caused by applications or system software/hardware such as orphan processes, memory leaks, and shared resource contention. Once an anomaly is detected, mitigative measures can be taken promptly by administrators, users, or automated management mechanisms.

To detect and classify anomalies, we propose an automated approach based on machine learning. Figure 3·1 shows an overview of our framework. We leverage historical resource usage and performance data that is collected from healthy and anomalous nodes to learn the signatures of target anomalies. As an HPC application can run on multiple nodes in parallel, if any of these nodes is anomalous, the entire application's resource usage and performance patterns may be affected. Hence, our training excludes data collected from any node that is used together with an anomalous node by

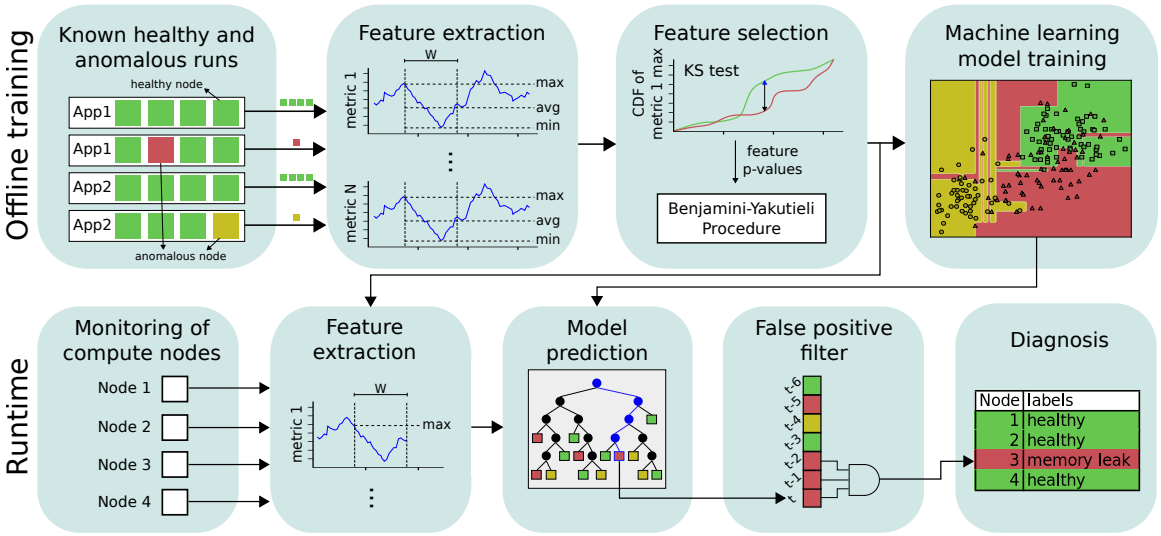


Figure 3·1: Our anomaly diagnosis framework. In the offline training phase, we use resource usage and performance data from known healthy and anomalous runs to identify statistical time series features that are useful to distinguish anomalies. The selected features are then used by machine learning algorithms to extract concise anomaly signatures. At runtime, we generate features from the recently observed resource usage and performance data, and predict the anomalies using the machine learning models. We raise an anomaly alarm only if the anomaly prediction is persistent across multiple sliding windows.

the same job. Using the data collected from known healthy and anomalous runs, we extract and identify the statistical features that are useful to detect these anomalies. Then, based on these features, we use machine learning algorithms to extract concise anomaly signatures. At runtime, we monitor the compute nodes and extract only the statistical features that are required for anomaly detection, and compare these features with the anomaly signatures. The remainder of this section explains these steps in detail.

Monitoring and Feature Extraction

We leverage data that is already periodically collected in HPC systems to assess system health. This data typically consists of time series of resource usage and per-

formance metrics such as CPU utilization, number of network packets received, and power consumption. Our framework does not depend on a specific set of collected metrics and can be coupled with a variety of HPC monitoring tools such as Ganglia[1].

For each collected metric, we keep track of the recently observed W values in a sliding window time series, and calculate the following statistical features:

- The minimum and the maximum values;
- Percentile values (5^{th}, 25^{th}, 50^{th}, 75^{th}, and 95^{th});
- The first four moments (i.e., mean, variance, skewness, and kurtosis).

The above features retain the time series characteristics and bring substantial computational and storage savings compared to directly using the raw values. To enable easy scaling, we extract these statistical features from individual compute nodes and do not account for the interaction and correlation between multiple nodes.

In each measurement time step, the features are calculated in at most $O(\log W)$ computational complexity for a sliding window size of W. With a constant and small W, this enables us to generate features at runtime with negligible overhead. The value of W is determined offline based on the target anomalies and the sampling frequency of the monitoring infrastructure (see Sec. 3.1.3 for details). While using a large window size typically makes anomaly detection easier, it delays anomaly detection.

Feature Selection

HPC system monitoring infrastructures may collect hundreds of resource usage and performance metrics per compute node (Agelastos et al., 2014). Performing anomaly diagnosis based on only a subset of these metrics and calculating only the statistical

[1]Ganglia monitoring system: `ganglia.info`

features that are useful for diagnosis can save significant computational overhead. Furthermore, feature selection can improve the accuracy of classification algorithms.

During training, we first generate all features, and then identify the features that are useful for anomaly detection using the *Kolmogorov-Smirnov* (KS) test (Massey Jr, 1951) together with the *Benjamini-Yakutieli* procedure (Benjamini and Yekutieli, 2001). This methodology has been shown to be successful for selection of time series features for regression and binary classification (Christ et al., 2016).

For a given feature that is extracted from an anomalous node, the KS test compares the cumulative distribution function (CDF) of that feature with the CDF of the same feature when running the same application without any anomaly. The divergence of these two CDFs is an indication of the statistical dependence of that feature and the anomaly (see the feature selection step in Fig. 3·1). Based on comparing the two CDFs, the KS test provides a p-value, which quantifies the probability of the given feature being relevant for the prediction of the target anomaly.

The p-value of each feature is then used in the Benjamini-Yakutieli procedure. The Benjamini-Yakutieli procedure determines the number of features to select based on a given *expected false discovery rate* (FDR), which is the proportion of useful features among all discarded features. As discussed in Sec. 3.1.3 in detail, the FDR parameter has negligible impact on the number of features selected in our dataset.

The feature selection methodology described above has been proposed for binary classification (Christ et al., 2016). We adapt this methodology for anomaly diagnosis via multi-class classification as follows: We select the useful features for each anomaly-application pair in the training data. We then use the union of the selected features to train machine learning models. This way, even if a feature is useful only to detect a specific anomaly when running a specific application, that feature is included in our analysis.

Model Training

We generate the selected features using the data collected from the individual nodes that are used in the known healthy and anomalous runs. We use these features to train supervised machine learning models where the label of each node is given as the type of the observed anomaly on that node (or healthy). In the absence of labeled data for anomalies, the training can be performed using data collected while running a diverse set of applications with *synthetic anomalies*, which are programs that mimic real-life anomalies.

With training data from a diverse set of applications that represent the expected workload characteristics in the target HPC system, machine learning algorithms extract the signatures of anomalies independent of the applications running on the compute nodes. This allows us to identify previously observed anomaly signatures on compute nodes even when the nodes are running an *unknown* application that is not used during training. Hence, our framework does not need any application-related information from workload managers such as Slurm[2].

We focus on tree-based machine learning models in our framework based on the following three observations: First, our earlier studies demonstrated that tree-based machine learning algorithms are superior for detecting anomalies in HPC systems compared to algorithms such as SVM and kNN (Tuncer et al., 2017a). Second, tree-based algorithms, in general, generate easy-to-understand models that lend themselves to scrutinization by domain experts as opposed to models that provide low observability into their decision process and reasoning (such as neural networks). Third, tree-based models allow us to further reduce the set of features that needs to be generated as follows: For each feature, the learning algorithms calculate how well the feature distinguishes individual classes in the training data using the *Gini index*

[2]Slurm Workload Manager: `https://slurm.schedmd.com/`

criterion, and discard the features that are not helpful for the classification. Hence, at runtime, we calculate only the features that are selected by both the feature selection phase and the learning algorithms, further reducing the computational overhead.

Runtime Diagnosis

At runtime, we monitor resource usage and performance metrics from individual nodes. In each monitoring time step, we use the last W collected metric data to generate the sliding window features that are selected during the training phase. These features are then used by the machine learning models to predict whether each node is anomalous along with the type of the anomaly.

As our goal is to detect anomalies that cause performance variations, raising a false anomaly alarm may waste the time of system administrators or even cause artificial performance variations if the anomaly alarm initiates an automated mitigative action. Hence, avoiding false alarms is more important for us than missing anomalies.

To increase robustness against false alarms, we do not raise an alarm when a node is predicted as anomalous based on data collected from a single sliding window. Instead, we consider an anomaly prediction as valid only if the same prediction persists for C consecutive sliding windows. Otherwise, we label the node as healthy. Increasing the parameter C, which we refer to as the *confidence threshold*, decreases the number of false anomaly alarms while delaying the detection time. We empirically select a confidence threshold as described in Sec. 3.1.3.

3.1.2 Experimental Methodology

Due to the lack of published and comprehensive labeled data on application resource usage patterns with and without anomalous behavior, we evaluate the efficacy of our framework by running controlled experiments on an HPC testbed. We mimic anomalies observed in this testbed by running synthetic programs simultaneously

with various HPC applications, and diagnose the anomalies using our framework and selected baseline techniques. This section presents the details on our target HPC testbed as well as the synthetic anomalies and the applications we use.

Large-scale Systems Used for Experiments

We evaluate our framework primarily on Volta, a Cray XC30m testbed supercomputer, located at Sandia National Laboratories. In addition, we show that our framework is applicable in different types of data centers by diagnosing anomalies on a virtualized cluster in Massachusetts Open Cloud (MOC)[3].

Volta: Volta consists of 52 compute nodes, organized in 13 fully connected switches with 4 nodes per switch. Each node has 64GB of memory and two sockets, each with an Intel Xeon E5-2695 v2 CPU with 12 2-way hyper-threaded cores, leading to a total of 48 threads per node.

Volta is monitored by the Lightweight Distributed Metric Service (LDMS) (Agelastos et al., 2014). At every second, LDMS collects 721 metrics described below:

- Memory metrics (e.g., free, cached, active, inactive, dirty memory);
- CPU metrics (e.g., per-core and overall idle time, I/O wait time, hard and soft interrupt counts, context switch count);
- Virtual memory statistics (e.g., free, active/inactive pages; read/write counts);
- Cray performance counters (e.g., power consumption, dirty, write-back counters; received/transmitted bytes/packets);
- Aries network interface controller counters (e.g., received/transmitted packets, flits, blocked packets).

[3]Massachusetts Open Cloud: `https://massopen.cloud/`

Out of the collected 721 metrics, we discard 158 metrics that provide system constants such as the memory page size. In addition, we convert the metrics that are incremental counters to the number of events that occurred over the sampling interval (e.g., interrupts per second) by taking their derivative.

MOC: MOC is an infrastructure as a service (IaaS) cloud running in the Massachusetts Green High Performance Computing Center, which is a 15 megawatt data center dedicated for research purposes.

In MOC, we use virtual machines (VMs) managed by OpenStack[4], where the compute nodes are VMs running on servers which communicate through the local area network. Although we take measurements from the VMs, we do not have control or visibility over other VMs running on the same host. Other VMs naturally add noise to our measurements, making anomaly detection more challenging.

We periodically collect resource usage data using the monitoring infrastructure built in MOC (Turk et al., 2016). Every 5 seconds, we collect 53 metrics, which are subset of node-level metrics read from the Linux `/proc/stat` and `/proc/meminfo` pseudo-files as well as `iostat` and `vmstat` tools. The specific set of collected metrics are selected by MOC developers and can be found in the public MOC code repository[5].

Applications

As the impact of performance anomalies are especially high for parallel HPC applications, we evaluate our framework using a diverse set of HPC benchmark applications with which we can obtain 10-15 minute running times using three different input configurations. This running time range is a typical average for the jobs submitted to various supercomputers (Feitelson et al., 2014).

[4]OpenStack: `https://www.openstack.org/`

[5]MOC Sensu-Puppet module: `github.com/CCI-MOC/kilo-puppet/tree/liberty/sensu`

Table 3.1: Applications used in evaluation

Benchmark	Application	# of MPI ranks	Description
NAS Parallel Benchmarks (Bailey et al., 1991)	bt	169	Block tri-diagnonal solver
	cg	128	Conjugate gradient
	ft	128	3D fast Fourier transform
	lu	192	Gauss-Seidel solver
	mg	128	Multi-grid on meshes
	sp	169	Scalar penta-diagonal solver
Mantevo Benchmark Suite (Heroux et al., 2009)	miniMD	192	Molecular dynamics
	CoMD	192	Molecular dynamics
	miniGhost	192	Partial differential equations
	miniAMR	192	Stencil calculation
Other	kripke (Kunen et al., 2015)	192	Particle transport

Table 3.1 presents the applications we use in our evaluation. The NAS Parallel Benchmarks (NPB) are widely used by the HPC community as a representative set of HPC applications. The Mantevo Benchmark Suite is developed by Sandia National Laboratories as proxy applications for performance and scaling experiments. These applications mimic the computational cores of various scientific workloads. In addition, we use kripke, which is another proxy application developed by Lawrence Livermore National Laboratory for the analysis of HPC system performance. All applications in Table 3.1 use MPI for inter-process and inter-node communication.

On our target platforms, we submit parallel applications that use four compute nodes, where the nodes are utilized as much as possible by using one MPI rank per core. In *bt* and *sp* applications, we do not fully utilize the compute nodes as these applications require the total number of MPI ranks to be the square of an integer. Similarly, *cg*, *ft*, and *mg* require the total number of MPI ranks to be a power of two.

In addition to the 4-node application runs, we experiment with 32-node runs with four applications (kripke, miniMD, miniAMR, and miniGhost), with which we can obtain 10-15 minute running times for two input configurations. Using these 32-node runs, we show that our framework can diagnose anomalies when running large

applications after being trained using small applications (see Sec. 3.1.3).

In our experiments, each application has three different input configurations, resulting in a different running time and resource usage behavior. For example, in miniMD, which is a molecular dynamics application that performs the simulation of a Lennard-Jones system, one input configuration uses the values of certain physical constants, while another configuration replaces all constants with the integer 1 and performs a unitless calculation.

The runs of the same application with the same input configuration leads to slightly different behavior as the benchmarks randomize the application input data and also due to the differences in the compute nodes allocated by the system software. Hence, we repeat each application run five times with the same input configuration but with different randomized input data and on different compute nodes.

Before generating features from an application run, we remove the first and last 30 seconds of the collected time series data to strip out the initialization and termination phases of the application. Note that larger applications may need a duration longer than 30 seconds to initialize or to terminate.

Synthetic Anomalies

The goal of our framework is to characterize and identify the *signatures* of previously observed anomalies. To evaluate our approach with controlled experiments, we design *synthetic anomalies* that mimic commonly observed performance anomalies caused by application- or system-level issues.

As shown in Table 3.2, we experiment with three types of anomalies. Orphan processes typically result from incorrect job termination. These processes continue to use system resources until their execution is completed or until they are forcefully killed by an external signal (Brandt et al., 2010; Brandt et al., 2009). Out-of-memory problems occur due to memory leaks or insufficient available memory on the compute

Table 3.2: Injected synthetic performance anomalies

Anomaly type	Synthetic anomaly name	Target subsystem
Orphan process/ CPU contention	dcopy	CPU, cache
	dial	CPU
Out-of-memory	leak	memory
	memeater	memory
Resource contention	linkclog	network

nodes (Agelastos et al., 2015). When there is not enough memory, the Linux out-of-memory killer will terminate jobs. Finally, contention of shared resources such as network links can significantly degrade performance (Bhatelé et al., 2013).

For each anomaly, we use six different *anomaly intensities* (2%, 5%, 10%, 20%, 50%, and 100%) to create various degrees of performance variation. We adjust the maximum intensities of *orphan process* and *resource contention* anomalies such that the anomaly increases the running time of the applications at most by 3X, which is in line with the performance variation observed in production systems (Skinner and Kramer, 2005). The intensity of the *out-of-memory* anomalies are limited by the available memory in the system, and add up to 10% overhead to the job running time without terminating the job. We do not mimic job termination due to out-of-memory errors as we primarily focus on performance variation rather than failures. We use the following programs to implement our synthetic anomalies:

1. *dcopy* allocates two equally sized matrices of `double` type, fills one matrix with a number, and copies it to the other matrix repeatedly, simulating CPU and cache interference. After 10^9 write operations, the matrix size is changed to cycle between 0.9, 5 and 10 times the sizes of each cache level when the intensity is 100%. The anomaly intensity scales the sizes of the matrices.

2. *dial* stresses a single CPU core by repeatedly generating random floating-point numbers and performing arithmetic operations. The intensity sets

the utilization of this process: Every 0.25 seconds, the program sleeps for $0.25 \times (1 - \text{intensity})$ seconds.

3. *leak* allocates a 200KB `char` array, which is scaled by the anomaly intensity. It fills this array with characters, sleeps for two seconds, and repeats the process. The allocated memory is never released, leading to a memory leak. After 10 iterations, the program restarts to avoid crashing the main program by consuming all available memory.

4. *memeater* allocates a 360KB `int` array, which is scaled by the anomaly intensity. It fills this array with random integers, and periodically increases the size of the array using `realloc` and fills in new elements. After 10 iterations, the program sleeps for 120 seconds and then restarts.

5. *linkclog* adds a delay before MPI messages sent out of and into the selected anomalous node by wrapping MPI functions. The duration of this delay is proportional to the message size and the anomaly intensity. We emulate network contention using message delays rather than using external jobs with heavy communication. This is because Volta's size and flattened butterfly network topology with fully-connected routers prevent us from clogging the network links with external jobs.

As discussed in detail in Sec. 3.1.3, our synthetic anomalies have different characteristics, allowing us to study various aspects of online anomaly diagnosis such as window size selection, feature selection, and sensitivity against anomaly intensities.

We perform our evaluation under two scenarios: (1) persistent anomalies, where an anomaly program executes during the entire application run, and (2) random-offset anomalies, where an anomaly starts at a randomly selected time while the application

is running. In the latter scenario, each application has two randomly selected anomaly start times throughout our experiments.

Implementation Details

We implement our framework in python. We use the *SciPy* package for the implemention of the KS test during feature selection and the *scikit-learn* package for the machine learning algorithms. We use three tree-based machine learning classifiers: *decision tree*, *adaptive boosting (AdaBoost)*, and *random forest*. In *scikit-learn*, the default parameters of these classifiers are tuned for a smaller feature set compared to ours. Hence, we increase the number of decision trees in AdaBoost and random forest classifiers to one hundred and set the maximum tree depth in AdaBoost to five. To avoid overfitting to our dataset, we do not extensively tune the classifier parameters.

While training the classifiers and the baseline techniques, we use periodic time series windows instead of sliding windows. This allows us to perform our extensive evaluation using hundreds of distinct training sets within a feasible total training duration. In a production environment, the models need to be trained only once unless the training set is expanded.

Evaluation Methodology

We collect resource usage and performance counter data during the following application runs: We run each application with three different input configurations and five repetitions for each input configuration. For each of these application runs, we inject one synthetic anomaly to one of four nodes used by the application during the entire run. We repeat these runs for every anomaly and three anomaly intensities (20%, 50%, and 100%) for each anomaly. For each above application run, we repeat the same run without any anomaly to generate a healthy data set. In total, the above experiments correspond to $11 \times 3 \times 5 \times 5 \times 3 \times 2 = 4950$ four-node application runs.

We use five fold stratified cross validation to divide the collected data into disjoint training and test sets as follows: We randomly divide our application runs into five disjoint equal-sized partitions where each partition has a balanced number of application runs for each anomaly. We use four of these partitions to train our framework and the baseline techniques, and the fifth partition for testing. We repeat this procedure five times where each partition is used once for testing. Furthermore, we repeat the entire analysis five times with different randomly-generated partitions.

Across all test sets, we calculate the following statistical measures to assess how well the anomaly detection techniques distinguish healthy and anomalous time windows from each other:

- *False alarm rate*: The percentage of the healthy windows that are predicted as anomalous (any anomaly type).
- *Anomaly miss rate*: The percentage of the anomalous windows that are predicted as healthy.

Additionally, we use the following measures for each anomaly type to assess how well the anomaly detection techniques diagnose different anomalies:

- *Accuracy*: The fraction of correct predictions to all predictions.
- *Precision*: The fraction of the number of windows correctly predicted with an anomaly type to the number of all predictions with the same anomaly type.
- *Recall*: The fraction of the number of windows correctly predicted with an anomaly type to the number of windows with the same anomaly type.
- *F-Score*: The harmonic mean of precision and recall.
- *Overall F-Score*: The F-score calculated using the weighted averages of precision and recall, where the precision and recall of each class is weighted by the number

of instances of that class. A naïve classifier that marks every window as healthy would achieve an overall F-score of 0.87 as approximately 87% of our data set consists of healthy windows.

We also evaluate the robustness of our anomaly detection framework against *unknown* anomaly intensities, where the framework is trained using certain anomaly intensities and tested with the remaining intensities. Additionally, we use the same approach for unknown application input configurations and unknown applications to show that we can detect anomaly signatures even when running an application that is not encountered during training.

In addition to the analyses using five fold stratified cross-validation, we perform the following three studies where our framework is trained with the entire data set used in the previous analyses: First, we demonstrate that our approach is not specific to four-node application runs by diagnosing anomalous compute nodes while running 32-node applications. Second, we evaluate our framework when running anomalies with low intensities (2%, 5%, and 10%), and demonstrate that we can successfully detect signatures of anomalies even when the anomaly intensities are lower than that observed during training. Third, we start synthetic anomalies while an application is running and measure the detection delay.

Baseline Methods

We implement two state-of-the-art algorithms for HPC anomaly detection as baselines to compare with our work: an independent component analysis based approach developed by Lan et al. (Lan et al., 2010), referred as "ICA-Lan", and a threshold-based fingerprinting approach by Bodik et al. (Bodik et al., 2010), referred as "FP-Bodik".

ICA-Lan: ICA-Lan (Lan et al., 2010) relies on Independent Component Analysis (ICA) (Comon, 1994) to detect anomalies on HPC systems. During training, ICA-Lan first normalizes the collected time series metrics and applies ICA to extract the

independent components of the data. Each of these features is a linear combination of the time series data of all metrics within a sliding window, where the coefficients are determined by ICA. ICA-Lan then constructs *feature vectors* composed of the top m independent components for each node and each sliding window.

To test for anomalies in an input sliding window, ICA-Lan constructs a feature vector with the coefficients used during training, and compares the Euclidean distance between the new feature vector and all the feature vectors generated during training. If the new feature vector is an outlier, the input is marked as anomalous. In the original paper (Lan et al., 2010), ICA-Lan can tell only whether a node is anomalous or not without classifying the anomaly. In our implementation, we generalize this method by replacing their distance-based outlier detector with a kNN (k Nearest Neighbor) classifier.

The original implementation of ICA-Lan uses the top $m = 3$ independent components. However, in Volta, we collect more than 25 times the number of metrics used in Lan et al.'s study. Hence, we need more independent components to capture the important features. As Lan et al. do not provide a methodology to select m, we study anomaly classification accuracy when using m within the [3-20] range on both Volta and MOC platforms.

As shown in Figure 3·2, while using $m = 3$ results in poor prediction performance

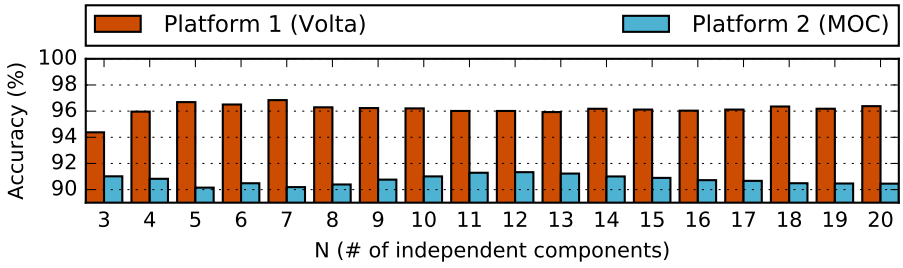


Figure 3·2: Classification accuracy of ICA-Lan w.r.t. number of independent components used in the algorithm. The data is obtained when using NPB applications (Tuncer et al., 2017a).

in Volta, the accuracy does not improve significantly for $m > 4$. $m = 7$ and $m = 12$ provide the highest accuracy on Volta and MOC, respectively. We settle on $m = 10$ as it provides a good middleground value that results in high accuracy.

FP-Bodik: FP-Bodik (Bodik et al., 2010) uses common statistical features and relies on thresholding to compress the collected time series data into feature vectors called *fingerprints*. Specifically, FP-Bodik first selects the metrics that are important for anomaly detection using logistic regression with L1 regularization. FP-Bodik then calculates the 25^{th}, 50^{th}, and 95^{th} percentiles of the selected time series metrics for each node and each sliding time series window. Then, a healthy range of these percentiles are identified from the healthy runs in the training phase. Next, each percentile is further replaced by a tripartite value (-1, 0, or 1) that represents whether the observed value is below, within, or beyond the healthy range, respectively. FP-Bodik constructs fingerprints that are composed of all the tripartite values for each node and each monitoring window. Finally, the fingerprints are compared to each other, and a fingerprint is marked as anomalous whenever its closest labeled fingerprint (in terms of L2 distance) belongs to an anomaly class.

3.1.3 Results

In this section, we first describe our parameter selection for sliding time series window size (W), the false discovery rate (FDR) during feature selection, and the confidence threshold (C). We then compare different anomaly detection and diagnosis methodologies in terms of classification accuracy and robustness against unknown applications and anomaly intensities that are not encountered during training. We demonstrate the generality of our framework by diagnosing anomalies during large application runs. Finally, we study the anomaly detection delay and the resource requirements of different diagnosis methodologies.

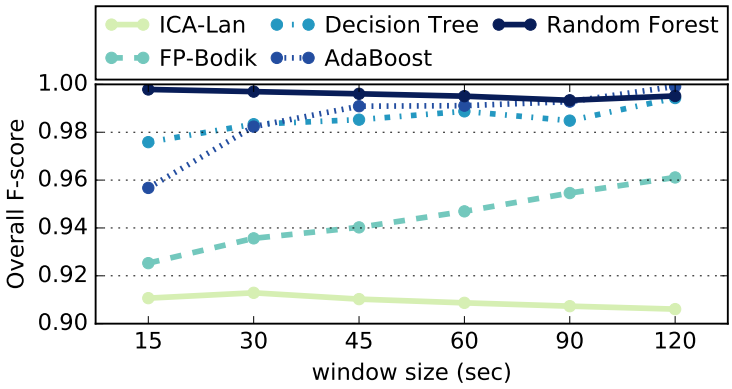


Figure 3·3: The impact of window size on the overall F-score. The classification is less effective with small window sizes where a window cannot capture the patterns in the time series adequately.

Window Size Selection

As discussed in Sec. 3.1.1, the size of the time series window that is used to generate statistical features may affect the efficacy of anomaly detection. While using a large window size allows capturing longer time series signatures in a window, it delays the anomaly detection.

Figure 3·3 shows the impact of window size on the overall F-score of baseline algorithms as well as our proposed framework with three different classifiers. The results presented in the figure is obtained using all features (i.e., without the feature selection step).

While the impact of window size on the overall F-score is below 3%, the F-scores of most classifiers tend to decrease with decreasing window size as small windows cannot capture the time series behavior adequately. The impact of the window size depends on the anomaly characteristics. This can be seen in Fig. 3·4, which depicts the per-class F-scores of the AdaBoost classifier for different window sizes. The F-score for the *memeater* anomaly decreases significantly with the decreasing window size. This is because the behavior of application runs with *memeater* is very similar to the healthy

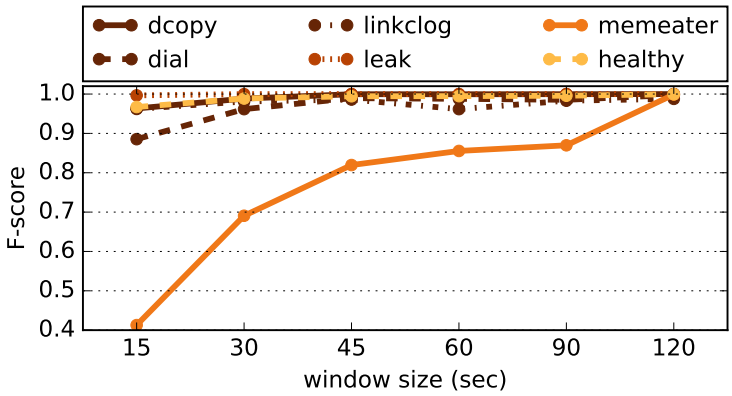


Figure 3·4: The impact of window size on the per-class F-scores of the AdaBoost classifier. The F-score of memeater anomaly decreases significantly as windows get shorter.

application behavior during *memeater*'s sleep phase, which is 120 seconds. Hence, as the window size gets smaller, more windows occur entirely within *memeater*'s sleep phase both in the training and the testing set, confusing the classifier on *memeater*'s signature. The reduction in the F-score of the healthy class due to this confusion is less significant than that of the *memeater* class because our dataset has 42 times more healthy windows than windows with *memeater*.

The window size in our framework needs to be determined based on the nature of the target anomalies and the system monitoring infrastructure. Based on the results in Figures 3·3 and 3·4, we conclude that a 45-second window size is a reasonable choice to accurately detect our target anomalies while keeping the detection delay low. For the rest of this section, we use a window size of 45 seconds.

Feature Selection

Feature selection is highly beneficial for reducing the computational overhead needed during online feature generation. Our framework's feature selection methodology identifies 43-44% of the 6193 features we generate as useful features to identify our target anomalies for an *expected False Discovery Rate* (FDR) range between 0.01%

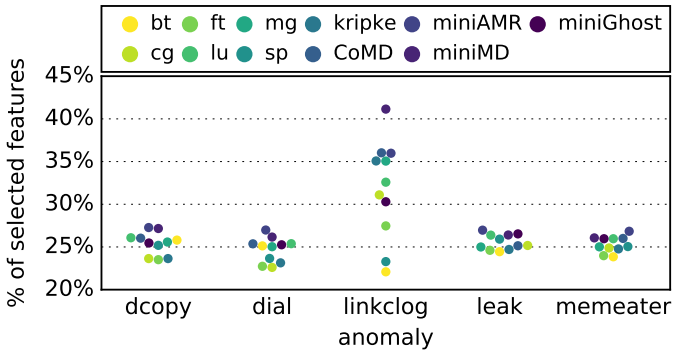


Figure 3·5: The percentage of features selected for different application-anomaly pairs. Less than 28% of the features are useful for the detection of target anomalies except for *linkclog*, where up to 41% of the features are useful depending on the running application.

and 10%. As the FDR parameter has a negligible impact on the number of selected features, we simply use FDR = 1%.

Figure 3·5 shows the percentage of selected features for each application-anomaly pair. While less than 28% of the features are identified as useful for detecting *dcopy*, *dial*, *leak*, and *memeater* anomalies, up to 41% of the features can be used as indicators of *linkclog*. For the applications where the *linkclog* anomaly is detrimental for performance such as miniMD, more features are marked as useful. This is because the resource usage patterns of such applications change significantly when suffering from *linkclog*. On the other hand, applications such as *bt* and *sp* are not affected by *linkclog* as they either have a light communication load or use non-blocking MPI communication. Fewer features are marked as useful for such applications. Overall, 43% of features are marked as useful, reducing the computational overhead of feature generation at runtime by more than 50%.

Using a reduced feature set can also improve the effectiveness of certain machine learning algorithms. For example, random forest contains decision trees that are trained using a subset of randomly selected features. In the absence of irrelevant features, the effectiveness of random forest slightly increases as shown in Table 3.3.

Table 3.3: The impact of feature selection on anomaly detection. The effectiveness of random forest improves when using only the selected features.

	False alarm rate		Anomaly miss rate	
	All features	Selected features	All features	Selected features
Decision tree	1.5%	1.5%	2.0%	2.0%
AdaBoost	0.8%	0.8%	1.9%	1.9%
Random forest	0.2%	0.1%	1.8%	1.4%

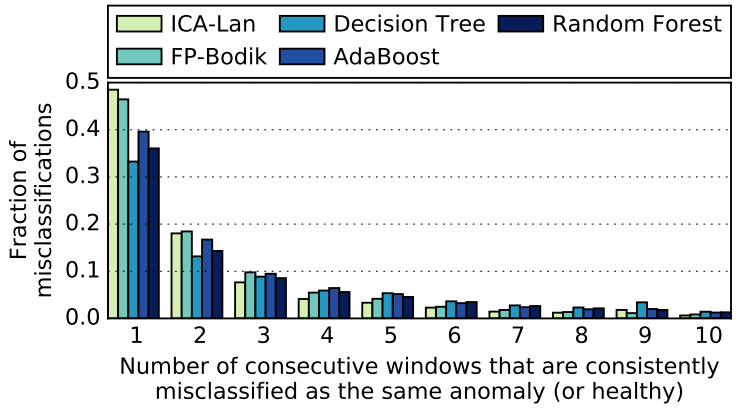


Figure 3·6: Distribution of consecutive misclassifications. Most misclassifications do not persist for more than a few consecutive windows.

The effectiveness of other learning algorithms such as decision tree and AdaBoost are not impacted by feature selection as feature selection is embedded in these algorithms.

The Impact of the Confidence Threshold

As shown in the Fig. 3·6, the majority of the misclassifications persist only for a few consecutive windows in all classifiers. To reduce false anomaly alarms, we filter out the non-persistent misclassifications using a confidence threshold, C. A prediction is considered as valid only if it persists for C consecutive windows.

Figure 3·7 shows the change in the false anomaly alarm rate and the anomaly miss rate when using various confidence thresholds. Using $C = 5$ reduces the false alarm rate by 23-44% when using the machine learning algorithms and by 25-69% when using

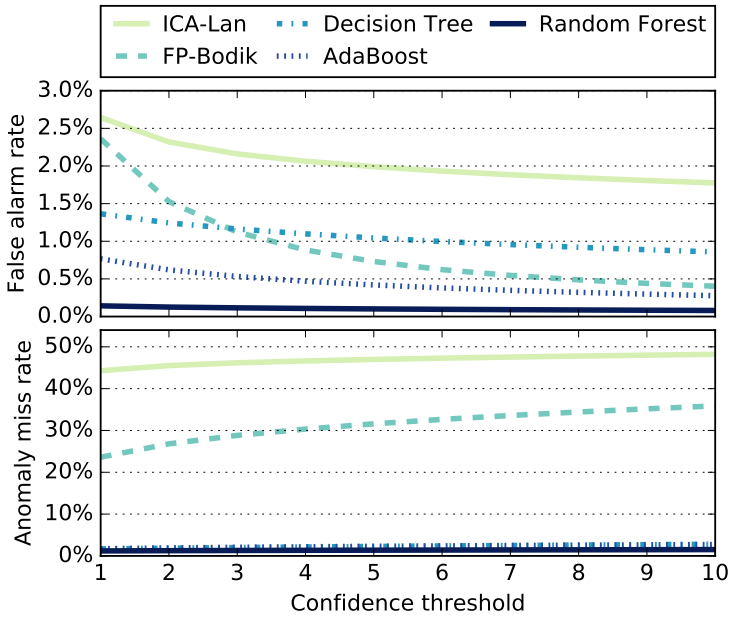
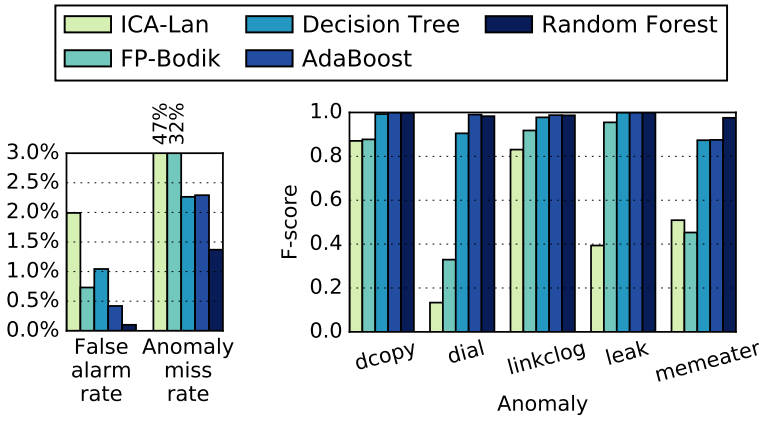


Figure 3·7: The impact of confidence threshold on the false alarm rate. Filtering out non-persistent anomaly predictions using a confidence threshold reduces the false alarm rate while increasing anomaly miss rate.

the baseline anomaly detection algorithms. On the other hand, the anomaly miss rate increase by 15-30% and 6-34% when using the machine learning algorithms and the baselines, respectively. To keep the anomaly detection delay low while decreasing the false alarm rate in all classifiers, we use a confidence threshold of $C = 5$.

Anomaly Detection and Classification

Figure 3·8 presents the false positive and negative rates for anomaly detection as well as F-scores for the anomaly types we study for the 5-fold stratified cross validation. Our proposed machine learning based framework consistently outperforms the baseline techniques in terms of anomaly miss rate and F-scores. While the FP-Bodik baseline can achieve fewer false alarms, it misses nearly a third of the anomalies. As decision tree is a building block of AdaBoost and random forest, it is simpler and consistently underperforms AdaBoost and random forest as expected. The best



(a) Detection **(b)** Diagnosis

Figure 3·8: Anomaly detection and diagnosis statistics of various classifiers using 5-fold stratified cross-validation. Random forest correctly identifies 98% of the anomalies while leading to only 0.08% false anomaly alarms.

overall performance is achieved using random forest, which misses only 1.7% of the anomalous windows and classifies the target anomalies nearly ideally. It raises false anomaly alarms only for 0.08% of the healthy windows.

Note that the false anomaly alarm rate can be reduced further by adjusting the confidence threshold as well as by tuning the parameters of machine learning algorithms. However, even with a false anomaly alarm rate of 0.08%, our framework can be used in a production environment with thousands of nodes to help administrators diagnose the root cause of performance anomalies observed in the system.

The F-scores corresponding to *memeater* are lower than those for other anomalies as the classifiers tend to mispredict *memeater* as healthy (and vice versa) during the sleep phase of *memeater*, where its behavior is similar to healthy application runs. Due to the randomized feature selection in random forest, random forest also uses the features that are not the primary anomaly indicators but are still helpful on anomaly detection. Thus, random forest is more robust against noise in the data, and can still detect *memeater* where other classifiers are unsuccessful. Also note that the *memeater*

anomaly degrades performance only by up to 10% while the *dcopy*, *dial*, and *linkclog* anomalies can degrade application performance by up to 300%. Hence, the detection of *memeater* is harder but is also less critical compared to other anomalies.

As shown in Fig. 3·8, FP-Bodik misses nearly a third of the anomalous windows. This is because even though FP-Bodik performs metric selection through L1 regularization, it gives equal importance to all the selected metrics. However, metrics should be weighted for robust anomaly detection. For instance, network-related metrics are more important for detecting network contention than memory-related metrics. Tree-based machine learning algorithms address this problem by prioritizing certain features through putting them closer to the root of a decision tree. Another reason for FP-Bodik's poor anomaly miss rate is that FP-Bodik only uses 25^{th}, 50^{th}, and 95^{th} percentiles in the time series data. Other statistics such as variance and skew are also needed to capture more complex patterns in the time series.

The F-scores achieved by ICA-Lan is lower than that of both FP-Bodik and our proposed framework primarily due to ICA-Lan's feature extraction methodology. ICA-Lan uses ICA to extract features from time series. This technique is commonly used for data analysis to reduce data dimensionality, and provides features that represent the *independent components* of the data. While these features successfully represent deviations, they are not necessarily able to capture anomaly signatures. We illustrate this by comparing the features and metrics that are deemed as important by ICA-Lan and random forest.

The most important ICA-Lan metrics are those with the highest absolute weight in the first ten independent components. In our models, the most important ICA-Lan metrics are the time series of idle time spent in various CPU cores. Idle CPU core time is indeed independent from other metrics in our data as some of our applications do not use all the available CPU cores in a node (see Sec. 3.1.2), and the decision

on which cores are used by an application is governed by the operating system of the compute nodes.

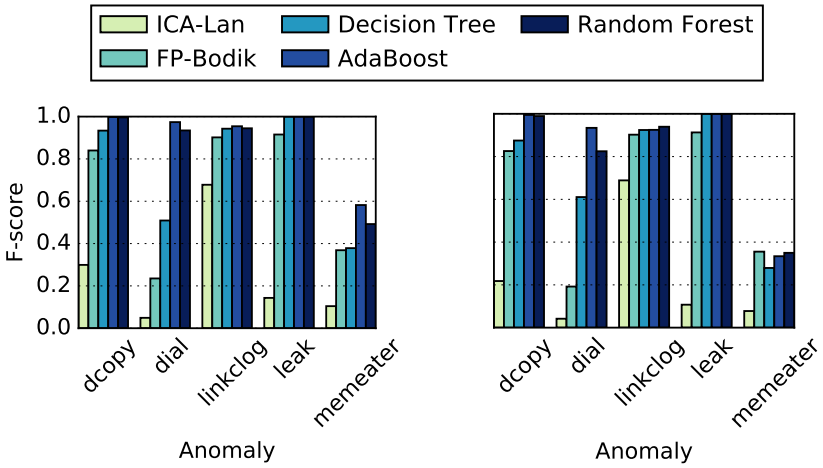
The most important random forest features are those that successfully distinguish different classes in the training data and are reported by the python *scikit-learn* package based on the normalized *Gini reduction* brought by each feature. In our random forest models, the most important features are calculated from time series metrics such as the number of context switches, the number of bytes and packets transmitted by fast memory access short messaging (a special form of point-to-point communication), total CPU load average, and the number of processes and threads created. These metrics are indeed different from those deemed important by ICA-Lan, indicating that the most important ICA-Lan metrics are not necessarily useful to distinguish anomalies.

Classification with Unknown Input Configurations

In a real supercomputer, it is not possible to know all input configurations for given applications during training. Hence, we evaluate the robustness of anomaly diagnosis when running applications with *unknown* input configurations that are not seen during training. For this purpose, we train our framework and the baseline techniques using application runs with certain input configurations and test only using the remaining input configurations.

Figure 3·9 shows F-scores for each anomaly obtained during our unknown input configuration study. Except for the *memeater* anomaly, our approach can diagnose anomalies with over 0.8 F-score even when the behavior of the applications are different than that observed during training due to the unknown input configurations. The F-scores tend to decrease when more input configurations are unknown.

There are two reasons for the decreasing F-scores: First, removing certain input configurations from the training set reduces the training set size, resulting in a less



(a) One unknown input configuration

(b) Two unknown input configurations

Figure 3·9: Anomaly diagnosis statistics when the training data excludes certain *unknown* input configurations for each application and the testing is done using only the excluded input configurations.

detailed modeling of the anomaly signatures. Second, the behavior of an application with an unknown input configuration may be similar to an anomaly, making diagnosis more difficult. One such example is the *memeater* anomaly, where healthy application runs with certain unknown input configurations are predicted as *memeater* by the classifiers.

Classification with Unknown Applications

In a production environment, we expect to encounter applications other than those used during offline training. To verify that our framework can diagnose anomalies when running unknown applications, we train our framework and the baseline techniques using all applications except for one application that is designated as the unknown application, and test using only that unknown application.

Figure 3·10 presents the anomaly detection results when we repeat this procedure where each applications is selected once as the unknown application. With the AdaBoost and random forest classifiers, the proposed framework achieves over

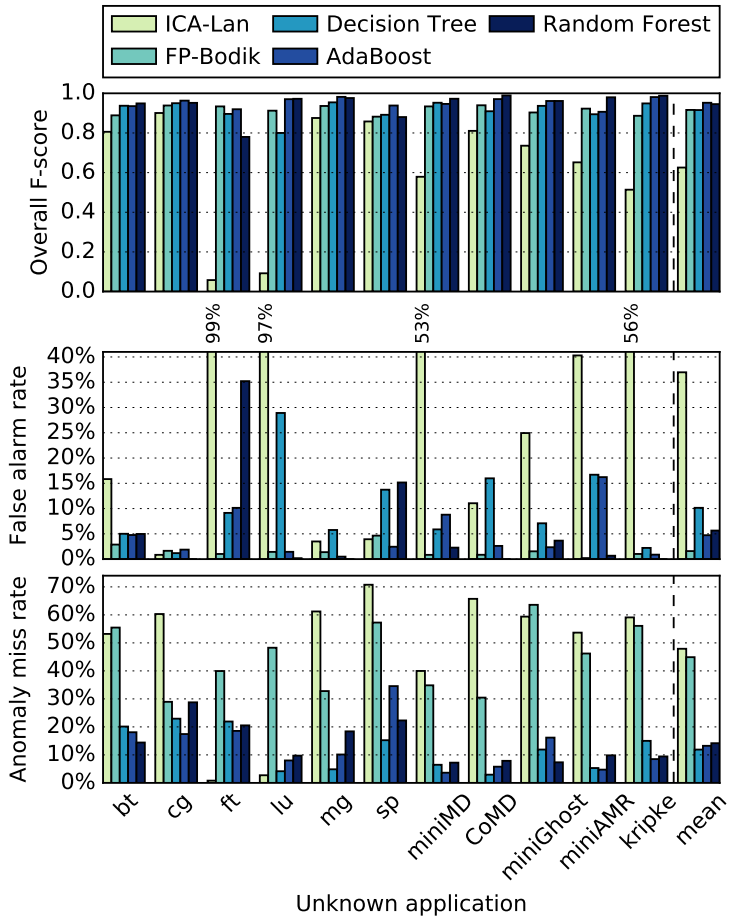
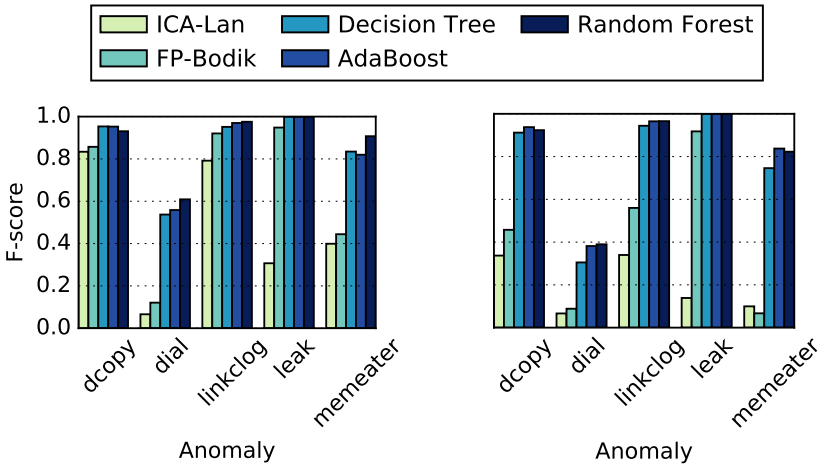


Figure 3·10: Anomaly detection and diagnosis statistics when the training data excludes one application and the testing is done using only the excluded *unknown* application. With the proposed framework, random forest achieves over 0.97 F-score on the average.

0.94 average F-score, while the average F-score of ICA-Lan is below 0.65. FP-Bodik achieves a similar F-score but misses 44% of the anomalous windows when unknown applications are running.

With random forest, the false alarm rate stays below 5% in all cases except when the unknown application is *ft* or *sp*. This rate can be further reduced by increasing the confidence threshold, C, at the expense of delaying anomaly detection.

When the characteristics of applications are significantly different than those ob-



(a) One unknown anomaly intensity

(b) Two unknown anomaly intensities

Figure 3·11: Anomaly diagnosis statistics when the training data excludes certain *unknown* anomaly intensities and the testing is done using only the excluded anomaly intensity.

served during training, the classifiers mispredict the healthy behavior of these applications as one of the anomalies. When such cases are encountered, the framework should be re-trained with a training set that includes the healthy resource usage and performance data of these applications. False alarm rates tend to increase with unknown applications as unknown applications lead to inconsistent consecutive predictions, which are filtered out during testing. Based on these results, we observe that our approach is robust against unknown applications.

Classification with Unknown Anomaly Intensities

We also study how the diagnosis effectiveness is impacted by unknown anomaly intensities where we use distinct anomaly intensities during training and the remaining intensities during testing. Figure 3·11 shows the F-scores for different anomalies with unknown intensities. High F-scores indicate that the anomaly signatures do not change significantly when their intensity changes, and our proposed framework can successfully diagnose anomalies. For example, the memory usage gradually increases

with the *leak* anomaly in all intensities. Hence, *leak* can be detected based on the skew in the time series of the allocated memory size metric.

The slight decrease in the F-scores is mainly caused by the reduction in the training set. In the *dial* anomaly, however, the intensity determines the utilization of the anomaly program. That means with an intensity of 20%, the anomaly sleeps 80% of the time, minimizing its impact on the application performance as well as its signature in resource usage patterns. Hence, when trained with high intensities, the algorithms tend to misclassify low intensity *dial* as healthy application behavior.

Diagnosing Anomalies with Low Intensities

We study anomaly diagnosis effectiveness when the anomalies have $1/10^{\text{th}}$ of the intensities we have used so far. In this subsection, we train the framework with the anomaly intensities 20%, 50%, and 100%, and test with the anomaly intensities 2%, 5%, and 10%. Figure 3·12 shows the resulting per-anomaly F-scores. In *dcopy*, *leak*, and *memeater*, the intensity determines the size of the memory used in the anomaly program. With low anomaly intensities, random forest diagnoses the signatures of these anomalies with

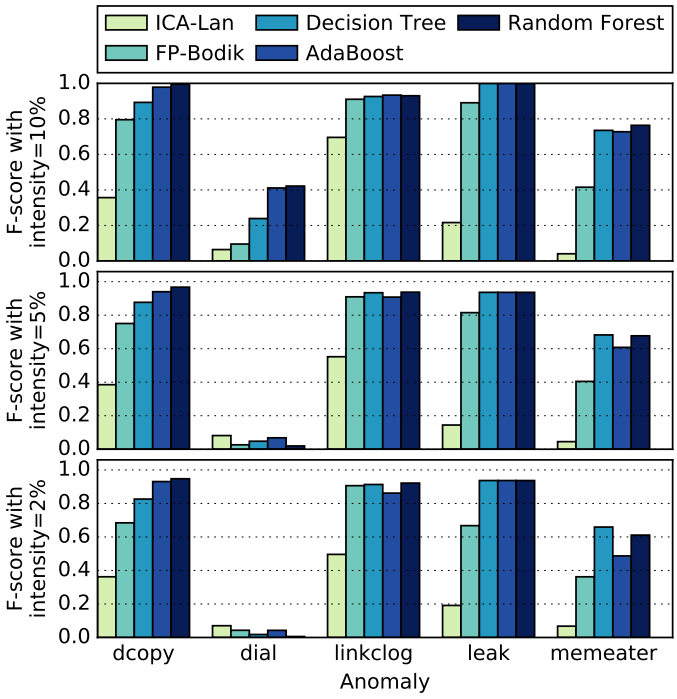
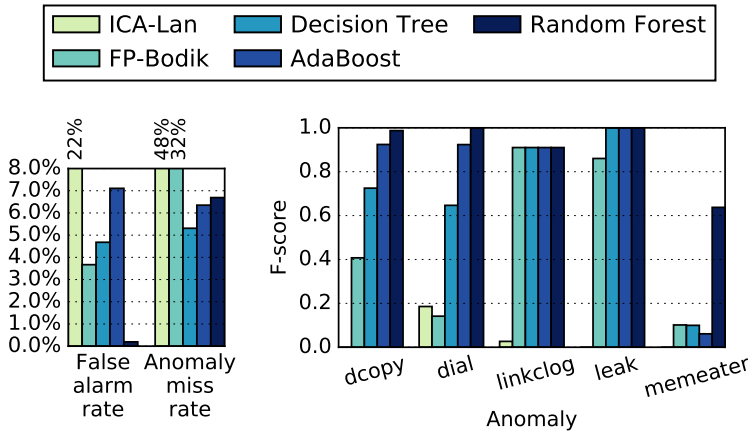


Figure 3·12: Anomaly diagnosis statistics when the models are trained with anomaly intensities 10, 20, and 100, and tested with low intensities. Most anomaly signatures are detected when the intensity is lowered. In the *dial* anomaly, the intensity sets the utilization of the synthetic anomaly program, making it harder to detect with low intensities.



(a) Detection

(b) Diagnosis

Figure 3·13: Anomaly detection and diagnosis statistics when the models are trained with 4-node application runs and tested with 32-node runs. The results are very similar to those with unknown input configurations as the 32-node runs use input configurations that are not used for training.

F-scores above 0.98. In the *linkclog* anomaly, the intensity determines the delay injected into the MPI communication functions, which is detected even with low intensities. In the intensity sets the utilization of the *dial* anomaly. As the intensity drops, the impact of *dial* on the application performance and resource usage patterns decreases, making it harder and also less critical to detect the anomaly.

Classification with Large Applications

Our framework can be used to diagnose anomalies when trained only with small application runs. We demonstrate this by using all 4-node application runs for training and 32-node runs for testing.

Figure 3·13 shows the detection and diagnosis statistics when running 32-node applications. As our framework checks individual nodes for anomalies and does not depend on how many nodes are being used in parallel by an application, the application size has minimal impact on the F-scores. The decrease in the F-scores is

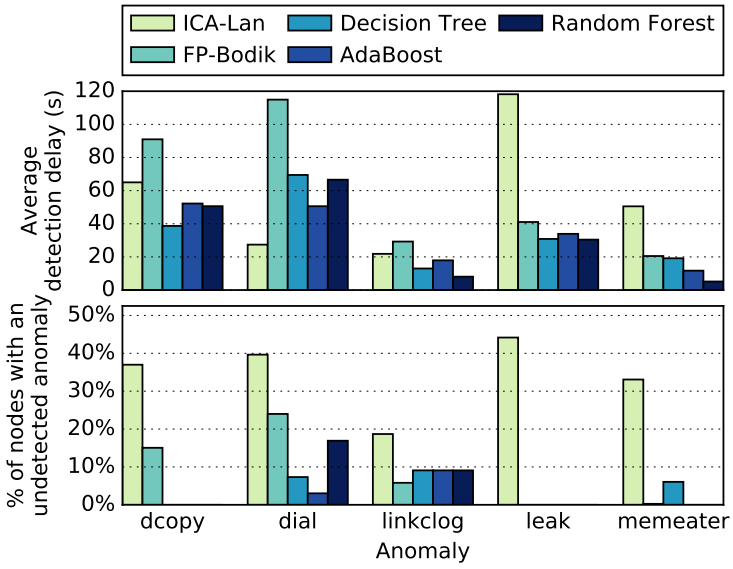


Figure 3·14: Anomaly detection delay and percentage of nodes with undetected anomalies when anomalies start at a random time while the application is running.

mainly because large application runs use input configurations that are unknown to the trained models. Hence, the F-scores in Figure 3·13 are similar to those in Figure 3·9, where the classifiers are trained with certain input configurations and tested with the remaining ones.

Diagnosis Delay

To analyze the delay during diagnosis, we use *random-offset* anomalies where the anomaly starts at a randomly selected time while the application is running. For each classifier, we record the time difference between the anomaly start time and the first window where C consecutive windows are labeled as anomalous with the correct anomaly type.

Figure 3·14 shows the diagnosis delay observed for each anomaly and each classifier as well as the fraction of the anomalies that are not detected until the application execution finished. Random forest achieves only five to ten seconds of delay

Table 3.4: Single-threaded computational overhead of model training and anomaly detection.

	Training time using the entire dataset		Testing time per sliding window per node	
	Feature generation & selection	Model training	Feature generation	Model comparison
ICA-Lan	10 days	30 s	62 ms	0.03 ms
FP-Bodik	5 days	10 mins	31 ms	148 ms
Decision tree	5 days	7 mins	4 ms	0.01 ms
AdaBoost	5 days	138 mins	13 ms	0.1 ms
Random forest	5 days	6 mins	13 ms	0.03 ms

while diagnosing linkclog and memeater. This delay is due to the fact that the proposed framework requires $C = 5$ consecutive windows to be diagnosed with the same anomaly before accepting that diagnosis. The delay is larger for other anomalies, where the diagnosis is performed with statistics that slowly change as the 45-second sliding window moves (e.g., mean and variance).

All of the nodes where the *linkclog* anomaly is not detected run *bt* and *sp* applications. These applications intensively use asynchronous MPI functions, which are in general not affected by *linkclog*. Hence, the impact of *linkclog* on the running time of *bt* and *sp* is negligible, and the detection of *linkclog* is harder for these applications.

Anomaly Diagnosis Framework Overhead

We assume that the node resource usage data is already being collected from the target nodes for the purposes of checking system health and diagnostics. State-of-the-art monitoring tools collect data with a sampling period of one second while using less than 0.1% of a core on most compute nodes (Agelastos et al., 2014).

Table 3.4 presents the average training and testing time of the baseline techniques as well as our framework with different machine learning algorithms when using a single thread on an Intel Xeon E5-2680 processor. As seen in Table 3.4, all approaches require the longest time for feature generation and selection. Note that feature gen-

eration and selection is an embarassingly-parallel process and is conducted only once during training.

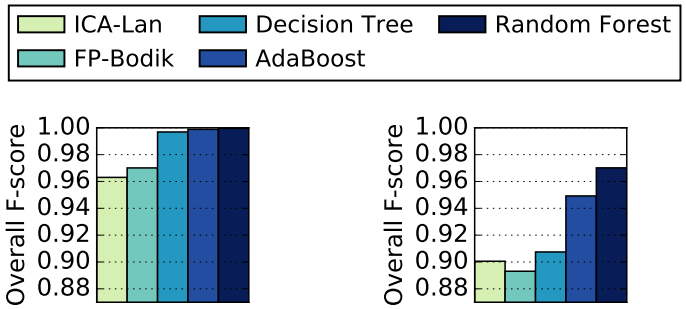
Decision tree and random forest classifiers are trained within ten minutes. Although a random forest classifier consists of 100 decision trees, its training is faster than the decision tree classifier. This is because random forest uses a subset of input features for each of its decision trees whereas all features are used in the decision tree classifier. The storage requirements for the trained models of decision tree, AdaBoost, and random forest is a 25KB, 150KB, and 4MB, respectively, whereas the baseline models both require approximately 60MB.

Detecting and diagnosing anomalies in a single sliding window of a single node with AdaBoost and random forest takes approximately 13ms using a single thread, which is negligible given that the window period is one second. Decision tree achieves the smallest feature generation overhead because it uses only a third of the features selected by our feature selection method (Sec. 3.1.1), whereas AdaBoost and random forest use nearly all selected features.

FP-Bodik has the highest overhead for runtime testing. This is because FP-Bodik calculates the L2 distance of the new fingerprint with all the fingerprints used for training to find the closest fingerprint. While ICA-Lan has a similar process during model prediction with kNN classification, the dimensionality of the space used in the ICA-Lan (10) is significantly smaller than that of FP-Bodik (>1000), leading to a much faster model prediction.

Anomaly Detection on the MOC

We use a virtual cluster in MOC to demonstrate that our framework is also applicable on platforms that are fundamentally different than Volta. In this analysis, we calculate the statistical features from a single time series window that contains the entire application run as opposed to using sliding windows. In addition, instead of



(a) Volta **(b)** MOC

Figure 3·15: Comparison of the overall F-scores achieved in Volta and MOC platforms.

linkclog, we use the *ddot* anomaly: *ddot* allocates two equally sized matrices of double type, fills them with a number, and calculates the dot product of the two matrices repeatedly (Tuncer et al., 2017a).

Figure 3·15 shows the F-scores different classifiers achieve in both platforms. Our framework consistently outperforms the baseline approaches and achieves an overall F-score above 0.97 in both platforms with random forest. F-scores are lower in MOC for all classification algorithms. There are three main factors that can cause the reduced F-scores in MOC: the number of collected metrics, sampling frequency, and platform-related noise. To measure the impact of the metric set difference, we manually choose 53 metrics from the Volta dataset that resemble MOC metrics based on their description and re-run our analysis with the reduced metric set. We find that reducing the metric set in Volta does not have a significant impact on the F-scores. Next, we measure the impact of data collection period by artificially increasing it to 5 seconds on Volta; however, the impact on classification accuracy is negligible. As a result of this analysis, we believe that the reduction in accuracy in MOC mainly stems from the noise in the virtualized environment, caused by the interference due to VM consolidation.

3.1.4 Summary

Performance anomalies lead to reduced efficiency and wasted resources in large-scale systems, especially in HPC clusters. The majority of the existing anomaly detection tools focus on binary classification of anomalous vs. healthy runs without identifying the type of the anomaly (i.e., diagnosis), which is a significantly more challenging problem. In this section, we have introduced our machine learning based framework that diagnoses performance anomalies at runtime. By leveraging machine learning algorithms, we are able to learn and identify signatures of anomalies in the collected resource usage data independent of running applications. Using experiments on real HPC systems, we demonstrate that our framework successfully diagnoses 98% of the injected anomalies and consistently outperforms existing techniques. This type of approach can be used in the future to enable higher levels of automation such as automated mitigation of anomalies through system management.

3.2 Software Configuration Analytics in the Cloud

So far, we have focused on performance anomalies based on application resource usage patterns. Another major anomaly type in large-scale computing systems is software misconfiguration. Configuration errors are among the leading causes of service disruptions and outages, especially in cloud platforms (Yin et al., 2011; Barroso et al., 2013). As cloud applications do not adequately verify the correctness of their own configurations (Xu et al., 2016), application-agnostic and automated verification of software configurations has received the attention of cloud researchers and engineers.

A promising approach for automated configuration verification is statistical and learning-based techniques that train on a corpus of configurations and learn common patterns. These techniques then identify configurations that deviate from the norm as potential errors. To perform a reliable analysis, these techniques need to be trained

with configuration data collected from a large number of working systems.

Cloud environments contain rich configuration data that are curated by different users for various purposes, and thus, provide a unique opportunity for applying learning-based configuration analysis. However, there is no state-of-the-art framework that enables discovery and extraction of configuration data from third-party cloud instances where the file system contents are unlabeled. Moreover, configurations are often stored in human readable text files with application-specific syntax. To apply existing configuration analysis techniques (e.g., (Zhang et al., 2014; Santolucito et al., 2017)), the information extracted from these files needs to be in the form of key-value pairs, where a key represents a specific configuration parameter.

We implement *ConfEx*, a novel framework that enables reliable software configuration analytics in the cloud. This section provides detailed information about software configurations, describes *ConfEx* in detail, and presents two use-cases for *ConfEx* to identify misconfigurations: outlier analysis and rule-based parameter type validation.

3.2.1 Cloud Software Configurations

Most cloud applications and system services store their configurations in human-readable text files or in configuration stores such as `etcd` and Windows registry. We focus on text file based configurations as it is prevalent for many of the building blocks of cloud applications (e.g., MySQL, Nginx, and Redis).

Figure 3·16 shows a snippet from an httpd configuration file. Each of the first two lines contains a *parameter* followed by a *value*, separated by a space. Lines 3-6 are in an application-specific format representing a conditional statement. While parsing these lines, one needs to retain the relational information between the parameters defined within the conditional statement, indicating that `User` and `Group` belong to the `IfModule unixd_module` section.

In some configuration files, the file schema is not embedded in the file itself and

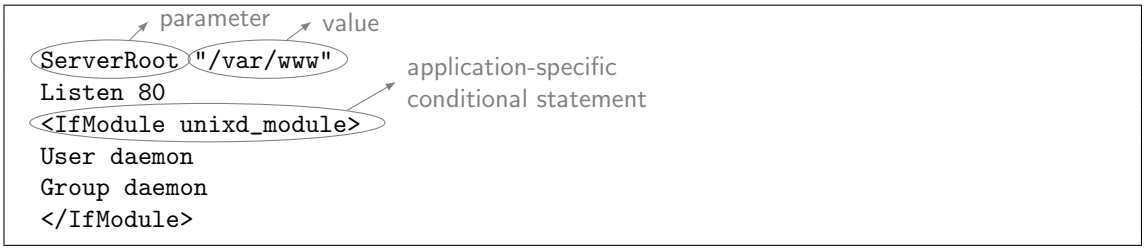


Figure 3·16: Httpd configuration file snippet. Configurations are stored in an XML-like format.

```
1 proc    swap     swap   pri=42          0 0
2 tmpfs   /dev/shm tmpfs  mode=0777       0 0
3 devpts /dev/pts devpts defaults,gid=5 0 0
```

Figure 3·17: `/etc/fstab` snippet. Configurations are stored in a table format where certain table cells contain multiple configuration entries.

requires domain knowledge to understand. One such example is the Linux filesystem configuration file (`/etc/fstab`), which defines available filesystems and their mount options. As shown in Figure 3·17, this file is structured in a table format where some columns may include parameter-value pairs such as `pri=42` (line 1) as well as multiple comma-separated entries such as `defaults,gid=5` (line 3).

Extracting configuration data from text-based files requires expertise on the specific application file format. Hence, to conduct corpus-based analysis using configuration files that are curated by different tools and users, one should use a community-driven parsing tool that allows contributions of application domain experts.

Configuration Errors

Table 3.5 summarizes common misconfiguration types we derived from related work (Ramachandran et al., 2009; Yin et al., 2011; Zhang et al., 2014; Xu et al., 2013; Chen et al., 2016) and online technical forums (e.g., `stackoverflow.com` and `serverfault.com`). *Illegal entries* can be identified through syntactic validation. Detecting *inconsistent entries* and *invalid ordering* requires extracting dependency and

Table 3.5: Common configuration error types and example constraints that lead to errors upon violation.

Error type	Example configuration constraint
Illegal entries	In PostgreSQL, parameter values that are not simple identifiers or numbers must be single-quoted.
	Variables must be in certain types (e.g., float).
Inconsistent entries	In PHP, `mysql.max_persistent` must be no larger than the `max_connections` in MySQL.
	In Cloudshare, service's `redis.host` entry (an IP address) must be a substring of Nginx's `upstream.msg.server` entry (IP address:port).
Invalid ordering	When using PHP in Apache, `recode.so` must be defined before `mysql.so`.
Environmental inconsistency	In MySQL, maximum allowed table size must be smaller than the memory available in the system
	In httpd, Apache user permissions must be set correctly to enable file uploads for website visitors.
Missing parameter	In OpenLDAP, a configuration entry must include `ppolicy[].schema` to enable password policy.
Valid entries that cause performance or security issues	MySQL's `Autocommit` parameter must be set to `False` to avoid poor performance under "insert" intensive workloads.
	In Nginx, setting server root location to "/" allows others to access the server's filesystem.
	Debug-level logging must be disabled to avoid performance degradation.

correlation information among various parameters. *Environmental inconsistencies* occur when application configurations do not match the environmental parameters such as file permissions and IP addresses. To find such inconsistencies, one needs to collect and analyze both application and environment configurations. Detecting *missing parameters* requires checking the existence of parameters rather than focusing on the values assigned to parameters. *Valid entries* that cause performance degradation or security vulnerabilities do not lead to crashes or error messages.

Configuration analysis tools commonly treat configurations as key-value pairs, in where each key corresponds to a specific configuration parameter (e.g., (Potharaju et al., 2015; Wang et al., 2004; Santolucito et al., 2017)). The configuration key-value pairs can be used for detecting the error types shown in Table 3.5 except for *invalid*

ordering. In this work, we use key-value pairs for configuration analysis and do not focus on *invalid ordering*.

3.2.2 Configuration Analytics using ConfEx

We propose a configuration analytics framework, *ConfEx*, for corpus-based configuration analysis in image repositories and multi-tenant cloud platforms. *ConfEx* discovers the configurations files in cloud system instances with unlabeled content, extracts consistent configuration data from these files, and applies statistical and learning-based analysis methods on the collected data to detect configuration errors.

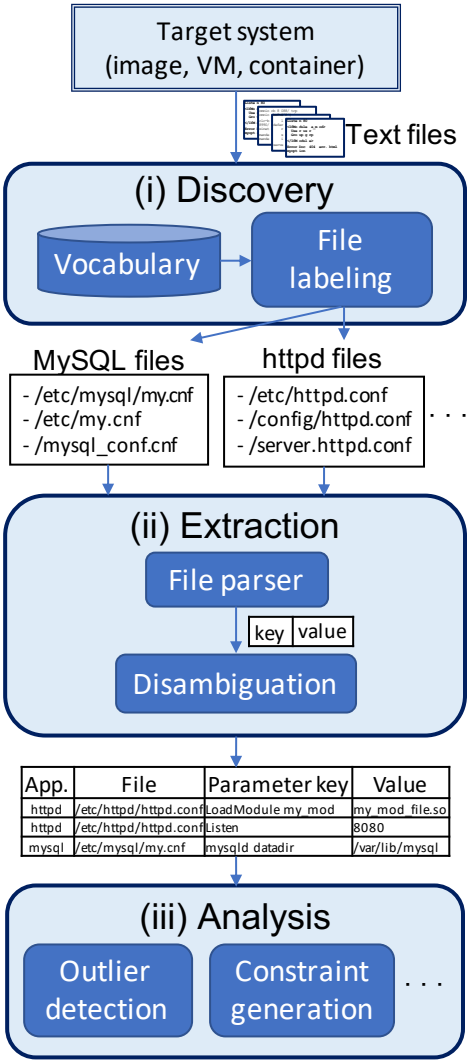


Figure 3·18: ConfEx overview.

Figure 3·18 shows an overview of *ConfEx*. In the discovery phase, *ConfEx* uses a vocabulary-based method we designed to discover configuration files. When a new cloud system instance with unlabeled configuration files is introduced (e.g., a new container), *ConfEx* reads and analyzes the text files in the given cloud instance and compares the contents of these files with a vocabulary database that is built offline (details not shown in Figure 3·18). To maintain low processing overhead during this discovery phase, *ConfEx* limits the size of text files inspected to 200KB. This threshold is supported by our investigation of 4581 Docker Hub images on which the largest configuration file found was 36KB. When *ConfEx* discovers a configuration file, it tags the file with a label identifying the soft-

ware that is associated with the file. These labels are then used in the extraction phase to apply software-specific file parsing and disambiguation rules. The extraction phase generates key-value pairs, which represent configuration data, using keys that consistently correspond to a single configuration parameter across different cloud instances. Finally, these key-value pairs are augmented with the software label and the source file path to enable a comprehensive and robust corpus-based configuration analysis. The rest of this section explains the phases of *ConfEx* in detail.

(i) Discovery

A common approach of locating configuration files is to check specific file system paths based on the locations of standard software installations. While system configuration file locations are typically consistent across different cloud instances, as we show in Section 3.2.3, depending on the application, 26-81% of valid configuration files are located in non-standard locations in popular Docker Hub images. These files are ignored by the configuration parsing tools. As a result, any configuration problems in these files will not be detected automatically using statistical and learning-based configuration analysis. To resolve this problem, the discovery phase of *ConfEx* identifies configuration files of known applications in cloud instances in an application-agnostic manner, regardless of where the files are located in the file system.

A straightforward way of checking whether a text file in the file system is a configuration file of an application is to compare the input text file's content with known configuration files of that application. If the similarity between the input file and a known configuration file is above a threshold, the input file can be labeled as a configuration of the target application. However, this approach is not reliable as the values assigned to configuration parameters may be unique to each user, especially for environment-specific parameters such as IP addresses, user names, or file paths. Thus, instead of using the entire content of configuration files for comparison, we

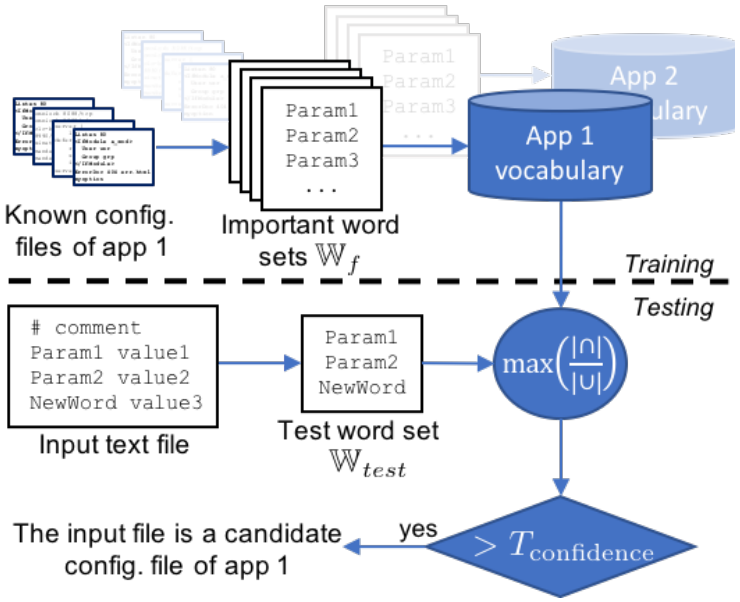


Figure 3·19: Discovery phase. A vocabulary is generated for each known application offline. Input text files are compared with each application vocabulary and selected as candidate configuration files upon a match that is larger than a confidence threshold.

identify *important configuration words* such as parameter names and configuration commands. These words are, in general, specific to applications and can be used to associate configuration files with applications.

Figure 3·19 depicts *ConfEx*'s discovery phase. During offline training, using known configuration files that are labeled with application names, we generate application-specific vocabularies by extracting the *important words* in these files as follows: We first discard commented-out lines as comments typically contain descriptions of configuration options with few or no application-specific words. We consider a line as a comment if it begins with `//`, `#`, or `%`, excluding the preceding white-space characters (i.e., tab and space). Then, we focus on the first word of the remaining non-comment lines as these words typically correspond to parameter names or configuration commands, whereas the subsequent words are user-provided values such as integers and file paths. While extracting the first word of a line, we use the following characters

as delimiters to account for the characters that are commonly used as part of a configuration file syntax: \t, , =, :, <, >, [,], ,. The *important words* of a text file is the set of first words of non-comment lines, in that file, and a collection of *important word* sets extracted from known configuration files form the application vocabulary.

During testing, we again extract the set of *important words* in an input text file using the same methodology. We calculate the similarity of the input *important word* set to each *important word* set in the vocabulary of each application. To calculate the similarity between two word sets, we use the Jaccard index (Real and Vargas, 1996), $J = |\mathbb{W}_1 \cap \mathbb{W}_2|/|\mathbb{W}_1 \cup \mathbb{W}_2|$, where $\mathbb{W}_1$ and $\mathbb{W}_2$ are two sets. If the maximum achieved similarity in a vocabulary is larger than a certain threshold, $T_{confidence}$, the file is labeled as a candidate configuration file of that application.

Note that the calculating the Jaccard index between the input word set, $\mathbb{W}_{test}$, and *important word* sets for all known configuration files is computationally expensive. We speed-up this process as follows: Let an *important word* set of a known configuration file f be $\mathbb{W}_f$, and the union of all keywords in a vocabulary be $\mathbb{W}_{vocab}$. Then, the Jaccard similarity (J) has the following upper bound:

$$J = \frac{|\mathbb{W}_{test} \cap \mathbb{W}_f|}{|\mathbb{W}_{test} \cup \mathbb{W}_f|} \leq J_{upper} = \frac{|\mathbb{W}_{test} \cap \mathbb{W}_{vocab}|}{|\mathbb{W}_{test}|} \tag{3.1}$$

as $|\mathbb{W}_{test} \cap \mathbb{W}_f| \leq |\mathbb{W}_{test} \cap \mathbb{W}_{vocab}|$ due to $\mathbb{W}_f \subseteq \mathbb{W}_{vocab}$, and $|\mathbb{W}_{test} \cup \mathbb{W}_f| \geq |\mathbb{W}_{test}|$. Checking J_{upper} once per vocabulary eliminates the need to compare $\mathbb{W}_{test}$ with all $\mathbb{W}_f$'s in the vocabulary if $J_{upper} < T_{confidence}$, which is common as most text files do not contain any configuration keywords of the target applications.

The syntax of all files that are labeled as configuration files is checked in the extraction phase. If the file does not conform with the configuration file syntax of the target application, users can be warned about a potential syntax error.

To discover the configuration files of a new application, a new vocabulary should be generated from a set of known configuration files of that application as described

above. This vocabulary can be extended simply by processing new labeled files without the need of re-processing the entire set of known configuration files.

(ii) Extraction

The purpose of the extraction phase is to parse the labeled configuration files and generate key-value pairs that represent configurations. For a robust corpus-based configuration analysis where the input configuration files are curated by different users, the extracted keys should have the following properties:

- *Consistency:* A specific key should always refer to the same parameter, both when observing configurations of a given cloud instance over time, and when comparing configurations across multiple systems.
- *Uniqueness:* Each parameter in a file should be represented by a unique key. However, if two parameters share the same name and context (such as parameters defined as a list), they should share the same key.
- *Context-preserving:* The keys of parameters that appear within the same block of a configuration file must retain this relational information. For example, in Figure 3·16, the keys of `User` and `Group` entires must express that both parameters are under the `IfModule` section. Such relations become more prevalent in file formats that keep hierarchical data such as JSON and XML.

While existing studies on configuration analysis have mostly focused on configuration stores that do not require data extraction such as Windows Registry (e.g., (Yuan et al., 2011)), or configurations with standard file formats such as `XML` and `JSON` (e.g., (Behrang et al., 2015; Zhang and Ernst, 2015)), most configuration files in today's cloud services (such as httpd and Nginx) are kept in human-readable text files that do not use standard file formats. These files require custom parsing rules based on

domain knowledge. However, the variety and rapid evolution of applications make it expensive and bug-prone to implement parsers for every configuration analysis tool.

Augeas for Parsing Configuration Files: To leverage the knowledge of domain experts on various applications and re-use an existing code-base that is continuously maintained, we build our extraction phase on top of Augeas (Lutterkort, 2008), which is one of the most popular tools available today for automatized configuration parsing and editing. Augeas has extensive application coverage with 182 *lenses*, which are file parsing rules to generate key-value pairs for different applications including httpd, MySQL, Nginx, PHP, and PostgreSQL. Augeas has been continuously maintained for more than ten years and has interfaces in different programming languages including Python, Ruby, Perl, and Java. As a result, Augeas is being used by other configuration management tools including Puppet (Loope, 2011) and bcfg2 (Desai, 2005), and also by Encore (Zhang et al., 2014), which is a state-of-the-art configuration analysis tool.

As Augeas is primarily intended for managing configurations in systems with uniform and known configuration structure, the keys in its output are often *ambiguous*, where a specific key does not necessarily correspond to the same parameter across different configuration files and the value assigned to a key does not necessarily correspond to the value of a configuration parameter. This ambiguity in the Augeas parser output complicates key-value-based analysis[6]. The reasons of this ambiguity can be seen in the example in Figure 3·20 and are summarized as follows:

The sample input httpd configuration file in Figure 3·20 has two `Listen` entries, but these two entries are represented by four key-value pairs by Augeas. Similarly, although both `Listen` entries represent the same configuration option, they are refer-

[6]Augeas output can be directly used for analysis (Zhang et al., 2014) only if the target configuration parameters are already defined by unique keys such as in PostgreSQL configurations or if the target files have the same parameter ordering.

red to using different keys to enforce a unique key per configuration entry. Such artificial key-value pairs reduce the reliability of configuration analysis.

Another challenge with the Augeas output stems from the indices assigned to the keys based on the ordering of entries in the file. Due to these indices, a different parameter ordering can result in a different key set. For example, in the httpd configuration file in Figure 3·20, if the second and the third lines were swapped, `/directive[2]` and `/directive[3]` keys would have referred to `Redirect` and `Listen`, respectively, unlike the Augeas output in Figure 3·20.

Because of the problems described above, the values of keys in Augeas' output do not necessarily correspond to the values of configuration parameters. For example, the value of the key `/directive[1]` in Figure 3·20 is `Listen`, which is a configuration parameter rather than a configuration value such as `8080`. Moreover, Augeas keys do not correspond to the same configuration consistently across different files. Hence, Augeas key-value pairs are not effective for corpus-

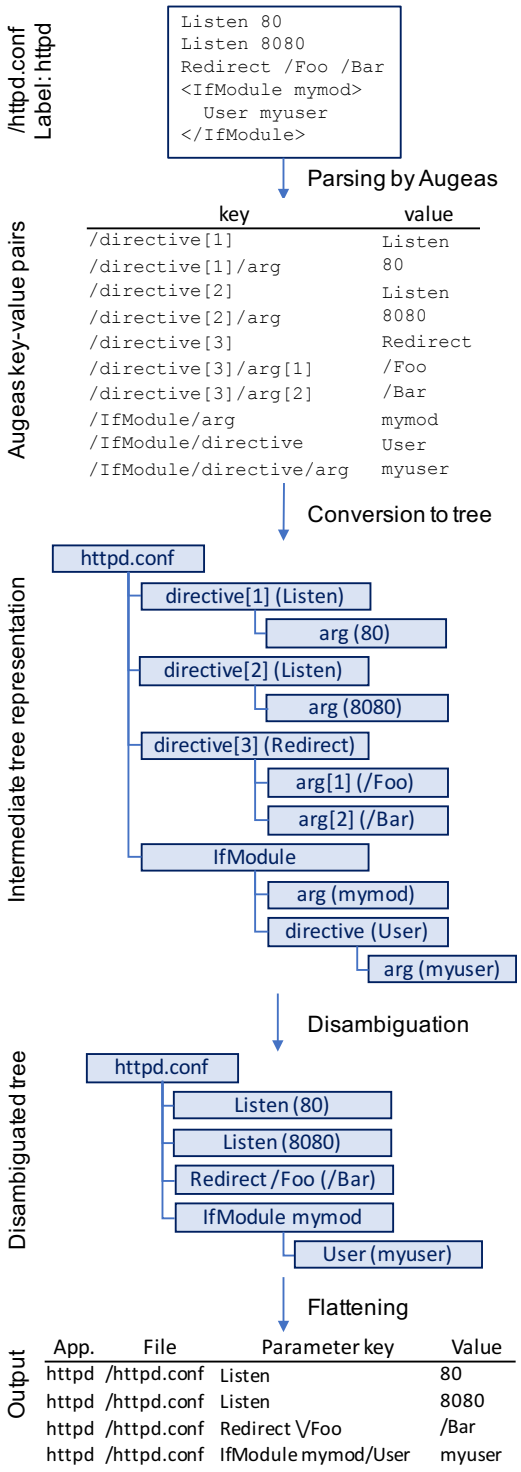


Figure 3·20: The extraction phase of our *ConfEx* framework.

based configuration analysis. Although the Augeas keys do not meet the consistency property, they can be used for analysis (Zhang et al., 2014) if the target configuration parameters are already defined by unique keys (such as in PostgreSQL) or if the target files have the same parameter ordering. However, an identical parameter ordering across different configuration files is not guaranteed in a multi-tenant cloud platform where the files are curated by different users.

Disambiguation of the Augeas Output: Figure 3·20 depicts the overall flow in our extraction phase. First, we parse the discovered files using Augeas, which discards any file that does not comply with the target application's configuration file format. We then *disambiguate* Augeas' output to generate reliable key-value pairs where a key consistently corresponds to the same parameter across different files. For this purpose, we convert the Augeas output into an *intermediate tree* that retains the hierarchical information in the configuration file. We transform this tree using a list of application-specific rules such that the transformed tree faithfully represents all parameters in the configuration files. We manually implement these rules using minimal domain knowledge and only by examining the document structure, parameters found in configuration files, and the corresponding output produced by Augeas.

Example disambiguation rules: By examining httpd configuration files and the Augeas output, we observe that the `directive` keys are redundant and do not correspond to parameters; hence, we extract the actual parameter names from the values of the `directive` keys. We also observe that specific entries such as `Redirect` do not represent parameters, but they are configuration commands with multiple arguments. From a configuration analysis perspective, we are interested in which arguments are being redirected (`/Foo` in Figure 3·20) and where they are directed to (`/Bar` in Figure 3·20). In this case, we use `Redirect /Foo` as the key, indicating that `/Foo` is

being redirected, and `/Bar` as the value assigned to this key. We identify fifteen such configuration commands in httpd by skimming through the application documentation. Our final observation is that nodes without values (such as `IfModule`) indicate configuration hierarchy. We use such nodes to preserve configuration hierarchy without assigning specific values to them. The above observations can be summarized in the following three tree transformation rules for httpd configurations:

- We replace `directive` nodes with the parameter names stored in the value of the directive. The value of the new node is the value of the child node `arg`.
- For specific keys that represent configuration commands (such as `Redirect`), we append the new key with the value of the child node named `arg[1]`. The value of the new node is the concatenation of the values of the remaining children whose name start with `arg`.
- We convert nodes without values (such as `IfModule`) into intermediate nodes where their key is appended with the value of the concatenation of the values of the children whose name start with `arg`.

After this rule-based transformation, the new tree is flattened and converted into a table as depicted in Figure 3·20. The application label and the file path are also appended to this table such that all configurations extracted from a cloud instance are represented in a single standardized format for robust analysis.

To extract reliable key-value pairs from the configuration files of new applications, one needs to implement application-specific tree transformation rules by examining the configuration file structure, configuration parameters, and the corresponding Augeas output using minimal domain knowledge as described above. A new Augeas lens may be required if the Augeas library does not support the new application.

(iii) Analysis

The discovery and extraction phases of *ConfEx* produces consistent key-value pairs, enabling the use of a rich variety of configuration analysis techniques in multi-tenant cloud platforms and image repositories. Analysis of software configurations can be used both to gain insight on user configuration practices and to detect misconfigurations. Existing automated misconfiguration detection techniques that can be applied as part of *ConfEx* include outlier value detection (Wang et al., 2004), parameter type inference (Zhang et al., 2014; Li et al., 2017), rule-based validation (Huang et al., 2015; Baset et al., 2017), parameter correlation analysis (Chen et al., 2016), and matching configuration parameters with the parameters in the source code for source-based analysis (Zhou et al., 2017; Zhou et al., 2016).

3.2.3 Evaluation

As there is no publicly available, comprehensive, and labeled misconfiguration data set, we evaluate *ConfEx* using public images in the Docker Hub repository and controlled injections of real configuration errors. To understand the individual benefits of discovery and extraction, we evaluate these two phases separately. In addition, we present two use cases of *ConfEx*: (1) detecting injected misconfigurations through outlier analysis and (2) syntactic configuration validation.

We focus on the Docker Hub images that contain either the network services system configuration file (`/etc/services`) or one of the three following popular cloud applications: httpd, MySQL, and Nginx. For `/etc/services`, we use the most downloaded thousand images and discard the images that do not have `/etc/services`. For each application, we use the images that are downloaded at least 50 times and contain the application name in their name or description. We have manually labeled the application configuration files in these images by examining file contents and file

Table 3.6: Statistics on the studied Docker Hub Images

application	# of images	total # of configuration files	total # of text files
httpd	272	9191	330106
MySQL	715	2600	509857
Nginx	2906	22450	313357
Network services	726	726	not used for discovery

Table 3.7: File paths checked by Augeas to identify httpd configuration files. "*" is a wildcard that represents any file name.

```
/etc/httpd/conf/httpd.conf
/etc/httpd/httpd.conf
/etc/httpd/conf.d/*.conf
/etc/apache2/sites-available/*
/etc/apache2/mods-available/*
/etc/apache2/conf-available/*.conf
/etc/apache2/conf.d/*
/etc/apache2/ports.conf
/etc/apache2/httpd.conf
/etc/apache2/apache2.conf
```

paths. Table 3.6 summarizes the number of images we use in our evaluation along with the number of text files and identified configuration files in these images. In total, we use 4581 images, where some images contain both the `/etc/services` file and one of the target applications.

Accuracy of Configuration File Discovery

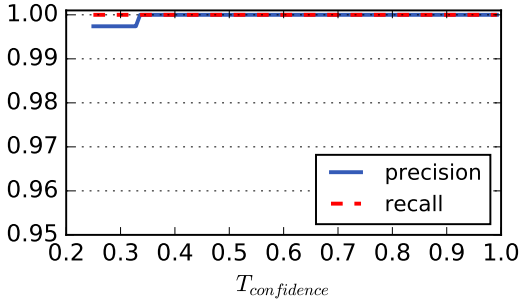
We compare *ConfEx*'s discovery phase with Augeas' configuration file discovery approach, which is checking the existence of files in specific file paths. Table 3.7 shows the paths checked by Augeas to discover httpd configuration files as an example. These file paths account for the default application installation paths in various Linux distributions. The paths checked by Augeas for other applications can be found on the Augeas website[7].

We measure the effectiveness of configuration file discovery separately for each

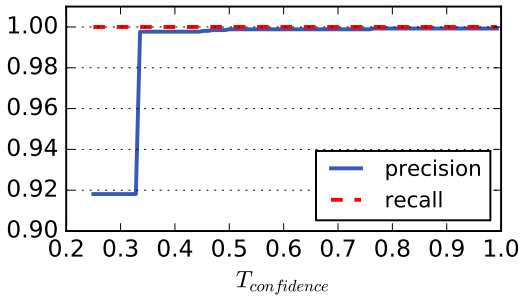
[7]Augeas lenses: `http://augeas.net/stock_lenses.html`

application and using five-fold cross validation. That is, for each application, we randomly divide the images that contain the application into five equal-sized partitions. We use four of these partitions to train our framework by generating an application vocabulary, and all the text files in the images of the fifth partition as testing set, where configuration discovery predicts whether the input text files are configuration files of the target application. We repeat this procedure five times, where each partition is used as a testing set once. Furthermore, we repeat the five-fold cross validation ten times with different randomly-selected partitions.

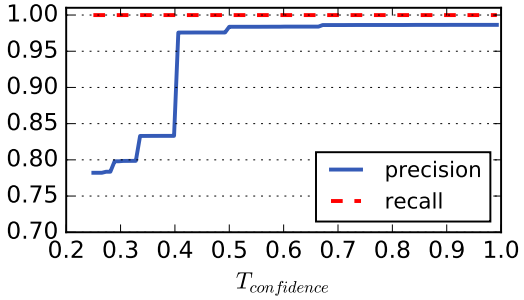
We use *precision* and *recall* as evaluation metrics. Precision is the fraction of true positives (i.e., correctly predicted configuration files) to the total number of files predicted as configuration files, and recall is the fraction of true positives to the total number of configuration files in the testing set.



(a) httpd



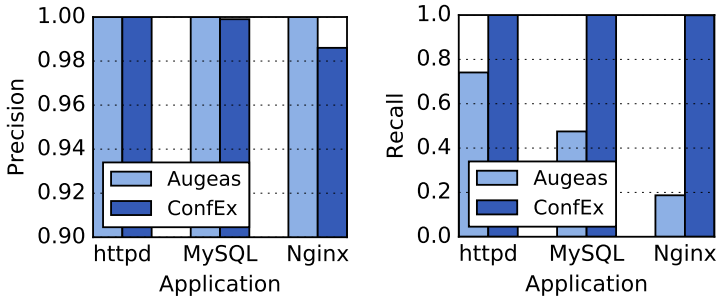
(b) MySQL



(c) Nginx

Figure 3·21: Configuration file discovery results using vocabulary-based discovery w.r.t. confidence threshold. With a confidence threshold above 0.5, *ConfEx*'s discovery approach achieves above 0.98 precision and recall in all applications we study.

Figure 3·21 shows the precision and recall *ConfEx* achieves on identifying the



(a) precision **(b)** recall

Figure 3·22: Comparison of the default path-based and *ConfEx*'s vocabulary-based discovery approaches ($T_{confidence} = 0.5$). The default approach can identify only 19-75% of the application configuration files, leading to low recall. *ConfEx* successfully identifies more than 98% of these files while resulting in less than 2% false positives in Nginx.

configuration files of the three target applications with various $T_{confidence}$. With $T_{confidence} > 0.5$, *ConfEx* achieves above 0.98 precision and recall for all three applications. The precision of discovery typically increases with the increasing $T_{confidence}$. This is because with a high $T_{confidence}$, the input text files with a few *important* words that don't exist in the vocabulary are labeled as non-configurations, reducing false positives. $T_{confidence}$ has a higher impact on Nginx's precision compared to httpd and MySQL as Nginx uses parameter names and command words that are commonly found in other text files (such as `user` and `include`). Increasing $T_{confidence}$ also decreases recall, but this decrease is negligible because the set of parameter names and configuration commands used in different configuration files is highly similar.

Having a lower precision is less critical than having a low recall during the discovery phase as ConfEx's extraction phase filters out the non-configuration files by checking their syntax. Hence, to avoid missing configuration files that have parameter names that are unseen during training, we use $T_{confidence} = 0.5$.

Figure 3·22 compares the baseline discovery approach of checking the standard configuration file paths with the proposed vocabulary-based discovery. The default

approach achieves ideal precision, i.e., all labeled configuration files are correctly labeled without false positives. This is because the standard configuration file paths (such as those shown in Table 3.7) do not contain text files that are not configurations. However, only 19% of Nginx configuration files can be found with this approach as the remaining configurations are not located in the default paths in the target images.

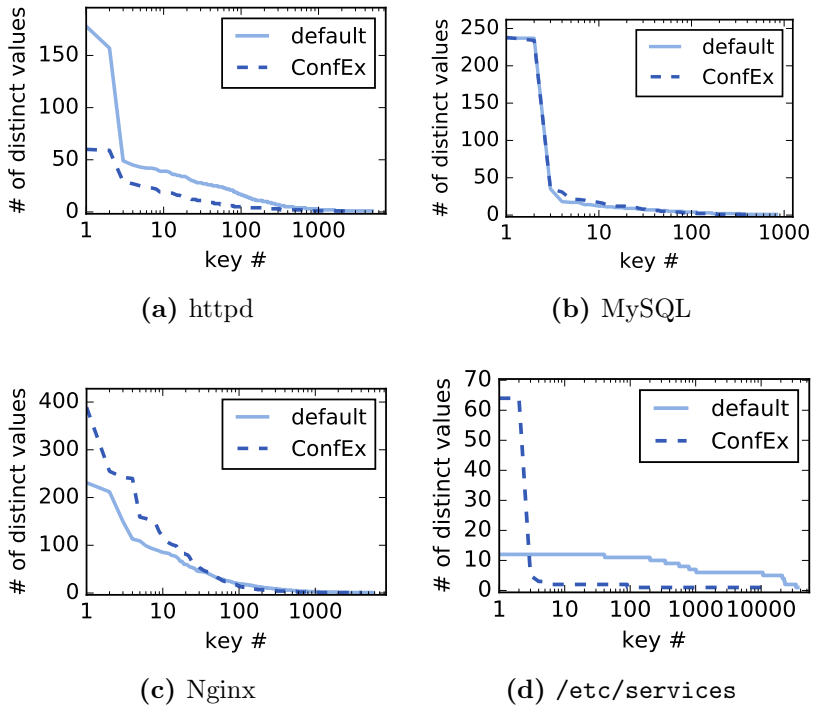
Overall, *ConfEx* successfully identifies 34156 target configuration files (out of 34241), while the baseline can identify only 12249 of the configuration files. In the remaining 85 files that are missed by *ConfEx*, approximately half of the parameter names are uncommon. As these parameter names are not seen during vocabulary generation, the corresponding files are labeled as non-configuration files. *ConfEx*'s lowest precision, which is over 98%, is observed with Nginx, where *ConfEx* labeled 347 of the non-configuration files as Nginx configurations among over 300,000 unlabeled text files. These mislabeled files contain words that are used as parameter names in Nginx such as the word `include` in file `/etc/ld.so.conf`.

The Impact of Disambiguating Augeas' Output

The disambiguation process described in Section 3.2.2 significantly impacts the parameter and value distributions observed across the configuration corpus. We demonstrate this impact by studying the total number of distinct values each key gets with and without disambiguation. As the configuration files of the same application may reside in different paths in different images, we analyze all extracted application key-value pairs regardless of the paths of the source files.

In Figure 3·23, we focus on the number of distinct values per configuration key across all known configuration files to show how *ConfEx*'s disambiguation step changes the distribution of the extracted key-value pairs.

When disambiguation is applied to the key-value pairs in the httpd configuration corpus, the number of distinct values assigned to individual keys reduce significantly.



(a) httpd **(b)** MySQL

(c) Nginx **(d)** `/etc/services`

Figure 3·23: The number of distinct values per key across application configuration corpora before and after *ConfEx*'s disambiguation. The keys are sorted individually for each line.

This is because an Augeas key can correspond to different parameters in different images. For example, `directive[1]` and `directive[2]` can correspond to `Listen` and `Redirect`, respectively, in one file, but `Redirect` and `Listen`, respectively, in another, giving the impression that the `directive[1]` and `directive[2]` keys both have two distinct values. This problem is resolved by *ConfEx*'s disambiguation, where a key consistently corresponds to the same parameter across different files and images.

In Figure 3·23b, the impact of disambiguation appears to be less significant for MySQL. However, upon further inspection, we observed that the keys do not correspond to the same parameters. For example, the first two disambiguated keys in Figure 3·23b correspond to the MySQL passwords assigned to `client` and `mysql_upgrade` in the configuration file, whereas the first two keys in the default keys

correspond to the passwords assigned to `mysql`, `connector_python`, and `mysqladmin` in addition to `client` and `mysql_upgrade`.

The Nginx distributions shows a decrease in the number of distinct values per key after disambiguation. The first key in both default and disambiguated distributions, `server/server_name`, illustrates how the number of distinct values per key decreases. When using the disambiguated keys, the key `server/server_name` covers all Nginx server name entries across the images. However, with the default keys, this key is used only if there is a single `server/server_name` is declared in the configuration file. If there are multiple server names in the same file, Augeas assigns an index to the key (`server/server_name[1]`, `server/server_name[2]`, etc.), preventing the comparison of server names across images using the same key.

Applying disambiguation to network services configurations reveals an interesting fact that is not visible when using the default Augeas key-value pairs: There is a single service, `X11`, that uses more than 60 ports for both tcp and udp connections. All other services use at most two ports.

3.2.4 Case Studies

We demonstrate the potential benefits of our framework, *ConfEx*, using two case studies to detect misconfigurations in Docker Hub images: detecting injected misconfigurations through outlier analysis, and rule-based validation of parameter types.

Detecting Misconfigurations with Outlier Analysis

PeerPressure (Wang et al., 2004) is a tool that finds the culprit configuration entry in an image with a single configuration error, where configurations are provided as key-value pairs. For each key-value pair, PeerPressure examines the values assigned to the key across a trusted corpus, and calculates the probability of the given value being a misconfiguration based on empirical Bayesian estimation. If a value is an

Table 3.8: Injected application misconfigurations

application	name	description
httpd	url	Error 401 points to a remote URL
httpd	dns	Unnecessary reverse DNS lookups
httpd	path	Wrong module path
httpd	mem	MaxMemFree should be in KB
httpd	req	Too low request limit per connection
MySQL	enum	Enumerators should be case-sensitive
MySQL	buf	Unusually large sort buffer
MySQL	limit	Too low connection error limit
MySQL	max	Invalid value for max number of connections
Nginx	files	Too few open files are allowed per worker
Nginx	debug	Logging debug outputs to a file
Nginx	access	Giving access to root directory
Nginx	host	Using hostname in a listen directive

outlier among the values that are assigned to the same key across the corpus, the corresponding entry has a high probability of being misconfigured. Finally, the key-value pairs are ranked based on the calculated probabilities, so that the pairs that are ranked higher are outliers, and hence, the most likely errors.

As PeerPressure is designed for Windows registry, it does not have configuration discovery and extraction capability. We use *ConfEx* to generate configuration key-value pairs for PeerPressure's outlier analysis. Additionally, we show the impact of *ConfEx*'s key-value pair disambiguation on PeerPressure's accuracy by using the default Augeas key-value pairs before disambiguation as a baseline.

We use PeerPressure to detect the misconfigurations listed in Table 3.8. We have identified the application misconfigurations from application websites and technical forums such as `stackoverflow.com`, and synthetically generate `/etc/services` misconfigurations. For applications, we inject each misconfiguration listed in Table 3.8 to a randomly selected image that contains the target parameter to be misconfigured. To generate `/etc/services` misconfigurations, we randomly select a service in a randomly chosen image, and change the port used by the selected service to a random integer between 1 and 10000. For each target misconfiguration, we repeat the randomized injection 1000 times. For each injection, we train PeerPressure using

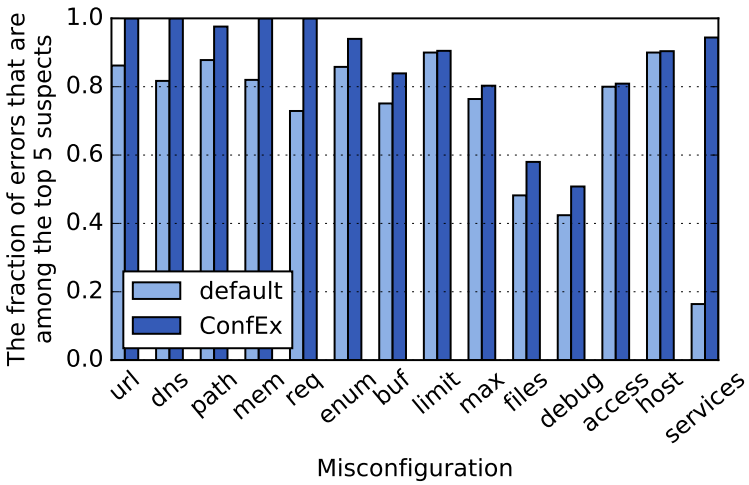


Figure 3·24: The fraction of injected errors that are ranked within the top five suspects by PeerPressure among 1000 randomized injections. `services` is the `/etc/services` misconfiguration.

the key-value pairs that belong to the target application from all images except for the misconfigured image. We then run PeerPressure and record its output ranking for the injected error.

Given an image with an injected misconfiguration, PeerPressure ranks all the configuration key-value pairs in the image with respect to their probability of being an error. Figure 3·24 shows the fraction of injected errors that are ranked within the top five suspects by PeerPressure among 1000 randomized injections of our target misconfigurations. Using *ConfEx*'s disambiguated keys consistently leads to similar or higher rankings compared to using default Augeas keys, making it easier to pinpoint the injected error.

With the default keys, PeerPressure suffers from having an incorrect view on the distribution of configurations as discussed in Section 3.2.2. This problem becomes more significant when the number of keys are used for the misconfigured parameter across the corpus is large (e.g., more than five), such as in `services` misconfigurations. Moreover, when the misconfigured image has files that have substantially different

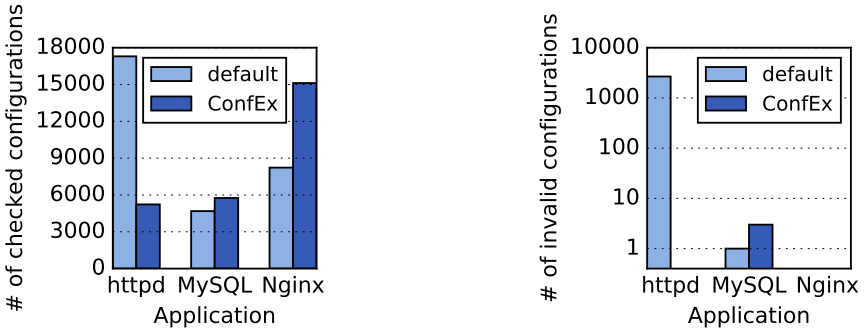
parameter ordering than the files seen during the training, the parameters in the image are represented by keys that use different indexing. As a result, common configuration entries become outliers in the corpus and have high PeerPressure rankings. However, PeerPressure can still detect the injected errors with the default keys if the parameter ordering in the misconfigured image is similar to those seen during training.

In *files* and *debug*, outlier detection performs poorly both with the default Augeas keys and *ConfEx*'s disambiguated keys. This is because compared to the other injected errors, the parameters being misconfigured in *files* and *debug* have a flatter value distribution with a large number of distinct values across the corpus. Hence, the injected erroneous value is not perceived as an outlier by PeerPressure. This is an inherent weakness of outlier analysis and can be avoided by using a larger corpus.

Rule-based Configuration Type Validation

We present a second use case of *ConfEx* by validating configuration types (such as integer or file path) through syntactic rules. For this purpose, we randomly select a configuration file, and for each parameter in this file, we write syntactic type validation rules using regular expressions similar to those used in prior work (Zhang et al., 2014; Li et al., 2017). For example, if the type of a parameter is IP address, the value assigned to this parameter is validated using the regular expression `^\d{1,3}(\.\d{1,3}){3}$`. For each rule, we map the rule to the key that points to the rule's target value in the selected file. For example, if the selected file is the one presented in Figure 3·20, we check whether the value assigned to the keys `/directive[1]/arg` (for Augeas keys) and `Listen` (for *ConfEx* keys) is a port number across all known configuration files in our corpus.

Based on the randomly selected file, we write syntactic rules for 25, 12, and 25 parameters in httpd, MySQL, and Nginx, respectively. We use these rules to validate the values assigned to the corresponding keys, and show the impact of disambiguation



(a) Number of checked configurations

(b) Number of values marked as invalid

Figure 3·25: The number of configuration entries checked and marked as invalid using rule-based type validation. With the default keys, all values that share the same key are checked using the same rule although they belong to different parameters in httpd, resulting in a high number false negatives. In MySQL and Nginx, default Augeas keys can capture only a subset of the target key-value pairs.

on syntactic configuration validation.

Figure 3·25a shows the number of configurations that use the keys for which we have written syntactic validation rules for both default Augeas and *ConfEx*'s disambiguated keys. As the same default Augeas key can correspond to multiple parameters in httpd, all values that share the same key are checked using the same rule even if they correspond to different parameters. This results in over 2500 values being marked as invalid as seen in Figure 3·25b. The same problem does not occur with MySQL and Nginx as their Augeas keys are not shared by different parameters. However, when using the default keys, the validation misses 1068 (19%) and 6894 (46%) of the target MySQL and Nginx parameters, respectively.

With *ConfEx*'s keys, the validation rules detect only three invalid values, and one of these values is also captured with the default keys. We have found that these values are to be replaced by a script (e.g., one of the values is `__PORT__`), and are indeed syntactically invalid.

3.2.5 Summary

Using analytics to accurately find misconfigurations in multi-tenant cloud platforms has been so far limited to the configurations that are located in default system locations or configuration stores such as `etcd` and Windows registry. While researchers has proposed automated techniques to detect misconfigurations in the cloud, we have demonstrated that these tools ignore up to 81% of the application configuration files and suffer from the *ambiguity* in the outputs of existing configuration parsers.

We implemented *ConfEx*, a novel framework that enables robust discovery and extraction of text-based software configurations from unlabeled cloud instances. By providing visibility into software configurations, *ConfEx* enables a new level of automated analysis in cloud platforms through the use of existing application-agnostic tools that are designed for key-value-based configurations. Given configurations collected from a large number of working systems, such tools can automatically learn parameter types (Li et al., 2017), value constraints (Zhang et al., 2014), parameter correlations (Chen et al., 2016), or common configuration patterns (Wang et al., 2004). As we show in our evaluation, these tools can then be used to detect configuration errors in an automated and application-agnostic way to improve service reliability in future cloud platforms.

Chapter 4

Data-driven Management for Improving Data Center Efficiency

Efficient data center management is a challenging task due to the complexity, size, and heterogeneity of today's data centers. The interactions among diverse applications and hardware components, dynamic constraints such as performance and power requirements, and physical restrictions such as thermal thresholds necessitate the use of automated dynamic system management policies that are driven by data collected from various data center layers. To design such data-driven management policies, one also needs accurate and scalable modeling of the data center to understand the impact of various management decisions. Even with accurate system modeling, it is difficult to find the most efficient operating point with low overhead, which is necessary for dynamic management at runtime.

In this thesis, we specifically focus on two aspects of data center management: (1) power management, which has a direct impact on the energy efficiency of data centers, and (2) workload management for highly-parallel HPC applications, which can significantly reduce wasted compute resources in HPC systems by decreasing application running times by up to 34% (Deveci et al., 2014).

We start this chapter by introducing *CoolBudget* (Tuncer et al., 2014), our cluster-level power management policy that improves the overall data center performance under power constraints. Based on power, performance, and thermal models of a data center, *CoolBudget* distributes the available power among cooling units and servers in

a workload-aware manner to maximize the overall data center performance without leading to power starvation in any server. To improve the thermal stability in servers, we also design a server cooling control policy (Zapater et al., 2015a) that proactively prevents thermal overshoots and minimizes thermally-induced power consumption.

In addition to our power management policies, we introduce our novel workload management policy, *PaCMap* (Tuncer et al., 2015), to reduce the communication overhead of HPC applications. *PaCMap* provides a holistic view on HPC job placement by considering both the network topology and applications' communication topology to obtain more efficient job placement compared to the state-of-the-art.

4.1 Cluster-level Power Management

Today's data centers typically have total power consumption limitations due to the capacity constraints of power infrastructures or to avoid peak electricity demand charges (Barroso et al., 2013). In addition, there is growing interest among data centers to employ sophisticated energy cost management techniques such as integration to smart grid through demand-side regulation programs (Chen et al., 2014a). These techniques require power budgeting, which refers to limiting the total data center power and distributing the available power across the servers in the data center while taking the application performance demands into account. As cooling can consume up to 50% of the total data center electricity (Dayarathna et al., 2016), power budgeting should also account for cooling.

A good strategy should intelligently react to changes in workloads and environment, and find the best management decisions to minimize the energy footprint without sacrificing performance. Designing such intelligent strategies requires accurate and scalable modeling of data center power and performance dynamics. In the following subsections, we first model the interactions between power, performance, and

cooling in a data center based on experiments on a real enterprise server. Our models enable safe reduction of the thermal headroom between the server internal temperatures and the critical thresholds. We then propose a novel power budgeting technique, *CoolBudget*, where we formulate an optimization problem to distribute the available power among servers and cooling units to improve the data center throughput and fairness among workloads. In Section 4.1.3, we demonstrate that *CoolBudget* improves the *fair speedup* of the data center by over 15% compared to the state-of-the-art.

4.1.1 Modeling of Power, Performance, and Cooling

To make efficient power budgeting decisions for a given set of jobs in a data center, we need the ability to estimate performance, power, and temperature for each server under various power distribution scenarios. For this purpose, we model the complex interactions between power, performance, cooling, and temperature in a data center.

We develop empirical models based on sensor and hardware performance counter data collected from an enterprise server. Our modeling methodology is applicable to a wide range of hardware and workload scenarios.

Methodology

We conduct our experiments on an enterprise server with 32 8GB memory modules and two SPARC T3 CPUs (Shin et al., 2010), providing a total of 256 hardware threads. Each CPU has 16 8-way hyper-threaded cores, providing a total of 256 simultaneous hardware threads. We collect sensor measurements through Continuous System Telemetry Harness (CSTH) (Lopez, 2007), which is a tool that runs on the service processor. Using CSTH, we measure processor voltage, current, and temperature for every CPU in the server, as well as total server power and server inlet temperature data every second. In addition to sensor measurements, we use Solaris 10 OS tools (`sar`, `cpustat`, `busstat` and `iostat`) to poll the hardware counters for

workload characterization. The overhead introduced by polling the counters during execution is negligible.

To train our developed models and to evaluate our policies, we use the following benchmarks: SPEC Power_ssj2008[1], SPEC CPU 2006 (Henning, 2006), and the PARSEC multi-threaded benchmark suite (Bienia, 2011) that assesses the performance of multiprocessor systems. For each PARSEC job, we launch 8 copies of a specific PARSEC application, each running with 32 threads, to stress our enterprise server in terms of power and temperature. The SPEC CPU applications are selected such that the subset comprises different workload characteristics (Phansalkar et al., 2007). As these applications are single-threaded, for each SPEC job, we launch as many copies of a single application or of an application pair as the number of threads we want to utilize. For each job, we experiment with 8, 12, 16, 20, 24, 28, and 32 active cores. Our total experimental database consists of 300 jobs, out of which, 75% are randomly selected for model training, and the remaining 25% are used for validation. In addition to the benchmarks, we use a custom-designed synthetic workload tool, *LoadGen*, to stress our server with any desired utilization level and to thoroughly model the temperature-power relationships.

We control the server power consumption using *thread packing* (Cochran et al., 2011), i.e., we allocate the software threads in a fewer number of hardware threads to decrease the power. We apply thread-packing at core level by either activating or deactivating all 8 threads in a core using Solaris `psrset` tool. The 32 cores in our server enables 32 possible power states, providing sufficient capping ability. However, finer-grained power control can also be implemented if desired (Hankendi et al., 2013; Shen et al., 2013).

Our target data center consists of 40 racks with 9 servers per rack, providing a total of 360 servers. The racks in the data center are distributed in 4 rows with a cool-

[1] `http://www.spec.org/power_ssj2008`

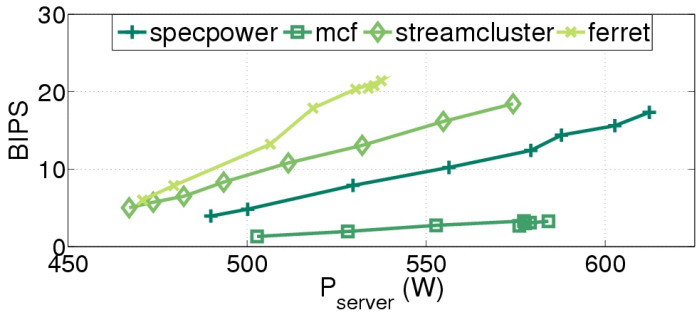


Figure 4·1: BIPS vs. server power relationship for various jobs, when each job is running with 8, 12, 16, 20, 24, 28, 32 cores.

aisle hot-aisle configuration. Two computer room air conditioning units (CRACs) are located at the same side of the two hot aisles and use under-floor cooling with room return. We use *TileFlow*[2] software to model the data center heat flow dynamics.

Server Power and Throughput

To improve the data center QoS using power budgeting, we first model the relationship between server power and throughput. We use billions of instructions per second (BIPS) as the throughput metric. A linear relationship between the server power cap and BIPS has been observed in prior work using DVFS (Zhan and Reda, 2013) and hypervisor resource limiting (Hankendi et al., 2013). We also observe a linear relationship in our system when using thread-packing as the control knob, as shown in Figure 4·1. Thus, we model BIPS as follows:

$$BIPS = k_0 \cdot P_{server} + k_1 \tag{4.1}$$

where k_i's are constants that depend on the job. We calculate the coefficient k_0 at runtime using performance counter data and power measurements with a model proposed in prior work (Zhan and Reda, 2013) as follows:

$$k_0 = k_2 + \frac{k_3 \cdot BIPS}{P_{server}} + k_4 e^{k_5 \cdot n_{dram}} \tag{4.2}$$

[2] `http://inres.com/products/tileflow`

where n_{dram} is the number of DRAM accesses per instruction and $k_{2,3,4,5}$ are machine-dependent coefficients found using offline regression analysis. k_0 is estimated every second at runtime. Then, k_1 is calculated by Equation 4.1 using BIPS and power measurements. Our BIPS prediction error has a mean of 0.6 BIPS (corresponding to only 3% average error) and a standard deviation of 2.5 BIPS.

We extend the model proposed by Zhan et al. by also predicting the BIPS-power scaling. Note that in Figure 4·1, the increase in throughput stops at a certain power level for each job. In our experiments, this is either due to a memory bottleneck (see *ferret* and *mcf*), or because increasing the number of active cores further than the number of software threads does not bring any benefits. We define $maxBIPS$ as the BIPS achieved by a job when using all 32 CPU cores. To predict the $maxBIPS$ of a job given its measured BIPS and the number of active hardware threads, we first assume linear scaling of BIPS with the number of threads:

$$maxBIPS_{threads} = BIPS \cdot \frac{n_{sw\_threads}}{n_{hw\_threads}} \tag{4.3}$$

where $n_{sw\_threads}$ and $n_{hw\_threads}$ are the number of software and active hardware threads, respectively, and $BIPS$ is a runtime measurement. Second, we observe that the number of DRAM accesses per instruction of an application puts an upper

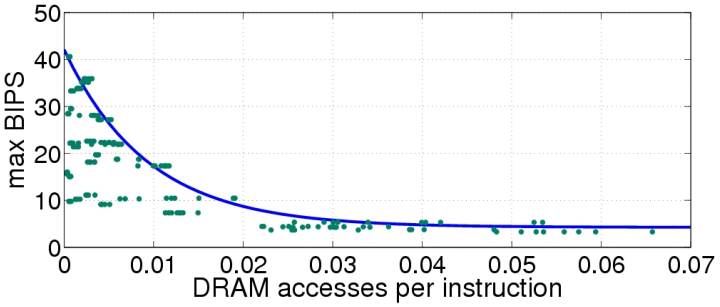


Figure 4·2: BIPS upper-bound set by the number of DRAM accesses per instruction. The dots represent individual jobs and the solid line is the regression result for the upper-bound.

bound on $maxBIPS$ as shown in Figure 4·2, where the data belongs to the jobs whose BIPS-power scaling is limited due to memory bottleneck. The BIPS upper bound on Figure 4·2 has the following form:

$$maxBIPS_{bottleneck} = k_6 + k_7 e^{k_8 \cdot n_{dram}} \tag{4.4}$$

where $k_{6,7,8}$ are regression coefficients. The predicted $maxBIPS$, is the minimum of Equations 4.3 and 4.4. Given $maxBIPS$, the server power consumed by a job when using all 32 CPU cores, $P_{server,max}$, is calculated using Equation 4.1. Predicting maximum server power in this way has a mean error of 11W and a standard deviation of 30W in our server, which consumes between 400-700W in our experiments.

Server Internal Temperatures

Conventional cooling systems target to limit servers' inlet temperature; however, our study shows that limiting server internal component temperatures instead leads to a significant improvement in efficiency. As CPUs are the hottest components in our server, we focus on CPU temperature, T_{cpu}. Our methodology can be used to include other server components such as memories and GPUs.

When assigning power limits to individual servers, we need to ensure that the CPU temperature threshold is not violated in any server. We achieve this by (1) using empirical models that correlate server power and internal component temperatures, and (2) using proactive server cooling (see Section 4.2).

We empirically model the steady-state CPU temperature based on a given server power cap to ensure reliable CPU temperatures during power budgeting. The steady-state temperature of a CPU depends on its power consumption, P_{cpu}, thermal resistance of the hardware, server fan speed, and server inlet temperature. We derive the thermal characteristics of our CPU at different utilization levels using *LoadGen*. We use a constant fan speed of 2400 rpm, which is an empirically selected value

that prevents the server leakage power from becoming dominant over the server fan power for the majority of our jobs in the inlet temperature range from 18°C to 35°C. Consequently, we model T_{cpu} using the resistance-capacitance thermal model of the chip (Pedram and Nazarian, 2006) as follows:

$$T_{cpu} = R_{cpu} \cdot P_{cpu} + T_{inlet} + k_9 \tag{4.5}$$

where k_9 represents an empirical ΔT that reflects the increase in the air temperature within the server enclosure before reaching the CPU. While using a constant provides sufficient accuracy for the thermal coupling in our server, other servers may require finer-grained modeling (Ayoub et al., 2012).

As P_{cpu} and BIPS are mostly governed by the same workload characteristics, we model P_{cpu} using the same methodology in BIPS estimation as follows:

$$P_{cpu} = k_{10} \cdot P_{server} + k_{11} \tag{4.6}$$

where $k_{10,11}$ are calculated in the same way as $k_{0,1}$ in Equation 4.1. Our combined CPU power-temperature model overpredicts the CPU temperature for a given server power with a mean of 2.9°C and a standard deviation of 1.5°C.

Data Center Thermal Dynamics

The power cap of a server not only changes the server internal temperatures, but also affects other servers due to heat recirculation. We model the heat recirculation using the methodology proposed by Tang et al. (Tang et al., 2006). In this model, the inlet temperature of a server is represented by a linear combination of the CRAC outlet temperature and the power consumption of each server, formulated as:

$$\boldsymbol{T_{inlet}} = \boldsymbol{D}\boldsymbol{P_{server}} + T_{crac} \tag{4.7}$$

where $\boldsymbol{T_{inlet}}$ and $\boldsymbol{P_{server}}$ are the server inlet temperature and power vectors, respectively, T_{crac} is the CRAC outlet temperature, and $\boldsymbol{D}$ is the heat distribution matrix.

The $\boldsymbol{D}$ matrix represents the heat recirculation as well as the impact of thermodynamic constants. We calculate the heat distribution matrix of our target data center using thermal simulations with *TileFlow*.

We model the CRAC unit power consumption using the coefficient of performance (CoP) approach. CoP is defined as $CoP = P_{compute}/P_{cool}$, where $P_{compute}$ is the total computing power (all servers) and P_{cool} is the cooling power. We use the CoP model given by Moore et al. (Moore et al., 2005) as follows:

$$CoP = 0.0068 \cdot T_{crac}^2 + 0.0008 \cdot T_{crac} + 0.458 \tag{4.8}$$

4.1.2 Telemetry-based Power Budgeting Using *CoolBudget*

We propose *CoolBudget*, a data center power budgeting technique, to improve the overall data center performance under a total power constraint without an unfair performance degradation for any of the jobs.

Policy Overview

For a workload-aware power budgeting, *CoolBudget* first collects performance counter data from all servers and constructs the power and temperature models described in Section 4.1.1. We then iteratively solve an optimization problem to find the most efficient power distribution for our data center model.

The policy maximizes the fair speedup, which corresponds to the harmonic mean of per server speedup, defined as:

$$\text{Fair Speedup} = \frac{N}{\sum_i^N (maxBIPS^i/BIPS^i)} \tag{4.9}$$

where N is the number of servers and $maxBIPS$ is the estimated maximum BIPS achievable by the executing job. Fair speedup is both an indicator of overall performance and a measure of fairness.

Our policy computes the optimum power distribution among the servers for a

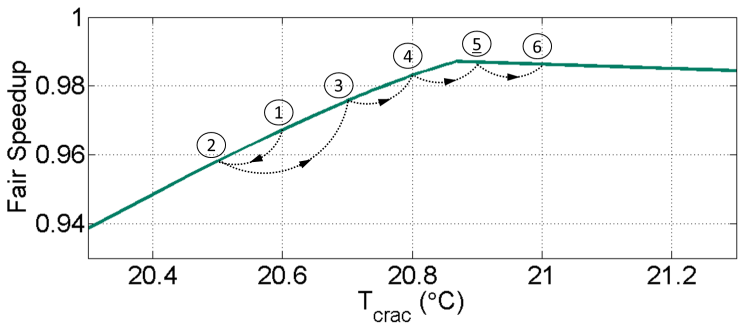


Figure 4·3: Typical trend in maximum fair speedup within the proximity of optimum T_{crac}, and the policy iteration steps with a starting point of 20.6°C

given T_{crac}. In order to find the most efficient T_{crac}, the problem is iteratively solved at different CRAC temperatures. Figure 4·3 shows the typical trend in maximum fair speedup with respect to T_{crac} for a given total budget. When T_{crac} is increasing, the fair speedup first increases because of the decrease in the cooling power due to Equation 4.8. This means that a larger portion of the total power budget is used for computation, leading to a higher throughput. When T_{crac} raises above a certain level, the performance of the hottest servers are degraded considerably to keep the temperature under the redline; thus, an increase in the room temperature is not useful anymore for the overall objective.

Based on the observation above, *CoolBudget* starts searching for the most efficient T_{crac} using its last known optimal value, which is 20.6°C in Figure 4·3. The policy first solves the optimization problem in the proximity of the last optimal T_{crac} (steps 1, 2, and 3 in the figure). Then, it iterates in the direction of increasing fair speedup (4 and 5) until fair speedup starts decreasing (6). Finally, the best solution is selected (5). We use 0.1°C resolution for the T_{crac} selection.

Optimization Problem

The optimization problem finds the best power distribution among the servers for a given T_{crac}, and formulated as follows:

$$\min_{\boldsymbol{P_{server}}} \quad \sum_{i}^{N}(maxBIPS^i/BIPS^i) \tag{4.10a}$$

$$\text{s.t.} \quad (1+1/CoP)\sum_i P^i_{server} \leq P_{budget} \tag{4.10b}$$

$$k_9(k_{10}P^i_{server}+k_{11}) + (\boldsymbol{DP_{server}})^i + T_{crac} + k_9 \leq T^i_{redline} \qquad \forall i \tag{4.10c}$$

$$P^i_{server,idle} \leq P^i_{server} \leq P^i_{server,max} \qquad \forall i \tag{4.10d}$$

The objective function in (4.10a) is the denominator of Equation 4.9. Constraint (4.10b) limits the total power usage, and is derived from the equation $P_{compute} + P_{cool} \leq P_{budget}$ and CoP equations (see Section 4.1.1). (4.10c) keeps the temperature of each processor under a given redline by combining Equations 4.5, 4.6, and 4.7. This constraint also introduces location-awareness to the problem through the heat recirculation matrix $\boldsymbol{D}$. Finally, (4.10d) ensures that the power given to a server falls between the idle power and maximum power for the given job. Note that as the constraints for each server are written individually, it is straightforward to use this methodology in a data center with heterogeneous servers or add performance constraints to specific servers.

CoolBudget is invoked every second. As the jobs we use generally have stable power profiles when they are executing, and because the thermal time constants of the CPUs in our server are in the order of tens of seconds, the periodic check of 1 second is sufficient to capture the changes in workload characteristics and to guarantee thermal constraints.

We solve the optimization problem using Matlab CVX[3] tool. The policy takes an average of 1 second on a computer with Intel i3 3.3GHz processor when solving for a data center of 360 servers. A 1-second overhead on an average desktop demonstrates that the algorithm can run sufficiently often without noticeable overhead in a data center environment.

4.1.3 Comparison with Other Policies

During our evaluation, we use data center simulations based on the linear data center model, and power, BIPS, and temperature data from real-life experiments.

We assume that job arrival times in our data center follow a Poisson distribution with a mean rate of 1 job per second, and each job has a mean service time of 3 minutes without power capping. This workload results in an approximate utilization of 50%, which is a typical value for an enterprise data center. An incoming job is randomly selected from our database, and it is allocated to the idle server whose temperature is the least affected by other servers. This corresponds to the idle server with the least row sum in $\mathbf{D}$.

We select the highest allowed processor temperature $T_{redline}$ as 75°C, based on two reasons: (1) Experimental analysis on our server shows that leakage power surpasses the fan power when $T_{cpu} > 75$°C, and therefore may adversely affect the energy savings; (2) it is desirable to operate with a margin from reliability-critical temperatures (e.g., 85-90C) of the CPUs to avoid throttling or accelerated failure probabilities.

First, we present an analysis on the impact of objective function selection. Second, we show the energy savings obtained only by collapsing the thermal headroom margin between server inlets and server internals (in our case, CPUs). Finally, we compare *CoolBudget* with a state-of-the-art policy to demonstrate the savings achieved by *CoolBudget*. We also show the impact of thermally-aware job allocation.

[3] `http://cvxr.com/cvx`

Objective Function	Normalized BIPS	Fair Speedup	Efficiency
Equal Power	1	79%	82%
max BIPS	1.27	0%	88%
min Max Degrad	0.98	78%	84%
max Fair Speedup	1.26	94%	89%

Table 4.1: Comparison of objective functions

Impact of Objective Function

Table 4.1 shows data center performance and efficiency using different objective functions, averaged over 100 random simulation snapshots with an average of 60% utilization and $P_{budget} = 200kW$. *Equal Power* function equally distributes the available power among active servers, *max BIPS* aims to maximize the total data center throughput, whereas *minMax Degrad* minimizes the maximum performance degradation across jobs. The BIPS results are normalized with respect to *Equal Power*. The results show that the proposed *Fair Speedup* objective function leads to the highest efficiency ($P_{compute}/P_{budget}$) and a total throughput close to *max BIPS*. 0% fair speedup in *max BIPS* means that minimal power is allocated to some servers, leading to starvation. The performance degradation results for active servers are given in Figure 4·4 as histograms. Maximizing fair speedup lets the performance of a few servers to degrade to increase the overall performance.

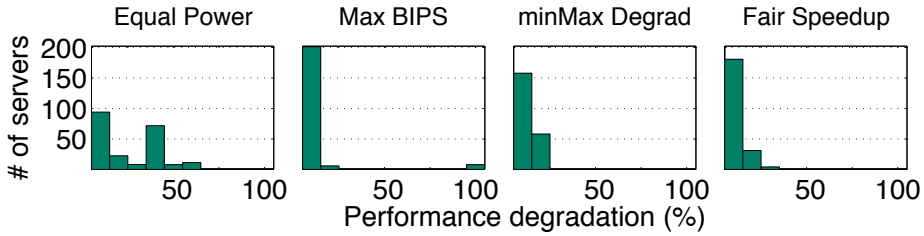


Figure 4·4: Performance degradation histograms for each objective function

Collapsing the Thermal Headroom in Servers

We compare our approach to a policy called server inlet based power budgeting (SI), where the only difference is that SI limits the server inlet temperature, T_{inlet}, instead of limiting T_{cpu}. As the CPU redline 75°C is the worst-case CPU temperature in our server at 24°C inlet and 2400 rpm fan speed, we select the T_{inlet} redline as 24°C.

With our default settings, SI cannot always find a feasible solution for the power constraints where *CoolBudget* displays no performance degradation. In other words, *CoolBudget* enables the support of much lower power limits. To be able to compare the two policies, we use a data center with reduced heat recirculation, where the recirculation matrix **D** is magnitude-wise halved.

Policy	Normalized BIPS	max(T_{cpu})	max(T_{inlet})	T_{crac}	Efficiency
Server Inlet (SI)	1	59.7°C	24.0°C	17.9°C	73%
CoolBudget	1.21	70.2°C	33.0°C	26.8°C	84%
ideal CoolBudget	1.28	75.0°C	37.9°C	31.7°C	88%

Table 4.2: Comparison of server inlet based cooling and *CoolBudget*

Table 4.2 shows the simulation results with reduced recirculation and with $P_{budget} = 230kW$, where *ideal CoolBudget* assumes perfect (zero error) modeling of the temperature and performance, and the BIPS results are normalized with respect to SI. Due to the over-prediction in the T_{cpu} estimation, *CoolBudget* leaves a temperature headroom of 4.8°C on the average. Increasing T_{crac} by 8.9°C leads to 21% increase in BIPS and 15% improvement in the efficiency.

Comparison with Baseline Budgeting Policy

We compare our policy with a prior approach called self-consistent budgeting policy, *SC* (Zhan and Reda, 2013). *SC* allocates a sufficient amount of power for cooling and budgets the remaining power among the servers. It does not, however, cool down

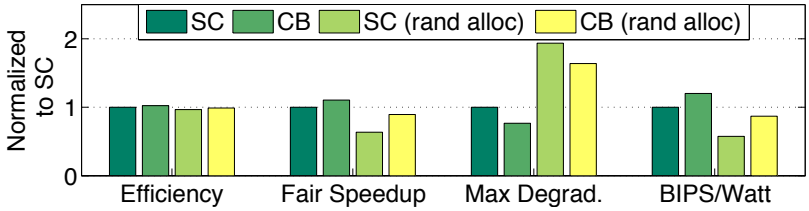


Figure 4·5: Comparison between self-consistent (SC) and *CoolBudget* (CB) policies under two job allocation scenarios with $P_{budget} = 200kW$

the hottest servers to reduce the cooling need as our policy does.

Figure 4·5 shows the comparison of *SC* and *CoolBudget* in both thermally-efficient and random job allocation scenarios. By redirecting more of the available power to computing, *CoolBudget* improves the fair speed-up by 10% and BIPS per Watt by 20% during efficient allocation. Random allocation does not affect the efficiency significantly; however, it decreases the fair speedup by 19-36% and the maximum degradation by more than 50%. This is because both *SC* and *CoolBudget* allocate lower power to thermally inefficient servers to keep a high efficiency, degrading the performance of the jobs in these servers.

4.1.4 Summary

Efficient data center power budgeting requires awareness of the power, performance, and temperature dynamics as well as intelligent policies that can improve efficiency based on the collected resource usage and physical sensor data at runtime. We have designed *CoolBudget*, which is based on a novel formulation of data center power budgeting problem and accurate power, performance, and temperature models that are validated on a real server. By leveraging data collected from both servers and other data center components such as cooling units, *CoolBudget* enables collapsing the thermal headroom between server inlets and internal components, increasing data center energy efficiency by over 10%. Furthermore, by using the *Fair Speedup* objective, we avoid significant unfairness among workloads while improving the overall efficiency.

4.2 Leakage-aware Server Cooling

In the preceding section, we have focused on distributing the available power across servers while taking data center cooling into consideration. In this section, we further improve the effectiveness of power budgeting and the overall data center efficiency by controlling the cooling of individual servers. To this end, we implement a novel server cooling control policy that proactively avoids thermal violations, providing a robust environment where the thermal headroom between server inlets and internal components can safely be collapsed (see Section 4.1). Furthermore, our policy reduces the thermally-related power consumption in servers by up to 6.4% compared to state-of-the-art. This section explains power and cooling dynamics in servers in detail and describes our proactive server cooling policy.

4.2.1 Cooling and Leakage Dynamics

The main purposes of the server fans are to remove the produced heat and to prevent overheating of the hottest components such as CPUs and memories. The fan speed should be carefully selected to avoid overcooling, which implies high cooling costs, and also overheating, which results in shorter component lifetimes and higher power. To clarify this point, Figure 4·6 shows the cubic increase in fan power with fan speed

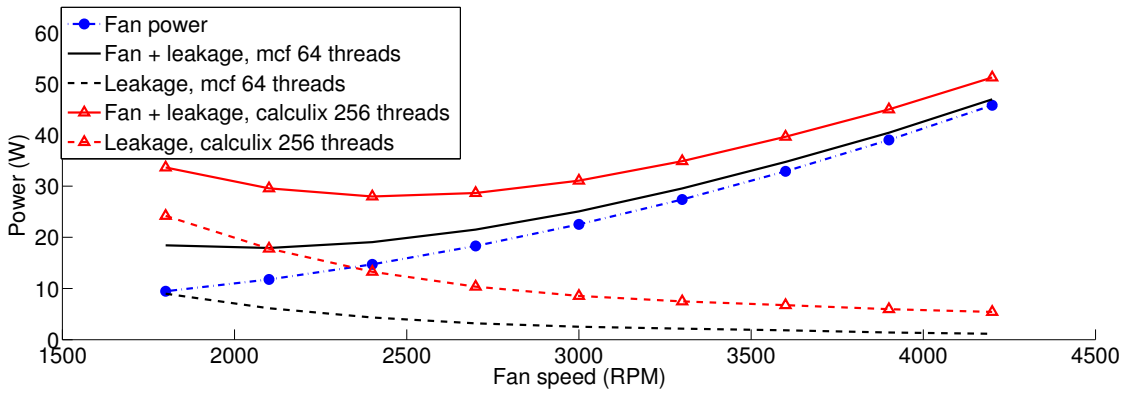


Figure 4·6: Fan and leakage power for various workloads.

as well as the exponential increase in leakage power when fan speed decreases for two particular workloads running on a highly multi-threaded enterprise server providing 256 hardware threads: (i) a memory intensive workload utilizing 25% of the server (64 copies of *mcf*) and (ii) a CPU intensive workload fully utilizing the server (256 copies of *calculix*). We observe that different RPM settings minimize the total fan plus leakage power for the two workload scenarios.

To find the fan speed where the overall power consumption is minimized, we propose a proactive fan control policy that is robust to different workloads, workload allocation policies, and ambient temperatures. To build this proactive policy, we develop accurate models that predict leakage and cooling power.

Modeling Methodology

We build models for fan power, temperature, and server power using the server setup and benchmarks described in Section 4.1.1. For model training, we only use the synthetic workload *LoadGen*, which allows us to stress the processors at any desired utilization level.

We enable customized fan control by setting the fan currents through external Agilent E3644A power supplies. We map the input current values to fan speeds, which are inferred with high accuracy by taking the Fourier transform of vibration sensors. In our work, we use a minimum fan speed of 1800RPM, and a maximum of 4200RPM. We avoid using lower fan speeds which lead to unstable fan behavior. On the other hand, 4200RPM overcools the server under our experimental conditions, and is above the maximum server default fan speed.

Processor Power

We focus on processors as they exhibit the majority of the temperature-dependent leakage power (Patterson, 2008). We also studied memory power using synthetic

benchmarks and verified that the temperature dependence of memory power is negligible in our system.

We divide CPU power into leakage and dynamic power as follows:

$$P_{CPU} = P_{idle} + P_{leak} + P_{dyn} \tag{4.11}$$

where P_{idle} is idle power consumption, P_{leak} is the increase in leakage power due to temperature, and P_{dyn} is the CPU dynamic power due to workload execution.

To model the relation between temperature and leakage power, we run *LoadGen* with 100% utilization for all the available fan speeds. Because the workload is constant in all experiments and the only control knob is fan speed, power consumption can only change due to leakage. We approximate the exponential dependence of leakage power on temperature using a second order polynomial:

$$P_{leak} = \alpha_0 + \alpha_1 \cdot T + \alpha_2 \cdot T^2 \tag{4.12}$$

where α_i's are regression coefficients, and T is the CPU temperature in Celsius. Figure 4·7 shows the data regression against the measured samples for both CPUs.

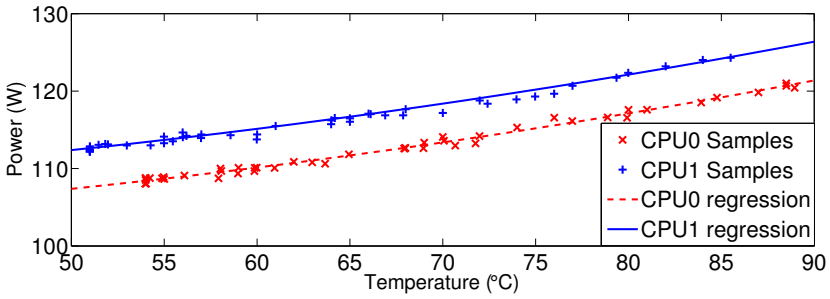


Figure 4·7: CPU leakage model regression for both CPUs.

We validate our model by running SPEC CPU and PARSEC workloads under different fan speeds, and subtract P_{leak} from the power traces. Because the executions of a given workload only differ in fan speed, the remaining power ($P_{CPU} - P_{leak} = P_{dyn}$) should be the same, and the difference between P_{dyn} traces under different fan speeds

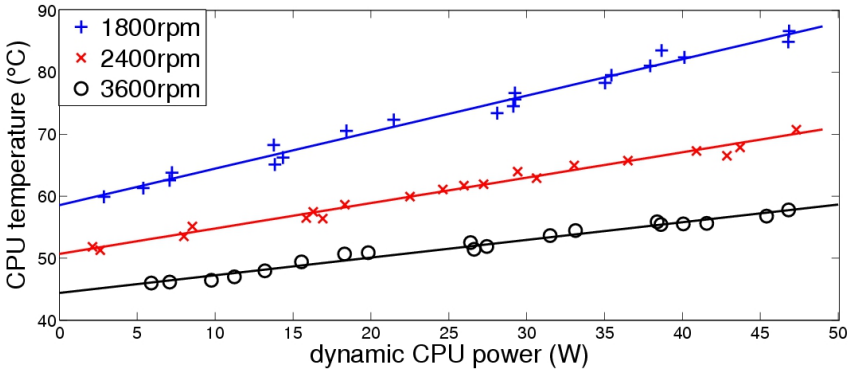


Figure 4·8: Steady-state temperature model and measured samples for three different fan speeds.

is a direct analytical estimate of the error of our model. The average error of 0.67W shows very high accuracy.

Processor Temperature

Using our power model, we estimate server leakage power at a given temperature. To adjust fan speed at runtime and minimize the energy consumption, we also need to predict future temperature to compensate for thermal delays associated with processor and heat sink. For this purpose, we propose a model which first predicts steady-state temperature based on power measurements and fan speed, and then estimates transient behavior.

The steady-state temperature of a processor that runs a constant workload is strongly correlated with dynamic power. Each dynamic power level has a corresponding steady-state CPU temperature for a given fan speed. Hence, we predict dynamic CPU power using our power model in Equation 4.11.

In our experiments, we observe a linear relationship between the steady-state chip temperature and the dynamic power consumption for each fan speed as demonstrated in Figure 4·8. To train our model, we run *LoadGen* with different duty cycles to vary average dynamic power, and record the steady-state temperature. We repeat the

same procedure for each available fan speed and derive models in the following form:

$$T_{ss} = k_0^s + k_1^s \cdot P_{dyn} + T_{amb} \tag{4.13}$$

where T_{ss} is the steady-state CPU temperature, k_0, k_1 are the model coefficients corresponding to fan speed s, and T_{amb} is the ambient temperature. We validate our model by running a set of SPEC CPU2006 workloads at two different ambient temperatures, 22°C and 27°C, where we obtain a maximum error of 6.6°C and root-mean-square error below 2.1°C. This accuracy is sufficient for our purposes.

When processor power varies, temperature changes exponentially with a time constant. We compute the thermal time constant of each fan speed by fitting exponential curves to the temperature measurements obtained while running *LoadGen* after the idle steady-state. As seen in Figure 4·9, the time constants, maximum observable temperatures and temperature range decrease as the fan speed increases. As the small changes in temperature do not affect the leakage power significantly, we only need to detect the large changes with time constants in the order of minutes. With such long time constants, we only predict the average temperature in the next minute using the closed form integration of our transient temperature model. A more fine-grained temperature prediction will lead to better approximations to the optimal fan speed by capturing small changes in the temperature; however, it will also induce

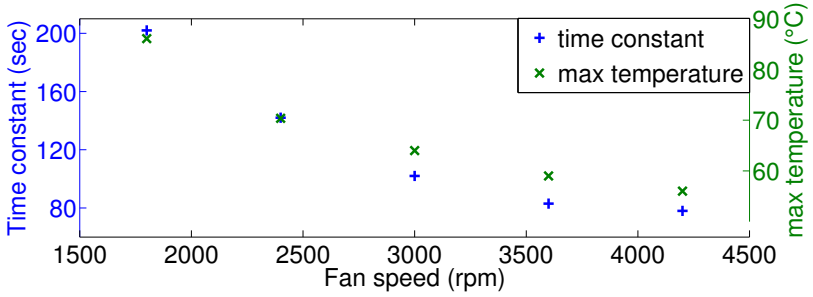


Figure 4·9: Thermal time constant and maximum temperature under various fan speeds

unnecessary changes in fan speed and decrease the lifetime of the fans. The duration of the temperature prediction should be selected considering this trade-off.

4.2.2 Leakage-aware Fan Control

Our fan control policy uses temperature and power measurements to proactively determine the fan speed that minimizes the *fan plus leakage* power. Algorithm 1 shows the pseudo-code of our policy, where f represents functions for models described in Section 4.2.1. The procedure is called every time epoch of length τ_{wait}.

Algorithm 1 Fan control procedure

```
1:  for each CPU p do
2:      P^p_leak = f_leakage(T^p_measured)
3:      P^p_dyn = P^p_measured − P^p_idle − P^p_leak
4:      for each fan speed s do
5:          T^{p,s}_ss = f_temperature(P^p_dyn, T_amb, s)
6:          T^{p,s}_avg = f_transient(T^p_measured, T^{p,s}_ss)
7:          if T^{p,s}_avg ≥ T_critical then
8:              P^{p,s}_leak = ∞
9:          else
10:             P^{p,s}_leak = f_leakage(T^{p,s}_avg)
11:         end if
12:     end for
13: end for
14: for each fan speed s do
15:     P^s_total = P^s_fan + Σ_p P^{p,s}_leak
16: end for
17: Set fan speed to argmin_s(P^s_total)
18: Wait τ_wait while monitoring P_dyn
```

Our policy first calculates the leakage power P_{leak} for each CPU p using the leakage power model in Equation 4.12 (line 2), and the dynamic CPU power using Equation 4.11 (line 3). As every P_{dyn} corresponds to a steady-state temperature T_{ss} under a given fan speed (Equation 4.13), the policy then computes the expected average temperature over the next τ_{wait} period, using a closed form integration of the transient temperature prediction (lines 5-6). Given the average temperature, expected leakage power is calculated. We prevent the selection of fan speeds that result in temperatures above the critical value $T_{critical}$, by setting the corresponding leakage power to infinity.

Finally, the total leakage + fan power is calculated for each fan speed (line 15), and the fan speed that minimizes power consumption is selected (line 17). The system then waits τ_{wait} seconds while monitoring the system for a workload change. If the dynamic CPU power changes significantly, this interval is interrupted and the

optimum fan speed is re-calculated. This waiting time ensures the stability of the controller and prevents the fan reliability issues that could arise with very frequent fan speed changes (i.e. in the order of seconds). For our system, we choose a τ_{wait} value of 1 minute, which is a safe choice considering the large thermal time constants.

We use a 300RPM resolution for the fan speed selection in our policy, which is a heuristically selected value. This resolution is selected such that the available fan speeds lead to a sufficient approximation to the optimal cooling conditions. Selecting an unnecessarily fine resolution will increase the computational overhead of the policy.

The fan speed control policy is run by the DLC-PC. The policy measures and averages power every second, and decides on the fan speed every minute using look-up tables and polynomials. The leakage and temperature prediction is computed only for 9 different fan speeds that cover the entire fan speed range (1800 to 4200RPM) with a resolution of 300RPM. As these are very simple operations with long periods, the policy has negligible overhead and can be implemented in the service processor.

4.2.3 Evaluation

This section presents several state-of-the-art policies and compare their performance against our proposed proactive fan control strategy. In addition, we provide an estimation of the impact of our policy on data center level power consumption based on power traces from a real data center.

Baseline Policies

Below are the baselines policies used in the evaluation of the proposed policy:

Best fixed: The default server fan policy sets a fixed fan speed that ensures the server reliability for a worst-case scenario for each ambient temperature, leading to overcooling. To ensure the fairness of the comparison and evaluate to benefits of dynamic fan speed selection, we use fixed 2400RPM as a baseline, which minimizes

leakage plus fan power for the majority of the workloads.

TAPO: The TAPO fan control policy (Huang et al., 2011) changes the thermal set point T_{sp} of the processor to indirectly control the fan speed. Assuming the workload is constant, once the thermal steady-state is reached, the policy changes T_{sp}. Then, it observes the change in the processor temperature and power to decide whether to increase or decrease T_{sp} to achieve lower power.

Bang-bang: The bang-bang is a multi-threshold controller that aims to keep CPU temperature within a desirable range. Our implementation tries to maintain temperature within the 65°C-75°C, thus: (i) if maximum observed temperature T_{max} goes below 60°C, fan speed is set to 1800RPM (lowest); (ii) if T_{max} is in between 60°C to 65°C, fan speed is lowered by 600RPM; (iii) if T_{max} is between 65 to 75 degrees, no action is taken; (iv) if T_{max} rises above 75°C, fan speed is increased by 600RPM; and, (v) if T_{max} is above 80°C, fan speed is increased to 4200RPM. The threshold values are heuristically chosen to optimize the tradeoff between high fan speed change frequency and high temperature overshoots while keeping temperature in a range that ensures low total power (Zapater et al., 2013).

Lookup table based fan control (LUT): This policy (Zapater et al., 2013) monitors CPU load periodically and tries to minimize the leakage plus cooling power by setting the optimum fan speed during run-time depending on the utilization of the server. For that purpose, a look-up table that holds the optimum fan speed for each utilization value is generated using LoadGen.

Workload Profiles

We generate 4 different workload profiles: (1) high power and long jobs, (2) low power and long jobs, (3) high power and short jobs, (4) low power and short jobs. Each profile consists of 10 randomly selected jobs from SPEC or PARSEC benchmarks with certain number of copies as described in Section 4.1.1, generated with a Poisson

distribution of arrival and service times. The mean arrival and service times are 25 and 20, respectively, for long jobs, and 15 and 10, respectively, for short jobs.

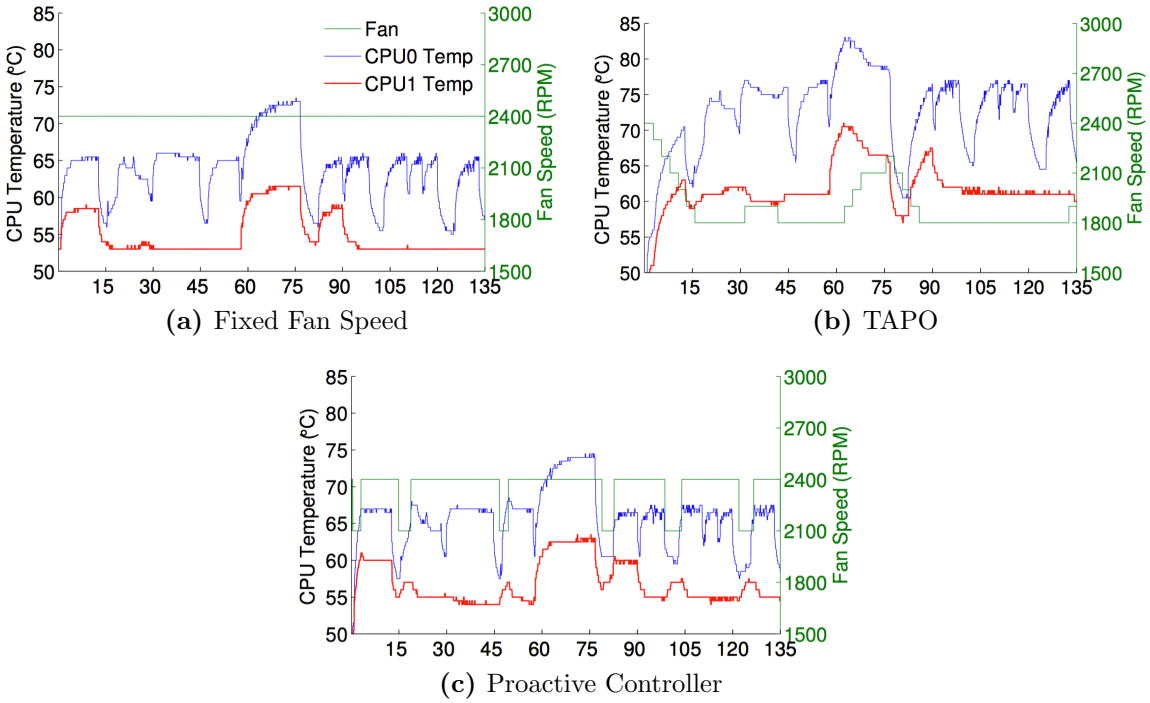
To generate profiles with variable stress in terms of power consumption, all benchmark applications are arranged into two classes: high power consumption and low power consumption.The probability of a job being selected from high power group is 0.8 and 0.2 for high and low power profiles, respectively.

Experimental Results

We evaluate the workloads under two allocation schemes: clustered and distributed. Clustered allocation packs all the threads together into the first N cores of the server, and distributed allocation spreads the workload as much as possible into all available cores. While distributed allocation reduces the maximum temperature, depending on the workload, it can increase energy consumption as it prolongs workload execution due to (1) added communication between two CPU sockets and (2) lower cache utilization. In our experiments, we do not observe a clear winner among the two allocation schemes in terms of energy consumption.

Figure 4·10: Leakage plus fan energy savings achieved by the proactive fan control policy compared to baseline policies.

Figure 4·10 shows the leakage plus fan energy savings achieved by our proactive policy under distributed allocation. The proactive policy consistently reduces energy consumption under all workload profiles and up to 6.4%. Note that the fixed fan



(a) Fixed Fan Speed

(b) TAPO

(c) Proactive Controller

Figure 4·11: CPU temperature and fan speed traces for workload profile 1 with clustered allocation using different fan controllers.

speed policy is already selected considering the leakage-cooling trade-offs and reduces the leakage plus fan energy by more than 8% compared to the default server policy.

Figure 4·11 shows the fan speed and the processor temperature trends of the fixed fan speed policy, TAPO policy, and the proactive policy running workload profile 4. As the proactive policy adjusts fan speed before the temperature has changed, it reduces the range of thermal oscillations, which are strongly linked to reducing lifetime reliability of chips (Xiang et al., 2010).

Impact at Data Center

We calculate the impact of our server fan control policy on the overall data center based on server power traces of a high-performance computing cluster consisting of 260 computer nodes in 9 racks at the Madrid Supercomputing and Visualization Center (CeSViMa). By using the telemetry deployed in CeSViMa, we gather 3 hours

of real server power traces for 256 servers. We use these power traces to simulate our proactive policy in a larger-scale scenario with a real workload trace, and compute the energy savings that our policy would achieve compared to the fixed fan speed policy and the server default policy.

We account for the effect of different ambient temperatures on the data center cooling based on data by Miller et al., in which each degree of increase in room temperature yields 4% energy savings in the cooling subsystem (Miller, 2007). As room temperature raises, the fan speed needed to keep servers within safe environmental conditions also increases. Hence, in our case study, we use a fixed fan speed of 2400RPM, 2700RPM, and 3000RPM as a baseline for comparison under 22°C, 27°C, and 32°C ambient temperature, respectively.

Our proactive policy outperforms both the fixed and the default server fan policies for all power traces and under every ambient temperature scenario. The savings obtained are 1.9% at 22°C ambient temperature, 5.5% at 27°C, and 10.3% at 32°C for the whole cluster in *leakage plus fan* power. This is translated into a reduction of 2.5% in the total CPU energy consumption of the cluster at 27°C ambient.

4.2.4 Summary

Most existing server fan control policies are reactive and unaware of leakage power. As we demonstrated in this section, the efficiency of existing fan control policies can be improved by leveraging the continuously collected CPU power data and using accurate models of temperature, leakage power, and cooling power. Compared to existing policies, our approach consistently reduces server energy consumption, reaching up to 6.4% of leakage plus fan energy savings. Moreover, our fan control policy proactively avoids thermal violations and is application-agnostic.

4.3 Efficient Topology Mapping in HPC Systems

This chapter so far focused on power management, which has a direct impact on data center energy efficiency. However, other aspects of data center management such as resource allocation can also improve overall energy efficiency by reducing wasted compute resources. In this thesis, we specifically focus on topology mapping for HPC applications, which is the placement of parallel application tasks (e.g., MPI ranks) onto the available compute nodes. As most HPC applications have a specific communication topology, which can be extracted from historical communication data (Zhai et al., 2009), topology mapping can be expressed as mapping an applications' communication graph onto the target machines' network graph. While topology mapping is an NP-hard problem (Hoefler and Snir, 2011), placing highly-communicating application tasks close to each other using heuristics has been shown to reduce execution times of HPC applications by up to 34% (Deveci et al., 2014).

In today's HPC systems, topology mapping includes decisions both from the system side and from the application side. The system is responsible for scheduling and allocation, whereas the application performs task mapping. In this section, we introduce our HPC job placement policy, PaCMap (Tuncer et al., 2015), which is the first to use a holistic view on topology mapping by combining the system-driven node allocation and the application-driven task mapping algorithms. This holistic view leads to a more efficient job placement compared to existing techniques, reducing application communication volume in terms of *hop-bytes* by up to 30%.

Figure 4·12 depicts conventional HPC workload management. The job submission only supports sending basic requests to the system, such as number of processing cores, memory requirement, and the worst case execution time (WCET), where a job refers to a specific instance of an application. Then, as part of system software, the scheduler determines when to run the job based on machine availability and job

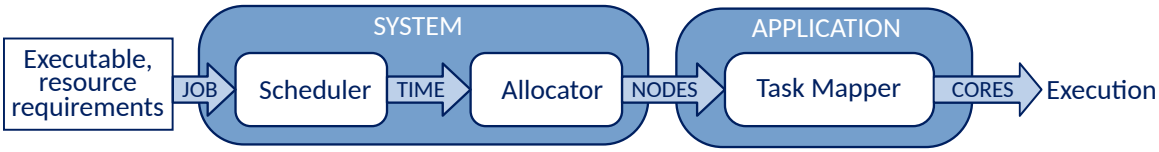


Figure 4·12: Workload management in conventional HPC machines.

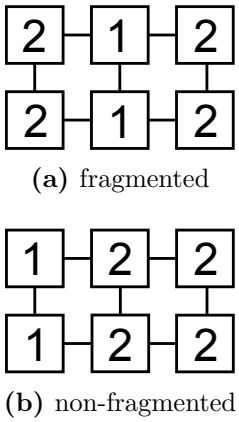
queue. Once the job is scheduled, the allocator assigns a set of machine nodes for the job. Finally, the application's task mapper places its tasks onto the allocated nodes. In this work, we assume there is no space sharing, i.e., a machine node cannot be shared by multiple jobs. This is common in HPC to enable tuning of applications to the available resources in compute nodes for improved performance.

Due to the limitations of the job submission framework, the system is unaware of the exact communication pattern of a job for unstructured applications. For task mapping, however, programmers can specify the communication pattern through interfaces such as MPI (MPI Forum, 2012). Alternatively, the pattern can be profiled during an application's first run and used in the future runs. In addition to the communication pattern, the application side can also discover the physical network topology through system calls (Subramoni et al., 2012). Based on these information, task mapping can make an efficient assignment of the application tasks to the machine nodes to minimize the communication overhead.

Job Allocation

There are two main considerations during job allocation. First, the selected nodes should be close to each other to reduce the communication distances. Second, the allocation should not lead to fragmented machine utilization. The allocation algorithms that disregard the second issue can lead to the segmentation of large empty blocks in the machine, potentially increasing the communication overhead of future jobs. Consider the example shown in Figure 4·13, where two jobs are scheduled in a

6-node machine with a 2D mesh topology. In case (a), the first job is allocated to the middle column. Thus, job 2 must be fragmented into smaller parts, increasing communication distance. However, case (b) assigns job 1 to the side of the machine, leaving sufficient space for job 2.



(a) fragmented

(b) non-fragmented

Figure 4·13: Fragmented and non-fragmented job allocation. The boxes represent machine nodes and the numbers represent allocated jobs.

The common techniques for non-contiguous job allocation can be classified into two categories: linear and clustered. We use one algorithm from each category.

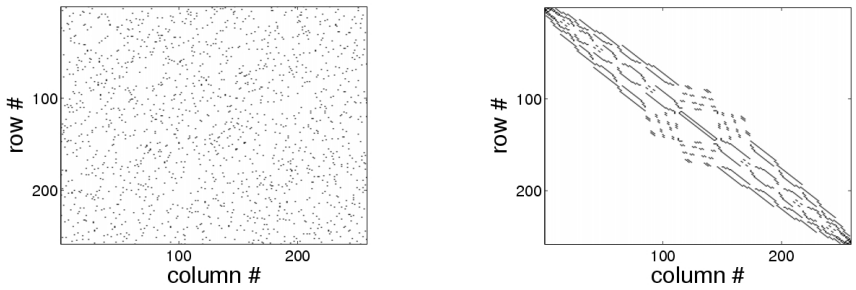
Best-fit (linear): The best-fit strategy is a combination of the ideas proposed by Lo et al. (Lo et al., 1997) and Leung et al. (Leung et al., 2002). The algorithm first linearly orders the machine nodes along a curve. Then, the free nodes are grouped into intervals along this curve. The job is allocated to the smallest interval that has sufficient nodes. If there is no such interval, the algorithm selects the nodes that minimize the maximum distance along the curve. This strategy is commonly used in real HPC machines due to its simplicity and its small time complexity of $O(M)$ in an M-node machine. In machines with a torus or mesh network topology, linear allocation strategies order the nodes using space-filling curves such as Hilbert curves to improve locality (Auble and Christiansen, 2014).

MC1x1 (clustered): The MC1x1 algorithm is a variant of the MC algorithm proposed by Mache et al. (Mache et al., 1997). It is a $(2 - 2/k)d$ approximation to the optimal solution for minimizing the average pairwise L1 distance of tasks in a d-dimensional mesh when allocating k processors (Bender et al., 2008). The algorithm aims to find a compact cluster of free nodes. For each free node n, MC1x1 calculates an allocation score by counting the number of free nodes in a d-dimensional hypercube centered on node n. Although MC1x1 finds clustered nodes, it does not address the

fragmented allocation problem and it is not applicable in network topologies other than mesh and torus. The time complexity of this approach depends on the machine state. In the worst case, calculating the score of a free node requires checking the availability of all the nodes in the machine, whereas the best case checks only the nearest J nodes of J free nodes. Hence, depending on the machine state, allocating nodes with MC1x1 takes between $O(M^2)$ and $\Omega(J^2)$, where M is the number of nodes in the machine.

Task Mapping

Task mapping considers the assignment of the individual application tasks to the machine nodes, which are selected by the job allocation stage (Section 4.3). Unlike job allocation, task mapping is able to use information provided by the application such as the communication pattern. We focus on three different task mapping techniques and use them as baselines for the evaluation of the proposed *PaCMap* algorithm.



(a) Original matrix **(b)** RCM permutation

Figure 4·14: RCM applied on a sample sparse matrix

Reverse Cuthill-Mckee (RCM): RCM (Cuthill and McKee, 1969) reduces the bandwidth of a symmetric matrix via permutation, i.e., it reorders the matrix such that the non-zero entries that are far from the diagonal are eliminated as demonstrated in Figure 4·14. When applied on a task communication matrix, which shows the communication links between the tasks, this corresponds to reducing the maximum

distance between the tasks when the tasks are linearly ordered.

In machines with contiguous allocation, RCM can be applied both on the network and the communication graphs with $O(JD \log D)$ complexity, where D is the maximum degree of the application graph. Then, in-order mapping of the tasks to the nodes can effectively reduce the communication distance for sparse communication patterns (Bhatelé et al., 2010). In machines that allow non-contiguous job allocation, the machine network cannot be directly used as it results in an unconnected graph. Instead, the tasks are mapped in-order along the curve of a linear allocation.

Recursive Graph Bisection (RGrB): RGrB algorithm uses the task communication graph and the network topology graph. It recursively splits both graphs into equal halves using minimum weighted edge-cuts, and maps the remaining task(s) to the remaining node at the end of the recursion. This algorithm has been used for contiguous allocation by software packages such as LibTopoMap (Hoefler and Snir, 2011) and SCOTCH (Pellegrini and Roman, 1996b). As there is no RGrB variant specific to non-contiguous allocation, we apply RGrB by building a virtual all-to-all graph for the machine network, where the edges are weighted based on the hop distance between the nodes. Due to the all-to-all graph, RGrB takes $O(J^3)$ for non-contiguous allocation for mapping J tasks (Hoefler and Snir, 2011). Our implementation of RGrB is based on LibTopoMap, and uses the METIS library (Karypis and Kumar, 1998) for bisectioning. Although this technique demonstrated efficient mappings, it is also shown that it may result in poor p-way partitions (Simon and Teng, 1997).

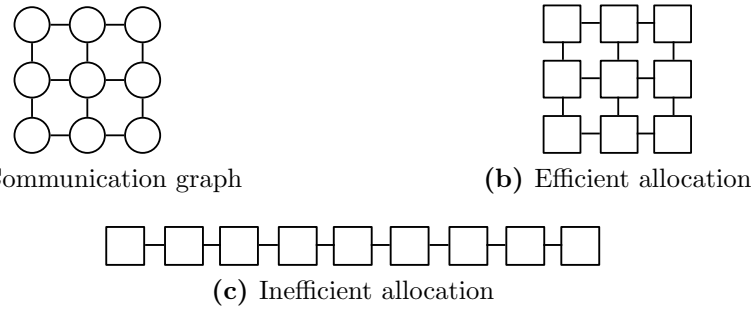
Recursive Geometry Bisection (RGeoB): Similar to the RGrB, RGeoB is based on recursive bisections. Instead of using the graphs, however, RGeoB splits the application and the machine geometries into equal halves such that the maximum dimension length is minimized (Deveci et al., 2014). The application geometry can be inherent to the application, such as the coordinates of an object in a computational

fluid dynamics simulation, or it can be generated from the communication graph. Similarly, the machine geometry can be defined as x, y, z coordinates of a 3D mesh network topology. When a generic sorting algorithm with $O(J \log J)$ is used for the geometric bisectioning, the complexity of RGeoB becomes $O(J \log^2 J)$ as shown by the master theorem (Cormen et al., 2001). The effectiveness of RGeoB strictly depends on how well the communication is represented by the given application geometry. For 3D stencil computations, this technique is shown to perform better than RGrB on a Cray XE6 (Deveci et al., 2014).

Interplay of Allocation and Mapping

As mentioned earlier in this section, the allocator is unaware of application communication patterns in current HPC systems. Hence, allocation decision can decrease the potential efficiency of the task mapping algorithm. Consider the example in Figure 4·15. A communication-aware allocation algorithm will allocate a 3x3 mesh (Figure 4·15b) for an application with a 3x3 stencil communication pattern, so that task mapping can reduce the average message hop distance to 1. However, if the allocator is unaware of the communication pattern, it can select the nodes as in Figure 4·15c, where the minimum achievable average message hop distance is 2.

Similar effects can also be observed in larger scales. Consider a 3D stencil ap-



(a) Communication graph **(b)** Efficient allocation

(c) Inefficient allocation

Figure 4·15: Communication graph of a 3x3 stencil application, and (b) communication-aware and (c) -unaware allocation examples

plication with dimensions 2x8x32. If this application is allocated to a 2x8x32 mesh, the minimum average hop distance would be 1. However, when it is allocated to a 8x8x8 cubical mesh instead, the RGeoB task mapping algorithm, being the best task mapper for this case, results in an average hop distance of 1.92, increasing the communication overhead. These example cases create sufficient motivation to develop algorithms that jointly consider allocation and mapping.

4.3.1 Joint Job Allocation and Task Mapping with PaCMap

We propose a topology mapping algorithm called *PaCMap* (Partitioning And Center MAPping) to improve the workload placement. *PaCMap* unifies system- and application-controlled workload placement algorithms to exploit asymmetries and irregularities in both data center network topology and application's communication topology. *PaCMap* can be used for any application with a distinct communication pattern in machines with both contiguous and non-contiguous allocation.

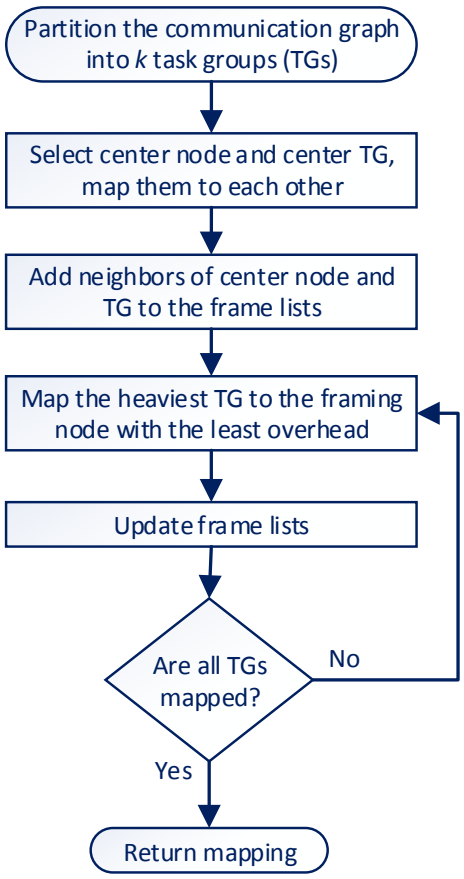


Figure 4·16: *PaCMap* overview

As shown in Figure 4·16, *PaCMap* first partitions the communication graph into k task groups (TGs) such that each group can fit into a single node in the cluster. This step consolidates the highly-communicating tasks to be executed on the same machine node. After this step, the problem reduces to mapping the TGs into the available machine nodes. In our implementation, the partitioning is done by the multilevel

k-way partitioning algorithm from the METIS library (Karypis and Kumar, 1998). Next, *PaCMap* selects a *center TG* from the partitioned graph and maps it to a selected *center node* in the cluster. Then, it expands the allocation by picking a node and mapping a TG to it based on the network topology and on the communication graph until all tasks are mapped.

PaCMap can also be used only for allocation by ignoring the mapping decision, or only for task mapping by limiting the nodes the algorithm can use. The rest of this section explains in detail how the center node and the center TG are selected and how the expansion is performed, along with a complexity analysis.

Center Machine Node Selection

As discussed in Section 4.3, the allocation determines how efficiently the cluster is utilized. The algorithm should select a collected group of nodes and should not lead to fragmented allocation of future jobs. Our solution is a heuristic that addresses both issues.

For each available node n, we look at the other available nodes in the proximity of n and calculate a node score $NS_{n,J}$, where J is the number of nodes to be allocated. To calculate the score, we first create a list of available nodes around n within a communication distance of R using breadth-first expansion. Then, we sort this list with respect to the distance to n. Starting from the closest node, we increase $NS_{n,J}$ for the first J nodes, and penalize it for the remaining extra nodes as follows:

$$NS_{n,J} = \sum_{i=1}^{J} f(dist_{n,i}) - \sum_{i=J+1} f(dist_{n,i}) \tag{4.14}$$

where $f(dist_{n,i})$ is a function of $dist_{n,i}$, which is the communication distance between n and node i in the list. To avoid fragmented allocation, R should be selected such that the number of nodes within the network distance of R is larger than J. Additionally, $f(dist_{n,i})$ should prioritize the nodes that are closer to node n to improve locality.

R and $f(dist_{n,i})$ are selected based on the network topology. We use the following heuristics for 3D torus topologies: The maximum number of nodes within a network hop distance of r equals $4r^3/3 + 2r^2 + 8r/3 + 1$ in a 3D torus. Using a pre-calculated look-up table based on this formula, we first find the minimum distance r_{min} that contains J nodes. We then select $R = r_{min} + 2$ to check the excessive availability around the node n. This number can be adjusted for different machines if the above formula becomes invalid due to asymmetrical torus dimensions. We use $f(dist_{n,i}) = 1/(4dist_{n,i}^2 + 2)$ so that the maximum total impact of the nodes equidistant to n on $NS_{n,J}$ is 1. We do not observe a significant change in the allocation performance by the selection of $f(dist_{n,i})$ as long as the nodes closer to n have more impact than others. In other network topologies such as Dragonfly (Kim et al., 2008), the same methodology can be followed to select R and $f(dist_{n,i})$.

Center Task Group Selection

PaCMap expands the allocation from the center node and the center TG, while allocating nodes and mapping TGs at each expansion step. As the center machine node selection stage scores the nodes based on the availability around them without prioritizing any direction, an efficient node selection requires the expansion to be symmetric in all directions. This can be done by selecting a proper center TG. The center TG is selected as the one with the minimum cumulative shortest-path distance to all other TGs. To find the center, we run Dijkstra's algorithm on all TGs.

Expansion

After allocating the center TG to the center node, we create two lists for the expansion: (1) a list of machine nodes that frame the current partial allocation, (2) a list of TGs that frame the currently allocated TGs. The frames are the neighboring nodes/TGs in the corresponding graphs, as demonstrated in Figure 4·17. The edge

(a) Communication graph **(b)** Network graph

Figure 4·17: Partially allocated application. The solid shapes are current allocation/mapping, the dashed shapes are the frames, and striped squares are busy nodes.

weights in the machine graph (the network links) are all 1, whereas the edge weights in the TG graph are given in the figure.

For each expansion step, we select the heaviest TG, i.e., the unmapped TG that has the largest communication volume with the currently allocated tasks. In Figure 4·17a, D would be selected for the next step with a total weight of $2 + 3 = 5$. We map this TG to the node that leads to the least total communication overhead, calculated as follows:

$$overhead_n = \sum_i dist_{n,i} \cdot W_{n,i} \tag{4.15}$$

where $W_{n,i}$ is the communication weight between the TGs mapped to nodes n and i. In Figure 4·17b, node #2 would be selected for task D with an overhead of $1 \cdot 2 + 1 \cdot 3 = 5$. If there is a tie between the nodes, we select the closest node to the center to maintain the symmetry of the expansion.

After mapping the heaviest TG to the selected node, we update the frames by adding the neighbors of the allocated node and the mapped TG. If there is no free node among the direct neighbors of the allocated node, we increase the search distance until a free node is found.

Overhead of PaCMap

Given that most HPC applications are re-run with different parameters or inputs, partitioning of the communication graph into TGs and selection of the center TG need to be done only once per application. The results can be re-used for the future submissions of the same application. Alternatively, this information can be passed to the system along with the job submission.

Selection of the center node and expansion should be done at runtime. The overhead of the center node selection strictly depends on the underlying network topology. We make use of the coordinate information of the 3D torus topology, and we check the availability of $O(R^3) = O(J)$ nodes for each available node. This can be done in parallel for the total number of nodes in the machine, M.

During expansion, finding and deleting the heaviest TG for J TGs takes $O(J \log J)$ with TG frame as Fibonacci heap and node frame as a linked list. Selecting the node that leads to the least communication requires calculating the communication overhead for all neighbors of the heaviest TG and for all nodes in the frame. During the entire mapping process, J nodes will be mapped and the communication weight of C links will be calculated. In addition, at any time, there will be a maximum of $O(J)$ nodes in the frame as the number of ports of a router in a 3D torus network is bounded. Hence, selecting nodes during mapping takes a total of $O(J^2 + JC) = O(JC)$ for connected graphs. Note that the best node can be also selected in parallel. Updating the task frame is done by breadth-first search, where each step includes updating the weights of the unmapped neighboring TGs in the Fibonacci heap, leading to a total of $O(J + C)$. The complexity of updating the node frame depends on the machine utilization, where worst-case requires checking the availability of $O(M)$ nodes, leading to $O(MJ)$ during the entire mapping process.

As a result, the complete allocation and mapping process has a time complexity

of $O(MJ + JC)$ for a connected communication graph and for 3D torus machines. This complexity is feasible in terms of real-life implementation and scalability, and it is comparable to our best-performing baselines.

4.3.2 Evaluation

As real HPC machines do not have the infrastructure to support combined allocation and task mapping, we use simulations for our evaluation. For this purpose, we use the Structural Simulation Toolkit (SST), which is an architectural simulation framework designed by Sandia National Laboratories to assist in the design, evaluation and optimization of HPC architectures and applications (Rodrigues et al., 2012). We extend the scheduler module in SST to consider task mapping and implement a communication-aware performance model to evaluate *PaCMap*. We also create realistic HPC workload traces and communication patterns to feed the simulator.

Target Machine

Our target machine uses static mapping, i.e., it does not support task migration at runtime. We use a 3D torus network topology, which is commonly used in HPC machines due to its low cost, ease of design and installation, and high bisection bandwidth (i.e., the total bandwidth of links placed between two equal-sized node sets after partitioning). Example machines that use 3D torus include IBM BlueGene/L, Cray XE6, and Cray XK7 (Alverson et al., 2012). During our simulations, we use static shortest path dimension-ordered (x-y-z) routing in the network.

Workloads

In order to compare the task mapping algorithms, we need a comprehensive set of unstructured sparse communication graphs as well as application geometries. For this purpose, we use the University of Florida Sparse Matrix Collection (Davis and

Hu, 2011) as a proxy for communication and geometry information. This collection is commonly used for the evaluation of graph algorithms such as bisectioning (e.g. (Holtgrewe et al., 2010)), and consists of data from real applications in various fields such as circuit simulations, financial modeling, and chemical process simulations. We use the applications in the collection with 2D or 3D geometry information with up to 115K tasks. We assume uniform communication between the tasks as it is the expected case for well-balanced HPC applications.

The comparison of non-contiguous allocation algorithms requires using an already populated machine. Additionally, our analysis should account for the impact of the allocation decision on the performance of future jobs. To address both issues, we use the logs provided in the Parallel Workloads Archive (PWA) (Feitelson et al., 2014) as inputs to our simulation, and evaluate entire workload traces. PWA logs are collected from real large scale parallel systems, and provide information on job arrival times, execution times, and job sizes, but no information on the communication. Hence, we assign communication patterns to the jobs by matching them with the proxy communication matrices in the sparse matrix collection. As the job sizes in PWA do not necessarily match with the matrix sizes in the collection, we apply *binning* to the PWA, in which the job sizes are changed to the closest available size in our application set.

Among the logs in the archive, LLNL-Atlas and CEA-Curie traces lead to the most balanced application counts after binning. LLNL-Atlas and CEA-Curie are collected from machines with 36864 and 93312 cores, respectively. These are the largest machine sizes in the archive after ANL-Intrepid, which leads to an unbalanced binning and biases the results by prioritizing only a few applications in our input set. Logs from the newer and larger machines are not available in the archive due to the explosive growth of machine sizes and the lag in collecting data for research. We

use the first two weeks of these two traces in our evaluation as machines like Cray Cielo[4] are usually taken down for maintenance every two weeks. Our LLNL-Atlas and CEA-Curie traces consist of 1001 and 3291 jobs, respectively.

Performance Model

During our simulations, we modify the execution time of a job based on its communication pattern and mapping. To extract the relationship between communication time and the network-related metrics, we conduct real-life experiments on the Cray XE6 Cielo supercomputer located at Los Alamos National Laboratories. Cielo consists of 8944 compute nodes and additional service nodes. Each compute node has a dual AMD Opteron 6136 eight-core "Magny-Cours" socket, providing a total of 16 cores. The nodes are connected using a Cray Gemini 3D torus network topology with the dimensions of 16x12x24 and two nodes per Gemini.

We use the miniGhost application (Barrett et al., 2012), which is a part of the miniapps developed by the DOE community to represent the computational core of various HPC applications. MiniGhost focuses on the nearest neighbor inter-process communication strategy, with computation mainly serving to provide enough data and separation of the boundary exchanges from some computation. Its core is based on CTH, an application for modeling complex multi-dimensional problems that are characterized by large deformations and/or strong shocks (Hertel et al., 1993).

We run miniGhost with sizes of powers of 2, from 64 to 65536. For each size, we run miniGhost under 5 allocation schemes and using 11 task mapping techniques, providing 55 executions for the same application. The task mapping techniques include all algorithms introduced in Section 4.3, all algorithms in LibTopoMap (Hoefler and Snir, 2011), and the system defaults. During the experiments, we collect time spent for communication and computation, number of bytes sent between machine

[4]Cray Cielo supercomputer: `http://www.lanl.gov/projects/cielo/index.php`

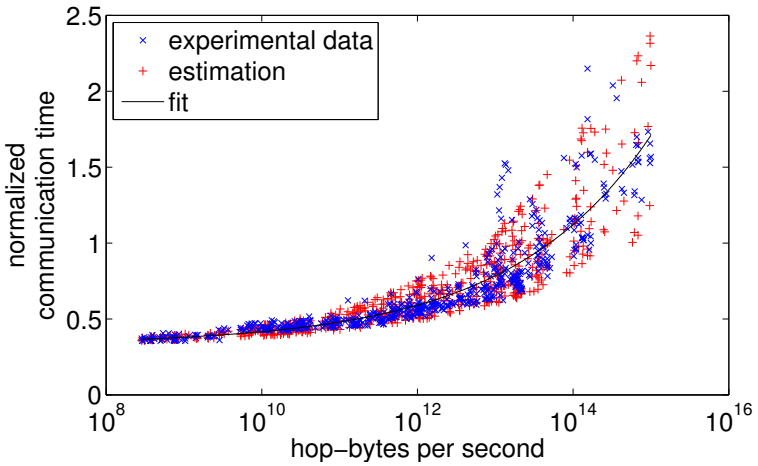


Figure 4·18: Relationship between communication and hop-bytes. The communication time represents the time spent in communication when the computation time equals 1.

nodes, maximum congestion, and message hop count.

We examine several metrics that have been associated with the communication overhead in the literature. These are maximum dilation (i.e., the maximum network hop distance a message travels), average dilation, average hops per byte, maximum network congestion, and hop-bytes as defined in Equation 4.16.

$$\textit{hop-bytes} = \sum_{i}^{\textit{messages}} \textit{hop-distance}_i \cdot \textit{bytes}_i \tag{4.16}$$

Hop-bytes represents the total communication volume in the network. Our experimental results show high correlation between hop-bytes and communication time, as shown in Figure 4·18. Based on the experimental data, we formulate the execution time of a job as follows:

$$T_{exec} = (1 + 0.0019 \cdot \textit{hop-bytes}^{0.16+\tau}) \cdot T_{base} \tag{4.17}$$

where T_{base} is the execution time without considering the overhead introduced by topology mapping decision, and τ is a uniformly distributed random number between -0.013 and 0.013 that represents the variation in the experimental results. As the

application tasks run on individual processing cores without consolidation, we assume that the computation time is independent of the mapping decision.

Results

We start by analyzing the task mappers independently from the allocation algorithms. For this purpose, we run each application in an empty machine using the same allocation. We use best-fit allocator with Hilbert curves to provide a fair comparison for RCM, which maps the tasks in-order to the allocated nodes. For each application, we use the smallest empty cubical machine with single core per node, where the machine dimensions are selected as powers of 2 so that efficient Hilbert curves can be generated. We use *PaCMap* only as a task mapper in this analysis.

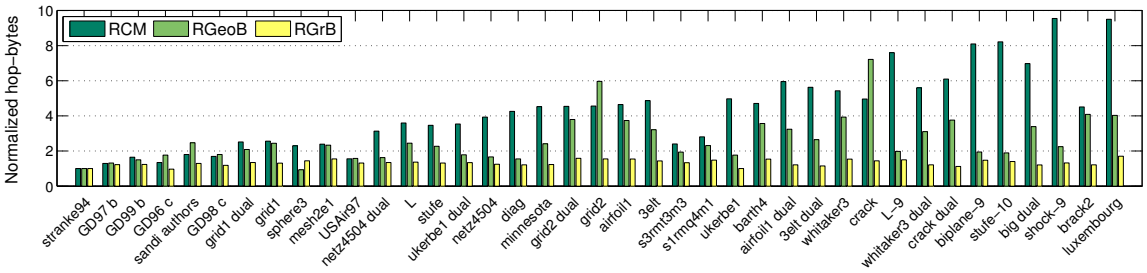


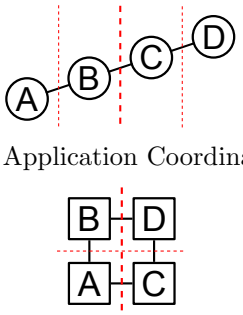
Figure 4·19: Hop-bytes comparison of task mappers for all applications in our input set. The values are normalized to the hop-bytes resulting from *PaCMap*.

Figure 4·19 shows the resulting hop-bytes for all applications in our input set. The results are normalized with respect to the hop-bytes of *PaCMap*. The applications in the x-axis are ordered from smallest with 10 tasks (*stranke94*) to the largest with 115K tasks (*luxembourg*). Note that all cases use the same allocator so that the benefits of combined allocation and mapping is not exploited.

In the figure, we observe that the RCM performance decreases as the application size increases. The first reason for this scalability problem is that RCM ignores the network links that are not along the curve. Second, the performance of RCM

depends on the correlation between the average and the maximum distance in the communication matrix, which typically decreases with the application size. For the application *diag*, for example, RCM increases the average communication distance more than 4 times of the original matrix while reducing the maximum distance.

The performance of the RGeoB depends on (1) how well the geometry represents actual communication, and (2) the similarity between the coordinates and the allocated node structure. For the latter, consider the example given in Figure 4·20. Because only coordinates are used during bisectioning, RGeoB places the tasks B and C far from each other and increases the message hop distance. This problem also occurs when the tasks have 2D coordinates but are mapped into a 3D topology. It has a significant impact on the communication overhead in larger scale, where thousands of tasks are placed far apart (e.g. *crack* in Figure 4·19). To avoid this problem in torus/mesh networks for structured communication, researchers have introduced methods such as folding (Bhatelé et al., 2010). However, no solution exists on this issue for arbitrary communication patterns.



(a) Application Coordinates

(b) Network Graph

Figure 4·20: Mismatching coordinates and allocation. The dashed lines represent the bisection cuts.

RGrB performance the best among our baselines. One weakness of RGrB is that the heuristics used for graph bisectioning, which is also an NP-hard problem, perform poorly with all-to-all graphs. Additionally, RGrB can lead to less efficient solutions depending on the communication and machine graphs as well as how the recursive branches of these two graphs are mapped to each other (Pellegrini and Roman, 1996a).

We have performed the same analysis using average message hop distance and per-job congestion metrics and verified that our algorithm do not worsen another network metric. The performance of *PaCMap* is reduced as the allocation is not

adapted to the communication graph during the aboves analyses. We compare our technique with others in the following section.

Analysis with Workload Traces

Analyzing how the allocation decision affects the task mapping performance in a machine with non-contiguous allocation requires an already-populated machine. For this purpose, we use entire workload traces and compare the execution time of the jobs. The target HPC machine used in this analysis has 16 processing cores per node as in the real-life experiments we use for calibrations. For each input trace, the machine size is selected as the actual machine size the logs are collected from.

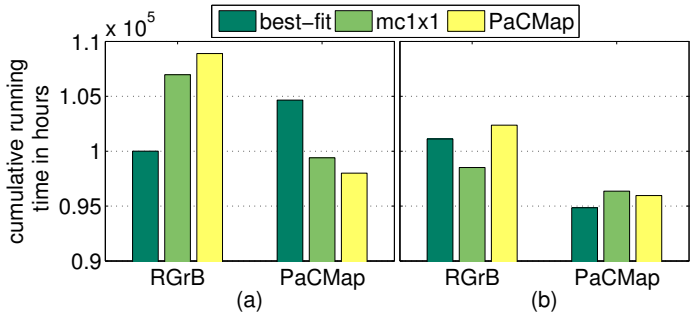
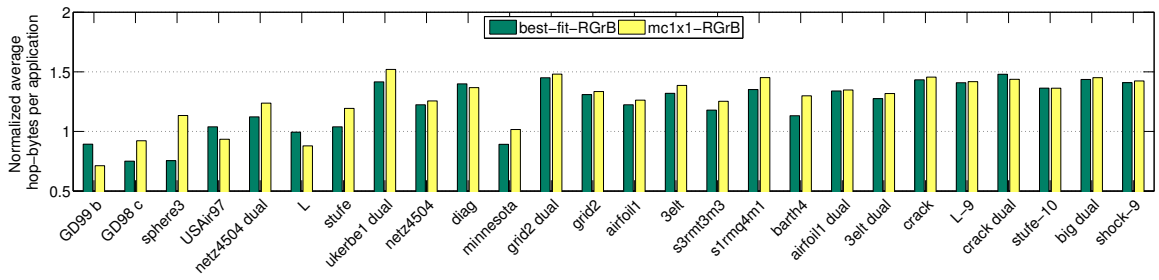


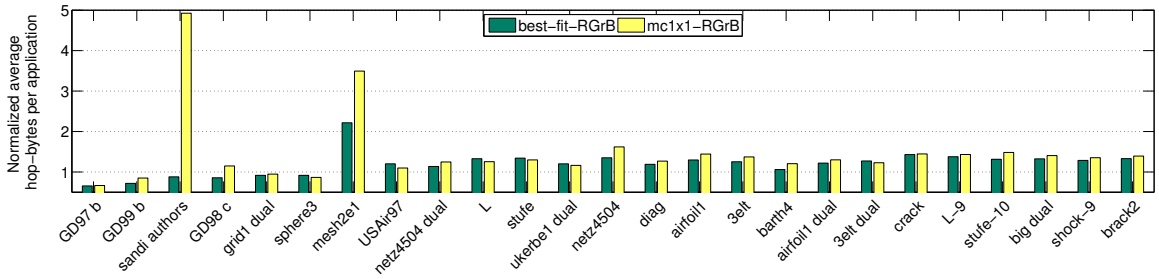
Figure 4·21: Cumulative running time of the jobs that use multiple nodes in (a) LLNL-Atlas and (b) CEA-Curie traces. The horizontal axis shows different task mappers; whereas bar colors are different allocators.

Figure 4·21 shows the cumulative execution time of all jobs that use multiple nodes (i.e., jobs with at least 16 tasks) in the traces with different allocator and task mapper pairs. Each bar group uses a different task mapper, and each bar in a group uses a different allocator. Note that *PaCMap* can be used as an allocator and/or as a task mapper.

Although intuitively clustered allocations should be more useful for RGrB than curve-following allocations, we observe that for LLNL-Atlas, RGrB leads to 7% less cumulative execution time than MC1x1 with the best-fit allocator. This is because



(a) LLNL-Atlas trace



(b) CEA-Curie trace

Figure 4·22: Average per-application hop-bytes in two workload traces with different allocator & task mapper pairs. The results are normalized with respect to *PaCMap*.

the Hilbert curves provide very high locality in this particular case. However, for CEA-Curie, the RGrB & MC1x1 pair leads to 3% shorter execution than with best-fit. The topology mapping performance not only depends on the allocator and the task mapper, but also on in which order the jobs arrive.

Complete topology mapping (both allocation and task mapping) with *PaCMap* decreases the cumulative execution time by 2% and 3% for LLNL-Atlas and CEA-Curie, respectively, compared to the best case of RGrB for each trace. Note that this reduction corresponds to 3000 hours of cumulative active node computation time in two weeks, which implies power and energy savings besides the execution time. The improvement is expected to grow with increasing application sizes based on our following analysis.

In order to verify that this difference in the execution time is not due to specific

workloads, we compare the average per-application hop-bytes in Figure 4·22. As we generate the communication patterns by binning job sizes in the traces to the closest available size in the sparse matrix collection, the traces do not contain all the applications. Thus, some applications are not present in the figures. The values in Figure 4·22 are normalized with respect to the results of *PaCMap*.

In both traces, RGrB performs better than *PaCMap* for applications with less than 1K tasks. As the application size increases, however, *PaCMap* leads to smaller hop-bytes up to 30% compared to the best case of RGrB, excluding *sandi authors* and *mesh2e1* in the CEA-Curie trace. These two applications have very large hop-bytes because they have a few instances in the trace, which are allocated poorly in this particular case.

4.3.3 Topology Mapping in Dragonfly Networks

Dragonfly (Kim et al., 2008) is a network topology that leverages high-radix routers reduce application communication overheads by constructing a constant-diameter network. This topology has been implemented in large-scale supercomputers such as Cray XK7 (Alverson et al., 2012), Cray cascade (Faanes et al., 2012), and IBM PERCS (Arimilli et al., 2010).

While recent real system experiments on dragonfly topologies argue that the impact of topology mapping is minimal on dragonflies (e.g., (Budiardja et al., 2013)), we show that the impact of topology mapping is still significant for large-scale applications in state-of-the-art HPC systems and demonstrate that topology mapping remains a major cause of performance variation.

Dragonfly Topology

A dragonfly consists of groups of routers which act as high-radix virtual routers connected to compute nodes. Figure 4·23 depicts a single dragonfly group consisting

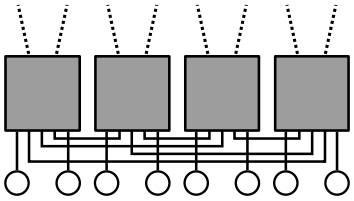


Figure 4·23: A dragonfly group with all-to-all local connections. Boxes are routers, circles are nodes, solid lines are electrical local links, and dashed lines are optical global links.

of 4 routers, where each router is connected to 2 nodes, 3 other routers within the group, and 2 routers in other groups that contain four routers and two compute nodes per router. While the intra-group connections use electrical links as in traditional HPC topologies such as torus, the inter-group connections use optical links.

The Impact of Task Mapping at Large Scale

We assess the impact of task mapping on dragonfly systems using the Trinity supercomputer[5], two mini-applications developed by the Department of Energy community for performance evaluation in HPC systems, and three task mapping algorithms.

Target System: Trinity is a 8.1PFlop/s, 4.2MW supercomputer with a Cray XC30 architecture. It consists of over 9000 compute nodes with 32 processing cores per node, and is the tenth most powerful supercomputer in the June 2017 Top500 list[6]. Trinity uses a dragonfly topology with 26 groups, each with 384 nodes. The nodes within a group are connected to each other with flattened butterfly topology, whereas the groups are connected to each other with all-to-all topology..

Applications: We use the Mantevo benchmark suite (Heroux et al., 2009), which is designed for performance evaluation and network scaling studies and represents the computational cores of various HPC applications. We select two mini-applications

[5]Trinity supercomputer: `http://www.lanl.gov/projects/trinity/`
[6]Top 500 Supercomputer Sites: http://www.top500.org/

that are sensitive to task mapping in torus networks: MiniGhost, which represents modeling of complex multi-dimensional problems such as large deformations and/or strong shocks, and MiniMD, which is a proxy for the force computations in molecular dynamics applications. While both applications focuses on a 3D problem with nearest-neighbor communication, MiniMD also uses `MPI_Allreduce` for FFT calculations and sends messages to the MPI ranks that are not nearest neighbors but a few hops away from the source rank in the problem geometry, in a single time step.

Task Mapping Algorithms: We use the following task mapping algorithms:

- *In-order* is the default task mapper. It assigns the MPI ranks in-order to the cores of the allocated compute nodes, which are sorted by the allocation order.
- *Random* randomly assigns the MPI ranks to cores.
- *RCB* (Deveci et al., 2014) recursively splits the allocated system nodes as well as the MPI ranks of a given 3-D application into equal halves based on the x, y, and z coordinates of the nodes/ranks. In both network space and the application space, the split is performed on the longest dimension. At the end of recursive splits, the remaining rank is mapped to the remaining core. RCB is originally built for 3-D mesh topologies. To adapt this algorithm to dragonfly, we use the group number of a compute node as its z-coordinate, and the row and column numbers within the group as the x- and y-coordinates of that node. While our adapted version loses some information on the exact dragonfly topology such as the global link locations, it can reduce the distance messages must travel.

Experiments Conducted: We run the selected applications on 1, 2, 4, ..., 4096 nodes. For each application size (i.e., number of nodes), we repeat our experiments 8 times using different sets of nodes that are assigned by the system software depending

on the system state. For each node allocation, we fully utilize the given nodes by running one thread on each core using 6 different openMP settings, where we use 1, 2, 4, ..., 32 threads per MPI rank. For each OpenMP setting, we re-run the same application using different task mappers.

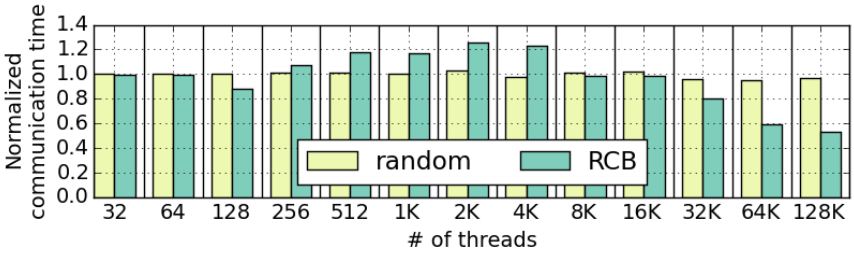
Results

Figure 4·24 shows the time spent during MPI communication as reported by the applications. For each set of parameters, the communication time with the task mappers are normalized with respect to the in-order (default) mapper. To eliminate the impact of node allocation on the results, we show the median communication time (out of the 8 runs). In our experiments, the mapping overhead is negligible compared to application communication times.

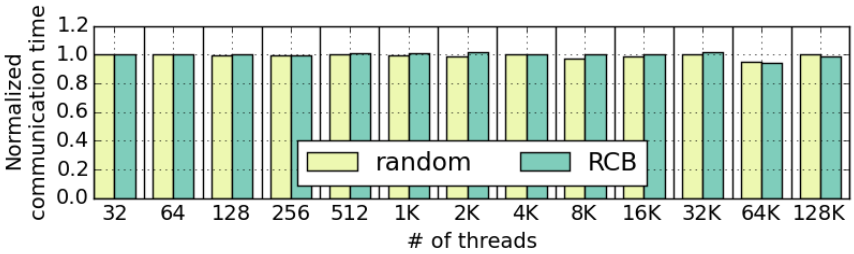
Our results demonstrate that task mapping can change the communication time significantly when running parallel programs on dragonfly networks. In Fig. 4·24(a), RCB mapper reduces the communication time by 47% when MiniGhost is running on 128K threads with 1 thread per rank, whereas in Fig. 4·24(c), random mapper increases the communication time by 210% when MiniMD is running on 64K threads with the OpenMP setting as 1 thread per rank.

Because the two applications differ in their communication patterns, they benefit from different task mapping strategies. In Fig. 4·24(a) when running MiniGhost, RCB is up to 47% better than in-order mapper; meanwhile in Fig. 4·24(c) when running MiniMD, in-order mapper is always better than the others.

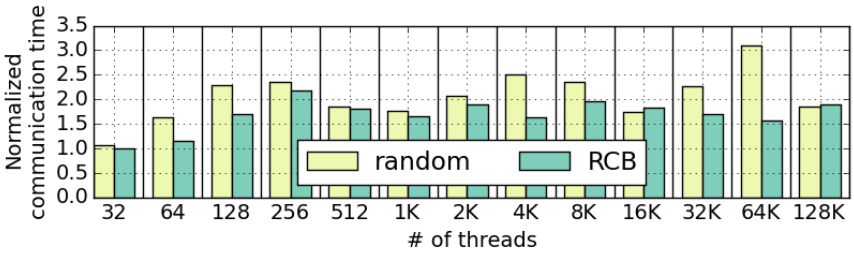
We find that the task mapper performance is also sensitive to application scale. Along the horizontal axis in Fig. 4·24(a), we see that the normalized communication time of RCB mapper varies more than 134%, from 0.53 to 1.25. RCB tends to perform worse than in-order mapper with less than 8K threads (corresponds to 256 nodes in our system), whereas with more than 8K threads, RCB turns out to be the best choice



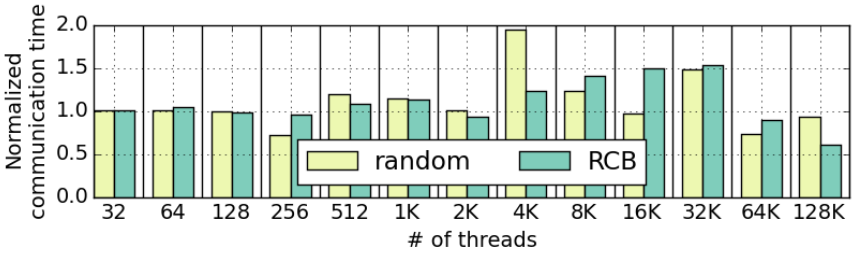
(a) MiniGhost with 1 thread per rank



(b) MiniGhost with 32 threads per rank



(c) MiniMD with 1 thread per rank



(d) MiniMD with 32 threads per rank

Figure 4·24: Application communication time normalized with respect to the in-order task mapper. The results show that task mapping affects the communication overhead significantly.

among the three task mappers. These results show that conclusions from small-scale experiments may not be extended to large-scale experiments.

Another factor that affects the impact of task mapping is the OpenMP settings (i.e., number of threads per rank). While the performance difference between task mappers is less than 7% in Fig. 4·24(b), with a different threads-per-rank setting in Fig. 4·24(a), the performance difference reaches up to 47%.

4.3.4 Summary

Topology mapping of HPC applications have a significant impact on the execution time, especially at the macro-scale, where the applications use thousands of machine nodes in parallel. We have designed a novel algorithm, *PaCMap*, that simultaneously applies job allocation and task mapping to minimize the execution time of applications with unstructured communication patterns. *PaCMap* is applicable to any network topology, and it can be also used as a mere allocator or task mapper. Furthermore, we have developed an execution time estimation model based on real life experiments on a Cray XE6, which is used to calibrate the simulations of long HPC workload traces. Our results show that *PaCMap* reduces application network traffic volume in terms of hop-bytes by up to 30% compared to state-of-the-art approaches in HPC machines with non-contiguous allocation.

In addition to presenting *PaCMap*, using large-scale experiments on the Trinity supercomputer, we have demonstrated that topology mapping remains an important factor for HPC application performance in exascale computing. Our experiments have shown that topology mapping is responsible for up to a 2X increase in communication times even in HPC systems that use state-of-the-art network infrastructures. We believe that topology mapping driven by application communication patterns can be targeted for novel network topologies in the future to further reduce application execution times.

Chapter 5

Conclusions and Future Directions

5.1 Summary of Major Contributions

Traditional problem diagnosis and system management techniques that are based on heuristics and manual labor are insufficient to achieve high efficiency in today's increasingly complex HPC and cloud data centers. This thesis has claimed that to improve robustness and efficiency of large-scale computing systems, significantly higher levels of automated support than what is available in today's systems are needed, and this automation should leverage the data continuously collected from various system layers. To this end, this thesis has provided frameworks to automatically diagnose performance and software configuration problems, and data-driven dynamic system management policies to improve the efficiency of large-scale computing systems.

This thesis has addressed automated problem diagnosis in two fronts: identification of performance anomalies and software configuration analytics. We first introduced a machine learning based online anomaly diagnosis framework that leverages the resource usage and performance data that are already being collected in large-scale systems to identify the signatures of previously-observed performance anomalies. We evaluated our framework using experiments on real HPC clusters and demonstrated that our approach effectively identifies 98% of the synthetic anomalies while leading to only 0.08% false anomaly alarms, consistently outperforming the state-of-the-art on anomaly diagnosis. We demonstrated that our approach can learn the anomaly signatures independent of the executing applications, enabling anomaly diagnosis even

when running applications that are not seen during training. This type of approach can be used in the future to enable higher levels of automation such as automated mitigation of anomalies through system management.

We have also addressed a different type of major large-scale system anomaly: software misconfiguration. As errors in software configurations are prominent in the cloud where developers do not need expertise on third-party applications to deploy services using these applications, we have focused on cloud platforms. We have designed *ConfEx*, a framework to discover and analyze text-based software configurations in the cloud. *ConfEx* achieves over 98% precision and recall on identifying configuration files and resolves the *ambiguation* in configuration parser outputs. Our approach enables a new source of information, software configurations, to improve the level of automation in cloud platforms. As an example for such automation, we have demonstrated two use cases of *ConfEx* with existing configuration analysis techniques that are designed for key-value pairs: outlier analysis and syntactic validation.

This thesis also proposed data-driven system management techniques to improve energy efficiency, particularly using power, thermal, and workload management. For cluster-level power management, we have proposed *CoolBudget*, a novel data center power budgeting policy that optimally partitions the given power limit across the servers and cooling units. Our policy uses temperature and power models to safely collapse the thermal headroom margins in the servers, and improves the overall throughput without an unfair performance degradation across the jobs. Experimental evaluation based on real servers shows that our policy achieves 21% higher throughput compared to the other state-of-the-art power budgeting techniques. The stable and predictable temperatures leveraged by *CoolBudget* are achieved using our leakage-aware proactive server fan control policy. This policy reduces the sum of server leakage power and cooling power by up to 6% compared to other policies without

incurring any performance overhead on the executing workloads.

This thesis also demonstrated via experiments on real HPC systems that topology mapping causes up to 50% variation in application running times when the application size reaches thousands of nodes. To mitigate this variation and to reduce application communication overhead, we proposed *PaCMap*, a topology mapping algorithm that unifies system-driven node allocation and application-driven task mapping. *PaCMap* finds topology mappings that are better-suited for the application communication patterns compared to existing approaches, reducing *hop-bytes* by up to 30%.

In summary, the proposed techniques in this thesis significantly improve the robustness and efficiency of large-scale computing systems using intelligent approaches based on data collected from various system layers. Based on our results, we believe that the automated problem diagnosis and data-driven system management can be combined in the future to enable automated mitigation of system anomalies in an energy-efficient way, paving the way to exascale computing. Next, we discuss open research problems and specific directions that could immediately follow this thesis.

5.2 Future Research Directions

5.2.1 Diagnosing Performance Anomalies in Production HPC Systems

The increasing size and complexity of large-scale computing systems such as supercomputers and cloud data centers necessitate automated anomaly diagnosis to maintain robust operation. In this thesis, we have proposed using machine learning algorithms for automated and robust anomaly diagnosis with application resource usage data, which is already being monitored in supercomputers. Due to the lack of comprehensive and labeled resource usage data of anomalous system behavior, we have evaluated our framework with synthetic anomaly injection and controlled experiments.

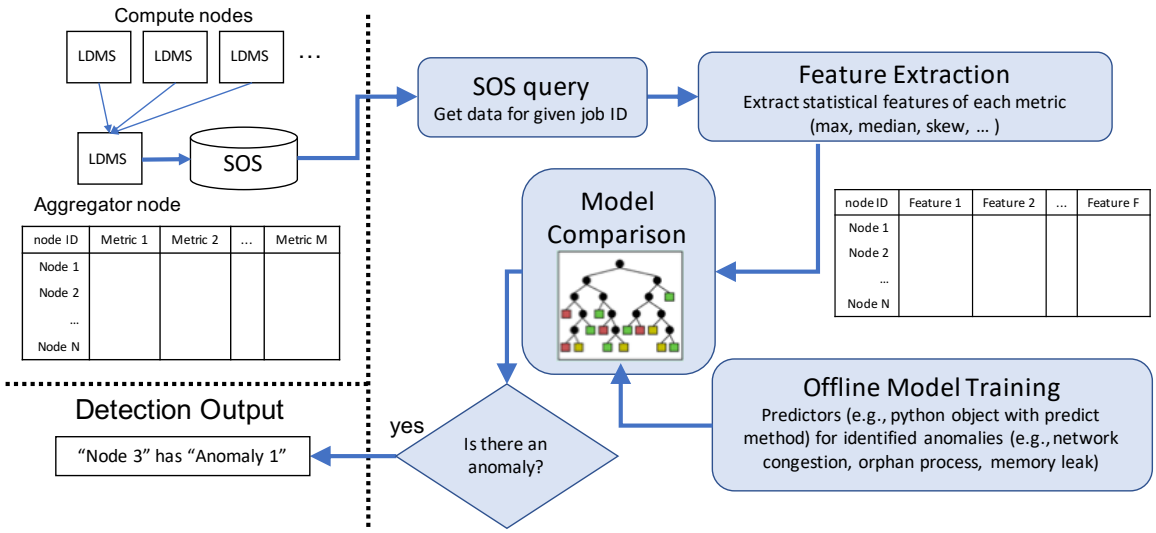


Figure 5·1: The integration automated our anomaly diagnosis framework with LDMS.

To address the lack of comprehensive and labeled data, a currently-ongoing stage of our research is the integration of our framework with the Light-weight Distributed Metric Service (Agelastos et al., 2014), which will enable us to evaluate our approach on production systems. Figure 5·1 depicts the initial design for this integration. The data collected by LDMS from indivudual compute nodes are aggregated into the aggregator node, and written into Scalable Object Store[1] (SOS), which is designed Open Grid Computing for continuous monitoring of HPC systems. After the resource usage data of a job is fed into SOS, our framework queries this data, extracts features, and uses offline-trained machine learning models to detect and diagnose anomalies. We envision that this integration would both enable the detection of known anomaly signatures in production systems and help with the labeling of anomalies via user feedback.

Another open problem is anomaly diagnosis without learning the exact signatures of anomalies. By combining our online anomaly diagnosis approach with existing anomaly detection (e.g., (Klinkenberg et al., 2017)) or log analysis (e.g., (Gainaru

[1]Scalable Object Store: `https://github.com/ovis-hpc/ovis`

et al., 2012)) approaches, one can identify the subsystem (e.g., memory, application, network interface) that causes the anomaly, which would be very helpful for system administrators to maintain robust operation.

5.2.2 Configuration Analytics

As the cloud infrastructure becomes more affordable and the open-source software becomes more prevalent, it gets easier for developers to deliver new cloud services without necessarily having the expertise needed for configuration tuning. As a result, configuration errors become increasingly widespread. While some researchers aim to overcome such errors by simplifying configurations (Xu et al., 2015; Tang et al., 2015), current trends in the open-source software configurations demonstrate that developers tend to prioritize software functionality over configuration usability, necessitating other approaches to avoid misconfigurations.

Our framework, *ConfEx*, enables configuration analytics using image repositories and third-party cloud instances. One gap in our configuration file discovery approach is that it can include the configuration files that are not actively used by applications in analysis, raising false positives for the errors existing in these inactive files. To address this issue, we are designing an active configuration file identification method that tracks system calls to filter out passive files during application initialization.

Another open research direction is finding configuration-related problems via correlating application resource usage patterns with configurations. One way of extracting the correlation between application behavior and configurations is to extend our performance anomaly diagnosis framework (see Section 3.1) to analyze configurations. By including configuration-related information during training, our framework can help understand what kind of application behavior is expected under given configuration settings. For this purpose, the learning algorithms in our anomaly diagnosis framework need to be modified to support semi-supervised or unsupervised learning.

5.2.3 Data-driven System Management

Power Management

As it is hard to conduct experiments on real data centers due to security and performance concerns, the research on data center power management has so far primarily relied on simulations (Kong and Liu, 2015). While some of these simulations use system models that need to be tuned for modern data center hardware and technologies (Moore et al., 2005), others use custom models based on technical documentations and published data (Zapater et al., 2015b) or conduct experiments on small-scale data centers with tens of servers. However, these approaches have not been evaluated on large data centers with hundreds or thousands of servers. Research-oriented data centers such as the Massachusetts Green High Performance Computing Center[2] can be leveraged in the future to explore new challenges in managing the power consumption of real large-scale computing systems.

Workload Management in HPC Systems

Workload management in HPC systems is a widely studied topic. The technological advances and increasing application and system sizes, however, create new challenges and inefficiencies in the HPC systems.

As discussed in Section 4.3.3, topology mapping has high impact on performance even in emerging networks such as dragonflies. However, efficient topology mapping has not yet been investigated thoroughly for these state-of-the-art networks. In our recent work (Zhang et al., 2018), we developed a job-size aware policy for machines with dragonfly topologies and demonstrated that by being aware of the size of the submitted jobs alone can reduce the application communication overhead by 16% on the average.

[2] `https://www.mghpcc.org/`

There is more information available on the communication patterns of HPC applications that can be exploited to further reduce communication overheads. One open problem is to use the dynamic communication traces of these applications during topology mapping. In prior work, topology mapping only considers the average communication traffic of an application. However, knowledge of how the communication of an application changes over the duration of its execution can help finding better mapping solutions that lowers network congestion. The information on an application's dynamic communication can be extended further by considering how the application interacts with the file system for data I/O as well as for checkpointing.

Another open problem on topology mapping is combining the scheduling and allocation stages of HPC workload management. For example, if the topology mapper had information that the future jobs in the queue require large chunks of contiguous nodes, it could leave empty blocks of nodes in the machine even if that means a less efficient mapping for the currently running jobs. Similarly, the scheduler could tune its backfilling algorithm with information on the topology of the available nodes.

References

Aceto, G., Botta, A., De Donato, W., and Pescapè, A. (2013). Cloud monitoring: A survey. *Computer Networks*, 57(9):2093–2115.

Adhianto, L., Banerjee, S., Fagan, M., Krentel, M., Marin, G., Mellor-Crummey, J., and Tallent, N. R. (2010). Hpctoolkit: Tools for performance analysis of optimized parallel programs. *Concurrency Computation Practice and Experience*, 22(6):685–701.

Agelastos, A., Allan, B., Brandt, J., Cassella, P., Enos, J., Fullop, J., Gentile, A., Monk, S., Naksinehaboon, N., Ogden, J., Rajan, M., Showerman, M., Stevenson, J., Taerat, N., and Tucker, T. (2014). The lightweight distributed metric service: A scalable infrastructure for continuous monitoring of large scale computing systems and applications. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 154–165.

Agelastos, A., Allan, B., Brandt, J., Gentile, A., Lefantzi, S., Monk, S., Ogden, J., Rajan, M., and Stevenson, J. (2015). Toward rapid understanding of production HPC applications and systems. In *IEEE International Conference on Cluster Computing*, pages 464–473.

Ahad, R., Chan, E., and Santos, A. (2015). Toward autonomic cloud: Automatic anomaly detection and resolution. *Proceedings - 2015 International Conference on Cloud and Autonomic Computing, ICCAC 2015*, pages 200–203.

Al-Roomi, M., Al-Ebrahim, S., Buqrais, S., and Ahmad, I. (2013). Cloud computing pricing models: a survey. *International Journal of Grid and Distributed Computing*, 6(5):93–106.

Albing, C. et al. (2011). Scalable node allocation for improved performance in regular and anisotropic 3d torus supercomputers. In *Proceedings of the European MPI Users' Group Conference on Recent Advances in the Message Passing Interface*, EuroMPI, pages 61–70.

Alverson, B., Froese, E., Kaplan, L., and Roweth, D. (2012). Cray XC series network. Technical report, Cray Inc. White paper.

Andrae, A. S. and Edler, T. (2015). On global electricity usage of communication technology: trends to 2030. *Challenges*, 6(1):117–157.

Arimilli, B., Arimilli, R., Chung, V., Clark, S., Denzel, W., Drerup, B., Hoefler, T., Joyner, J., Lewis, J., Li, J., Ni, N., and Rajamony, R. (2010). The PERCS high-performance interconnect. In *IEEE Symposium on High Performance Interconnects*.

Arzani, B. and Outhred, G. (2016). Taking the blame game out of data centers operations with netpoirot. *ACM Conference of the Special Interest Group on Data Communication (SIGCOMM)*, pages 440–453.

Attariyan, M., Chow, M., and Flinn, J. (2012). X-ray: Automating root-cause diagnosis of performance anomalies in production software. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI, pages 307–320.

Auble, D. and Christiansen, B. (2014). SLURM workload manager overview. Poster presented at the ACM/IEEE International Conference on High Performance Computing, Networking, Storage and Analysis (SC).

Ayoub, R., Nath, R., and Rosing, T. (2012). JETC: Joint energy thermal and cooling management for memory and CPU subsystems in servers. In *IEEE Symposium on High Performance Computer Architecture (HPCA)*, pages 1–12.

Bailey, D. H., Barszcz, E., Barton, J. T., Browning, D. S., Carter, R. L., Dagum, L., Fatoohi, R. A., Frederickson, P. O., Lasinski, T. A., Schreiber, R. S., Simon, H. D., Venkatakrishnan, V., and Weeratunga, S. K. (1991). The NAS parallel benchmarks - summary and preliminary results. In *Proceedings of the ACM/IEEE Conference on Supercomputing*, pages 158–165.

Balzuweit, E., Bunde, D. P., Leung, V. J., Finley, A., and Lee, A. C. (2016). Local search to improve coordinate-based task mapping. *Parallel Computing*, 51:67–78.

Barrett, R. F., Vaughan, C. T., and Heroux, M. A. (2012). Minighost: a miniapp for exploring boundary exchange strategies using stencil computations in scientific parallel computing. Technical report, Sandia National Laboratories, Albuquerque, NM.

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(3):1–154.

Baset, S., Suneja, S., Bila, N., Tuncer, O., , and Isci, C. (2017). Usable declarative configuration specification and validation for applications, systems, and cloud. In *Proceedings of the Industrial Track of the International Middleware Conference*.

Behrang, F., Cohen, M. B., and Orso, A. (2015). Users beware: Preference inconsistencies ahead. In *Proceedings of the 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE, pages 295–306.

Bender, M. A., Bunde, D. P., Demaine, E. D., Fekete, S. P., Leung, V. J., Meijer, H., and Phillips, C. A. (2008). Communication-aware processor allocation for supercomputers: Finding point sets of small average distance. *Algorithmica*, pages 279–298.

Benjamini, Y. and Yekutieli, D. (2001). The control of the false discovery rate in multiple testing under dependency. *Annals of Statistics*, 29:1165–1188.

Bhatelé, A., Gupta, G. R., Kalé, L. V., and Chung, I. H. (2010). Automated mapping of regular communication graphs on mesh interconnects. In *International Conference on High Performance Computing (HiPC)*, pages 1–10.

Bhatelé, A. and Kalé, L. (2011). Heuristic-based techniques for mapping irregular communication graphs to mesh topologies. In *IEEE International Conference on High Performance Computing and Communications (HPCC)*, pages 765–771.

Bhatelé, A., Mohror, K., Langer, S. H., and Isaacs, K. E. (2013). There goes the neighborhood: Performance degradation due to nearby jobs. In *Proceedings of International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 41:1–41:12.

Bianchini, R. (2012). Leveraging renewable energy in data centers: present and future. In *International symposium on High-Performance Parallel and Distributed Computing*, pages 135–136.

Bienia, C. (2011). *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, Princeton, NJ, USA.

Bodik, P., Goldszmidt, M., Fox, A., Woodard, D. B., and Andersen, H. (2010). Fingerprinting the datacenter: Automated classification of performance crises. In *Proceedings of the 5th European Conference on Computer Systems*, pages 111–124.

Brandt, J., Chen, F., De Sapio, V., Gentile, A., Mayo, J., Pebay, P., Roe, D., Thompson, D., and Wong, M. (2010). Quantifying effectiveness of failure prediction and response in HPC systems: Methodology and example. In *Proceedings of the International Conference on Dependable Systems and Networks Workshops*, pages 2–7.

Brandt, J., DeBonis, D., Gentile, A., Lujan, J., Martin, C., Martinez, D., Olivier, S., Pedretti, K., Taerat, N., and Velarde, R. (2015). Enabling advanced operational analysis through multi-subsystem data integration on trinity. *Proc. Cray Users Group*.

Brandt, J., Gentile, A., Mayo, J., Pébay, P., Roe, D., Thompson, D., and Wong, M. (2009). Methodologies for advance warning of compute cluster problems via statistical analysis: A case study. In *Proceedings of the 2009 Workshop on Resiliency in High Performance*, pages 7–14.

Budiardja, R. D., Crosby, L., and You, H. (2013). Effect of rank placement on cray xc30 communication cost. In *The Cray User Group Meeting*.

Burgess, M. and Ralston, R. (1997). Distributed resource administration using cfengine. *Software: practice and experience*, 27(9):1083–1101.

Chan, C. S., Jin, Y., Wu, Y.-K., Gross, K., Vaidyanathan, K., and Rosing, T. (2012). Fan-speed-aware scheduling of data intensive jobs. In *Proceedings of the International Symposium on Low Power Electronics and Design*, pages 409–414.

Chaudhry, M. T., Ling, T. C., Manzoor, A., Hussain, S. A., and Kim, J. (2015). Thermal-aware scheduling in green data centers. *ACM Computing Surveys*, 47(3):39:1–39:48.

Chen, H., Caramanis, M. C., and Coskun, A. K. (2014a). Reducing the data center electricity costs through participation in smart grid programs. In *International Green Computing Conference (IGCC)*, pages 1–10.

Chen, M., Mao, S., and Liu, Y. (2014b). Big data: A survey. *Mobile networks and applications*, 19(2):171–209.

Chen, W., Wu, H., Wei, J., Zhong, H., and Huang, T. (2016). Determine configuration entry correlations for web application systems. In *IEEE Annual Computer Software and Applications Conference (COMPSAC)*, volume 1, pages 42–52.

Chen, X., He, X., Guo, H., and Wang, Y. (2011). Design and evaluation of an online anomaly detector for distributed storage systems. *Journal of Software*, 6(12 SPEC. ISSUE):2379–2390.

Christ, M., Kempa-Liehr, A. W., and Feindt, M. (2016). Distributed and parallel time series feature extraction for industrial big data applications. *arXiv preprint arXiv:1610.07717*.

Cisco (2017). Cisco bug: Csctf52095 - manually flushing os cache during load impacts server. `https://goo.gl/375oDm`.

Cochran, R., Hankendi, C., Coskun, A. K., and Reda, S. (2011). Pack & cap: adaptive dvfs and thread packing under power caps. In *Proceedings of the IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 175–185.

Comon, P. (1994). Independent component analysis, a new concept? *Signal processing*, 36(3):287–314.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2001). *Introduction to Algorithms*. MIT Press and McGraw-Hill, 2nd edition.

Cuthill, E. and McKee, J. (1969). Reducing the bandwidth of sparse symmetric matrices. In *Proceedings of the National Conference*, ACM, pages 157–172.

Dalmazo, B. L., Vilela, J. P., Simoes, P., and Curado, M. (2016). Expedite feature extraction for enhanced cloud anomaly detection. *Proceedings of the IEEE/IFIP Network Operations and Management Symposium (NOMS)*, pages 1215–1220.

Davis, T. A. and Hu, Y. (2011). The university of florida sparse matrix collection. *ACM Transactions on Mathematical Software (TOMS)*, 38(1):1:1–1:25.

Dayarathna, M., Wen, Y., and Fan, R. (2016). Data center energy consumption modeling: A survey. *IEEE Communications Surveys Tutorials*, 18(1):732–794.

Dean, J. and Barroso, L. A. (2013). The tail at scale. *Communications of the ACM*, 56(2):74–80.

Depoorter, V., Oró, E., and Salom, J. (2015). The location as an energy efficiency and renewable energy supply measure for data centres in europe. *Applied Energy*, 140:338–349.

Desai, N. (2005). Bcfg2: A pay as you go approach to configuration complexity. *Australian Unix Users Group (AUUG2005)*, 10.

Deveci, M., Rajamanickam, S., Leung, V. J., Pedretti, K., Olivier, S. L., Bunde, D. P., atalyrek, U. V., and Devine, K. (2014). Exploiting geometric partitioning in task mapping for parallel computers. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 27–36.

Dongarra, J., Meuer, M., Simon, H., and Strohmaier, E. (2017). Top500 supercomputer ranking. `https://www.top500.org/lists/2017/11/`.

Dorier, M., Antoniu, G., Ross, R., Kimpe, D., and Ibrahim, S. (2014). Calciom: Mitigating I/O interference in HPC systems through cross-application coordination. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 155–164.

Egwutuoha, I. P., Levy, D., Selic, B., and Chen, S. (2013). A survey of fault tolerance mechanisms and checkpoint/restart implementations for high performance computing systems. *The Journal of Supercomputing*, 65(3):1302–1326.

Faanes, G., Bataineh, A., Roweth, D., Court, T., Froese, E., Alverson, B., Johnson, T., Kopnick, J., Higgins, M., and Reinhard, J. (2012). Cray cascade: A scalable hpc system based on a dragonfly network. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 1–9.

Feitelson, D. G., Rudolph, L., Schwiegelshohn, U., Sevcik, K. C., and Wong, P. (1997). Theory and practice in parallel job scheduling. In *Workshop on Job Scheduling Strategies for Parallel Processing*, pages 1–34.

Feitelson, D. G., Tsafrir, D., and Krakov, D. (2014). Experience with using the parallel workloads archive. *Journal of Parallel and Distributed Computing*, 74(10):2967 – 2982.

Gainaru, A., Cappello, F., Snir, M., and Kramer, W. (2012). Fault prediction under the microscope: A closer look into HPC systems. In *Proceedings of International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 77:1–77:11.

Guan, Q., Fu, S., De Bardeleben, N., and Blanchard, S. (2013). Exploring time and frequency domains for accurate and automated anomaly detection in cloud computing systems. *Proceedings of IEEE Pacific Rim International Symposium on Dependable Computing, PRDC*, pages 196–205.

Gurumdimma, N., Jhumka, A., Liakata, M., Chuah, E., and Browne, J. (2016). CRUDE: Combining resource usage data and error logs for accurate error detection in large-scale distributed systems. *IEEE Symposium on Reliable Distributed Systems (SRDS)*.

Han, X. and Joshi, Y. (2012). Energy reduction in server cooling via real time thermal control. In *IEEE Semiconductor Thermal Measurement and Management Symposium (SEMI-THERM)*, pages 20–27.

Hankendi, C., Reda, S., and Coskun, A. K. (2013). vcap: Adaptive power capping for virtualized servers. In *IEEE International Symposium on Low Power Electronics and Design (ISLPED)*, pages 415–420.

He, L., Jarvis, S. A., Spooner, D. P., Chen, X., and Nudd, G. R. (2004). Dynamic scheduling of parallel jobs with qos demands in multiclusters and grids. In *IEEE/ACM International Workshop on Grid Computing*, pages 402–409.

Henning, J. L. (2006). Spec cpu2006 benchmark descriptions. *ACM SIGARCH Computer Architecture News*, 34(4):1–17.

Heroux, M. A., Doerfler, D. W., Crozier, P. S., Willenbring, J. M., Edwards, H. C., Williams, A., Rajan, M., Keiter, E. R., Thornquist, H. K., and Numrich, R. W. (2009). Improving performance via mini-applications. Technical report, Sandia National Laboratories, Albuquerque, NM.

Hertel, E. S., Bell, R. L., Elrick, M. G., Farnsworth, A. V., Kerley, G. I., McGlaun, J. M., Petney, S. V., Silling, S. A., Taylor, P. A., and Yarrington, L. (1993). CTH:

A software family for multi-dimensional shock physics analysis. In *Proceedings of the International Symposium on Shock Waves*, pages 377–382.

Hochstein, L. (2014). *Ansible: Up and Running: Automating Configuration Management and Deployment the Easy Way*. O'Reilly Media, Inc.

Hoefler, T. and Snir, M. (2011). Generic topology mapping strategies for large-scale parallel architectures. In *Proceedings of the International Conference on Supercomputing (ISC)*, pages 75–84.

Holtgrewe, M., Sanders, P., and Schulz, C. (2010). Engineering a scalable high quality graph partitioner. In *IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*, pages 1–12.

Huang, P., Bolosky, W. J., Singh, A., and Zhou, Y. (2015). Confvalley: A systematic configuration validation framework for cloud services. In *Proceedings of the Tenth European Conference on Computer Systems*, EuroSys '15, pages 19:1–19:16.

Huang, W., Allen-Ware, M., Carter, J. B., Elnozahy, E., Hamann, H., Keller, T., Lefurgy, C., Li, J., Rajamani, K., and Rubio, J. (2011). TAPO: Thermal-aware power optimization techniques for servers and data centers. In *International Green Computing Conference and Workshops (IGCC)*, pages 1–8.

Ibidunmoye, O., Hernández-Rodriguez, F., and Elmroth, E. (2015). Performance anomaly detection and bottleneck identification. *ACM Computing Surveys*, 48(1):1–35.

Ibidunmoye, O., Metsch, T., and Elmroth, E. (2016). Real-time detection of performance anomalies for cloud services. *IEEE/ACM 24th International Symposium on Quality of Service (IWQoS)*, pages 1–2.

IDC (2014). IDC finds growth, consolidation, and changing ownership patterns in worldwide datacenter forecast. Technical report, International Data Corporation. `https://www.idc.com/getdoc.jsp?containerId=prUS25237514`.

Jain, N., Bhatele, A., Ni, X., Wright, N. J., and Kale, L. V. (2014). Maximizing throughput on a dragonfly network. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 336–347.

Jayathilaka, H., Krintz, C., and Wolski, R. (2017). Performance monitoring and root cause analysis for cloud-hosted web applications. *Proceedings of the 26th International Conference on World Wide Web (WWW)*, pages 469–478.

Jin, D., Cohen, M. B., Qu, X., and Robinson, B. (2014). Preffinder: Getting the right preference in configurable software systems. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ASE, pages 151–162.

Jin, S., Zhang, Z., Chakrabarty, K., and Gu, X. (2016). Accurate anomaly detection using correlation-based time-series analysis in a core router system. *IEEE International Test Conference (ITC)*, pages 1–10.

Kachris, C. and Tomkos, I. (2012). A survey on optical interconnects for data centers. *IEEE Communications Surveys & Tutorials*, 14(4):1021–1036.

Karypis, G. and Kumar, V. (1998). Multilevelk-way partitioning scheme for irregular graphs. *Journal of Parallel and Distributed computing*, 48(1):96–129.

Kim, J., Dally, W. J., Scott, S., and Abts, D. (2008). Technology-driven, highly-scalable dragonfly topology. In *International Symposium on Computer Architecture (ISCA)*, pages 77–88.

Klinkenberg, J., Terboven, C., Lankes, S., and Muller, M. S. (2017). Data mining-based analysis of hpc center operations. *2017 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 766–773.

Kong, F. and Liu, X. (2015). A survey on green-energy-aware power management for datacenters. *ACM Computing Surveys (CSUR)*, 47(2):30.

Kreutz, D., Ramos, F. M., Verissimo, P. E., Rothenberg, C. E., Azodolmolky, S., and Uhlig, S. (2015). Software-defined networking: A comprehensive survey. *Proceedings of the IEEE*, 103(1):14–76.

Kumar, M. R. V. and Raghunathan, S. (2016). Heterogeneity and thermal aware adaptive heuristics for energy efficient consolidation of virtual machines in infrastructure clouds. *Journal of Computer and System Sciences*, 82(2):191–212.

Kunen, A., Bailey, T., and Brown, P. (2015). Kripke-a massively parallel transport mini-app. Technical report, Lawrence Livermore National Laboratory (LLNL), Livermore, CA.

Lan, Z., Zheng, Z., and Li, Y. (2010). Toward automated anomaly identification in large-scale systems. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 21(2):174–187.

Laptev, N., Amizadeh, S., and Flint, I. (2015). Generic and scalable framework for automated time-series anomaly detection. *Proceedings of the ACM International Conference on Knowledge Discovery and Data Mining (KDD)*, pages 1939–1947.

Lawson, B. G. and Smirni, E. (2002). Multiple-queue backfilling scheduling with priorities and reservations for parallel systems. In *Job Scheduling Strategies for Parallel Processing*, pages 72–87.

Leung, V. J., Arkin, E. M., Bender, M., Bunde, D., Johnston, J., Lal, A., Mitchell, J. S. B., Phillips, C., and Seiden, S. S. (2002). Processor allocation on cplant: achieving general processor locality using one-dimensional allocation strategies. In *IEEE International Conference on Cluster Computing*, pages 296–304.

Li, W., Li, S., Liao, X., Xu, X., Zhou, S., and Jia, Z. (2017). Conftest: Generating comprehensive misconfiguration for system reaction ability evaluation. In *Proceedings of the International Conference on Evaluation and Assessment in Software Engineering (EASE)*, pages 88–97.

Lim, H., Kansal, A., and Liu, J. (2011). Power budgeting for virtualized data centers. In *USENIX Annual Technical Conference (ATC)*, volume 59.

Liu, Z., Chen, Y., Bash, C., Wierman, A., Gmach, D., Wang, Z., Marwah, M., and Hyser, C. (2012). Renewable and cooling aware workload management for sustainable data centers. In *Proceedings of the ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems*, pages 175–186.

Lo, V., Windisch, K. J., Liu, W., and Nitzberg, B. (1997). Noncontiguous processor allocation algorithms for mesh-connected multicomputers. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 8(7):712–726.

Loope, J. (2011). *Managing Infrastructure with Puppet: Configuration Management at Scale*. "O'Reilly Media, Inc.".

Lopez, L. (2007). Advanced electronic prognostics through system telemetry and pattern recognition methods. *Microelectronics Reliability*, 47:1865–1873.

Lutterkort, D. (2008). Augeas–a configuration api. In *Linux Symposium, Ottawa, ON*, pages 47–56.

Mache, J., Lo, V., and Windisch, K. (1997). Minimizing message-passing contention in fragmentation-free processor allocation. In *Proceedings of the International Conference on Parallel and Distributed Computing Systems*, pages 120–124.

Massey Jr, F. J. (1951). The kolmogorov-smirnov test for goodness of fit. *Journal of the American statistical Association*, 46(253):68–78.

Miller, R. (2007). Data center cooling set points debated. *Data Center Knowledge*.

Moore, J., Chase, J., Ranganathan, P., and Sharma, R. (2005). Making scheduling "cool": Temperature-aware workload placement in data centers. In *Proceedings of the USENIX Annual Technical Conference (ATEC)*, pages 5–5.

MPI Forum (2012). MPI: A message-passing interface standart. Version 3.0. `http://www.mpi-forum.org`.

Nadi, S., Berger, T., Kästner, C., and Czarnecki, K. (2014). Mining configuration constraints: Static analyses and empirical results. In *Proceedings of the International Conference on Software Engineering*, ICSE, pages 140–151.

Nair, V., Raul, A., Khanduja, S., Sundararajan, S., Keerthi, S., Bahirwani, V., Shao, Q., Herbert, S., and Dhulipalla, S. (2015). Learning a hierarchical monitoring system for detecting and diagnosing service issues. *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 2029–2038.

Nguyen, H., Shen, Z., Tan, Y., and Gu, X. (2013). Fchain: Toward black-box online fault localization for cloud systems. *Proceedings - International Conference on Distributed Computing Systems*, pages 21–30.

O'Shea, D., Emeakaroha, V. C., Pendlebury, J., Cafferkey, N., Morrison, J. P., and Lynn, T. (2016). A wavelet-inspired anomaly detection framework for cloud platforms. *CLOSER 2016 - Proceedings of the 6th International Conference on Cloud Computing and Services Science*, 1(April).

Oxley, M. A., Jonardi, E., Pasricha, S., Maciejewski, A. A., Siegel, H. J., Burns, P. J., and Koenig, G. A. (2018). Rate-based thermal, power, and co-location aware resource management for heterogeneous data centers. *Journal of Parallel and Distributed Computing*, 112:126–139.

Patterson, M. (2008). The effect of data center temperature on energy efficiency. In *Thermal and Thermomechanical Phenomena in Electronic Systems (ITHERM)*, pages 1167 –1174.

Pedram, M. and Nazarian, S. (2006). Thermal modeling, analysis, and management in vlsi circuits: Principles and methods. *Proceedings of the IEEE*, 94(8):1487–1501.

Pellegrini, F. and Roman, J. (1996a). Experimental analysis of the dual recursive bipartitioning algorithm for static mapping. Technical report, TR 1038-96, LaBRI, URA CNRS 1304, Univ. Bordeaux I.

Pellegrini, F. and Roman, J. (1996b). Scotch: A software package for static mapping by dual recursive bipartitioning of process and architecture graphs. In *Proceedings of the International Conference and Exhibition on High-Performance Computing and Networking*, HPCN Europe, pages 493–498.

Phansalkar, A., Joshi, A., and John, L. K. (2007). Subsetting the spec cpu2006 benchmark suite. *ACM SIGARCH Computer Architecture News*, 35(1):69–76.

Potharaju, R., Chan, J., Hu, L., Nita-Rotaru, C., Wang, M., Zhang, L., and Jain, N. (2015). Confseer: Leveraging customer support knowledge bases for automated

misconfiguration detection. *Proceedings of the VLDB Endowment*, 8(12):1828–1839.

Pradelle, B., Triquenaux, N., Beyler, J. C., and Jalby, W. (2014). Energy-centric dynamic fan control. *Computer Science-Research and Development*, 29(3-4):177–185.

Preissl, R., Schulz, M., Kranzlmüller, D., de Supinski, B. R., and Quinlan, D. J. (2010). Transforming MPI source code based on communication patterns. *Future Generation Computer Systems*, 26(1):147–154.

Rabkin, A. and Katz, R. H. (2013). How hadoop clusters break. *IEEE Software*, 30(4):88–94.

Ramachandran, V., Gupta, M., Sethi, M., and Chowdhury, S. R. (2009). Determining configuration parameter dependencies via analysis of configuration data from multi-tiered enterprise applications. In *Proceedings of the 6th International Conference on Autonomic Computing*, ICAC, pages 169–178.

Real, R. and Vargas, J. M. (1996). The probabilistic basis of jaccard's index of similarity. *Systematic biology*, 45(3):380–385.

Rodrigues, A., Cooper-Balis, E., Bergman, K., Ferreira, K., Bunde, D., and Hemmert, K. S. (2012). Improvements to the structural simulation toolkit. In *Proceedings of the International Conference on Simulation Tools and Techniques*, pages 190–195.

Roy, S., König, A. C., Dvorkin, I., and Kumar, M. (2015). Perfaugur: Robust diagnostics for performance anomalies in cloud services. *Proceedings - International Conference on Data Engineering*, 2015-May:1167–1178.

Santolucito, M., Zhai, E., Dhodapkar, R., Shim, A., and Piskac, R. (2017). Synthesizing configuration file specifications with association rule learning. *Proceedings of the ACM on Programming Languages*, 1:64:1–64:20.

Shen, K., Shriraman, A., Dwarkadas, S., Zhang, X., and Chen, Z. (2013). Power containers: An os facility for fine-grained power and energy management on multicore servers. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 65–76.

Shin, J. L., Tam, K., Huang, D., Petrick, B., Pham, H., Hwang, C., Li, H., Smith, A., Johnson, T., Schumacher, F., Greenhill, D., Leon, A. S., and Strong, A. (2010). A 40nm 16-core 128-thread cmt sparc soc processor. In *International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*.

Simon, H. D. and Teng, S.-H. (1997). How good is recursive bisection? *SIAM Journal on Scientific Computing*, 18(5):1436–1445.

Skinner, D. and Kramer, W. (2005). Understanding the causes of performance variability in HPC workloads. In *IEEE International Symposium on Workload Characterization*, pages 137–149.

Snir, M., Wisniewski, R. W., Abraham, J. A., Adve, S. V., Bagchi, S., Balaji, P., Belak, J., Bose, P., Cappello, F., Carlson, B., Chien, A. A., Coteus, P., Debardeleben, N. A., Diniz, P. C., Engelmann, C., Erez, M., Fazzari, S., Geist, A., Gupta, R., Johnson, F., Krishnamoorthy, S., Leyffer, S., Liberty, D., Mitra, S., Munson, T., Schreiber, R., Stearley, J., and Hensbergen, E. V. (2014). Addressing failures in exascale computing. *International Journal of High Performance Computing Applications*, pages 129–173.

Subramani, V., Kettimuthu, R., Srinivasan, S., Johnston, J., and Sadayappan, P. (2002). Selective buddy allocation for scheduling parallel jobs on clusters. In *IEEE International Conference on Cluster Computing*, pages 107–116.

Subramoni, H., Potluri, S., Kandalla, K., Barth, B., Vienne, J., Keasler, J., Tomko, K., Schulz, K., Moody, A., and Panda, D. K. (2012). Design of a scalable infiniband topology service to enable network topology aware placement of processes. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 70:1–70:12.

Talby, D. and Feitelson, D. G. (1999). Supporting priorities and improving utilization of the ibm sp scheduler using slack-based backfilling. In *Proceedings of the International Symposium on Parallel and Distributed Processing*, pages 513–517.

Tang, C., Kooburat, T., Venkatachalam, P., Chander, A., Wen, Z., Narayanan, A., Dowell, P., and Karl, R. (2015). Holistic configuration management at facebook. In *Proceedings of the Symposium on Operating Systems Principles*, pages 328–343.

Tang, Q., Mukherjee, T., Gupta, S. K., and Cayton, P. (2006). Sensor-based fast thermal evaluation model for energy efficient high-performance datacenters. In *International Conference on Intelligent Sensing and Information Processing (ICISIP)*, pages 203–208.

Taylor, M. and Vargo, S. (2014). *Learning Chef: A Guide to Configuration Management and Automation*. "O'Reilly Media, Inc.".

Technavio (2017). Global data center server market 2017-2021. Technical report, Technavio. `https://goo.gl/ues2i8`.

Tsafrir, D., Etsion, Y., and Feitelson, D. G. (2007). Backfilling using system-generated predictions rather than user runtime estimates. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 18(6):789–803.

Tuncer, O., Ates, E., Zhang, Y., Turk, A., Brandt, J., Leung, V., Egele, M., and Coskun, A. K. (2017a). Diagnosing performance variations in hpc applications using machine learning. In *International Supercomputing Conference (ISC-HPC)*, pages 355–373.

Tuncer, O., Leung, V. J., and Coskun, A. K. (2015). Pacmap: Topology mapping of unstructured communication patterns onto non-contiguous allocations. In *Proceedings of the ACM International Conference on Supercomputing*, pages 37–46.

Tuncer, O., Vaidyanathan, K., Gross, K., and Coskun, A. K. (2014). Coolbudget: Data center power budgeting with workload and cooling asymmetry awareness. In *IEEE International Conference on Computer Design (ICCD)*, pages 497–500.

Tuncer, O., Zhang, Y., Leung, V. J., and Coskun, A. K. (2017b). Task mapping on a dragonfly supercomputer. In *IEEE High Performance Extreme Computing Conference (HPEC)*.

Turk, A., Chen, H., Tuncer, O., Li, H., Li, Q., Krieger, O., and Coskun, A. K. (2016). Seeing into a public cloud: Monitoring the massachusetts open cloud. In *USENIX Workshop on Cool Topics on Sustainable Data Centers*.

Verma, A., Pedrosa, L., Korupolu, M., Oppenheimer, D., Tune, E., and Wilkes, J. (2015). Large-scale cluster management at google with borg. In *Proceedings of the European Conference on Computer Systems (EuroSys)*, page 18.

Villars, R. L., Perry, R., and Scaramella, J. (2012). Converging the datacenter infrastructure: Why, how, so, what? Technical report, International Data Corporation. `https://goo.gl/mMCBfk`.

Wang, D., Bhatelé, A., and Ghosal, D. (2014). Performance variability due to job placement on edison. In *Poster presented at the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*.

Wang, G., Yang, J., and Li, R. (2016). An anomaly detection framework based on ica and bayesian classification for iaas platforms. *KSII Transactions on Internet and Information Systems (TIIS)*, 10(8):3865–3883.

Wang, H. J., Platt, J. C., Chen, Y., Zhang, R., and Wang, Y.-M. (2004). Automatic misconfiguration troubleshooting with peerpressure. In *Proceedings of the USENIX Symposium on Opearting Systems Design & Implementation*, OSDI, pages 17–17.

Wang, Z., Tolia, N., and Bash, C. (2010). Opportunities and challenges to unify workload, power, and cooling management in data centers. In *Proceedings of the Fifth International Workshop on Feedback Control Implementation and Design in Computing Systems and Networks*, pages 1–6.

Xiang, Y., Chantem, T., Dick, R. P., Hu, X. S., and Shang, L. (2010). System-level reliability modeling for mpsocs. In *International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pages 297–306.

Xu, T., Jin, L., Fan, X., Zhou, Y., Pasupathy, S., and Talwadker, R. (2015). Hey, you have given me too many knobs!: Understanding and dealing with over-designed configuration in system software. In *Proceedings of the 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE, pages 307–319.

Xu, T., Jin, X., Huang, P., Zhou, Y., Lu, S., Jin, L., and Pasupathy, S. (2016). Early detection of configuration errors to reduce failure damage. In *12th USENIX Symposium on Operating Systems Design and Implementation*, OSDI.

Xu, T., Zhang, J., Huang, P., Zheng, J., Sheng, T., Yuan, D., Zhou, Y., and Pasupathy, S. (2013). Do not blame users for misconfigurations. In *Proceedings of the ACM Symposium on Operating Systems Principles*, SOSP, pages 244–259.

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 ACM Symposium on Operating Systems Principles*, SOSP, pages 159–172.

Yu, L. and Lan, Z. (2016). A scalable, non-parametric method for detecting performance anomaly in large scale computing. *IEEE Transactions on Parallel and Distributed Systems*, 27(7):1902–1914.

Yuan, D., Xie, Y., Panigrahy, R., Yang, J., Verbowski, C., and Kumar, A. (2011). Context-based online configuration-error detection. In *Proceedings of the USENIX Annual Technical Conference*, USENIXATC, pages 28–28.

Zapater, M., Ayala, J. L., Moya, J. M., Vaidyanathan, K., Gross, K., and Coskun, A. K. (2013). Leakage and temperature aware server control for improving energy efficiency in data centers. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 266–269.

Zapater, M., Tuncer, O., Ayala, J. L., Moya, J. M., Vaidyanathan, K., Gross, K., and Coskun, A. K. (2015a). Leakage-aware cooling management for improving server energy efficiency. *IEEE Transactions on Parallel and Distributed Systems*, 26(10):2764–2777.

Zapater, M., Turk, A., Moya, J. M., Ayala, J. L., and Coskun, A. K. (2015b). Dynamic workload and cooling management in high-efficiency data centers. In *International Green and Sustainable Computing Conference (IGSC)*, pages 1–8.

Zhai, J., Sheng, T., He, J., Chen, W., and Zheng, W. (2009). Fact: Fast communication trace collection for parallel applications through program slicing. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis (SC)*, pages 27:1–27:12.

Zhan, X. and Reda, S. (2013). Techniques for energy-efficient power budgeting in data centers. In *Design Automation Conference (DAC)*, pages 1–7.

Zhan, X. and Reda, S. (2015). Power budgeting techniques for data centers. *IEEE Transactions on Computers*, 64(8):2267–2278.

Zhang, J., Renganarayana, L., Zhang, X., Ge, N., Bala, V., Xu, T., and Zhou, Y. (2014). Encore: Exploiting system environment and correlation information for misconfiguration detection. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS, pages 687–700.

Zhang, S. and Ernst, M. D. (2014). Which configuration option should i change? In *Proceedings of the International Conference on Software Engineering*, ICSE, pages 152–163.

Zhang, S. and Ernst, M. D. (2015). Proactive detection of inadequate diagnostic messages for software configuration errors. In *Proceedings of the International Symposium on Software Testing and Analysis*, ISSTA, pages 12–23.

Zhang, X., Meng, F., Chen, P., and Xu, J. (2016). Taskinsight : A fine-grained performace anomaly detection and problem locating system. *IEEE International Conference on Cloud Computing (CLOUD)*, pages 2–5.

Zhang, Y., Tuncer, O., Kaplan, F., Olcoz, K., Leung, V. J., and Coskun, A. K. (2018). Level-spread: A new job allocation policy for dragonfly networks. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. Manuscript accepted for publication.

Zhou, S., Li, S., Liu, X., Xu, X., Zheng, S., Liao, X., and Xiong, Y. (2017). Easier said than done: Diagnosing misconfiguration via configuration constraints analysis: A study of the variance of configuration constraints in source code. In *International Conference on Evaluation and Assessment in Software Engineering*, EASE'17, pages 196–201.

Zhou, S., Liu, X., Li, S., Dong, W., Liao, X., and Xiong, Y. (2016). Confmapper: automated variable finding for configuration items in source code. In *IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C)*, pages 228–235.

CURRICULUM VITAE