2020

Automating telemetry- and trace-based analytics on large-scale distributed systems

BOSTON UNIVERSITY

COLLEGE OF ENGINEERING

Dissertation

AUTOMATING TELEMETRY- AND TRACE-BASED ANALYTICS ON LARGE-SCALE DISTRIBUTED SYSTEMS

by

EMRE ATEŞ

B.S., Middle East Technical University, 2015

Submitted in partial fulfillment of the

requirements for the degree of

Doctor of Philosophy

2020

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

Wenchao Li, Ph.D.
Assistant Professor of Electrical and Computer Engineering
Assistant Professor of Systems Engineering

Fourth Reader

Raja R. Sambasivan, Ph.D.
Ankur and Mari Sahu Assistant Professor of Computer Science
Tufts University

Dedicated to my mother.

Acknowledgments

There are so many people without whose support and contributions this dissertation would not have happened.

First, I would like to thank my advisor Prof. Ayşe K. Coşkun for guiding me during my PhD in both technical matters and otherwise. Her drive, intelligence, and vision have helped both my growth as a researcher and this dissertation immensely.

I would like to thank my almost co-advisors Prof. Manuel Egele and Prof. Raja R. Sambasivan, who helped me with insightful discussions, technical advice, and creative ideas. I would also like to thank Prof. Wenchao Li for being an attentive and active part of my dissertation committee.

I would like to thank Dr. Vitus J. Leung and Jim Brandt from Sandia National Laboratories for their close collaboration on most of my dissertation and for advising me during my internship. Thanks to Dr. Ata Turk for the discussions and his advice on many topics and for always being very encouraging.

Dr. Tapasya Patki and Dr. Jayaraman J. Thiagajaran from Lawrence Livermore National Laboratory, and Chris Kennelly from Google mentored me through my internships during my PhD. Thanks to their efforts, I gained a much wider experience and worked on a variety of projects.

I would like to thank all the other graduate students, with whom I have had the pleasure of collaborating, Dr. Ozan Tuncer, Yijia Zhang, Burak Aksar, Mert Toslali and Anthony Byrne. I would also like to thank past and current members of PeacLab and residents of PHO340 for making the PhD period more fun.

I will always be grateful to my parents, siblings, and extended family, who have always encouraged me in my endeavors and provided me with everything I needed, and for giving me the mixture of curiosity, resilience and self-confidence that is needed to complete a PhD. I would also like to specifically thank Arzu Ateş for not letting me

worry about anything back home when I moved half-way around the world to pursue my PhD.

Finally, I would like to thank my wife and collaborator, Dr. Qingqing Xiong, for always being with me in the worst and the best parts of the last 3 years and hopefully many more years to come. Meeting you was the best thing that happened to me.

Emre Ateş

AUTOMATING TELEMETRY- AND TRACE-BASED ANALYTICS ON LARGE-SCALE DISTRIBUTED SYSTEMS

EMRE ATEŞ

Boston University, College of Engineering, 2020

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

ABSTRACT

Large-scale distributed systems—such as supercomputers, cloud computing platforms, and distributed applications—routinely suffer from slowdowns and crashes due to software and hardware problems, resulting in reduced efficiency and wasted resources. These large-scale systems typically deploy monitoring or tracing systems that gather a variety of statistics about the state of the hardware and the software. State-of-the-art methods either analyze this data manually, or design unique automated methods for each specific problem. This thesis builds on the vision that generalized automated analytics methods on the data sets collected from these complex computing systems provide critical information about the causes of the problems, and this analysis can then enable proactive management to improve performance, resilience, efficiency, or security significantly beyond current limits.

This thesis seeks to design scalable, automated analytics methods and frameworks for large-scale distributed systems that minimize dependency on expert knowledge, automate parts of the solution process, and help make systems more resilient. In

addition to analyzing data that is already collected from systems, our frameworks also identify what to collect from where in the system, such that the collected data would be concise and useful for manual analytics. We focus on two data sources for conducting analytics: numeric telemetry data, which is typically collected from operating system or hardware counters, and end-to-end traces collected from distributed applications.

This thesis makes the following contributions in large-scale distributed systems: (1) Designing a framework for accurately diagnosing previously encountered performance variations, (2) designing a technique for detecting (unwanted) applications running on the systems, (3) developing a suite for reproducing performance variations that can be used to systematically develop analytics methods, (4) designing a method to explain predictions of black-box machine learning frameworks, and (5) constructing an end-to-end tracing framework that can dynamically adjust instrumentation for effective diagnosis of performance problems.

Contents

List of Tables

List of Figures

List of Abbreviations

AI	.....	Artificial Intelligence
API	.....	Application Programming Interface
CC	.....	Chameleon Cloud [Keahey et al., 2018]
CCA	.....	Canonical Correlation Analysis [Hotelling, 1936]
CPU	.....	Central Processing Unit
GPU	.....	Graphics Processing Unit
DAG	.....	Directed Acyclic Graph
DB	.....	Database
DT	.....	Decision Tree [Breiman, 2017]
HDFS	.....	Hadoop Distributed File System [Apache Software Foundation, 2019]
HPAS	.....	HPC Performance Anomaly Suite
HPC	.....	High Performance Computing
ICA	.....	Independent Component Analysis
ID	.....	Identifier
IPM	.....	Integrated Performance Monitoring [Skinner et al., 2009]
kNN	.....	k-nearest neighbors
LDMS	.....	Lightweight Distributed Metric Service [Agelastos et al., 2014]
ML	.....	Machine Learning
MOC	.....	Massachusetts Open Cloud [Mass Open Cloud, 2020]
MPI	.....	Message Passing Interface [Message Passing Interface Forum, 1994]
NFS	.....	Network File System
NIC	.....	Network Interface Card
OS	.....	Operating System
PAPI	.....	Performance Application Programming Interface [Terpstra et al., 2010]
RF	.....	Random Forest [Breiman, 2001a]
RPC	.....	Remote Procedure Call
STAIF	.....	Statistical Trace-driven Automated Instrumentation Framework [Ates et al., 2020d]
SVC	.....	Support Vector Classifier

SVM	.....	Support Vector Machine
UUID	.....	Universally Unique Identifier
VM	.....	Virtual Machine

Chapter 1

Introduction

From e-commerce, video hosting and social networks to particle physics and drug discovery, large-scale computing systems have enabled the advancement of science in many areas, have helped improve the quality of life, and they are among the forefronts of computer engineering research. However, diagnosing problems in these systems is incredibly challenging. Developers often spend nearly 50% of their time on debugging [O'Dell, 2017].

As these large-scale distributed systems grow more complex, understanding the system operation sufficiently to diagnose problems also gets more difficult. For example, operators of supercomputers typically do not know what applications are executing in their systems, and as a result, fraud, waste, and abuse of computing resources has grown to be a more prominent problem [Peisert et al., 2015]. Another example is performance variability. Because of unresolved performance problems, performance variability of jobs in modern high performance computing (HPC) systems can reach up to 100% [Bhatele et al., 2013].

In order to help with problem diagnosis, engineers and programmers have added instrumentation of various types to most components of a computer. The data collected is varied in the degrees of structure, but can broadly be grouped into numeric data, i.e., *metrics*, and event-based data, i.e., *logs*. Examples of numeric data include the performance counters exposed by the CPUs, network components, or the operating system. Event-based data include logs with various metadata such as timestamps,

request IDs, or application-specific state information.

The traditional approach for problem diagnosis is to manually inspect the instrumentation collected, or to have automated heuristics designed to detect one or a few known problems. However, due to the growing scale and complexity of large-scale computing systems, finding out which hardware or software component a problem stems from, which node's metrics or which service's logs to inspect, or deciding which of the many possible problem scenarios has occurred has gotten dramatically more challenging. The data collected from large-scale systems can easily reach multiple terabytes per day [Tuncer et al., 2017], making any type of analysis a substantial engineering effort.

To make things worse, there are constraints on how much instrumentation data can be collected from a system. For all current CPUs, collecting all hardware performance counter data at the same time is not possible due to hardware restrictions. The collected data needs to be stored, and storage capacity dedicated to telemetry and trace data may be limited. The telemetry data may need to be transmitted over the network, consuming bandwidth, and collected consuming memory and CPU time. Furthermore, certain processes in log or metric collection need to be performed on the same thread as the applications, causing a direct slowdown on applications.

Because of instrumentation constraints, operators need to decide what to collect from the systems to help diagnosis, and engineers need to decide what instrumentation to make available when building systems. However, it is almost impossible to know what instrumentation to add *a priori* to help diagnose problems that may happen in the future. For example, Zhao et al. [Zhao et al., 2017] state that Hadoop, HBase, and Zookeeper have been patched over 28,821 times over their lifetimes to add, remove, or modify static log statements embedded in their code.

1.1 Thesis Statement

In this thesis, we design automated methods for large-scale distributed systems that reduce dependency on expert knowledge, substantially improve the speed of problem detection and diagnosis compared to manual analysis, and improve the accuracy compared to state-of-the-art automated methods. We also show that the frameworks we propose are flexible and generally-applicable, which is important for real-world feasibility of these automated frameworks, since building or using a specialized framework for every new problem, system, or application is neither feasible nor scalable.

There are several unique properties of large-scale computing systems that make such automated methods feasible. First, repeated behavior is common in large-scale computing systems. Supercomputers often run the same simulation many times with different inputs as part of scientific computing applications. Many nodes of a supercomputer have identical hardware, and run similar computations as part of a larger simulation. Distributed applications also show repeated behavior; e.g., distributed data stores repeatedly process similar READ or WRITE requests. Furthermore, problems in large-scale systems are also likely to repeat. Therefore, methods such as machine learning that make use of this repeated behavior are good matches to solve such problems.

The second aspect that makes automated methods feasible is the availability of different structured data sources. Even though data collection imposes an overhead, the collected data is still abundant. For example, Facebook collects 1.16 gigabyte of instrumentation data per second [Kaldor et al., 2017]. Therefore, data-driven automated methods can be designed and implemented to make use of this rich data.

1.2 Contributions

In order to demonstrate the application of automated analytics methods, we choose three common categories of problems in large-scale computing systems: (1) Identifying which application is executing in a supercomputer, (2) diagnosing node-level performance variations in large-scale computers, and (3) performance debugging of distributed applications. For each category, we develop an automated analytics method that accelerates problem resolution by processing raw telemetry and trace data and providing refined high-level semantic information to operators. This high level information can be in the form of direct diagnosis, e.g., where network contention is occurring, or traces with only relevant trace points included so that the operators can easily use the selected data for debugging.

We also observe two areas of improvement common to many automated analytics methods that make developing, comparing, debugging and deploying them practically feasible. First, we observe that performance variation is widely studied by researchers, and synthetically generating performance variation is a common method; however, there is no commonly used performance variation generator, which results in a fragmented research space (e.g., [Tuncer et al., 2017, Lan et al., 2010, Kasick et al., 2010]). Second, many automated analytics tools contain machine learning algorithms that are essentially black-boxes to operators as they do not provide reasoning about their predictions, which makes such frameworks more difficult to trust, and challenging to debug and deploy. We make contributions in both of these directions.

All of our frameworks aim to automate the initial steps of problem resolution such as identifying when contention occurs or enabling instrumentation that is useful to debug a problem. This automation leaves the remainder of the problem resolution to human operators, who then can find the root cause of the problem and make code or configuration changes to resolve the problem, or make policy-level decisions to resolve

the issues with their systems. It is beyond the scope of this thesis to fully automate the debugging cycle from problem detection to the application of the resolution.

1.2.1 Application Detection

Modern supercomputers are shared among thousands of users running a variety of applications. Knowing which applications are running in the system can bring substantial benefits to system owners and operators: knowledge of applications that intensively use shared resources can aid scheduling; unwanted applications such as cryptocurrency mining or password cracking can be blocked; system architects can make future design decisions based on system usage. However, identifying applications on supercomputers is challenging because applications are executed using esoteric scripts along with binaries that are compiled and named by users.

We propose an automated analytics method, *Taxonomist*, that uses metrics collected from supercomputers to detect which application is running. Compared to existing methods [DeMasi et al., 2013, Combs et al., 2014], our method causes less overhead on the running applications, thus can be deployed in production systems, and it has higher accuracy. Our method operates by extracting statistical features from the system telemetry data, which is in the form of time series, and using machine learning methods such as random forests to classify applications.

We show that when trained with a set of target applications, Taxonomist can identify those applications with an F-score over 0.95, even when running with different inputs or in the presence of a large number of unknown applications. We also demonstrate the efficacy of Taxonomist in detecting cryptocurrency mining.

1.2.2 Performance Variation Diagnosis

There is significant performance variability in current HPC systems [Bhatele et al., 2013]. However, users of HPC systems often do not know if their job had good or

poor performance unless the problem is extreme. This challenge, coupled with the scale of the systems and the data collected, makes locating performance problems, diagnosing the root causes, and resolving them difficult problems for large-scale computing systems.

To address this problem, we propose an automated analytics framework that processes metrics collected from supercomputers and cloud computing systems. Our framework detects when and where a node-level performance *anomaly*[1] occurs, and classifies the type of anomaly. Our framework operates in a similar manner with Taxonomist, extracting statistical features from metrics and using machine learning methods to classify the anomalies. We show that our framework diagnoses the cause of performance variations at runtime with over an F-score over 0.97.

1.2.3 Reproducing Performance Variations

Past work on detection of performance variability on HPC systems [Klinkenberg et al., 2017, Borghesi et al., 2019b], including our own [Tuncer et al., 2017, Tuncer et al., 2019], rely on synthetically reproducing performance variations to collect performance data. However, there is no standard way of reproducing these variations, resulting in a fragmented research space where results from different teams are difficult to compare, difficult-to-reproduce results, and wasted researcher time.

We develop and open-source a performance variability generator for HPC systems, HPC Performance Anomaly Suite, *HPAS*. HPAS is configurable, easy to use, and targets various subsystems in a large-scale computing system. We study the performance effects of HPAS and demonstrate several use cases for it. For example, our suite is able to create resource contention on many of the supercomputer subsystems, and thus, can be used to generate a data set for the evaluation of machine learning methods that detect performance problems.

[1] We refer to events or conditions that result in suboptimal performance as anomalies.

1.2.4 Explaining Automated Analytics Frameworks

A major roadblock in the development and deployment of automated analytics methods is the black-box nature of the machine learning algorithms these analytics frameworks use. To address this problem, machine learning practitioners have developed various *explainability* techniques that can explain past decisions by machine learning algorithms. We find that none of the existing explainability methods (e.g., [Ribeiro et al., 2016, Lundberg and Lee, 2017]) can tackle the complexity of time series metric data that is collected on large-scale systems.

In this dissertation, we define the "counterfactual time series explainability problem," and develop heuristics to solve this problem. We compare the explanations generated by our method with state-of-the-art explainability methods, and show that our explanations are more comprehensible, providing explanations that in the typical case contain only 2 or 3 time series, and provide equal or better *robustness* to changing problems and *faithfulness* to the original classifier.

1.2.5 Debugging Performance Problems in Distributed Applications

In addition to system-level performance problems, distributed applications can also have significant application-level performance problems that are challenging to debug. On request-based distributed applications, developers and operators use workflow-centric traces [Sambasivan et al., 2016], which are graphs of events in a distributed application that represent work done on behalf of a request, to diagnose problems. Traces are more useful than node-level metrics for such systems, since metrics provide node-level information and one node can be hosting many services, and application-level performance problems may not even be reflected in node-level metrics. One major challenge in using traces to diagnose such problems is the difficulty of knowing what instrumentation to collect, since tracing can cause significant overhead.

We propose an automated analytics framework that can decide what instrumentation to collect in response to an ongoing performance problem in distributed applications. Our framework starts with a system with only minimal instrumentation enabled. As the system executes requests, our framework collects workflow-centric traces and analyzes them to localize performance problems. After localizing problems, the framework extends the enabled instrumentation to further localize the problem and collect useful traces. In production systems, all available instrumentation contains over 4000 unique trace points. We show that our framework automatically locates performance problems by exploring only 30% of the available instrumentation while staying under a user-provided instrumentation budget.

1.3 Organization

The rest of the dissertation is organized as follows: In Chapter 2, we provide a brief background on large-scale computing systems, the types of problems that automated analytics methods can address, and how these problems are currently addressed in state-of-the-art systems. In Chapter 3, we describe our automated application detection framework, Taxonomist, followed by Chapter 4, which describes our automated anomaly diagnosis framework. Chapter 5 describes our performance variability generation tool, HPAS, and Chapter 6 presents our explainability method for multivariate time series models. In Chapter 7 we present our automated workflow-centric tracing framework that controls what instrumentation to collect from distributed applications. Finally, we conclude in Chapter 8 and discuss several important open questions.

1.4 Previously Published Work

Most work in this dissertation has been previously published at journals and conferences, and some is currently under review. The application detection work described in

Chapter 3 has been published in the European Conference on Parallel and Distributed Systems [Ates et al., 2018a]. The related code and data are also available as an artifact [Ates et al., 2018b]. The anomaly diagnosis work described in Chapter 4 has been published in the International Supercomputing Conference [Tuncer et al., 2017] and subsequent work has been published in the IEEE Transactions on Distributed and Parallel Systems [Tuncer et al., 2019]. The performance anomaly generation tool in Chapter 5 has been published in the International Conference on Parallel Processing [Ates et al., 2019b]. The tool itself is available online [PeacLab, 2019].

The explainability method in Chapter 6 is under review [Ates et al., 2020c]. The code and data used are available online [Ates et al., 2020a, Ates et al., 2020b]. The automated instrumentation framework described in Chapter 7 is also under review [Ates et al., 2020d]. Preliminary work on Chapter 7 has been published at the Symposium on Cloud Computing [Ates et al., 2019a].

Chapter 2

Background and Related Work

In this chapter, we provide an introduction to large-scale distributed systems and common problems in such systems, an overview of the instrumentation that is already available in these systems, and how the collected data is analyzed today to address the problems we aim to address with our proposed automated analytics frameworks.

2.1 Large-Scale Distributed Systems

A distributed system can be defined as: *"A collection of autonomous computing elements that appears to its users as a single coherent system"* [Tanenbaum and Van Steen, 2007]. Some examples of distributed systems are modern supercomputers, sensor networks, or distributed applications. The scale of these systems can be measured from several angles. For the purposes of this dissertation, large-scale distributed systems are systems to which traditional debugging and operational tools are no longer applicable in an efficient or meaningful way. Examples include supercomputers where scanning log records are so large that using existing tools to analyze them is impossible [Taerat et al., 2011], or distributed applications that are so complex that visualizing microservice interactions in a comprehensible way is no longer feasible [Cockcroft, 2016].

The existing definition for distributed systems make it challenging to define when one distributed system ends and another begins. Modern internet services can be hosted on multiple cloud platforms, may be using software-as-a-service components

such as distributed object stores, and include significant code running on the client-side, such as mobile applications. To the user, the mobile application "appears a single coherent system;" however, to the operators and developers of the internet service, the distributed object store may, and should, "appear as a single coherent system." For the purposes of this dissertation, we define a single distributed system as: "A collection of autonomous computing elements *that are operated by a single entity* and appears to its users as a single coherent system." We distinguish based on operators because the instrumentation data that is collected from these systems are typically only available to the operators. For example, in a distributed application hosted on a cloud platform, the operators have no visibility to the application's performance or health status, thus they can not diagnose problems in the application. Vice versa, the application operators can detect performance problems in the cloud platform if they are impacted by it, but they can not diagnose it further because of the lack of data/expertise.

The benefits of large-scale computing systems are numerous; however, these benefits come with trade-offs. One of these trade-offs is the complexity of the systems. Modern large-scale computing systems are incredibly complex, and this complexity is the cause of many problems that these systems suffer from. This section describes the problems that we focus on in this dissertation using automated analytics methods. These problems include performance problems in supercomputers, cloud computing systems, and distributed applications, as well as fraud, waste and abuse of supercomputers.

One of the most significant challenges in today's HPC systems is application performance variance. For example, the amount of variation in application running times can exceed 100% on production systems [Bhatele et al., 2013, Leung et al., 2003, Inadomi et al., 2015, Skinner and Kramer, 2005]. In addition to leading to unpredictable application running times, performance variations also cause premature

job terminations and wasted compute cycles. Similar performance variations exist on cloud platforms [Ueda and Nakatani, 2010, Mehrotra et al., 2012], resulting in wasted compute cycles.

While resource utilization and efficiency of supercomputers are top concerns for both system operators and users, fraud, waste, and abuse of resources have also been major concerns in HPC [Peisert et al., 2015]. Wasted resources in supercomputing stem from a variety of sources such as application hangs due to software and hardware faults, contention in shared resources (such as high speed networks, shared parallel file systems or memory). Fraudulent use includes bitcoin mining and password cracking, which has recently been gaining media attention [Office of Inspector General, 2014, Rosenberg, 2018]. In the next section, we discuss one of the major causes for wasted resources for large-scale computing systems, performance variability.

2.2 Performance Variability on Large-Scale Computing Systems

Performance debugging is challenging; therefore, many of the automated analytics methods we propose in this dissertation are designed to diagnose performance problems. In this section, we provide an overview of reported causes of performance variability in large-scale computing systems.

It is important to note the difference between performance variability and faults. *Faults* include behaviors in the computer system that result in errors in the correctness or premature termination of the executed program. We focus on *performance variability*, which results in sub-optimal execution time while still obtaining correct results. For example, our focus on performance variability includes *anomalies*, such as network contention, which do not result in wrong results but may lead to reduced performance in systems where different applications share network resources.

Performance variability adversely affects supercomputers in many ways. Users request more time than required for their jobs because they cannot reliably predict job running time, which in turn harms scheduling. Researchers measuring performance need to take a large number of measurements since results may be invalid if insufficient measurements are taken [Hoefler and Belli, 2015, Maricq et al., 2018]. Programmers are unable to decide whether a code change improves or degrades performance due to high run-to-run performance variability.

Performance variations on HPC systems have been studied by many researchers. Skinner et al. report more than 100% slowdown compared to the average in production supercomputers [Skinner and Kramer, 2005]. They examine several different systems and report that cache contention, network contention, file system contention, kernel process scheduling, and system activity are the main causes of performance variations. Bhatele et al. show that on a Cray XE system, the execution time of communication-intensive applications ranges from 28% faster to 41% slower than the average performance [Bhatele et al., 2013]. They investigate various causes for this performance variability such as operating system activity and job allocation strategy, and concur that interference on network links is the principal cause.

We group the root causes of performance variability by the subsystem where the performance variation causing anomaly manifests. This grouping is useful because the method of replication depends on the subsystem. The major subsystems affecting performance in a supercomputer are the CPU, the cache hierarchy, the memory, the high speed network, and the storage system. We provide examples causes of performance variability in each subsystem below.

CPU: Several examples of performance variations stem from the CPU. Orphan processes left from previous jobs that are still consuming CPU cycles are among the anomalies that clog the CPU [Brandt et al., 2010,Brandt et al., 2009]. System processes

may also use a high amount of CPU because of software errors or miscalibration [Cisco, 2017], and OS scheduling may result in unpredictable execution times—also known as "OS jitter". Manufacturing variability of CPUs has also been reported to affect the performance of HPC jobs [Inadomi et al., 2015, Marathe et al., 2017].

Cache Hierarchy: Modern CPUs' cache hierarchies consist of several levels. The cache hierarchy is a typical source of performance variation for both distributed HPC applications and single-server or even single-CPU applications. Some of the cache-related performance variations are unavoidable (e.g., the cold-start effect), while some of them are caused by software problems (e.g., false sharing) or a combination of software and hardware problems [Cook et al., 2017, McCalpin, 2018].

Memory: In systems where nodes are shared between different applications, memory is shared as well, causing contention in the memory. In systems without node-sharing, placement of application data into different memory banks may affect performance [McCurdy and Vetter, 2010]. Other examples causing performance variability in an HPC system are memory intensive orphan processes and memory leaks.

Network: The high speed network is typically shared between many applications, and certain usage patterns may cause network contention, adversely affecting other applications [Bhatele et al., 2016, Grant et al., 2015, Evans et al., 2003]. The location and severity of contention in a network depend on the network topology, whether nodes are shared between different jobs, the number of NICs per node, the number of links between two nodes in the network, and other factors.

Shared Storage: Interference or other problems in shared file systems can also cause performance variations [Dorier et al., 2014]. In most cases, the Message-Passing Interface (MPI) [Message Passing Interface Forum, 1994] requires all of the communicating nodes to have the same binaries in the same paths, and using shared

file systems is a common way to accomplish this. Such file systems are also used for checkpointing data and other inputs/outputs. Both the speed of checkpointing and the speed of I/O depend on the performance of the shared file system, which can vary significantly over time depending on aggregate file system load.

2.3 Instrumentation on Large-Scale Computing Systems

> *"Data! Data! Data!" he cried impatiently.*
> *"I can't make bricks without clay."*
>
> Sir Arthur Conan Doyle

In order to help operators diagnose problems in a running system, instrumentation is typically added to every layer and every component. The instrumentation data that is commonly, and continuously, collected from these systems at runtime can be divided into *time series data* and *logs* [Snodgrass, 1988, Turk et al., 2016]. Other types of data can also be collected such as core dumps or file system changes [Byrne et al., 2020]. However, this data is not commonly and continuously collected because the perceived benefits may be limited, it might incur high overhead to collect the data, or the data is not collected at a high enough volume to be useful for automated analytics. We will next discuss the two types of instrumentation data that we focus on in this dissertation separately.

2.3.1 Time Series Data

Almost every component of every stack layer contains numeric data that can be collected. The facility layer exposes power and temperature metrics, network layer keeps packet statistics, the CPUs contain numerous performance counters, both operating systems and middleware have counters that record various usage and performance statistics. Typically, this data is periodically sampled and stored for later use, which results in a data stream of multivariate time series coming from each node

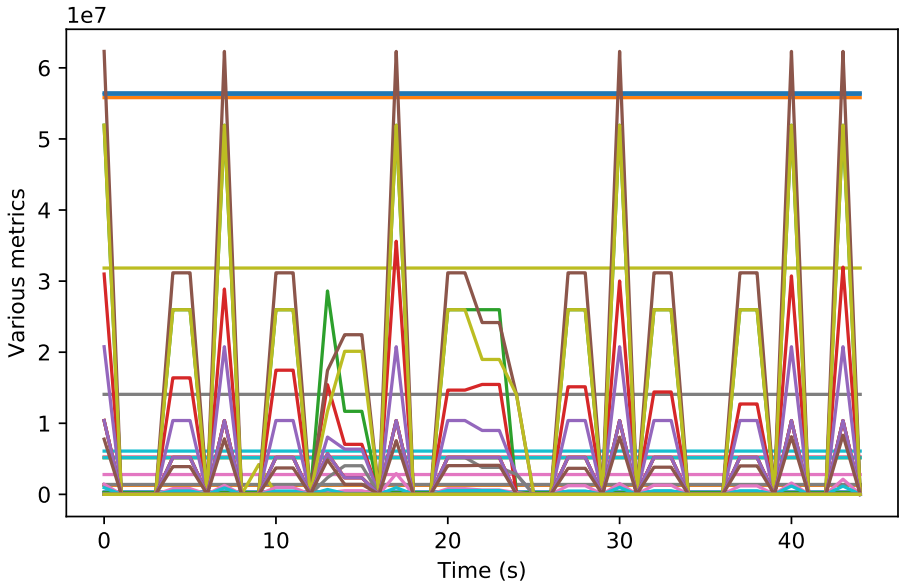


Figure 2·1: A 45-second time window sample from a single node of a multi-node application execution on Voltrino, a Cray XC30m supercomputer with 56 nodes.

in the system. We refer to each of the variables in the multivariate time series as a *metric*. Each metric typically represents one resource/counter.

The mechanism for collecting time series data depends on the system. The amount of data collected and the frequency is also system-dependent. Common open-source tools include Ganglia [Ganglia, 2020], Nagios [Nagios, 2020], Prometheus [Prometheus, 2020]. It is common to collect hundreds to thousands of metrics per node [Agelastos et al., 2014, Borghesi et al., 2019b, Bartolini et al., 2018, Nie et al., 2018]. The tool we commonly use throughout the dissertation is the Lightweight Distributed Metric Service (LDMS) [Agelastos et al., 2014]. LDMS is capable of collecting 100s of time series per node with 1-second sampling intervals from more than 1000 nodes with overhead low enough to be below measurement noise [Agelastos et al., 2015].

There are several distinguishing factors of time series data collected from large-scale computing systems compared to other time series data sources or data sets commonly used by the machine learning community. The HPC system telemetry data contains

1000s of time series, which is larger than any of the data sets at the UCI Machine Learning repository [Bagnall et al., 2020]. Furthermore, only 4 of the 160 data sets in that repository have more than 100 time series. From our experience working with large-scale systems time series data, we have found that the value of the time series is as important as, or more important than, the trend of the time series, unlike traditional machine learning data sets, in domains like motion classification, speech recognition, EKG analysis, etc. where the trend is typically very singificant. These differences result in a need for specialized data pipelines for large-scale systems time series data.

2.3.2 Log Data

Log data can be characterized as a collection of events that is collected from the system with associated metadata. This metadata often contains log messages, information on the state of the system, log messages, severity, process/thread information and request IDs. In this dissertation, we focus on structured logs with workflow information, also known as *workflow-centric traces* [Sambasivan et al., 2016].

An example trace is given in Fig. 2·2. Traces are structured as a Directed Acyclic Graph (DAG), where nodes are events in the distributed system and edges are happened-before relationships. Nodes may contain additional information, similar to log entries, like key-value pairs describing system state and time stamps. Edges typically contain latency information. Nodes with multiple outgoing edges indicate concurrency points and multiple incoming edges indicate synchronization points.

Workflow-centric traces can be collected from distributed systems using various tools, such as X-trace [Fonseca et al., 2007], Dapper [Sigelman et al., 2010], Jaeger [Jaeger, 2020] and Zipkin [Zipkin Foundation, 2020]. These tools typically operate by propagating the request ID with the request, as well as the last event's ID. Each event is written to a common log store, which can then be processed to construct

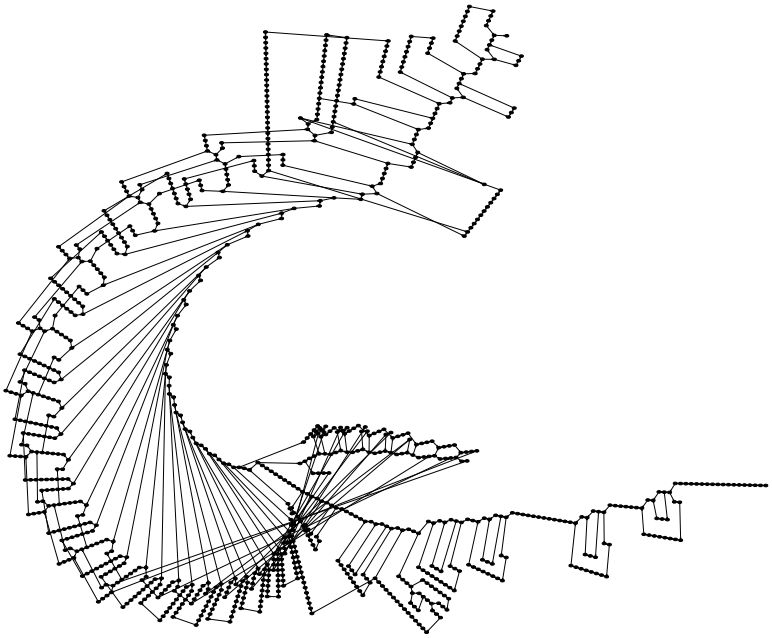


Figure 2·2: A workflow-centric trace collected from an HDFS write operation. The nodes in the graph represent events and the edges represent happens-before relationships. It is very challenging to use this sort of graphs for performance debugging.

the traces.

2.4 Related Work on Instrumentation Analytics

In this section, we describe existing methods for analyzing instrumentation data. We group existing methods based on our contributions: existing methods for application detection, performance variation detection, and trace analysis. We also describe existing methods for performance variation generation, and multivariate time series explainability.

2.4.1 Application Detection

Typically, supercomputer operators have not visibility into which applications are executing on their systems because of the flexibility the users have in compiling and

submitting their applications. Several prior approaches have explored identifying applications. Peisert has identified application detection as a problem in supercomputers [Peisert, 2010]. He focused on using MPI calls through Integrated Performance Monitoring (IPM) [Skinner et al., 2009] to identify application communication patterns. Further work by Whalen et al. refined the method to classify applications based on their communication graphs [Whalen et al., 2013], and DeMasi et al. used system utilization data collected by IPM to identify applications [DeMasi et al., 2013]. These works are based on IPM, which is a tool that monitors the MPI calls in HPC applications. IPM needs to be linked with the applications and introduces up to 5% performance overhead [DeMasi et al., 2013].

Combs et al. have studied the applicability of using power signatures to identify applications [Combs et al., 2014]. As Combs et al. observed, power traces from different servers are not consistently comparable, so such a method is not scalable for large-scale systems. Our evaluation in Sec. 3.4 confirms that using only power signatures is insufficient to identify a diverse set of applications in large-scale systems.

In contrast to related work, Taxonomist [Ates et al., 2018a] (Chap. 3) uses LDMS which has negligible overhead [Agelastos et al., 2014], and is capable of monitoring every application regardless of MPI use, and does not need to be linked with the applications. Taxonomist can be trained with a selection of applications of interest, and can reliably distinguish these applications from the remaining applications. Our method can also detect unknown applications it has not been trained with, which is very important for practical real-world scenarios, since training with every application is infeasible for the very large number of applications that real-world systems may run.

Another line of work aims at blocking unwanted applications. One way to block cryptocurrency mining in supercomputers is to prevent miners from getting the most

recent blockchain additions using firewalls [RedLock CSI Team, 2018]. However, many unwanted applications such as password crackers do not need to be connected to the Internet. Furthermore, firewalls may result in packet losses, and it has been shown that even very small packet loss is unacceptable for scientific computing because of the high bandwidth requirements [Dart et al., 2013]. Another approach to prevent waste might be to whitelist only applications compiled by the system administrators. However, availability is considered to be an important aspect of HPC systems, and limiting the users to use only specific applications would harm the user experience and limit the flexibility and usability of the systems. Therefore, knowledge of the applications running on the system can be a very important aid in blocking unwanted applications.

2.4.2 Performance Variation Diagnosis

In the last decade, there has been growing interest in building automatic performance anomaly detection tools for cloud or HPC systems [Ibidunmoye et al., 2015]. A number of tools have been proposed to detect anomalies of either a specific type (such as network anomalies) or multiple types. These tools in general rely on rule-based methods, time-series prediction, or machine learning algorithms.

Rule-based anomaly detection methods are commonly deployed in large-scale systems. These methods use threshold-based rules on the monitored resource usage and performance metrics, where the rules are set by domain experts based on the performance and resource usage constraints, application characteristics, and the target HPC system [Ahad et al., 2015, Jayathilaka et al., 2017]. The reliance of such methods on expert knowledge limits their applicability. In addition, these rules significantly depend on the target HPC infrastructure and are not easily generalizable to other systems.

Time-series based approaches build a time-series model and make predictions

based on the collected metrics. These methods raise an anomaly alert whenever the prediction does not match the observed metric value beyond an acceptable range of differences. 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], spectral Kalman filter [Laptev et al., 2015], and Holt-Winters forecasting [Ibidunmoye et al., 2016]. While such methods successfully detect anomalous behavior, they are not designed to identify the type of the anomalies (i.e., diagnosis). Moreover, these methods lead to unacceptable computational overhead when the collected set of metrics is large.

A number of machine learning based approaches 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], hierarchical clustering [Gurumdimma et al., 2016], k-means clustering [Bhuyan et al., 2011], and kernel density estimation [Ibidunmoye 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]. Although these studies can detect anomalies on HPC and cloud systems with high accuracy, only a few studies aim at diagnosing the anomaly types [Bodik et al., 2010]. As we show in our evaluation (Sec. 4.4), our approach of using tree-based algorithms on features that summarize time series behavior outperforms existing techniques on detecting and diagnosing anomalies.

Feature extraction is essential to reduce computation overhead and to improve the detection accuracy in learning-based approaches. In addition to using common statistical features such as mean/variance [Klinkenberg et al., 2017], previous works on HPC anomaly detection have also studied features such as correlation coefficients [Chen

et al., 2011], Shannon entropy [De Assis et al., 2013], and mutual information gain [Gurumdimma et al., 2016]. Some approaches also explored advanced feature extraction methods including principal component analysis (PCA) [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]. In Sec. 4.4, we demonstrate that combining statistical feature extraction results in superior anomaly indicators compared to the features selected using statistical techniques such as ICA.

Our anomaly diagnosis work [Tuncer et al., 2017, Tuncer et al., 2019] (Chap. 4) is different from related work in the following aspects: Our proposed framework can *detect* and *diagnose* previously observed performance anomalies, which do not necessarily lead to failures. Our approach does not rely on expert knowledge beyond initial labeling or determination of anomalies of interest, and has negligible computational overhead. In Sec. 4.4, we show that our technique consistently outperforms the state-of-the-art methods in diagnosing performance anomalies in HPC systems.

2.4.3 Performance Variation Generation

Researchers that study methods to detect and analyze the cause of performance variations typically generate their own synthetic anomalies to evaluate their proposed methods [Tuncer et al., 2017, Tuncer et al., 2019, Kasick et al., 2010, Guan et al., 2013, Lan et al., 2010]. However, since synthetic anomaly generation is not the focus of these studies, the methods for performance variability generation are not explained in sufficient detail to replicate the results, and they have not released their codes for generating these synthetic anomalies.

There are various existing tools for creating performance variability on computer systems. Delimitrou et al. build a workload suite for data centers called iBench that induces interference in various shared resources, mostly architectural CPU components [Delimitrou and Kozyrakis, 2013]. Their tool helps quantify the contention

created by applications as well as the contention that can be tolerated by the applications. However, the released version is substantially limited compared to the tool described in the publication. Specifically for networked systems, Sato et al. build a tool called NINJA that mimics network noise by injecting sleep before MPI calls and, thus, creates a message race for MPI applications [Sato et al., 2017]. They demonstrate their tool can manifest subtle message races in MPI applications more frequently; however, their approach is not applicable for most forms of anomaly diagnosis since no actual network contention occurs. Netti et al. introduce a framework called FINJ that enables injecting anomalies into HPC systems [Netti et al., 2019], but their focus is on the mechanism for injecting anomalies, not the anomalies themselves; thus, they do not analyze the behavior and effect of the anomalies. Gremlins is a suite for emulating future HPC systems, e.g., power constrained systems, on current hardware [Schilz et al., 2014]; their methodology is not explicitly targeting performance variability, thus they miss significant components such as network and I/O contention. Our work is the first performance variation generator that addresses all subsystems of HPC systems. It also requires minimal user permissions and no modifications to system software or hardware.

2.4.4 Multivariate Time Series Explainability

As machine learning models started to be used in critical applications such as healthcare, explainability has been the topic of much research. We use the helpful classification of Arya et al. when discussing the existing literature [Arya et al., 2019]. We focus on explainable models and local sample- and feature-based explanations for black-box models.

Explainable models learn a model that is inherently understandable by untrained operators. They include simple models like logistic regression and decision trees and newer models like CORELS, which learns minimal rule lists that are easy to

understand [Angelino et al., 2017]. Our experiences with using CORELS with HPC time series data in Sec. 8.1.3 show that CORELS fails to learn a usable model, and decision trees using HPC data become too complex to be understood by human operators.

Sample-based explanations provide samples that explain decisions by giving supportive or counter examples, or prototypes of certain classes. Koh and Liang use influence functions to find training set samples that are most impactful for a specific decision [Koh and Liang, 2017]. Contrastive explanations method (CEM) provides a synthetic sample with a different classification result that is as similar to the input sample as possible, while using autoencoders to ensure the generated sample is realistic [Dhurandhar et al., 2018]. Sample-based methods do not address the inherent complexity of HPC data, since they focus on tabular data or images where the test sample and the explanation sample are easy to compare manually. HPC time series data with thousands of time series per sample is challenging for users to compare and contrast without additional computing.

Feature-based explanations highlight certain features that are impactful during classification. Global feature-based explanations for some classifiers such as random forests and logistic regression use the learned weights of the classifiers [Palczewska et al., 2014]. Local feature-based explanations include LIME, which fits a linear model to classifier decisions for samples in the neighborhood of the sample to be explained [Ribeiro et al., 2016]. SHAP derives additive Shapley values for each feature to represent the impact of that feature [Lundberg and Lee, 2017]. These feature-based models do not support using time series directly; however, many HPC frameworks use feature transformations and feature-based explanations can explain these models' classifications in terms of the features they use. In this case, interpreting the meaning of complex features such as kurtosis or B-splines is left to the user [Endrei et al., 2018].

Time series specific explanations have also been foci of research. Schegel et al. evaluate different general purpose explainability methods on time series [Schlegel et al., 2019]. Karlsson et al. propose a method to synthetically generate sample-based explanations for time series [Karlsson et al., 2018]. Gee et al. propose a method to learn prototypes from time series [Gee et al., 2019]. All three of these methods operate on univariate time series and do not address the problem of explaining multivariate time series. Assaf and Roy propose a method to extract visual saliency maps for multivariate time series [Assaf and Schumann, 2019]; however, their method is specific to deep neural networks. Furthermore, saliency maps lose their simplicity when scaled to hundreds or thousands of time series.

Counterfactual explanations produce a sample that is similar to the test sample, but with a different classification label, such that the differences can be used as a guide to understand important factors in classifications. Wachter et al. are among the first to use the term "counterfactual" for ML explanations [Wachter et al., 2017]. Counterfactual explanations have also been used in the domains of image [Goyal et al., 2019], document [Martens and Provost, 2014] and univariate time series [Karlsson et al., 2018] classification. DiCE is an open source counterfactual explanation method for black-box classifiers [Mothilal et al., 2020]. To the best of our knowledge, there are no existing methods to generate counterfactual explanations for high-dimensional multivariate time series. However, in our experience counterfactual explanations are the most comprehensible explanations for this type of models (Chap 6). Furthermore, many of the counterfactual explanation methods generate synthetic data without specifically addressing the difficulties of generating highly-correlated multivariate time series data, which is the type of data used in HPC ML frameworks.

Explainability techniques targeting HPC ML frameworks need to consider several properties of HPC data that general-purpose explainability techniques do not consider.

HPC time series data is more complex than traditional machine learning data sets by several aspects. A single sample is most often not explainable because of the volume of data contained. As shown in Fig. 2·1, one sample contains hundreds of time series and understanding which part of which time series is important is very challenging. Therefore, sample-based methods fail to provide comprehensible explanations. Furthermore, each metric in the time series requires research to understand. For example, performance counters with the same name can have different meanings based on the CPU model. Furthermore, the values of metrics may not be meaningful without comparison points. In order to address these challenges, our explainability approach is *both* sample- and feature-based. We provide a counterfactual sample from the training set, and indicate which of the many time series in the sample need to be modified to have a different classification result. This results in an explanation that is easy to understand by human operators, since it requires interpreting only a minimal number of metrics. To help users interpret metric values, we provide examples of the same metric for multiple classes (e.g, both normal and anomalous).

Existing explainability methods often make two assumptions that are not applicable to HPC data: The ability to generate realistic data synthetically, and the availability of gradients. Synthetically generated data is used as part of many methods [Ribeiro et al., 2016, Lundberg and Lee, 2017, Dhurandhar et al., 2018, Wachter et al., 2017]; however, generating realistic HPC time series data is challenging because (1) the data represents physical resources which obey various physical constraints, and (2) metrics are highly correlated with each other—e.g., free memory and used memory. Other explainability methods rely on the gradients of the model [Koh and Liang, 2017, Goyal et al., 2019, Lundberg and Lee, 2017, Wachter et al., 2017]; which are not available for many popular HPC ML frameworks [Tuncer et al., 2017, Ates et al., 2018a, Tuncer et al., 2019, Klinkenberg et al., 2017, Bodik et al., 2010], because these

frameworks use features like percentiles and classifiers such as the random forest for which gradients can not be calculated. Our proposed method (Chap. 6) does not rely on synthetically generated time series and does not require any gradients, which makes it widely applicable to HPC ML frameworks.

Chapter 3

Taxonomist: Application Detection through Rich Monitoring Data

Resource utilization and efficiency of supercomputers are top concerns for both system operators and users. It is typical to use figures of merit such as occupation of compute nodes or total CPU usage to assess utilization and efficiency; however, these metrics do not measure if the compute capacity is used meaningfully.

In fact, fraud, waste, and abuse of resources have been major concerns in high performance computing (HPC) [Peisert et al., 2015]. Wasted resources in supercomputing stem from a variety of sources such as application hangs due to software and hardware faults, contention in shared resources (such as high speed networks, shared parallel file systems or memory), and fraudulent use (e.g., bitcoin mining, password cracking). Bitcoin mining in supercomputing environments has recently been gaining media attention [Office of Inspector General, 2014, Rosenberg, 2018]. Knowing which applications are running on the system is a strong aid in addressing fraud, waste, and abuse problems.

Knowledge of applications running on the system can also be used for various system-level optimizations. Bhatele et al. have shown that network-intensive applications can slow down other applications significantly [Bhatele et al., 2013]. Similarly, Auweter et al. presented a scheduling method that leverages application-specific energy consumption models to reduce overall power consumption [Auweter et al., 2014]. Knowing the most common applications and their characteristics is also useful to

system architects who make design decisions, or to the supercomputer procurers who can make better funding and procurement decisions based on knowledge of typical application requirements.

Typically, supercomputer operators and system management software running on these large computers have no knowledge of which applications are executing in the supercomputer at a given time. A supercomputer is shared by many users and runs hundreds to thousands of applications concurrently per day [NERSC, 2016]. These applications are compiled by users using different compiler settings, which result in vastly different executables even if compiled from the same source. It has been shown that static analysis of the binaries is not enough to detect the same application compiled with different compilers or flags [Egele et al., 2014]. Furthermore, users tend to use non-descriptive names for the binaries and scripts used in their job submission (e.g., `submit128.sh`, `a.out`, `app_runner.sh`). Therefore, naive methods for detecting applications such as looking at the names of the processes and scripts are not useful.

To address these challenges, we present *Taxonomist*, an automated technique for identifying applications running in supercomputers. Each application has (often non-obvious) resource utilization patterns that can be observed in the metrics collected from supercomputers. Taxonomist uses machine learning techniques to learn these patterns in the metrics collected. Taxonomist can then identify known applications, even when they are running with new input configurations, and also new (unknown) applications.

In this chapter, we present Taxonomist: a novel technique that uses machine learning to identify known and unknown applications running in a supercomputer based on readily available system monitoring data (§ 3.2). Taxonomist is able to detect applications that are new to the system, as well as previously unseen input

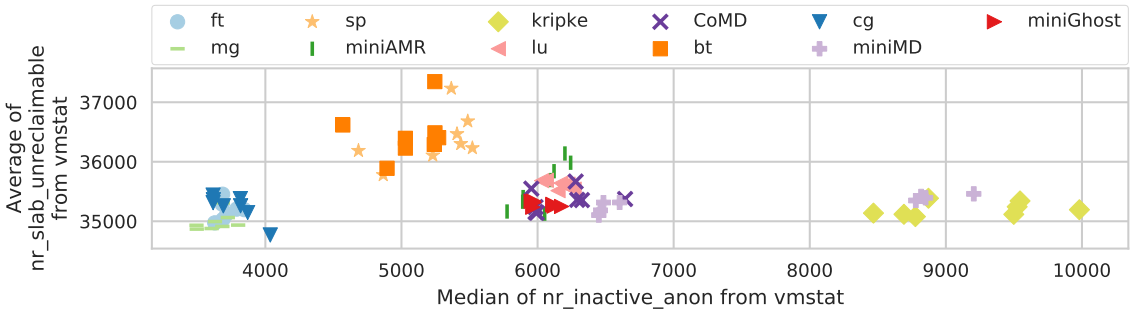


Figure 3·1: Two example metrics from `/proc/vmstat` for 11 applications with two different input configurations, where each application is running on 4 nodes. These two metrics can be used to distinguish among some applications, but cannot be used to reliably detect each of the 11 applications.

configurations of known applications. We demonstrate the effectiveness of Taxonomist on a production supercomputer using over 50,000 production HPC application runs collected over 6 months of cluster usage, a wide selection of benchmarks, and cryptocurrency miners (§ 3.3). We report greater than 95% *F-score* with this data set (§ 3.4).

3.1 Motivation

Taxonomist uses monitoring data to identify applications. Each applications uses the HPC system in a different way, and we claim that this results in unique resource usage fingerprints for applications to identify different applications. Modern monitoring systems are able to continuously collect hundreds of metrics per second from every compute node in an HPC system [Agelastos et al., 2014]. It is infeasible to manually inspect this data and identify applications relying on rules of thumb and expert knowledge; therefore, we design an automated approach to systematically discover the differences between the applications. To test our claim, we run 11 applications on a supercomputer while collecting data and analyze the collected data.

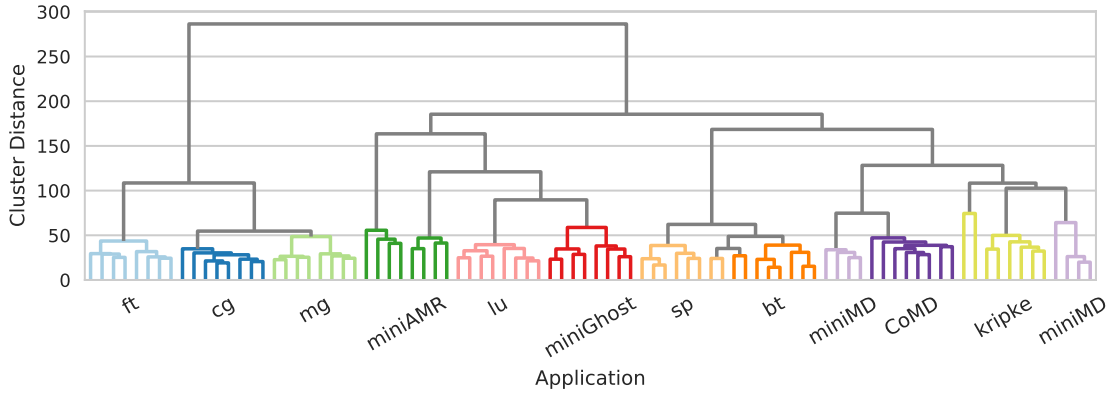


Figure 3·2: Clustering of 11 different applications, where each application is running on 4 nodes with two different input configurations. We manually assign different colors to represent different applications.

Figure 3·1 shows two example metrics for a set of 11 applications we run on a supercomputer (see § 3.3 for details on experimental setup). The x-axis shows the median of `nr_inactive_anon`, which represents the number of anonymous memory pages that are inactive, and the y-axis shows the mean of `nr_slab_unreclaimable`, which is the number of pages in the slab memory that cannot be reclaimed. As seen in the figure, applications have different resource usage characteristics. However, these two metrics are not sufficient to distinguish between all applications. It is rather challenging to determine the best metrics to distinguish among a large set of applications using intuition or simple methods.

Figure 3·2 demonstrates clustering of the same 11 applications using all 721 metrics we collect (see § 3.2.1 for details of the metrics). To construct this figure, we extract statistical features such as percentiles and standard deviation from the collected data (see § 3.2.2), and cluster the statistics corresponding to the compute nodes. For clustering, we use Ward's method and standardized Euclidean distance[1]. The results indicate that nodes running the same application are close to each other in the feature space, but the clustering is not perfect (e.g., miniMD is clustered incorrectly).

[1]our implementation uses Python `scipy.cluster.hierarchy.linkage`

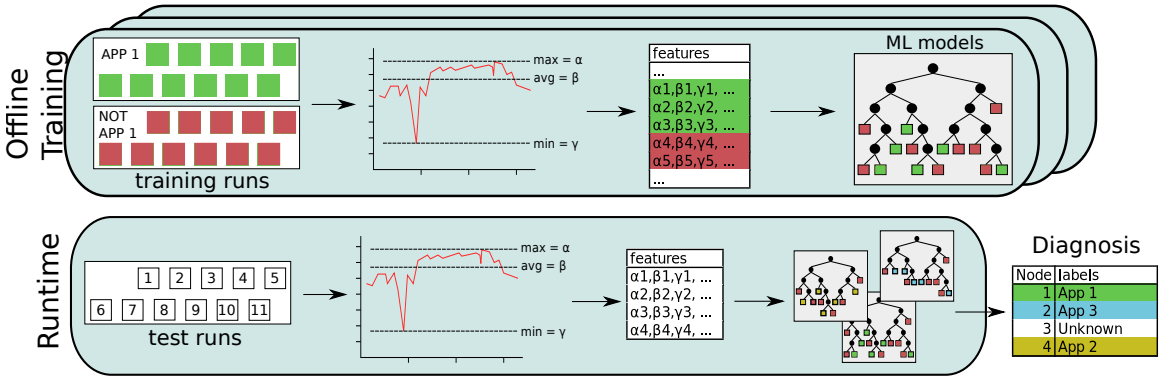


Figure 3·3: Overview of Taxonomist.

Manually finding which metrics are important to distinguish each application among hundreds of monitored metrics requires extensive knowledge on the metrics and applications. With supervised learning, the most relevant features can be automatically selected, and applications can be reliably identified. Thus, Taxonomist uses supervised learning techniques.

3.2 Taxonomist: A Technique for Identifying Applications

Taxonomist, outlined in Fig. 3·3, is a technique for identifying applications in large-scale systems using monitoring data collected from the machine. The monitoring data is collected from every compute node in a time series format. We then generate statistical features that reduce our storage and computation overhead, while enabling us to retain meaningful information in the time series. Finally, we train a classifier for each application to separate that application from the rest of the applications using labeled historical data. At runtime, Taxonomist analyzes monitoring data and labels each node's application according to the predictions from the classifiers. We also mark applications as *unknown*, based on the confidence of each classifier.

3.2.1 Monitoring

The first step of our technique is data collection. Typically, some form of monitoring is in place in supercomputers. These systems collect numeric information about the usage of the network, memory, CPUs and other subsystems.

We monitor individual nodes and consider data from all nodes that are running a specific application separately. This enables us to recognize a known application that possibly runs on a different number of nodes than the number of nodes in that application's training runs.

3.2.2 Statistical Feature Extraction

After collecting monitoring data, Taxonomist removes a segment (40 seconds in our implementation) from each end of the time series to account for the transient initialization and finalization phases from the applications. We have observed 40 seconds to be sufficient for all applications in this study; however, this duration is application dependent. We also remove any constant metrics and convert metrics that represent counter values to their deltas.

Taxonomist generates statistics from the time series data gathered from the compute nodes. The statistics used are the minimum, maximum, mean, standard deviation, skew, kurtosis and the 5^{th}, 25^{th}, 50^{th}, 75^{th} and 95^{th} percentiles. Each metric's time series is distilled into these 11 features. These statistics have been shown to be useful in analyzing time series from supercomputers [Tuncer et al., 2017, Wang et al., 2006]. They are also easy to calculate, reduce storage requirements, and enable us to compare applications that have different durations. We scale each feature to the $[0, 1]$ range according to the values observed in the training set. The same scaling factors are used at runtime.

3.2.3 Classification

To distinguish a set of given applications, we train a machine learning model using a training set of these labeled applications. Taxonomist labels each run with the corresponding application or it can also label new runs as *unknown*.

For each classifier, we use a *one-versus-rest* version of that classifier: i.e., for each application in the training set, we train a separate classifier that differentiates the application. This approach makes it easy to add a new application to the ensemble of classifiers and to get information about the nature of each application. This approach also enables us to train for only applications of interest, and we do not have to re-train every classifier when a new application is added.

For evaluation purposes, we compare the following classification algorithms: random forests, forests of extremely randomized trees (ExtraTrees), decision trees and the support vector machine classifier (SVC) with linear and radial basis function kernels. In terms of accuracy, the best performing one for our data is the random forest (§ 3.4).

From every classifier, we obtain confidence values on whether a new observation belongs to one of the existing training classes. For example, the confidence threshold for the random forest is the percentage of trees in the forest that agree with the final classification. If none of the confidence values are above a predetermined *confidence threshold*, we mark this new observation as *unknown*.

Confidence Threshold Selection: A very high threshold would result in conservatively labeling new inputs of known applications as unknown, while too low values would result in unknown applications being labeled as a similar known application. To select the confidence threshold we first remove each application from the training set and perform testing with examples of that application in the training set while changing the confidence threshold. Then, we remove one input of each application and perform the same test. We select the threshold that results in the highest average

F-score for both scenarios.

Hyperparameter Selection: Most classifiers have hyperparameters that describe the configuration of the algorithm. We find the best hyperparameters by splitting the training set into 5 cross validation folds. With 4/5 of the training data we train classifiers with different hyperparameters, and pick the best performing one using 1/5 of the training set. We choose the important hyperparameters for each classifier and over a certain range we train all combinations of hyperparameters, i.e., grid search. We find the best hyperparameter separately for each application's classifier. Note that we never use any test data during training or hyperparameter selection.

3.2.4 Operation of Taxonomist

During normal operation, Taxonomist uses the monitoring data to label each node of each application after a job finishes. These labels can be used to raise alarms in the case of cryptocurrency mining and to generate system usage reports or other summaries. They can also be used in further research and development on application-specific system optimizations. Furthermore, identifying fraud, waste, and abuse after application completion is still valuable.

As Taxonomist relies on machine learning, it requires a labeled training data set as input. This data set can be collected by a collaboration of users, operations staff, and analysts. After the applications of interest are determined, data can be collected by running them with different input configurations. This training is a one-time effort unless the applications of interest change.

In our current implementation, the application needs to finish before we identify it; however, Taxonomist can be modified to work with only the first few minutes of application data. There are existing methods that propose execution of applications for a short time before the main run is scheduled [Thebe et al., 2009]. Such strategies can be used with Taxonomist to collect data.

3.3 Experimental Methodology

We run our experiments on a production supercomputer, using LDMS [Agelastos et al., 2014] already in place. We evaluate our system with 11 benchmarks, 5 different *unwanted* applications, and also with 6 months of typical supercomputer usage.

3.3.1 Platform

We run all of our experiments on Volta, a Cray XC30m supercomputer located at Sandia National Laboratories. Volta is composed of 13 fully-connected routers, with 4 nodes each, leading to a total of 52 compute nodes. The operating system used is SLES 11 (SUSE Linux Enterprise Server) with kernel version 3.0.101. Each node has 64 GB of memory and two Intel Xeon E5–2695 v2 CPUs with 12 2-way hyper-threaded cores.

LDMS is a scalable monitoring system deployed on Volta. We use the memory metrics collected from `/proc/meminfo` and `/proc/vmstat`, CPU usage information from `/proc/stat`, and network usage information from Cray network interface card (NIC) counters. 721 metrics from every node every second in total.

3.3.2 Applications

Representative Applications: We pick a collection of 11 benchmarks and proxy applications, described in the upper section of Table 3.1. We choose these applications to be representative of characteristic HPC workloads. All representative applications use MPI, and are compiled with the Cray compilers. For each application, we use 3 different input configurations, and we run the applications on 4 nodes. We also run miniAMR, miniMD, miniGhost and Kripke on 32 nodes with an additional input. We

[2]Zcash competition: [Zcash Electric Coin Company, 2016], minerd: `github.com/pooler/cpuminer`, BFGminer: `github.com/luke-jr/bfgminer`, xenon: `github.com/xenoncat/equihash-xenon`, davidjaenson: `github.com/davidjaenson/equihash`, tromp: `github.com/tromp/equihash`, John the Ripper: `openwall.com/john`

Table 3.1: Applications used for the evaluation of Taxonomist.

	Application	# of Inputs	# of Ranks	Description
Representative Applications	BT [Bailey et al., 1991]	3	169	Block tri-diagonal solver
	CG [Bailey et al., 1991]	3	128	Conjugate gradient
	FT [Bailey et al., 1991]	3	128	Fourier transform
	LU [Bailey et al., 1991]	3	192	Gauss-Seidel solver
	MG [Bailey et al., 1991]	3	128	Multi-grid on meshes
	SP [Bailey et al., 1991]	3	169	Scalar penta-diagonal solver
	miniAMR [Heroux et al., 2009]	4	192/1536	Adaptive mesh refinement
	miniMD [Heroux et al., 2009]	4	192/1536	Molecular dynamics
	CoMD [Heroux et al., 2009]	3	192	Molecular dynamics
	miniGhost [Heroux et al., 2009]	4	192/1536	Structured PDE solver
	Kripke [Kunen et al., 2015]	4	192/1536	S_N transport
Unwanted Applications[2]	minerd	10	2/4	CPU cryptocurrency miner
	BFGminer	2	2/4	Cryptocurrency miner
	xenon	2	96/192	Zcash competition winner
	davidjaenson	1	2/4	Zcash competitor
	tromp	1	2/4	Zcash competitor
	John the Ripper	194	96/192	Password cracker

run each application on the maximum number of hardware threads available that the application can utilize.

Unwanted Applications: These are applications that are usually not allowed on supercomputers such as cryptocurrency miners and password crackers. The tromp, davidjaenson, and xenon miners are from an open source miner competition [Zcash Electric Coin Company, 2016]; BFGminer and minerd are popular miners for mining with CPUs. Xenon is single-threaded, so we execute 48 copies per node. Other cryptocurrency miners are multi-threaded, so we execute them one copy per node, using 48 threads. John the Ripper is a popular password cracking application which supports MPI; we execute it one rank per hardware thread. The inputs for John the Ripper are various password formats; and for the cryptocurrency miners, the inputs are the different types of cryptocurrencies. Due to ethical considerations, we ran all unwanted applications in benchmark mode to ensure that none of the cryptocurrency mined was connected to the main blockchains.

Typical Volta Usage: This data includes unlabeled applications run by 28

unique Volta users, consisting of 58,366 jobs, from August 2016 until January 2017. Our controlled experiments are removed from these runs.

3.3.3 Baseline Application Detection Technique

Combs et al. [Combs et al., 2014] have proposed a technique (referred to as *Combs* in the remainder of this chapter) for application detection using power data instead of performance monitoring data. Combs uses a similar feature extraction approach, but in contrast to our method, it extracts serial correlation, non-linearity, self-similarity, chaos, and trend from the time series, as well as skew, kurtosis, serial correlation and non-linearity from the time series with the trend component removed. Furthermore, Combs et al. normalized maximum and median with the minimum for each time series to generate two additional features. Their method uses a random forest classifier and does not have a method for labeling unknown applications, so we do not implement any thresholding for Combs' method.

3.4 Evaluation

We evaluate the capability of Taxonomist in detecting applications with a variety of workloads and scenarios. First, we examine the classification performance in identifying known applications with new input configurations. Then, we evaluate the performance in labeling unknown applications.

For all tests, we first perform 5-fold cross validation, where we split the whole data into five sets with equal distributions of applications with the original data set. We then train five different Taxonomist instances using four of the sets. For testing, we use the fifth set that was removed from training data. For the normalization and hyperparameter selection steps, Taxonomist performs another 5-fold cross-validation on the training set.

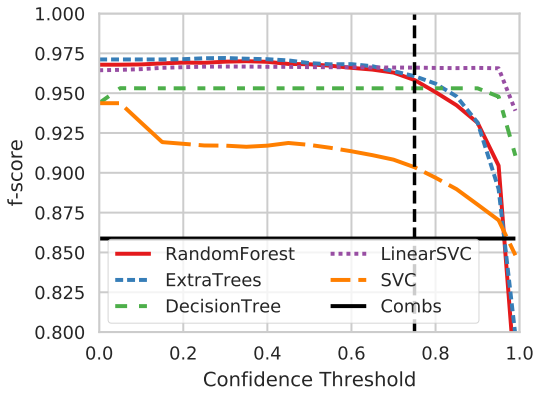
For the results, we report the *F-Score*, which is a widely used measure of classifier performance. For binary classification, F-Score is defined as the harmonic mean of *precision* and *recall*. Precision is the ratio of true positives to the number of all positive predictions, and recall is the ratio of true positives to the number of all actual positives in the data set. F-Score ranges between 1 (best) and 0 (worst). All of our results are multi-class; therefore we calculate the average precision and recall for each class, and take the harmonic mean to calculate the overall F-score.

Full Data Set: Table 3.2 shows the 5-fold cross validation results on the 11 representative applications. All of the results except the baseline technique (Combs) have an F-Score of over 0.99. However, this scenario where the training data contains all applications and all input configurations is unrealistic. SVM with the linear kernel (LinearSVC) performs better than the rbf kernel (SVC). This is likely due to the large data set with many features and data points, and this behavior is consistent with the literature [Hsu et al., 2003].

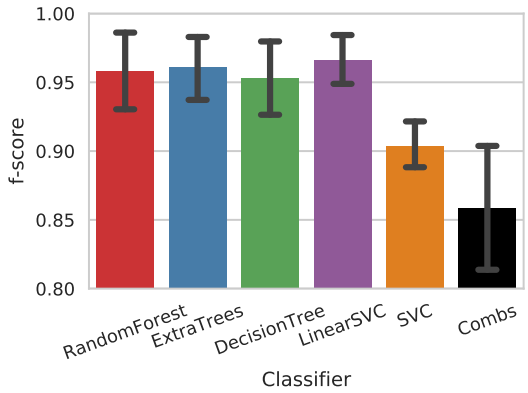
Table 3.2: Five-fold cross validation results with the full data set.

Classifier	Precision	Recall	F-score
RandomForest	1.000	1.000	1.000
ExtraTrees	1.000	1.000	1.000
DecisionTree	0.998	0.998	0.998
LinearSVC	0.999	0.999	0.999
SVC	0.994	0.994	0.994
Combs	0.932	0.931	0.931

Detecting Applications with Unknown Input Configurations: Applications' resource usage is affected by their input configurations. To evaluate Taxonomist's robustness against input configurations that are not in the training set, we remove one of the input sets from the training set. For the test set, we keep the cross validation folds the same. Figure 3·4 shows that the classification is successful unless the confidence threshold is over 0.9, in which case the unknown input configurations are marked as unknown applications.

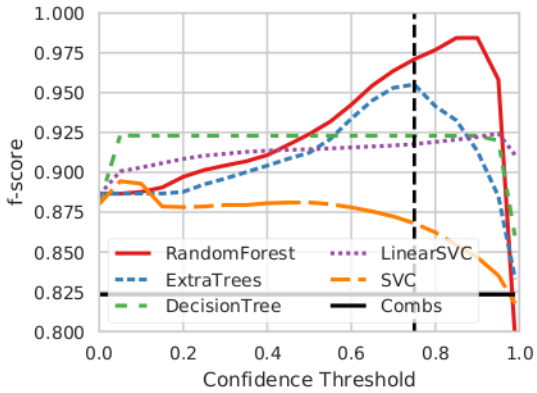


(a) F-scores for classifiers, vertical dashed line indicates the chosen confidence threshold.

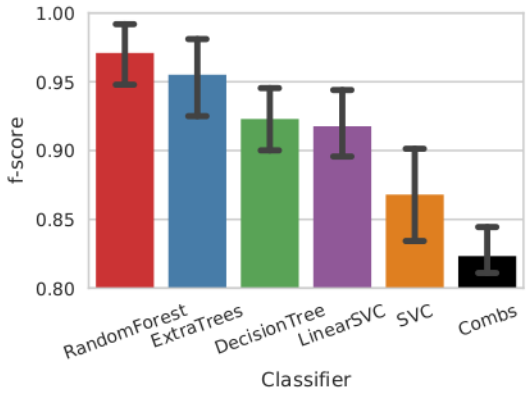


(b) F-scores for classifiers at the chosen confidence threshold, 0.75. Error bars indicate the 95% confidence interval.

Figure 3·4: F-scores with one input configuration removed from training. In most cases, the applications are correctly identified in spite of the unknown input configuration.

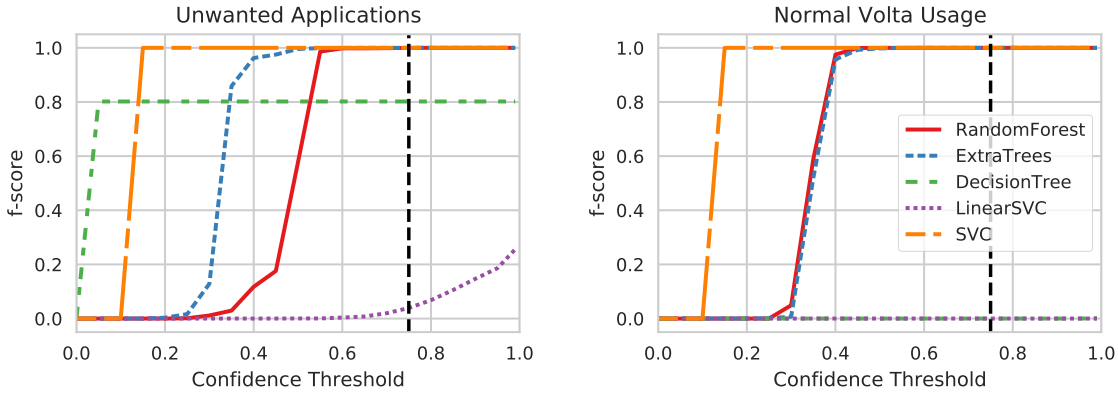


(a) F-scores for classifiers, vertical dashed line indicates the chosen confidence threshold.



(b) F-scores for classifiers at the chosen confidence threshold, 0.75. Error bars indicate the 95% confidence interval.

Figure 3·5: F-scores with one application removed from the training set. With the correct confidence threshold choice, the unknown application can be correctly identified.



(a) F-scores when tested with bitcoin miners and password crackers.

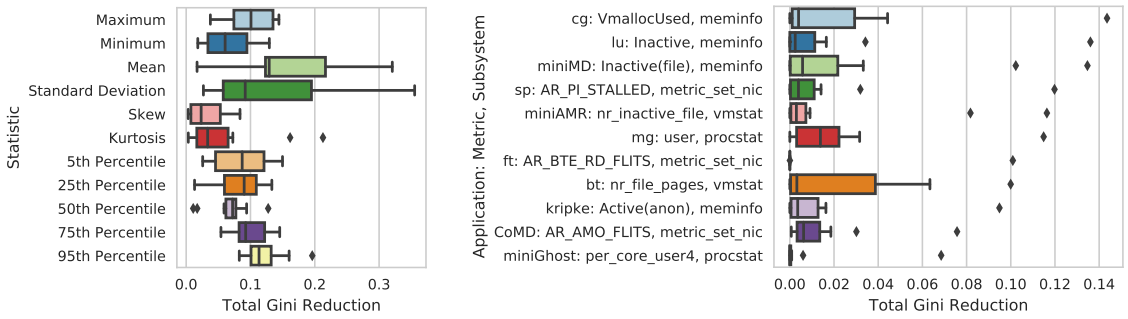
(b) F-scores when tested with HPC applications that are not known to the classifiers.

Figure 3·6: The classifiers can correctly identify unknown applications, whether they are HPC applications or bitcoin miners and password crackers.

Detecting Unknown Applications: Figure 3·5 shows classification results with one application removed from the training set. If the removed application is labeled as unknown, we mark it as a correct prediction. In more than 95% of the cases, the unknown application is correctly identified as such. The lowest F-Scores are for the BT and SP applications, which are both partial differential equation solvers, and they have been shown to have similar behavior [Ma et al., 2009]. Hence, the classifiers tend to mispredict SP and BT.

The confidence threshold that gives the maximum value for the average F-scores of the unknown input and unknown application cases is 0.75, and Random Forest is the classifier that gives the best average F-score.

Unwanted Applications and Typical Volta Usage: We show Taxonomist's ability to identify unknown applications from different domains by testing with unwanted applications such as bitcoin miners, shown in Fig. 3·6a, and with 6 months of Volta usage data, shown in Fig. 3·6b. In both of these tests, we train Taxonomist



(a) The importance of each statistical measure.

(b) The most important metric for each 11 applications and the metric's source subsystem.

Figure 3·7: The importance of different metrics and statistics. Box-plots are constructed using the different decision trees for each application. The box shows the quartiles while the whiskers show the rest of the distribution except outliers, which are points away from the low and high quartiles by more than $1.5 \times IQR$.

with the 11 representative applications, and consider the unknown label to be correct. Random Forest, Extra Trees and SVC have an almost perfect F-score for identifying any of these applications as unknown. Combs is not shown, because it is unable to identify unknown applications.

Feature Importance: In order to present the importance of different statistical features and metrics, we train a decision tree for each application, using all of the data from the 11 applications. To compare feature importances, we use *Gini reduction*, which is used to measure the reduction of heterogeneity in the data. A feature that can divide the data set well has a high Gini reduction, which means the resulting divided data sets are more homogeneous. We use the implementation in Python `scikit-learn` library (`sklearn.DecisionTreeClassifier.feature_importances_`).

In the decision trees corresponding to our 11 applications, we calculate the total Gini reduction of features extracted using the 11 statistics (§ 3.2.2), and report it in Fig. 3·7a. The box-plots are constructed using the data from the decision trees, and the individual importance values from the trees are summed up. Fig. 3·7b shows the

most important metric from each decision tree. The important metric and subsystem[3] are highly application specific.

3.5 Conclusion

We have presented Taxonomist, a technique for classifying applications in supercomputers with the help of readily available monitoring data. We evaluated our claim that each application has a unique resource usage fingerprint that can be used to differentiate applications. We described the technique, which builds classifiers from historical data, and detects new applications while being robust to new input configurations of applications.

We have evaluated Taxonomist using a comprehensive data set including controlled experiments and real-world workloads and demonstrated F-scores of over 95%. We have found that tree-based classifiers perform best, in terms of accuracy, with our multivariate time series data, and that the random forest is the best performing one.

We have shown that cryptocurrency miners and password crackers, which are typical unwanted applications for HPC systems, have very unique resource usage fingerprints compared to HPC applications, so they can be easily detected using the approach of Taxonomist.

[3] metric-set-nic: Cray network counters [Cray, 2018], vmstat: `/proc/vmstat`, meminfo: `/proc/meminfo`, procstat: `/proc/stat`, AR stands for AR-NIC-RSPMON-PARB-EVENT-CNTR

Chapter 4

Diagnosing Performance Variations in HPC Applications

Application performance variations are among the significant challenges in today's high performance computing (HPC) systems as they adversely impact system efficiency. For example, the amount of variation in application running times can reach 100% on real-life systems [Bhatele et al., 2013, Leung et al., 2003, Skinner and Kramer, 2005]. In addition to leading to unpredictable application running times, performance variations can also cause premature job terminations and wasted compute cycles. Common examples of *anomalies* that can lead to performance variation include orphan processes left over from previous jobs consuming system resources [Brandt et al., 2010], firmware bugs [Cisco, 2017], memory leaks [Agelastos et al., 2015], CPU throttling for thermal control [Brandt et al., 2015], and resource contention [Bhatele et al., 2013, Dorier et al., 2014, Leung et al., 2003]. These anomalies manifest themselves in system logs, performance counters, or resource usage data.

To detect performance variations and determine the associated root causes, HPC operators typically monitor system health by continuously collecting performance counters and resource usage data such as available network link bandwidth and CPU utilization. Hundreds of metrics collected from thousands of nodes at frequencies suitable for performance analysis translate to billions of data points per day [Agelastos et al., 2014]. As HPC systems grow in size and complexity, it is becoming increasingly impractical to analyze this data manually. Thus, it is essential to have tools that

automatically identify problems through continuous and/or periodic analysis of data.

In this chapter, we describe a machine learning framework that automatically detects compute nodes that have exhibited known performance anomalies and also diagnoses the type of the anomaly. Our framework avoids data deluge by using easy-to-compute statistical features extracted from applications' resource utilization patterns. We evaluate the effectiveness of our framework in two environments: a Cray XC30m machine, and a public cloud hosted on a Beowulf-like cluster [Sterling et al., 1995]. We demonstrate that our framework can detect and classify anomalies with an *F-score* above 0.97, while the *F-score* of the state-of-the-art techniques are between 0.89 and 0.97.

The rest of the chapter is organized as follows: Sec. 4.1 describes our machine-learning-based anomaly detection framework, Sec. 4.2 describes the state-of-the-art algorithms that we implement as baselines, Sec. 4.3 explains our experimental methodology, and Sec. 4.4 provides our experimental findings. Finally, we conclude in Sec. 4.5. An overview of related work can be found in Sec. 2.4.2.

4.1 Anomaly Detection and Classification

Our goal is to detect anomalies that cause performance variations and to classify the anomaly into one of the previously encountered anomaly types. To this end, we propose an automated anomaly detection technique, which takes advantage of historical data collected from known healthy runs and anomalies, and builds generic machine learning models that can distinguish anomaly characteristics in the collected data. In this way, we are able to detect and classify anomalies when running applications with a variety of previously unobserved inputs. With a training set that represents the expected application characteristics, our technique is successful even when a known anomaly impacts an application we have not encountered during training.

Directly using raw time series data that is continuously collected from thousands of nodes for anomaly detection can incur unacceptable computational overhead. This can lead to significant time gaps between data collection and analysis, delayed remedies, and wasted compute resources. Instead of using raw time series data, we extract concise statistical features that retain the characteristics of the time series. This significantly reduces our data set, thus decreasing the computational overhead and storage requirements of our approach. In this chapter we apply our anomaly diagnosis offline after application runs are complete, but we also demonstrate that online anomaly detection can be performed using a similar framework [Tuncer et al., 2019]. In the next subsections, we explain the details of our proposed approach on feature extraction and machine learning.

4.1.1 Feature Extraction

HPC monitoring infrastructures are rapidly evolving and new monitoring systems are able to periodically collect resource usage metrics (e.g., CPU utilization, available memory) and performance counters (e.g., received/transmitted network packets, CPU interrupt counts) during application runs [Agelastos et al., 2014]. This data provides a detailed view on applications' runtime characteristics.

While an application is running, we periodically collect resource usage and performance counter metrics from each node during the entire application run. Note that our technique is also applicable when metrics are collected for a sliding history window to investigate only recent data. The metrics we collect, as described in detail in Sec. 4.3.1, are not specific to any monitoring infrastructure and the proposed framework can be coupled with different HPC monitoring systems (e.g., [Ganglia, 2020, Nagios, 2020, Agelastos et al., 2014]). From the time series of collected metrics, we extract the following easy-to-compute features to enable fast anomaly analysis:

- Simple order statistics that help differentiate between healthy and anomalous

behavior: the minimum value, 5^{th}, 25^{th}, 50^{th}, 75^{th}, and 95^{th} percentile values, the maximum value, and the standard deviation;

- Features that are known to be useful for time-series clustering [Wang et al., 2006]:
  - *Skewness* indicates lack of symmetry. In a time series X_t, skewness S is defined by $S = \frac{1}{n\sigma^3} \sum_{t=1}^{n} \left(X_t - \overline{X_t}\right)^3$, where $\overline{X_t}$ is the mean, σ is standard deviation, and n is the number of data points.
  - *Kurtosis* refers to the heaviness of the tails of a distribution. The kurtosis coefficient is defined as $K = \frac{1}{n\sigma^4} \sum_{t=1}^{n} \left(X_t - \overline{X_t}\right)^4$.
  - *Serial correlation* measures the noisiness in given data, and can be estimated by the Box-Pierce statistic [Wheelwright et al., 1998].
  - *Linearity* is a measure of how well a time series can be forecasted with traditional linear models [Giannerini, 2012].
  - *Self-similarity* measures the long-range dependence, i.e., the correlation of X_t and X_{t+k} in time series X_t for large values of k.

The calculation of statistical features is a low-overhead procedure, and can be further optimized to work with data streams for on the fly feature generation. We provide an evaluation of the overhead of our implementation in Sec. 4.4.3.

4.1.2 Anomaly Diagnosis Using Machine Learning

Our machine-learning-based anomaly diagnosis approach is depicted in Fig. 4·1. As seen in the figure, during offline training, we run various types of applications (denoted as A, B, C in the figure) using different input sizes and input data (denoted with subscripts 1, 2, etc. in the figure). We gather resource usage and performance counter metrics from the nodes used by each application both when running without any

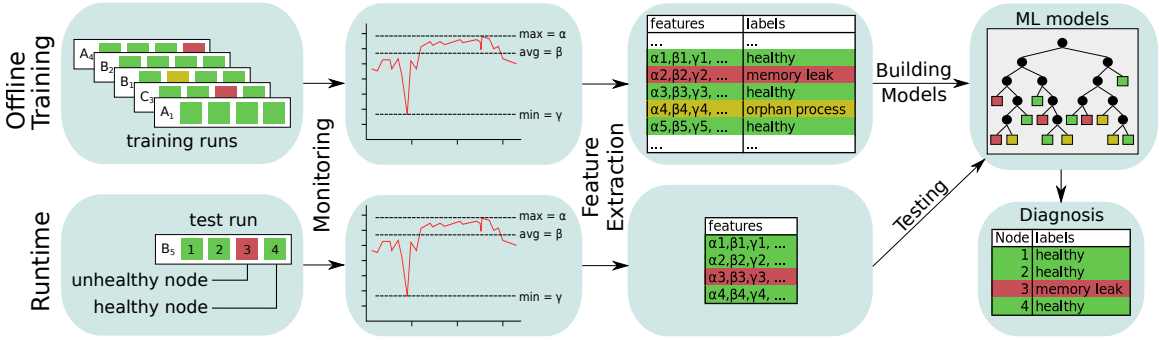


Figure 4·1: Overall system architecture. Machine learning models built offline are used for classifying observations at runtime.

anomaly and when we inject a synthetic anomaly to one of the nodes (see Sec. 4.3.2 for details on injected anomalies). When an application finishes executing, we compute statistical features using the metrics collected from individual nodes as described in Sec. 4.1.1. We label each node with the type of the introduced anomaly (or healthy). We use these labels and computed per-node features as input data to train various machine learning algorithms such as k-nearest neighbors and random forests. As machine learning algorithms do not use application type as input, they extract anomaly characteristics independent of applications.

At runtime, we again monitor application resource usage and performance counter metrics and extract their statistical features. We then use the machine learning models built during the training phase to detect anomalies and identify the types of anomalies in the nodes used by the application.

4.2 Baseline Methods

We implemented two state-of-the art methods as baselines of comparisons: the statistical approach proposed by Lan et al. [Lan et al., 2010] (referred as “ST-Lan”), and the fingerprinting approach of Bodik et al. [Bodik et al., 2010] (referred as “FP-Bodik”).

4.2.1 ST-Lan [Lan et al., 2010]

The core idea of ST-Lan is to detect anomalies based on distances between time series. ST-Lan applies Independent Component Analysis (ICA) to transform the monitored time series data into *independent components*, which represent the directions of maximal independence in data by using a linear combination of metrics. The first 3 independent components are used as a behavioral profile of a node. At runtime, the authors compare the behavioral profiles of the nodes that are used in new application runs to the profiles of known healthy nodes to identify whether the collected time series is an outlier using a distance-based outlier detection algorithm.

4.2.2 FP-Bodik [Bodik et al., 2010]

This method first divides each metric's time series into equal-sized epochs. Each epoch is represented by three values: 25^{th}, 50^{th}, and 95^{th} percentiles within that epoch. FP-Bodik further reduces data by selecting a subset of monitored metrics that are indicative of anomalies in the training set using logistic regression with L1 regularization. Next, a healthy range for the percentiles of each metric is identified using the values observed in healthy nodes while running applications. FP-Bodik then creates a *summary vector* for each percentile of each epoch based on whether observed metrics are within healthy ranges. The average of all summary vectors from a node constructs a *fingerprint* vector of the node. In order to find and classify anomalies, FP-Bodik compares L2 distances among these fingerprint vectors and chooses the nearest neighbor's category as the predicted anomaly type.

4.3 Experimental Methodology

Our experiments aim to provide a realistic evaluation of the proposed method in comparison with the baseline techniques. We run kernels representing common HPC

workloads and infuse synthetic anomalies to mimic anomalies observed in real-world HPC systems. This section describes our anomaly generation techniques, experimental environments, and the HPC applications we run in detail.

4.3.1 HPC Systems and Monitoring Infrastructures

We use two fundamentally different environments to evaluate our anomaly detection technique: a supercomputer, specifically a Cray XC30m cluster named Volta, and the Massachusetts Open Cloud (MOC), a public cloud running on a Beowulf-like [Sterling et al., 1995] cluster. We select these two environments as they represent modern deployment options for HPC systems.

Volta is a Cray XC30m cluster located at Sandia National Laboratories and accessed through Sandia External Collaboration Network[1]. It consists of 52 compute nodes, organized in 13 fully connected switches with 4 nodes per switch. The nodes run SUSE Linux with kernel version 3.0.101. Each node has 64 GB 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]. This service enables aggregation of a number of metrics from a large number of nodes. At every second, LDMS collects 721 different metrics as 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)

[1] `http://www.sandia.gov/FSO/docs/ECN_Account_Process.pdf`

- Cray performance counters (e.g., power consumption, dirty, writeback counters; received/transmitted bytes/packets)
- Aries network interface controller counters (e.g., received/transmitted packets, flits, blocked packets)

Massachusetts Open Cloud (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 [Mass Open Cloud, 2020].

In MOC, we use virtual machines (VMs) managed by OpenStack [Sefraoui et al., 2012], where the compute nodes are VMs running on commodity-grade 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, this infrastructure collects 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 [CCI-MOC, 2020].

4.3.2 Synthetic Anomalies

We focus on node-level anomalies that create performance variations. These anomalies can result from system or application-level issues. Examples of such anomalies are as follows:

- *Out-of-memory:* When the system memory is exhausted in an HPC platform, the Linux out-of-memory killer terminates the executing application. This is

typically caused by memory leaks [Agelastos et al., 2015].

- *Orphan processes:* When a job terminates incorrectly, it may result in orphan processes that continue using system resources such as memory and CPU [Brandt et al., 2010, Brandt et al., 2009].

- *Hidden hardware problems:* Automatic compensation mechanisms for hardware faults can lead to poor overall system performance. An example of such problems was experienced in Sandia National Laboratories' Redstorm system as slower performance in specific nodes, where several CPUs were running at 2.0 GHz instead of 2.2GHz [Snir et al., 2014].

We run synthetic anomalies on a single node of a multi-node HPC application to mimic the anomalies seen in real-life systems by stressing individual components of the node (e.g., CPU or memory), emulating interference or malfunction in that component. As synthetic anomalies, we use the following programs with two different *anomaly intensities*:

1. *leak:* This program allocates a 16 MB `char` array, fills the array with characters, and sleeps for two seconds in an infinite loop. The allocated memory is never released, leading to a memory leak. If the available system memory is consumed before the running application finishes, the *leak* program restarts. In the low intensity mode, a 4 MB array is used.

2. *memeater:* This program allocates a 36 MB `int` array and fills the array with random integers. It then periodically increases the size of the array using `realloc` and fills in new elements. After 10 iterations, the application restarts. In the low intensity mode, an 18 MB array is used.

3. *ddot:* This program allocates two equally sized matrices of `double` type, using `memalign`, fills them with a number, and calculates the dot product of the two

matrices repeatedly. We change the matrix size periodically to be 0.9, 5 and 10 times the sizes of the caches. It simulates CPU and cache interference by re-using the same array. The low intensity mode allocates arrays half the size of the original.

4. *dcopy:* This program again allocates two matrices of sizes equal to those of *ddot*, however it copies one matrix to the other one repeatedly. Compared to *ddot*, it has less CPU interference and writes back to the matrix.

5. *dial:* Repeatedly generates random floating point numbers, and performs arithmetic operations, thus stresses the ALU. In low intensity mode, the anomaly sleeps for 125ms every 250ms.

4.3.3 Applications

In order to test our system with a variety of applications, we use the NAS Parallel Benchmarks (NPB) [Bailey et al., 1991]. We pick five NPB applications (bt, cg, ft, lu and sp), with which we can obtain feasible running times (10-15 minutes) for three different custom input sizes. Each application run uses 4 nodes on Volta and 4 VMs on MOC. As some of our applications require the number of MPI ranks to be the square of an integer or to be a power of two, we adjust the number of ranks used in our experiments to meet these requirements and run applications with 64 and 16 ranks in Volta and MOC, respectively.

In our experiments, we run the selected 5 NPB applications for every combination of 3 different application input sizes and 20 and 10 randomized input data set in Volta and MOC, respectively. We repeat each of these runs 20 times: 10 without any anomaly, and 10 with one of the 4 application nodes having a synthetic anomaly for every combination of 5 anomaly types and 2 anomaly intensities. This results in 3000 application runs in Volta and 1500 in MOC, half of which use a single unhealthy node,

i.e., a node with an anomaly.

We have observed that for the application runs with a single unhealthy node, the characteristics of the remaining (i.e., healthy) nodes are more similar to the nodes in a completely healthy application run than to an unhealthy node. This is because even when the runtime of an application changes due to inclusion of an unhealthy node, the characteristics that we evaluate do not change significantly on the remaining healthy nodes for the applications we use.

4.3.4 Implementation Details

We implement most of our preprocessing and classification steps in Python. Before feature generation, we remove the first and last 30 seconds of the collected time series data to strip out the initialization and termination phases of the applications. Note that the empirical choice of 30 seconds is based on these particular applications and the configuration parameters used.

During preprocessing, we take the derivative of the performance counters so that the resulting metrics represent the number of events that occurred over the sample interval (e.g., interrupts per second). This is automatically done in MOC, and can be easily integrated into LDMS in Volta.

Proposed Framework: For feature generation, we use the Python `scipy-stats` package to calculate skewness and kurtosis. We use R to calculate Box-Pierce statistics, the `tseries` R package to calculate the Teräsvirta neural network test for linearity, and the `fracdiff` R package for self-similarity.

We evaluate the following machine learning algorithms: k-nearest neighbors, support vector classifiers with the radial basis function kernel, decision trees, random forests, and AdaBoost. We use Python's `scikit-learn` packages [Pedregosa et al., 2011] for the implementations of these algorithms.

ST-Lan: This algorithm uses the first $N = 3$ independent components determined

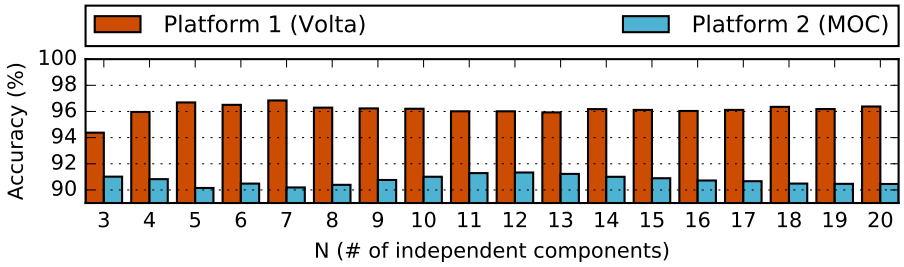


Figure 4·2: Classification accuracy of ST-Lan w.r.t. number of independent components used in the algorithm for the two platforms used in this study.

by ICA as the behavioral profile of a node. The first 3 independent components do not capture the independent dimensions in our data because the metric set we monitor is significantly larger than that used by Lan et al. As the authors do not provide a methodology to select N, we have swept N values within $[3, 20]$ range and compared accuracy in terms of the percentage of correctly labeled nodes on both of our experimental platforms as shown in Fig. 4·2. $N = 7$ and $N = 12$ provide the highest accuracy on Volta and MOC, respectively. We settled on $N = 10$ as it provides a good middle ground value that results in high accuracy on both platforms. In addition to selecting N, we extend ST-Lan to be able to do multi-class classification (i.e., to identify the type of anomaly) as well by using a kNN classifier instead of the distance-based outlier detection algorithm used by the authors.

FP-Bodik: This algorithm uses divides the collected metric time series into epochs before generating fingerprints. In their work [Bodik et al., 2010], Bodik et al. select the epoch length as 15 minutes with a sampling rate of a few minutes due to the restrictions in their monitoring infrastructure. In our implementation, we use the epoch length as 100 measurements, which corresponds to 100 seconds.

4.4 Results

We evaluate the detection algorithms using 5-fold stratified cross validation, which is a standard technique for evaluating machine learning algorithms and is performed as follows: We randomly divide our data set into 5 equal-sized partitions with each partition having data from a balanced number of application runs for each anomaly. We use a single partition for testing while using the other 4 disjoint partitions for training; and repeat this procedure 5 times, where each partition is used once for testing. Furthermore, we repeat the 5-fold cross validation 10 times with different randomly-selected partitions.

We calculate the average *precision* and *recall* for each class across all test sets, where the classes are the 5 anomalies we use and "healthy", and precision and recall of class C_i are defined as follows:

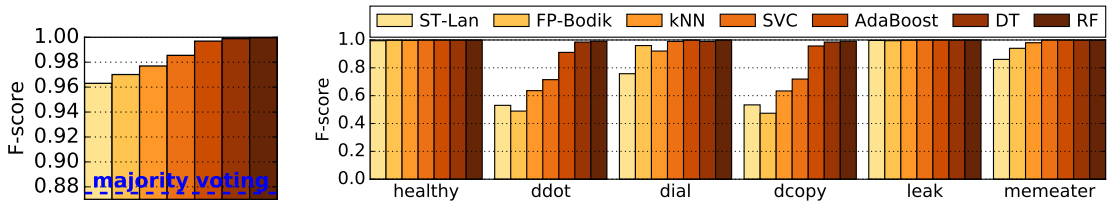
$$precision_{C_i} = (\#\text{ of correct predictions})_{C_i}/(\#\text{ of predictions})_{C_i} \tag{4.1}$$

$$recall_{C_i} = (\#\text{ of correct predictions})_{C_i}/(\#\text{ of elements})_{C_i} \tag{4.2}$$

For each class, we report *F-score*, which is the harmonic mean of precision and recall. In addition, we calculate an overall F-score for each algorithm as follows: We first calculate the weighted average of precision and recall, where the precision and recall of each class is weighted by the number of instances of that class in our data set. The harmonic mean of these weighted average values is the overall F-score.

We use the following classifiers in our machine learning framework: k-nearest neighbors (kNN), support vector classifier (SVC), AdaBoost, decision tree (DT), and random forest (RF).

The rest of this section begins with comparing anomaly detection techniques when the disjoint training and test sets include data from the same applications,



(a) Overall F-score

(b) F-scores for each anomaly

Figure 4·3: F-scores for anomaly classification in Volta. ST-Lan and FP-Bodik are baseline algorithms. Majority voting in (a) marks everything as "healthy."

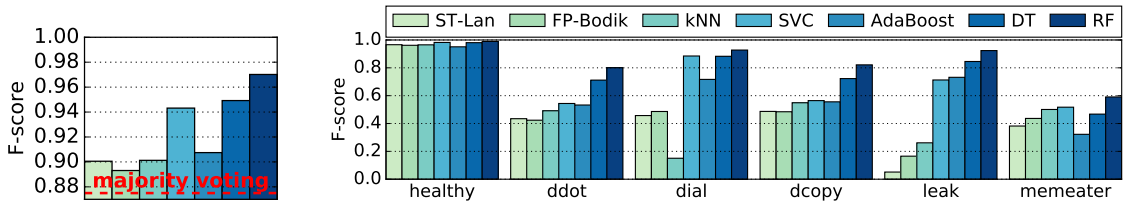
application input sizes, and anomaly intensities, but using different application input data. However, it is not a realistic scenario to know all the possible jobs that will run on an HPC system. Hence, in the following subsections, we evaluate the robustness of our approach and the baseline techniques to unknown application input sizes, unknown applications, and unknown anomaly intensities. Finally, we provide an experimental evaluation of the computational overhead of our anomaly detection approach.

4.4.1 Anomaly Detection and Classification

Figures 4·3 and 4·4 show the effectiveness of the anomaly detection approaches in terms of overall and per-anomaly F-scores in Volta and MOC environments, respectively. Note that half of our application runs use 4 healthy nodes and the other half use 3 healthy nodes and a single unhealthy node. Hence, the overall F-score of *majority voting*, which simply marks every node as "healthy", is 0.875 (represented by a dashed line in Fig. 4·3a and 4·4a).

In Volta, DT and RF result in close to ideal detection accuracy. As *ddot* and *dcopy* anomalies both stress caches, all algorithms tend to mislabel them as each other, resulting in lower F-scores.

The relatively poor performance on ST-Lan in Fig. 4·3 demonstrates the importance



(a) Overall F-score

(b) F-scores for each anomaly

Figure 4·4: F-scores for anomaly classification in MOC.

Table 4.1: The most important 10 features selected by RF in Volta.

source	feature
`/proc/stat`	avg_user
`/proc/stat`	perc5_idle
`/proc/stat`	perc95_softirq
`/proc/vmstat`	std_dirty_backgnd_thrshld
`/proc/stat`	perc25_idle
`/proc/vmstat`	std_dirty_threshold
`cray_aries_r`	std_current_freemem
`/proc/stat`	perc50_idle
`/proc/vmstat`	perc95_pgfault
`/proc/vmstat`	min_numa_hit

Table 4.2: The most important 10 metrics selected by ST-Lan in Volta.

source	metric
`nic`	WC_FLITS
`/proc/meminfo`	VmallocUsed
`nic`	WC_PKTS
`/proc/meminfo`	Committed_AS
`/proc/vmstat`	nr_page_table_pages
`/proc/meminfo`	PageTables
`/proc/meminfo`	VmallocChunk
`nic`	WC_BLOCKED
`/proc/vmstat`	nr_active_anon
`/proc/meminfo`	Active(anon)

of feature selection. ST-Lan leverages ICA for dimensionality reduction and uses features that represent the maximal independence in data but are not necessarily relevant for anomaly detection. Table 4.1 presents the most useful 10 features selected by random forests based on the normalized total Gini reduction brought by each feature as reported by Python `scikit-learn` package. For comparison, we present the metrics with the 10 highest absolute weight in the independent components used in ST-Lan in Table 4.2. Indeed, none of the top-level metrics used by ST-Lan is used in the most important features of RF.

In MOC, however, the important metrics in the independent components match with the important features of RF as shown in Tables 4.3 and 4.4. The reason is that we collect 53 metrics in MOC compared to 721 metrics in Volta; and hence, there is a higher overlap between the metrics in the first 10 independent components and those

Table 4.3: The most important 10 features selected by RF in MOC.

source	feature
/proc/meminfo	std_free
/proc/meminfo	std_used
/proc/stat	avg_cpu_idle
vmstat	std_free_memory
/proc/meminfo	std_freeWOBuffersCaches
/proc/meminfo	std_used_percentage
vmstat	perc75_cpu_user
/proc/stat	max_cpu_idle
/proc/meminfo	std_usedWOBuffersCaches
/proc/stat	perc75_cpu_idle

Table 4.4: The most important 10 metrics selected by ST-Lan in MOC.

source	metric
vmstat	cpu_user
/proc/stat	cpu_user
iostat	user
vmstat	cpu_idle
iostat	idle
vmstat	cpu_system
iostat	system
/proc/stat	cpu_system
/proc/meminfo	freeWOBuffersCaches
/proc/meminfo	usedWOBuffersCaches

selected by decision trees. As the metric space increases, the independent components become less relevant for anomaly detection.

The overall detection performance in MOC is lower for all algorithms. There are 4 main factors that can cause the reduced accuracy in MOC: the number of collected metrics, dataset size, sampling frequency, and platform-related noise. To measure the impact of the difference in the metric set, we choose 53 metrics from the Volta dataset that are closest to the MOC metrics and re-run our analysis with the reduced metric set. This decreases F-score by 0.01 for SVC and kNN and poses no significant reduction for DT, RF, and AdaBoost. Next, we reduce the size of the Volta dataset and use 5 randomized application input data instead of 10. The combined F-score reduction due to reduced dataset size and metric set is around 0.02 except for DT,

RF, and AdaBoost, where the F-score reduction is insignificant. We also measure the impact of data collection period by increasing it to 5 seconds; however, the impact on classification accuracy is negligible. 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 and migration.

Considering both MOC and Volta results, our results indicate that RF is the best-performing algorithm with overall F-scores between 0.97 and 1.0 on both platforms, while the baselines have overall F-scores between 0.89 and 0.97. This is consistent with other work that uses system telemetry data from HPC systems. One reason is because the random forest operates by applying thresholds to individual metrics, which is a natural fit for HPC telemetry data, since thresholding is a commonly used method for heuristics as well.

4.4.2 Classification with Unknown Applications, Inputs and Anomaly Intensities

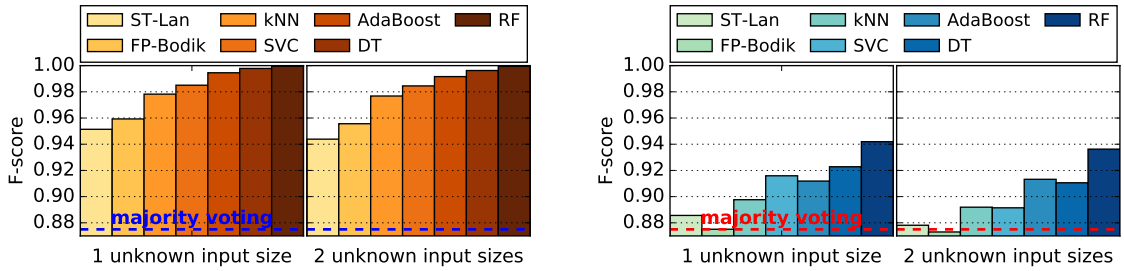
Classification with Unknown Application Inputs: In a real-world scenario, we expect to encounter application input sizes other than those used during training. This difference between training and test sets can result in observing application resource usage and performance characteristics that are new to the anomaly detection algorithms. To evaluate the robustness of our approach against input sizes that have not been encountered before, we modify our training and test sets in our 5-fold cross validation, where we remove an unknown input size from all training sets and the other input sizes from all test sets. We repeat this procedure 3 times so that all input sizes are selected as the unknown size once. We also evaluate detection algorithms when two input sizes are simultaneously removed from the training sets, for all input size combinations.

Figure 4·5 presents the overall F-score achieved by anomaly detection algorithms for

unknown input sizes. As we train the algorithms with a smaller variety of application input sizes, their effectiveness decrease as expected. In MOC, FP-Bodik's F-score decreases down to the majority voting level. However, the proposed machine learning approach consistently outperforms the baselines, with RF keeping its near-ideal accuracy in Volta.

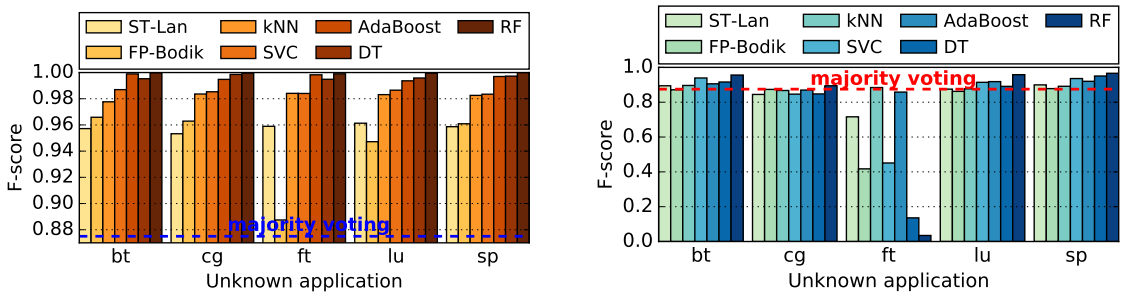
Classification with Unknown Applications: In order to evaluate how well our anomaly detection technique identifies anomaly characteristics independent of specific applications, we remove all runs of an application from the training sets, and then, remove all the other applications from the test sets. We repeat this procedure for all 5 applications we use.

Figure 4·6 shows the overall F-score of the detection algorithms for each unknown application. The most prominent result in the figure is that most algorithms have very poor classification accuracy in MOC when the unknown application is *ft*. Figure 4·7a illustrates how *ft* is different from other applications in terms of the most important two features used by DT to classify healthy runs. When not trained with *ft*, DT uses the threshold indicated by the dashed line to identify the majority of the healthy nodes, which results in most healthy *ft* nodes being marked as unhealthy. In Volta, however, the data has less noise due to the absence of VM interference and the number



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

Figure 4·5: Overall F-score when the training data excludes one or two input sizes and the testing is done using only the excluded input sizes.



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

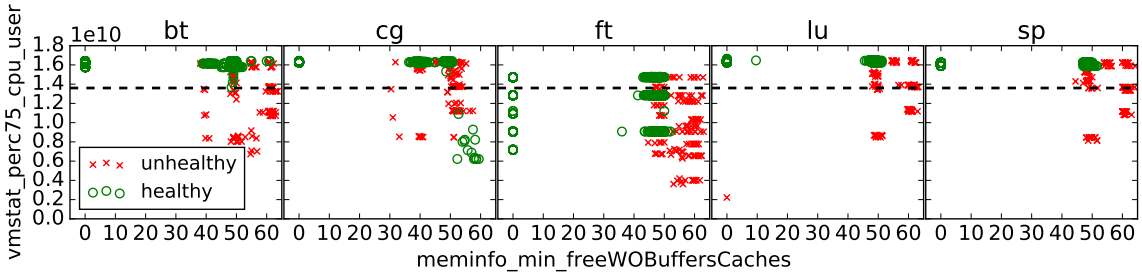
Figure 4·6: Overall F-score when the training data excludes one application and the testing is done using only the excluded application.

of metrics is significantly larger. Hence, DT is able to find more reliable features to classify healthy runs as depicted in Fig. 4·7b.

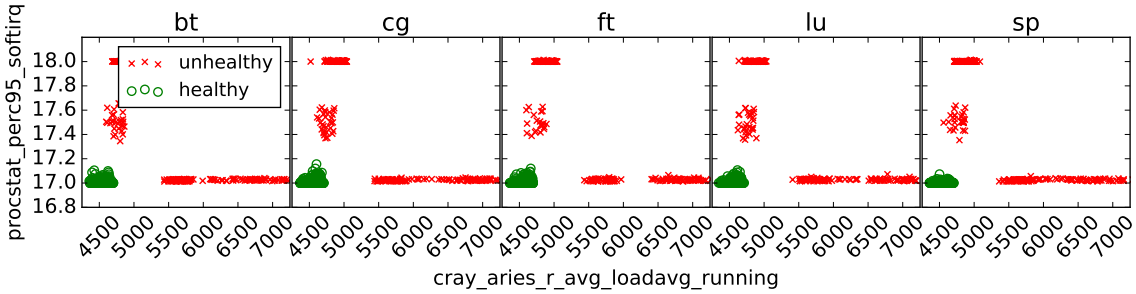
Figure 4·6 shows that the *F-score* of FP-Bodik also decreases significantly in both Volta and MOC when *ft* is the unknown application. This is because when not trained with *ft*, the generated fingerprint of the *memeater* anomaly by FP-Bodik is similar to the fingerprint of healthy *ft*, resulting in FP-Bodik marking healthy *ft* nodes as *memeater*.

These examples show that when the training set does not represent the expected application runtime characteristics, both our framework and the baseline algorithms may mislabel the nodes where unknown applications run. To avoid such problems, a diverse and representative set of applications should be used during training.

Classification with Unknown Anomaly Intensities: We evaluate the robustness of the anomaly detection algorithms when they encounter previously-unknown anomaly intensities. Thus, we train the algorithms with data collected when running with either high- or low-intensity anomalies test with the other intensity. Figure 4·8 shows the resulting F-scores in Volta and MOC environments. When the detection algorithms are trained with anomalies with high intensity, the thresholds placed by the algorithms are adjusted for highly anomalous behavior. Hence, when tested with

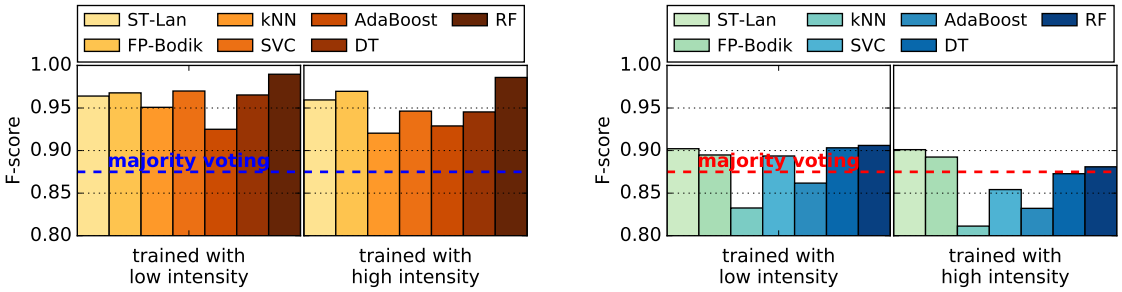


(a) MOC. When ft is excluded from the training set, DT classifies runs below the dashed line as unhealthy, which causes healthy ft nodes to be classified as unhealthy.



(b) Volta. The distinction between healthy and unhealthy clusters is clearly visible.

Figure 4·7: The scatter plots of the datasets for the most important two features used by DT to classify healthy data.



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

Figure 4·8: Overall F-score when the training data excludes one anomaly intensity and the testing is done using only the excluded anomaly intensity.

low anomaly intensity, the algorithms misclassify some unhealthy nodes as healthy, leading to a slightly lower F-score. The baseline algorithms demonstrate a more robust behavior against unknown anomaly intensities compared to our approach except for RF, which outperforms the baselines on Volta and performs similarly on MOC when

trained with low anomaly intensity.

4.4.3 Overhead

In our framework, the most computationally intensive part is feature generation. Generating features for a 900-second time window in Volta, i.e., from a 48-thread server for 721 metrics with 1 second sampling period, takes 10.1 seconds on average using a single thread. This translates into 11ms single-thread computational overhead per second to calculate features for the metrics collected from a 48-thread server. Assuming that these features are calculated on the server by monitoring agents, this corresponds to a total of $11/48 = 0.23ms$ computational overhead per second (0.02%) on Volta servers. Performing classification with trained machine learning algorithms takes approximately 10ms and this overhead is negligible compared to application running times. With our implementations, the classification overheads of FP-Bodik and ST-Lan are 0.01% and below 0.01%, respectively. The training overhead of both the machine learning algorithms and the baseline algorithms is negligible as it can be done offline.

Regarding the storage savings, the data collected for a 4-node 15-minute run on Volta takes 6.2 MB as raw time series, and only 252 KB as features (4% of the raw data). This number can be further reduced for tree-based classifiers by storing only the features that are deemed to be important by the classifiers.

4.5 Conclusion

Performance variation is an important factor that degrades efficiency and resiliency of HPC systems. Detection and diagnosis of the root causes of performance variation is a hard task due to the complexity and size of HPC systems. Because of this, performance variation is a major challenge in today's production systems.

In this chapter, we present an automated, low-overhead, and highly-accurate frame-

work for detection and identification of anomalies in HPC systems. Our framework works by extracting statistical features from readily available system telemetry data and using machine learning models to learn performance variations. We evaluate our proposed framework on two fundamentally different platforms and demonstrate that our framework is superior to other state-of-the-art approaches in detecting and diagnosing anomalies, and robust to previously unencountered applications and application characteristics.

Chapter 5

HPAS: An HPC Performance Anomaly Suite for Reproducing Performance Variations

In the previous chapter (Chapter 4), we introduced an automated framework that can detect and classify performance variations. In order to evaluate this method, we developed our synthetic performance variation generators. Due to the lack of an open-source, widely-applicable method for reproducing realistic scenarios that create performance variability, it is common for researchers to develop their own synthetic reproduction of performance variations (i.e., anomaly injectors) [Lan et al., 2010, Kasick et al., 2010, Tuncer et al., 2017, Tuncer et al., 2019]. This lack of a common methodology for generating realistic performance variation causing anomalies, results in a fragmented research space, difficulty in repeatability/comparison of results across different research teams, and loss of valuable research and system time.

In this chapter, we introduce HPAS, a new *HPC Performance Anomaly Suite*[1], with the goal of enabling researchers, engineers and administrators to repeatably and systematically study realistic performance variability in HPC systems. We follow the scientific intuition that standardized benchmarks play an important role in the development of computer hardware and software as well as in the evaluation of middleware and policies. Such benchmarks relieve engineers and scientists from the

[1]The source code and additional documentation of the anomalies described are available at `www.github.com/peaclab/HPAS`.

burden of developing representative workloads. In addition, as discussed above, coming up with realistic examples or use cases may often be difficult—or impossible—for many researchers. Using benchmarks, researchers can compare different approaches in computer systems in a fair manner and help advance science. It is our intent to advance the state-of-the-art in reproducible HPC research with this anomaly suite contribution.

Our anomaly suite, HPAS, consists of a set of synthetic "anomalies" that reproduce common root causes of performance variations in supercomputers: CPU contention, cache evictions, memory bandwidth interference, memory intensive processes, memory leaks, network contention, and contention in the shared file system metadata servers and storage servers. We design these synthetic anomalies using processes that run in user space; thus, our suite does not require modifications to hardware, any other applications, or the kernel. Our anomalies can be configured for various intensities at runtime using several knobs. Furthermore, each anomaly is designed to minimize its interference in the subsystems that it is not targeting.

The remainder of this chapter first discusses each of the HPAS anomaly generators targeting individual subsystems, (§ 5.1); we follow with an analysis of the characteristics of each of these anomalies by themselves and their impact on various co-located applications (§ 5.2); and demonstrate possible uses of these anomalies, including generating synthetic data for the evaluation of anomaly diagnosis methods and comparing the effects of performance variability on load balancing and system management policies (§ 5.3).

5.1 Synthetic Anomalies

Our anomaly suite, HPAS, implements eight performance anomalies, shown in Table 5.1, in order to replicate the types of performance variability mentioned above.

Table 5.1: A list of HPAS anomalies and their details. Every anomaly has configurable start/end times as well.

Anomaly type	Anomaly name	Anomaly behavior	Runtime configuration options
CPU intensive process	CPUOCCUPY	Arithmetic operations	utilization %
Cache contention	CACHECOPY	Cache read & write	cache (L1/L2/L3), multiplier, rate
Memory bandwidth contention	MEMBW	Uncached memory write	buffer size, rate
Memory intensive process	MEMEATER	Allocate, fill, & release memory	buffer size, rate
Memory leak	MEMLEAK	Increasingly allocate & fill memory	buffer size, rate
Network contention	NETOCCUPY	Send messages between two nodes	message size, rate, number of tasks (ntasks)
I/O metadata server contention	IOMETADATA	File creation & deletion	rate, ntasks
I/O bandwidth contention	IOBANDWIDTH	File read & write	file size, ntasks

Each anomaly targets a single subsystem and the behavior can be configured to change the intensities of the anomalies.

Our goal when designing HPAS is to accurately reproduce performance variations that are commonly encountered in HPC systems. This section first describes our design constraints for the synthetic anomalies and then provides the details of each synthetic anomaly. We select the sources of interference and design the anomalies based on both discussions with experts in and a literature review (Section 2.2). Four of the anomalies we present are based on those used in our previous work [Tuncer et al., 2017, Tuncer et al., 2019], which have also been used by Netti et al. [Netti et al., 2019] (CPUOCCUPY, CACHECOPY, MEMLEAK, MEMEATER in our suite).

When designing the anomalies, we seek to balance the usability of the anomalies with realistic reproduction. We implement all anomalies such that no modifications to the benchmark applications, shared libraries, operating system kernel, drivers or supercomputer hardware are required. For example, even though a memory leak could be more realistically reproduced by modifications to application code, such an

approach would require separate modifications to each benchmark and would reduce the reusability of the anomaly suite. Instead, we use a separate user space process that has a similar impact on the system.

We also design the anomalies such that the intensity of the anomaly can be adjusted using command line options. For example, for anomalies that require memory allocation, the amount and rate of memory allocated is adjustable; or, for some anomalies, a variable amount of sleep is inserted between periods of activity to reduce the intensity of the anomaly. This configurability also enables composing more complicated variability patterns (e.g., [Kuo et al., 2014]) by using multiple anomaly instances.

We consider each of the major subsystems in an HPC system, i.e., the CPU, the cache hierarchy, the memory, the high speed network, and the storage system. For each subsystem, we design synthetic anomalies that replicate known causes of performance variations in that subsystem. Table 5.1 provides a summary of our anomalies that will be elaborated upon in the following subsections.

5.1.1 CPU

We model CPU-based performance variability with the CPUOCCUPY anomaly. This anomaly performs arithmetic operations on random values in a loop and sleeps for a given percentage of the time, using SETITIMER(). In this way, the activity of the anomaly has negligible impact on the cache or memory, and the utilization of the CPU can be adjusted to a given percentage. The CPU consists of many components that may independently affect performance; however, the contention in HPC systems is typically between separate processes, and different processes contend for CPU time, which can be adequately reproduced using CPUOCCUPY.

The CPUOCCUPY anomaly can be executed on the same node with the application to emulate CPU contention, which may be caused by system processes or CPU-

intensive orphan processes, or it can emulate OS jitter by setting the consumed CPU time to a low value and impacting the scheduling behavior of the OS.

5.1.2 Cache Hierarchy

We model cache-related performance variations with the CACHECOPY anomaly that intensively uses the cache. The anomaly generator allocates two arrays, each of which are half the size of the L1, L2 or L3 caches, based on user-chosen parameters and repeatedly copies the contents of one array to the other one. The two arrays are contiguous in memory and are allocated using POSIX_MEMALIGN(). In this way, the specific level of the cache is effectively utilized by the anomaly, and the cache lines belonging to applications that share the same level of cache as the anomaly are expected to be frequently evicted.

CACHECOPY can be used to emulate cache contention, other hardware or software problems that may cause cache lines to be unexpectedly evicted, or running on a machine with a smaller cache.

5.1.3 Memory

We create three synthetic anomalies to model memory-related performance variability: Memory-intensive orphan processes, memory leak, and memory bandwidth contention. These anomalies can be used to mimic different types of memory contention or dead memory regions.

Memory Intensive Process: The MEMEATER anomaly allocates an array of a given size (35 MB by default, but adjustable) and fills it with random values. Later, it uses REALLOC() to increase the array's size by the same amount, fills the remaining area with random values, and repeats until the time or size limit given by the user is reached.

Memory Leaks: We model memory leaks using the MEMLEAK anomaly, which

```
#include <xmminitrin.h>
void temporal_copy(double **orig, double **swap) {
  for (int i = 0; i < SIZE; i++) {
    for (int j = 0; j < SIZE; j++) {
      _mm_stream_pi((__m64 *) (&swap[j][i]),
        *(__m64 *) (&orig[i][j])); // MOVNTQ
      _mm_empty(); // EMMS
    }
  }
}
```

Figure 5·1: A C code sample for creating memory bandwidth contention.

allocates an array of characters of a given size (20 MB by default) and fills it with random characters in each iteration. The addresses of the arrays are not stored and are not freed at each iteration, causing a memory leak.

Memory Bandwidth: The MEMEATER anomaly uses a large amount of cache as well, so we also design MEMBW to create contention only in the memory bandwidth. We model memory bandwidth contention by using the x86 SSE non-temporal memory instructions such as MOVNT*. These instructions are accessible from intrinsic functions which are supported by most compilers. When the data is marked with the non-temporal hint, i.e., that it will only be used once, it is not loaded into the cache. Our anomaly, MEMBW, first allocates two 2D matrices in the stack and fills one of them with random values. Then, it writes the transpose of the first matrix into the second matrix using the non-temporal hint, as shown in Figure 5·1. The transpose operation repeats for the duration of the anomaly.

5.1.4 Network

There are several methods for implementing inter-node communication in supercomputers, and when designing the NETOCCUPY anomaly we choose the method that introduces the least software overhead and the most emphasis on the network. MPI is the dominant parallel programming model, and many MPI implementations offer OS-bypass and other optimizations, allowing for faster communication compared to

using raw sockets. However, we choose the SHMEM API since it has been shown to have a lower latency, thus higher load on the network, compared to MPI on the Cray Aries network [Alverson et al., 2012], and it also offers similar optimizations as MPI. We focus on the Cray Aries because it is one of the most commonly used networks in the top 10 computers in the Top500 list (tied with Mellanox EDR Infiniband).

Our network interference generator can be executed in any two nodes provided that the link or router to be congested lies in the main communication path between the nodes. The anomaly then pairs the ranks on either node, such that the ranks on one node send messages to their corresponding rank on the other node using SHMEM_PUTMEM(). We use 100 MB messages because we observe that using messages smaller than 100 MB results in less contention, while messages larger than 100 MB do not noticeably increase bandwidth usage of the anomaly.

The main usage of the NETOCCUPY anomaly is to emulate network contention. The bandwidth consumption of the anomaly can be tuned to emulate different levels of network contention.

5.1.5 Shared Storage

We target a common shared file system architecture, where there are one or a few metadata servers that manage the creation/deletion of files and other metadata such as locks and the locations and permissions of the files. Each metadata operation first passes through these metadata servers, and the actual contents of the files are located in storage nodes. The communication between the file system and the compute nodes is performed using either a separate network or the same interconnect that is used for inter compute-node communication.

Using the POSIX API, we can stress both the metadata servers and the storage servers separately; therefore, we design two anomalies. The metadata server is stressed using the IOMETADATA anomaly that creates and opens files, writes one character

to each in a loop, closes all open files, and deletes them after 10 iterations. The IOBANDWIDTH anomaly uses `dd` [IEEE and The Open Group, 2018] to copy random data into a file. It then copies that file to another file and so on. This anomaly causes contention in the disks of the storage servers, as well as the interconnect between the file system and compute nodes. Both of these anomalies can be used standalone, or they can be started using MPI to achieve higher contention, in which case they use separate files for each rank.

5.2 Evaluation

To evaluate the proposed anomaly suite, we inspect the effect of each anomaly on target subsystems and different applications. We run our experiments on two systems: Voltrino, a Cray XC40m supercomputer located at Sandia National Laboratories, and Chameleon Cloud [Keahey et al., 2018] (which we use as a cluster of bare-metal servers). Voltrino has 24 nodes with two Intel Xeon E5-2698 v3 processors with 16 cores per socket and 24 nodes with one Intel Xeon Phi 7250 processor with 68 cores. We run all of the experiments on Voltrino using the nodes with Haswell Xeon E5-2698. Our Chameleon Cloud (CC) experiments use two 12-core Intel Xeon E5-2670 v3 processors per node. Both systems have 125 GB memory per node.

On Voltrino, we collect monitoring data using the open-source Lightweight Distributed Metric Service (LDMS) [Agelastos et al., 2014]. LDMS on Voltrino uses several samplers: PROCSTAT collects processor metrics from /PROC/STAT; MEMINFO and VMSTAT collects memory metrics from /PROC/MEMINFO and /PROC/VMSTAT, respectively; ARIES_NIC_MMR collects hardware counters from the Aries Network Interface Cards (NICs); CRAY_ARIES_R collects Cray-specific hardware counters; and SPAPIHASW collects hardware counters using PAPI [Terpstra et al., 2010]. In the rest of the chapter, we indicate the sampler for each metric using '::'. For example,

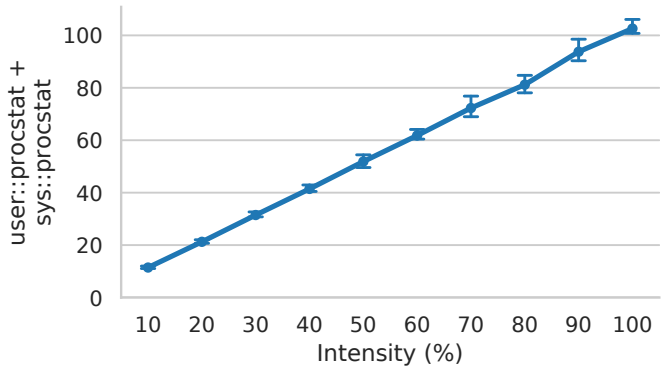


Figure 5·2: CPUOCCUPY intensity vs. CPU utilization in Voltrino. CPUOCCUPY uses the given percentage of the CPU.

USER::PROCSTAT indicates the metric USER from /PROC/STAT. In Voltrino, LDMS is configured to collect 2121 metrics per second from each node in our experiments.

5.2.1 Effects of the Anomalies on Their Respective Subsystems

We evaluate the effectiveness of each anomaly on its target subsystems in this section. Figure 5·2 shows the total CPU utilization in one node (i.e., USER::PROCSTAT + SYS::PROCSTAT) against the chosen intensity for CPUOCCUPY. Aside from the variability caused by the operating system, CPUOCCUPY can accurately use the given percentage of the CPU, and can be used to model CPU contention. The results from CC agree with Voltrino results, CPUOCCUPY can accurately consume the indicated percentage of CPU time.

The effects of CACHECOPY are demonstrated in Figure 5·3. For this experiment, a single-rank instance of miniGhost [Heroux et al., 2009] and CACHECOPY is placed on the same physical core, but two different logical cores using hyperthreading, causing them to share L1, L2 and L3 caches. We increase the working set size of the anomaly from the size of L1 cache to the size of L3. As the working set size is increased, more last level cache misses are observed for miniGhost. As CC has a smaller L3 cache than Voltrino, it suffers from more L3 cache misses with the anomaly.

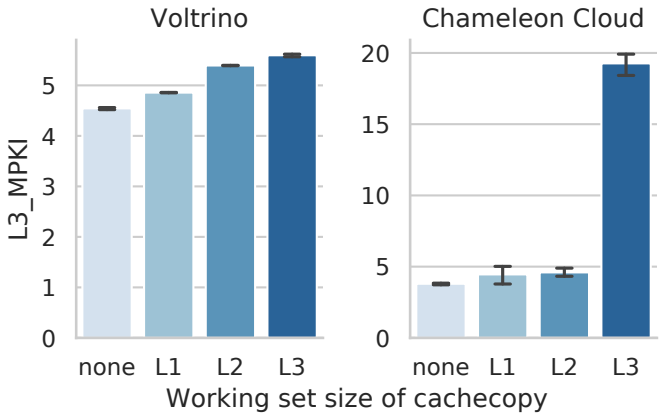


Figure 5·3: CACHECOPY vs. L3 misses per 1000 instructions (MPKI) in miniGhost in Voltrino and Chameleon Cloud.

Figure 5·4 shows the memory bandwidth as measured by the STREAM benchmark [McCalpin, 1995] in presence of the MEMBW or CACHECOPY anomalies. We place STREAM on core 0 and place the anomalies on cores other than 0 until we use all the other 15 cores of the socket for the anomaly. We also report results for CACHECOPY, which has negligible effect on memory bandwidth as expected, even though it uses 15 cores. The results from CC agree with those from Voltrino.

We show the memory behavior over time for MEMLEAK and MEMEATER in Figure 5·5. While MEMEATER behaves like a memory-intensive application and allocates a large amount of memory at initialization, it does not increase the total memory footprint. On the other hand, MEMLEAK displays the typical pathological memory allocation pattern that keeps increasing. Both anomalies terminate after the given duration. The amount of memory allocated and the behavior over time can be tuned in our anomaly generators. The results from CC agree with those from Voltrino.

To quantify the effectiveness of the NETOCCUPY anomaly, we measure the bandwidth between two nodes in two different switches in Voltrino using the OSU benchmark [Panda et al., 2018], as shown in Figure 5·6. The Aries interconnect has 4 nodes connected to each switch; thus, we allocate the remaining 6 nodes for the network

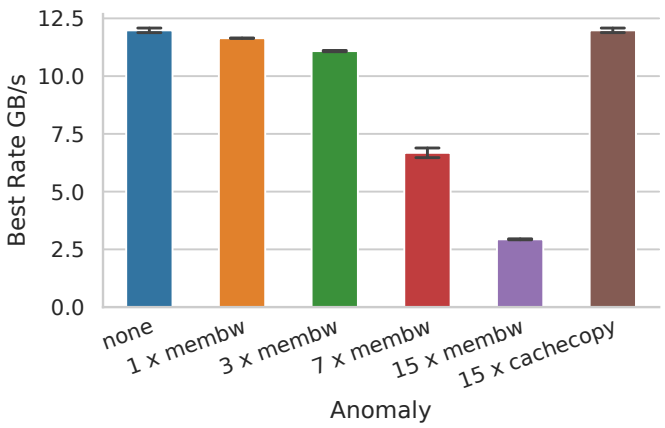


Figure 5·4: MEMBW and CACHECOPY effects on memory bandwidth on Voltrino. As expected, CACHECOPY has no significant impact on memory bandwidth while MEMBW significantly reduces memory bandwidth available to the application.

anomalies. We use 1, 2, and 3 pairs of nodes for the anomaly (corresponding to 2, 4, 6 nodes in the figure). The anomalies reduce the effective bandwidth of the OSU benchmark. Note that we use Cray MPI's MPICH_GNI_GET_MAXSIZE parameter to observe the effect of network congestion for smaller message sizes. The reduction of bandwidth is limited because of the topology of Voltrino, which has many redundant links and uses adaptive routing to avoid congested links. Another consequence of this adaptive routing is the possible congestion caused in links not directly targeted by the anomaly. We cannot evaluate the network anomaly in CC because of its simple star network topology, which means that the network links are only between the single router and the nodes.

For evaluation of the I/O anomalies, we only use Chameleon Cloud (CC) because the file system in Voltrino is shared by many systems other than Voltrino, resulting in inherent performance variability even when there are no applications running on Voltrino. Furthermore, our initial experiments using 256 instances of IOMETADATA and IOBANDWIDTH anomalies caused outages in Voltrino's Lustre file system. On CC,

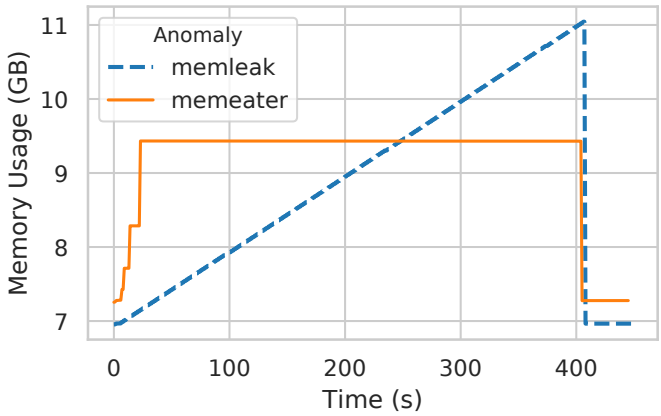


Figure 5·5: Memory usage over time for MEMLEAK and MEMEATER on Voltrino.

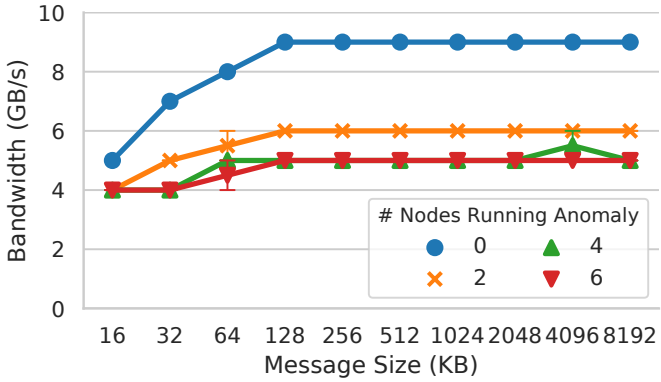


Figure 5·6: Message size of the OSU benchmark vs. network bandwidth in presence of NETOCCUPY anomaly in Voltrino.

we use the "Network File System (NFS) share" complex appliance of CC[2] to set up one NFS server and five clients. The storage server has one 250 GB ST9250610NS disk, and has the same CPU as the compute nodes. We run the IOMETADATA or IOBANDWIDTH anomalies on four nodes (48 instances per node) while measuring file system performance by running the IOR application [Lawrence Livermore National Laboratory, 2018] on the remaining node. Figure 5·7 demonstrates that IOBANDWIDTH reduces the effective bandwidth of IOR placed in the other node by clogging the disk on

[2] `https://www.chameleoncloud.org/appliances/25/`

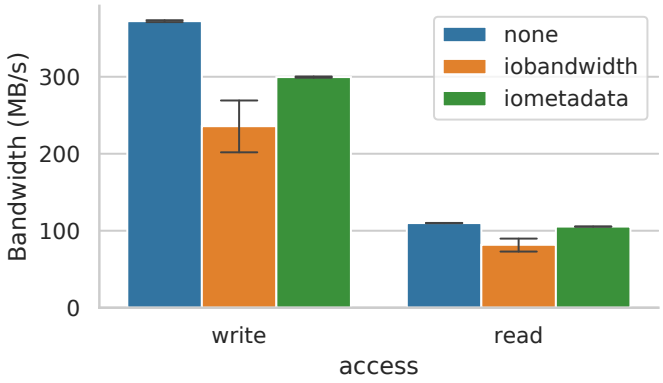


Figure 5·7: Impact of I/O anomalies when run on Chameleon Cloud.

Table 5.2: Characteristics of the benchmark applications.

	Cloverleaf	CoMD	Kripke	MILC	miniAMR	miniGhost	miniMD	SW4lite
CPU-intensive		✓					✓	✓
Memory-intensive	✓			✓	✓	✓		
Network-intensive	✓		✓	✓	✓	✓		

the storage node. The IOMETADATA anomaly also affects the bandwidth, since the CC file system does not have a separate metadata server. The impact of IOBANDWIDTH is higher in our case because the NFS server is using a single disk and 24 threads for metadata operations.

5.2.2 Effects of the Anomalies on HPC Applications

We analyze the impact of our synthetic anomalies on a diverse set of eight benchmark applications shown in Table 5.2. Among them, Cloverleaf, CoMD, miniAMR, miniGhost, and miniMD are from the Mantevo Benchmark Suite [Heroux et al., 2009], which are proxy applications mimicking different scientific computation kernels. Kripke is a proxy application for a particle transport simulation developed to study the performance characteristics of data layouts and sweep algorithms [Kunen et al., 2015]. MILC represents part of the codes written by the MIMD Lattice Computation collaboration to study quantum chromodynamics [The MIMD Lattice Computation (MILC) Collaboration, 2016]. SW4lite is also a proxy application containing the computational

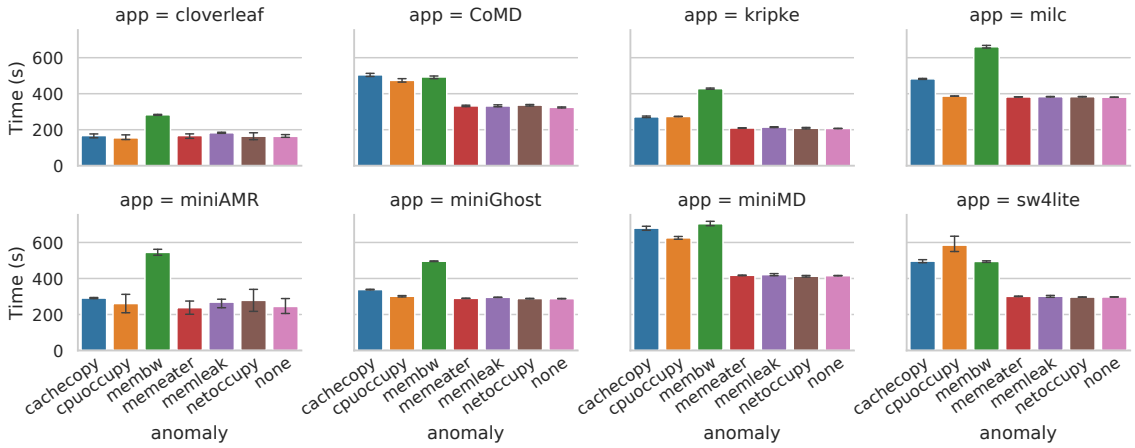


Figure 5·8: Execution time of each application with each anomaly on Voltrino.

kernels of SW4 which solves an elastic wave equation for seismic simulations [Sjogreen, 2018].

To first understand how intensively the benchmark applications use certain system resources, we analyze the characteristics of the selected benchmark applications based on collected performance metrics (without any anomalies). We evaluate CPU-intensiveness by instruction per second (IPS) through the metric INST_RETIRED:ANY::SPAPIHASW; we evaluate memory-intensiveness by observing cache misses through the metric L2_RQSTS:MISS::SPAPIHASW. We evaluate network-intensiveness by a network traffic counter through the metric AR_NIC_NETMON_ORB_EVENT_CNTR_REQ_FLITS::ARIES_NIC_MMR. Based on our analysis, we summarize the characteristics of the benchmark applications in Table 5.2.

Figure 5·8 shows the running time of applications when they are run with the anomalies. We use application running time as a measure of application performance. Each anomaly affects application performance in different ways. The anomalies that affect performance the most are CACHECOPY, CPUOCCUPY and MEMBW. For example, CPU-intensive applications, including CoMD, miniMD, and SW4lite, are all heavily affected by CACHECOPY and CPUOCCUPY. The memory-intensive applications,

including Cloverleaf, MILC, miniAMR, and miniGhost, are more impacted by MEMBW than other anomalies. None of the applications are affected significantly by the network anomaly because of the highly connected network of Voltrino, that is designed for much larger supercomputers with adaptive routing. Also, memory anomalies such as MEMLEAK and MEMEATER do not visibly affect performance because Voltrino does not use swap and applications are killed when they run out of memory. Indeed, if the size of the memory anomalies are set too large, they result in application crashes.

5.3 Use Cases for HPAS

In this section, we show, using experiments on Voltrino, that our anomaly suite can be used in the following three example cases: (1) Evaluate tools that diagnose performance deviation on HPC systems, (2) systematically evaluate the performance of system management policies under the conditions of resource contention, and (3) develop applications and systems resilient to performance variability. We envision that the usage of HPAS will be advantageous in many other performance or resilience studies as well.

5.3.1 Evaluating Anomaly Diagnosis Tools

Anomaly detection and diagnosis methods are typically based on machine learning algorithms and require a considerable amount of training data to use and evaluate [Tuncer et al., 2017, Tuncer et al., 2019, Klinkenberg et al., 2017, Kasick et al., 2010, Lan et al., 2010]. Since collecting ground truth anomaly data on HPC systems is not an easy task, data generated using our anomaly suite can be used instead. Furthermore, using the same methods for anomaly generation can make it easier for researchers to compare anomaly diagnosis methods.

To demonstrate the use of our anomaly suite in evaluating anomaly diagnosis tools, we use our anomaly detection framework described in Chapter 4. Our framework

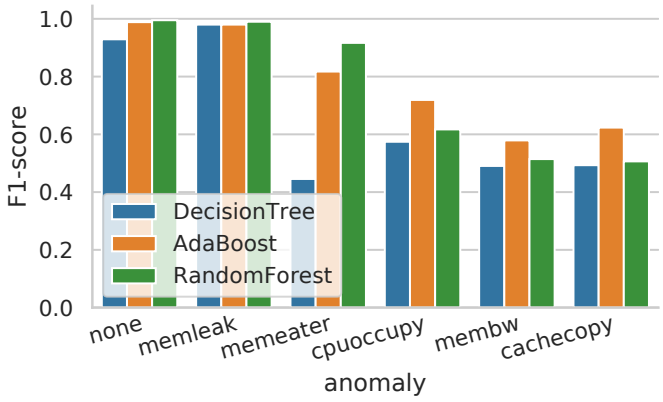


Figure 5·9: Results for classification of the anomalies. The overall F1-score using Random Forest algorithm is 0.94.

contains an offline training phase and a runtime diagnosis phase. In the offline training phase, we first use resource usage and performance counter data from known healthy and anomalous runs to extract useful statistical features calculated from time series. Then, these features are used to train the tree-based machine learning algorithms. At runtime, we generate statistical features from resource usage and performance counter data. Using these features, the machine learning model predicts the root cause (e.g., CPU contention, memory leak, network contention) of performance variations occurring at certain times. In a very similar manner to the anomaly diagnosis framework described in Chapter 4, we collect similar metrics using LDMS and generate the statistical features mentioned in that chapter. We use decision tree, AdaBoost, and random forest algorithms for training and prediction.

We run eight benchmark applications with and without our anomalies and use the data generated to evaluate the diagnosis framework using 3-fold cross-validation. The F1-scores for individual anomalies are reported in Figure 5·9 and the confusion matrix which shows accuracy for each class is reported in Figure 5·10. While the framework is good at identifying whether there is an anomaly or not, the CACHECOPY, CPUOCCUPY, and MEMBW anomalies are sometimes mistaken for each other. This

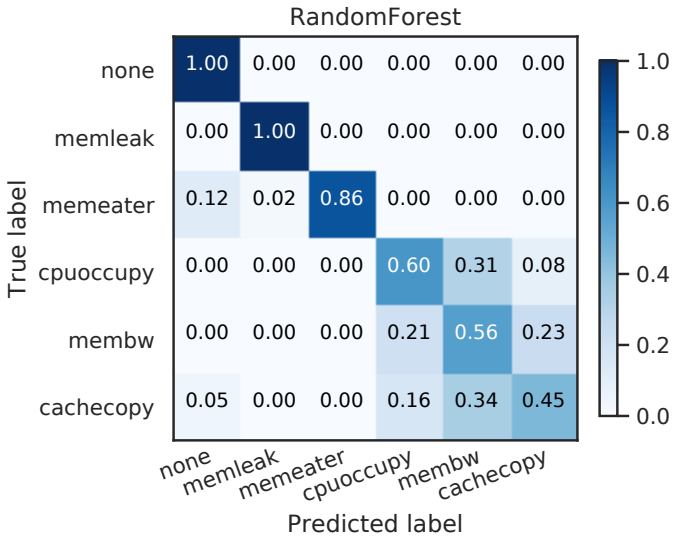


Figure 5·10: Confusion matrix for anomaly diagnosis using Random Forest.

could be due to the lack of metrics representing memory bandwidth in the monitoring data. In general, the results are compatible with our earlier results, demonstrating the usability of our suite in the evaluation of anomaly diagnosis methods. Our new anomalies also demonstrate room for improvement for better diagnosis of cache and CPU anomalies in Figure 5·10.

5.3.2 Evaluating System Management Policies

System management policies on HPC systems, such as job scheduling, job allocation, or task mapping, play a vital role in efficient usage of system resources [Xiong et al., 2018, Yang et al., 2011]. Anomalies in a system may affect the behavior of a system management policy. Knowing how the scheduling/allocation of jobs changes when there are performance anomalies helps evaluate system management policies in a more realistic manner and select a policy that is resilient to anomalies.

In the following, we demonstrate how two job allocation policies react differently under presence of CPU and memory anomalies. The two policies are the Round-

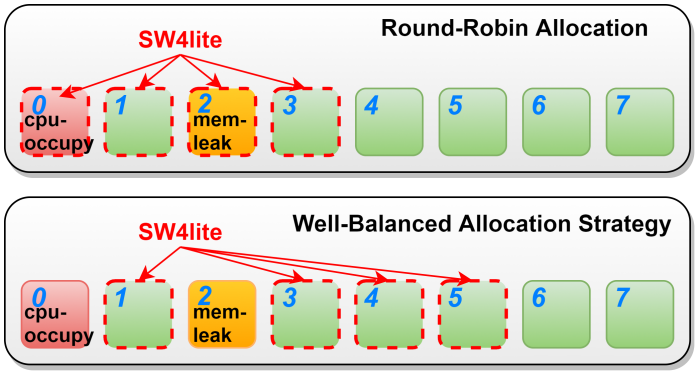


Figure 5·11: Allocation of SW4lite with two policies.

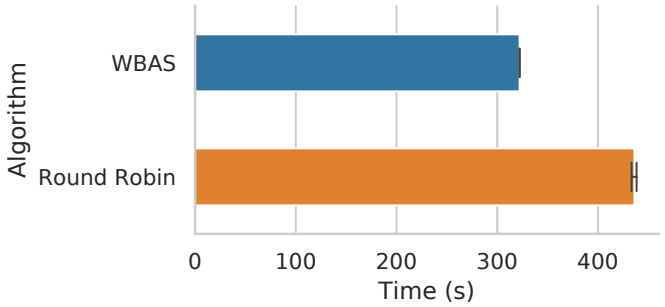


Figure 5·12: Evaluating the impact of anomalies on two different job allocation policies. Figure shows average running times for two allocation policies in the presence of anomalies.

Robin (RR) policy and the Well-Balanced Allocation Strategy (WBAS) by Yang et al. [Yang et al., 2011]. The RR policy simply allocates a job to the available nodes in the system following the label order. The WBAS policy prioritizes selecting the nodes with lower CPU load and larger free memory. To accomplish this, the WBAS policy calculates a computing capacity (CP) value for each node by $CP = (1 - Load\%) \times Mem_{free}$. Here, the CPU load $Load\%$ is derived from both current load and the average load of the recent several minutes according to the formula $Load = \frac{5}{6}Load_{current} + \frac{1}{6}Load_{5minAvg}$. In our system, we collect the current CPU load of the node using the metric USER::PROCSTAT, and we monitor the free memory (Mem_{free}) by the metric MEMFREE::MEMINFO.

In our case study, we run the SW4lite application on 4 nodes of Voltrino out of 8 available nodes (referred to as Nodes [0..7]), as shown in Figure 5·11. To create an anomaly, we run CPUOCCUPY on Node 0, and run MEMLEAK on Node 2. CPUOCCUPY can be used to change the CPU load to any given value between 0% and 100% for each core, we set it to 100% for one core. MEMLEAK can be used to reduce free memory on Node 1 to any given value, we set it to 1 GB. The WBAS policy avoids using the two nodes with the anomalies and allocates the job to Nodes $[1, 3..5]$ instead. Meanwhile, the RR policy allocates the job to Nodes [0..3].

With each of the two allocation policies, we run the SW4lite application 3 times and report the execution time in Figure 5·12. On average, the job execution time is 322 s with the WBAS policy, and it is 436 s with the RR policy. These results show that for this case, compared to the RR policy, the WBAS policy reduces the execution time by 26% on average through actively avoiding the anomalous nodes. This experiment provides an example of how our synthetic anomaly suite can be utilized to evaluate and compare different job allocation policies in the presence of these types of anomalies. HPAS brings the ability to independently change the $Load\%$ and Mem_{free} components of the CP equation, enabling a systematic evaluation of the equation and motivating more complicated models perhaps with cache or network components as well. Without the usage of our suite, it is more difficult to systematically test and compare different system management techniques under controlled anomalies.

5.3.3 Developing Applications Resilient to Performance Variability

One way of developing applications resilient to performance variations is to be aware of how much a given application is affected by anomalies in different subsystems. As an example, we show the use of HPAS in demonstrating the effect of a load balancing algorithm by using the Charm++ runtime system.

We use a simple 3D stencil application given in the Charm++ examples and

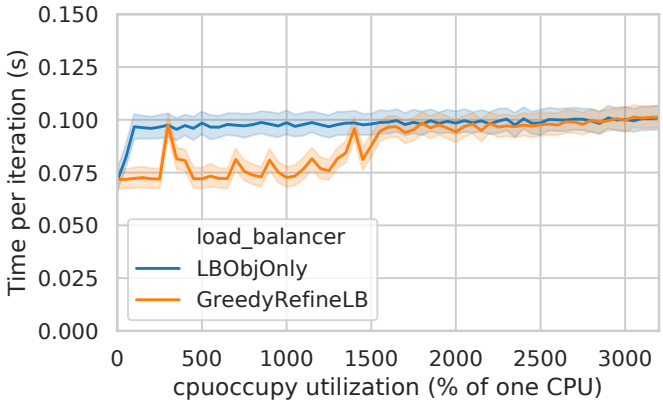


Figure 5·13: Performance of 3D stencil with different load balancers with increasing CPUOCCUPY intensity on Voltrino.

execute it on one node while changing the intensity of the CPUOCCUPY anomaly from zero to 100% of 32 CPUs. Figure 5·13 shows the performance of two load balancers: LBOBJONLY that only uses object properties and GREEDYREFINELB load balancer that measures CPU capacity before scheduling tasks. The two load balancers perform similarly when there are no anomalies (utilization = 0), and when more than 16 CPUs are used by the anomaly. However, in most cases where the anomaly uses fewer than 16 CPUs, GREEDYREFINELB, which measures CPU capacity, outperforms the other one. Notably, a degradation in performance where the anomaly uses 4 CPUs may indicate room for improvement in the load balancing strategy, as anomaly intensities at 5 or 6 CPUs can be mitigated successfully.

This example use case illustrates how our anomaly suite can be used to inform the choice of the load balancer, the development of new load balancers, or the decision to use a load balancer or not.

5.4 Conclusion

Performance variability in HPC systems is a significant challenge, and studying this variability is a key step in the reduction of variability for future systems. The study

of variability is impacted by the lack of a commonly used and open-source tool for generating synthetic performance variations. To address this, we have presented HPAS, a suite of anomaly replication tools that realistically replicate performance variability in specific subsystems such as the CPU, cache, memory, network, or storage.

We demonstrated compelling use cases for HPAS, including performance variation diagnosis and evaluation of system management policies and applications. In many of the use cases, HPAS has shown that there is room for improvement in the state-of-the-art. We believe that the adoption of this suite will have a positive impact on research and development efforts on resolving performance variability in HPC systems.

Chapter 6

Counterfactual Explanations for Machine Learning on Multivariate HPC Time Series Data

We have so far introduced two automated analytics methods that use numeric monitoring data from large-scale computing systems to detect which applications are running (Chapter 3) and diagnose performance variations (Chapter 4). Other automated analytics methods have been shown to diagnose performance variations [Borghesi et al., 2019b, Klinkenberg et al., 2017] or improve scheduling [Yang et al., 2011, Xiong et al., 2018] using multivariate time series data. In this chapter, we refer to such ML frameworks that use multivariate time series HPC system telemetry data as *HPC ML frameworks*.

While many advantages of ML are well-studied, there are also common drawbacks that HPC ML frameworks need to address before they can be widely used in production. These frameworks commonly have a taciturn nature, e.g., reporting only the final diagnosis "`network contention on router-123`," without providing reasoning relating to the underlying data. Furthermore, the ML models within these frameworks are black boxes which may perform multiple data transformations before arriving at a classification, and thus are often challenging to understand. The black-box nature of these frameworks causes a multitude of drawbacks, including making debugging mispredictions challenging, degrading user trust, and reducing the overall usefulness of the systems.

To address this ML *explainability* problem, a number of methods have been proposed by researchers. Some classifiers such as linear regression [Rencher and Christensen, 2012] and decision trees [Quinlan, 1986] are *inherently interpretable* and model weights can shed light on the model decisions. There are also methods that explain black-box classifiers [Arya et al., 2019]. These methods can be divided into *local* and *global* explanations, based on whether they explain a single prediction or the complete classifier. Local explanations can also be divided into *sample-based* explanations that provide different samples as explanations and *feature-based* explanations that indicate the features that impact the decision the most. However, most of existing explainability methods are not designed for multivariate time series data, and they fail to generate sufficiently simple explanations when tasked with explaining HPC ML frameworks.

Why do existing explainability methods fail to provide satisfactory explanations for HPC time series data? One differentiating factor is the complexity of the data. Existing sample-based methods provide samples from the training set or synthetically generate samples [Koh and Liang, 2017, Dhurandhar et al., 2018]. These methods are designed with the assumption that one sample is self-explanatory and users can visually distinguish between two samples; however, providing another sample HPC time series data with hundreds of metrics is not an adequate explanation. On the other hand, existing feature-based methods [Ribeiro et al., 2016, Lundberg and Lee, 2017] provide a set of features and expect the users to know the meaning of each feature, as well as normal and abnormal values for them, which is often not possible for HPC time series data.

In this chapter, we introduce a novel explainability method for time series data that provides *counterfactual* explanations for individual predictions. The counterfactual explanations consist of hypothetical samples that are as similar as possible to the sample

that is explained, while having a different classification label, i.e., "if these metrics were different in the given sample, the classification label would have been different." The format of the counterfactual explanation is identical to the format of the training data with the same number of time series and same window length. The counterfactual explanations are generated by selecting a minimum number of time series from the training set and substituting them in the sample under investigation to obtain different classification results. In this way, system administrators or users can understand the expected behavior by examining a limited number of substituted metrics. These explanations can be then used to debug misclassifications, understand how the classifier makes decisions, provide adaptive dashboards that highlight important metrics from ongoing application runs and extract knowledge on the nature of normal or anomalous behavior of a system.

Our specific contributions are as follows:

- Demonstration of existing general-purpose explainability methods and how they are inadequate to explain HPC time series frameworks (§ 6.5.1),
- design of a formal problem statement for multivariate time series explainability and a proof of NP-hardness of the problem (§ 6.2, Appendix A),
- design of a heuristic algorithm for the time series explainability problem (§ 6.3),
- demonstration of the application of the proposed explainability method to several ML-based HPC frameworks and data sets (§ 6.4, 6.5),
- comparison of our method with state-of-the-art explainability methods using a set of novel and standard metrics. Our method generates comprehensible explanations for HPC time series data, and performs better than baselines in terms of *faithfulness* and *robustness* (§ 6.5).

6.1 Motivation

The growth in ML-based methods indicates that ML is becoming a key part of HPC system analysis and management. Widespread adoption of such methods in production systems, however, depends on solving important challenges in accountability, accuracy, and explainability. In other words, we need to ensure that administrators and users are able to make sense of the ML classifications and predictions, and accordingly iterate on their design, fix possible bugs, or determine additional probes for data collection. In many fields where human operators exist as the last layer of decision-making, such as precision medicine, explainability is already a requirement [Tjoa and Guan, 2019]. In the HPC domain, it is also crucial to have models that provide explanations for their predictions so that system administrators and users can make important and possibly costly decisions with confidence. For example, a system administrator should expect questions like "Is this system working as expected?", "Do the results seem reasonable?" or "What could be a case that can improve or change the current result?" to be answered by a production ML-based HPC management framework.

The field of explainability aims to answer such questions by generating explanations for existing black-box models. Deployed black-boxes can model extremely complex and interdependent relationships, which are usually not feasible for humans to comprehend easily, or at all [Murdoch et al., 2019]. Using explainability methods to explain ML models is expected to bring a multitude of advantages to HPC systems. Explanations help non-expert users understand the model behavior without requiring a deep understanding of the entire model. Explanations can also help identify possible adversarial perturbations to increase the robustness of the systems [Koh and Liang, 2017], they can be used to assure that features which are effective in the classification represent metrics that have a causal relationship to the anomalies [Hall, 2019]. Operators can use explanations to ensure that decision-making is less biased, i.e., help detect and

ultimately fix existing bias in the training data set and resulting model.

6.2 Counterfactual Time Series Explanation Problem

Our goal is to provide counterfactual explanations for ML methods that operate on time series data. We define the *counterfactual time series explanation problem* as follows. Given a black-box ML framework that takes multivariate time series as input and returns class probabilities, the explanations show which time series need to be modified, and how, to change the classification result of a sample in the desired way, e.g., "if `MemFree::meminfo` was not decreasing over time, this run would not be classified as a memory leak." For a given sample and a class of interest, our counterfactual explanation finds a *minimum set of substitutions* to the sample from a *distractor*[1] chosen from the training set that belongs to the class of interest, such that the resulting sample is predicted as the class of interest. We assume a black box model for the classifier, thus having no access to the internal weights or gradients. We next define our problem formally.

6.2.1 Problem Statement

In this chapter, we represent multivariate time series classification models using $f(x) = y : \mathbb{R}^{m \times t} \rightarrow \mathbb{R}^k$ where the model f takes m time series of length t and returns the probability for k classes. We use the shorthand $f_c(x)$ as the probability for class $c \in [1, k]$. Our goal is to find the *optimum counterfactual explanation* for a given test sample x_{test} and class of interest c. We define an optimum counterfactual explanation as a modified sample x' that is constructed using x_{test} such that $f_c(x')$ is maximized. The class of modifications that we consider to construct x' are substitutions of entire time series from a distractor sample x_{dist}, chosen from the training set, to x_{test}. Our

[1]The distractor is a sample chosen from the training set that our methods "distracts" the classifier with, resulting in a new classification result.

second objective is to minimize the number of substitutions made to x_{test} in order to obtain x'.

The optimum counterfactual explanation can be constructed by finding x_{dist} among the training set and A which minimizes

$$L\left(f, c, A, x'\right) = \left(1 - f_c(x')\right)^2 + \lambda ||A||_1, \tag{6.1}$$

where

$$x' = (I_m - A)x_{test} + Ax_{dist}, \tag{6.2}$$

λ is a tuning parameter, I_m is the $m \times m$ identity matrix, and A is a binary diagonal matrix where $A_{j,j} = 1$ if metric j of x_{test} is going to be swapped with that of x_{dist}, 0 otherwise.

We prove in the Appendix A that the problem of finding a counterfactual explanation, x_{dist} and A, that maximize $f_c(x')$ is NP-hard. Because of this, it is unlikely that a polynomial-time solution for our explainability problem exists; therefore, we focus on designing approximation algorithms and heuristics that generate acceptable explanations in a practical duration.

6.2.2 Rationale for Chosen Explanation

Many existing explainability techniques rely on synthetic data generation either as part of their method or as the end result [Ribeiro et al., 2016, Dhurandhar et al., 2018, Lundberg and Lee, 2017]. These explanations assume the synthetic modifications lead to meaningful and feasible time series. Generating synthetic HPC time series data is challenging because many of the time series collected represent resource utilization values that have many constraints, i.e.; the rate of change of certain counters and the maximum/minimum value of metrics are bounded by physical constraints.

In our method, we choose x_{dist} from the training set as part of the explanation,

in contrast to providing synthetic samples, which guarantees that the time series in the explanation are feasible and realistic, because they were collected from the same system. Choosing a distractor x_{dist} from the training set also enables administrators to inspect the logs and other information besides time series that belong to the sample.

We keep the number of distractors to 1, instead of substituting individual time series from various distractors, in order to guarantee a possible solution. As long as $\arg\max_{j\in[1,k]} f_j(x_{dist}) = c$, a solution with $||A||_1 \leq m$ exists. Furthermore, in cases where administrators need to inspect logs or other related data, keeping the distractor count small helps improve the usability of our method.

6.3 Our Method: Counterfactual Explanations

What are some algorithms that can be used to obtain counterfactual explanations? We present a greedy search algorithm that generates counterfactual explanations for a black-box classifier and a faster optimization of this algorithm.

As we described in Sec. 6.2, our goal is to find counterfactual explanations for a given test sample x_{test}. Recall that a counterfactual explanation is a minimal modification to x_{test} such that the probability of being part of the class of interest is maximized. Our method aims to find the minimal number of time series substitutions from the chosen distractor x_{dist} instance that will flip the prediction.

We relax the loss function L (6.1), using

$$L\left(f, c, A, x'\right) = \left((\tau - f_c(x'))^+\right)^2 + \lambda(||A||_1 - \delta)^+, \tag{6.3}$$

where x' is defined in (6.2), τ is the target probability for the classifier, δ is the desired number of features in an explanation and $x^+ = \max(0, x)$, which is the rectified linear unit (ReLU). ReLU is used to avoid penalizing explanations shorter than δ. Running optimization algorithms until $f_c(x')$ becomes 1 is usually not feasible and the resulting

explanations do not significantly change; thus, we empirically set $\tau = 0.95$. We set $\delta = 3$, as it is shown to be a suitable number of features in an explanation [Miller, 2019].

Our explainability method operates by choosing multiple distractor candidates and, then, finding the best A for each distractor. Among the different A matrices, we choose the matrix with the smallest loss value. We present our method for choosing distractors, and two different algorithms for choosing matrix A for a given distractor.

6.3.1 Choosing Distractors

After finding the best A for each x_{dist}, we return the best overall solution as the explanation. As we seek to find the minimum number of substitutions, it is intuitive to start with distractors that are as similar to the test sample as possible. Hence, we use the n nearest neighbors of x_{test} in the training set that are correctly classified as the class of interest for the distractor. Especially for data sets where samples of the same class can have different characteristics, e.g., runs of different HPC applications undergoing the same type of performance anomaly, choosing a distractor similar to x_{test} would intuitively yield more minimal and meaningful explanations.

To quickly query for nearest neighbors, we keep all correctly classified training set instances in a different *KD-Tree* per class. The number of distractors to try out is given by the user as an input to our algorithm, depending on the running time that is acceptable for the user. If the number of training instances is large, users may choose to either randomly sample or use algorithms like k-means to reduce the number of training instances before constructing the KD-tree. The distance measure we use is euclidean distance, and we use the KD-tree implementation in scikit-learn [Pedregosa et al., 2011].

Algorithm 1 Sequential Greedy Search

Require: Instance to be explained x_{test}, class of interest c, model f, distractor x_{dist}, stopping condition τ
Require: $f_c(x_{dist}) \geq \tau$
Ensure: $f_c(x') \geq \tau$
1: $AF \leftarrow 0_{m \times m}$ // AF is final A
2: **loop**
3: $x' \leftarrow (I_m - AF)x_{test} + AFx_{dist}$
4: $p \leftarrow f_c(x')$
5: **if** $p \geq \tau$ **then return** AF **end if**
6: **for** $i \in [0, m]$ **do**
7: $A \leftarrow AF$
8: $A_{i,i} = 1$
9: $x' \leftarrow (I_m - A)x_{test} + Ax_{dist}$
10: improvement $\leftarrow f_c(x') - p$
11: **end for**
12: Set $AF_{i,i} = 1$ for i that gives best improvement
13: **end loop**

6.3.2 Sequential Greedy Approach

The greedy algorithm for solving the hitting set problem is shown to have an approximation factor of $log_2|U|$, where U is the union of all the sets [Chandrasekaran et al., 2011]. Thus, one algorithm we use to generate explanations is the *Sequential Greedy Approach*, shown in Algorithm 1. We replace each feature in x_{test} by the corresponding feature from x_{dist}. In each iteration, we choose the feature that leads to the highest increase in the prediction probability. After we replace a feature in x_{test}, we continue the greedy search with the remaining feature set until the prediction probability exceeds τ, which is the predefined threshold for probabilities.

6.3.3 Random-Restart Hill Climbing

Although the greedy method is able to find minimal explanations, searching for the best explanation by substituting the metrics one by one can become slow for data sets with many metrics. For a faster algorithm, we use derivative-free optimization

Algorithm 2 Random Restart Hill Climbing

Require: Instance to be explained x_{test}, class of interest c, model f, distractor x_{dist}, loss function $L(f, c, A, x')$, max attempts, max iters

for $i \in [0, num_{restarts}]$ **do**
  Randomly initialize A; attempts $\leftarrow 0$; iters $\leftarrow 0$
  $x' \leftarrow (I_m - A)x_{test} + Ax_{dist}$
  $l \leftarrow L(f, c, A, x')$
  while attempts $\leq$ max attempts **and** iters $\leq$ max iters **do**
    iters++
    $A_{tmp} \leftarrow$ RandomNeighbor(A)
    $x' \leftarrow (I_m - A_{tmp})x_{test} + A_{tmp}x_{dist}$
    if $L(f, c, A_{tmp}, x') \leq l$ **then**
      attempts $\leftarrow 0$; $A \leftarrow A_{tmp}$; $l \leftarrow L(f, c, A, x')$
    else
      attempts++
    end if
  end while
end for

algorithms to minimize the loss L (6.3).

For optimizing running time, we use a hill-climbing optimization method, which attempts to iteratively improve the current state by choosing the best successor state under the evaluation function. This method does not construct a search tree to search for available solutions and instead it only looks at the current state and possible states in the near future [Russell and Norvig, 2009]. It is easy for hill-climbing to settle in local minima, and one easy modification is *random restarting*, which leads to a so-called *Random Restart Hill-Climbing*, shown in Algorithm 2.

This algorithm starts with a random initialization point for A, and evaluates L for random neighbors of A until it finds a better neighbor. If a better neighbor is found, the search continues from the new A. In our implementation, we use the Python package `mlrose` [Hayes, 2019].

In some cases, hill climbing does not find a viable set A that increases the target probability. We check for this possible scenario by pruning the output, i.e., removing

metrics that do not impact target probability. Then, if no metrics are left, we use greedy search (§ 6.3.2) to find a viable solution.

6.3.4 How to Measure Good Explanations?

The goal of a local explanation is to provide more information to human operators to let them understand a particular decision made by an ML model, learn more about the model, and hypothesize about the future decisions the model may make. However, in analogy to the Japanese movie Rashomon, where characters provide vastly different tellings of the same incident, the same classification can have many possible explanations [Breiman, 2001b]. Thus, it is necessary to choose the best one among possible explanations.

There is no consensus on metrics for comparing explainability methods in academia [Lipton, 2018, Schmidt and Biessmann, 2019]. In this work, we aim to provide several tenets of good explanations with our explainability method.

Faithfulness to the original model: An explanation is faithful to the classifier if it reflects the actual reasoning process of the model. It is a first-order requirement of any explainability method to accurately reflect the decision process of the classifier and not mislead users [Ribeiro et al., 2016]. However, most of the time it is challenging to understand the actual reasoning of complicated ML models. To test the faithfulness of our method, we explain a simple model with a known reasoning process and report the precision and recall of our explanations.

Comprehensibility by human operators: Understanding an explanation should not require specialized knowledge about ML. According to a survey by Miller [Miller, 2019], papers from philosophy, cognitive psychology/science and social psychology should be studied by explainable artificial intelligence researchers. In the same survey, it is stated that humans prefer only 1 or 2 causes instead of an explanation that covers the actual and full list of causes. This is especially important for HPC time

series data, since each time series represents a different metric and each metric typically requires research to understand the meaning. Thus, to evaluate comprehensibility, we compare the *number of time series* that are returned in explanations by different explainability methods.

Robustness to changes in the sample: A good explanation would not only explain the given sample, but provide similar explanations for similar samples [Alvarez-Melis and Jaakkola, 2018, Alvarez Melis and Jaakkola, 2018], painting a clearer picture in the minds of human operators. Of course, if similar samples cause drastic changes in model behavior, the explanations should also reflect this. A measure that have been used to measure robustness is the *local Lipschitz constant* $\mathcal{L}$ [Alvarez-Melis and Jaakkola, 2018], which is defined as follows for a given x_{test} instance:

$$\mathcal{L}(x_{test}) = \max_{x_j \in \mathcal{N}_k(x_{test})} \frac{\|\xi(x_{test}) - \xi(x_j)\|_2}{\|x_{test} - x_j\|_2}, \tag{6.4}$$

where $\xi(x)$ is the explanation for instance x, and $\mathcal{N}_k(x)$ is the k-nearest neighbors of x_{test} in the training set. We use nearest neighbors, instead of randomly generated samples, because it is challenging to generate realistic random time series. The maximum constant is chosen because the explanations should be robust against the worst-case. Intuitively, the Lipschitz constant measures the ratio of change of explanations to changes in the samples. We change explanations to $1 \times m$ binary matrices (1 if metric is in explanation, 0 otherwise) to be able to subtract them.

Generalizability of explanations: Each explanation should be generalizable to similar samples; otherwise, human operators using the explanations would not be able to gain an intuitive understanding of the model. Furthermore, for misclassifications, it is more useful for the explanations to uncover classes of misclassifications instead of a single mishap.

We measure generalizability by applying an explanation's substitutions to other

samples. If the same metric substitutions from the same distractor can flip the prediction of other samples, that means the explanation is generalizable.

6.4 Experimental Setup

In this section, we describe the data sets and ML frameworks we use to evaluate our explainability method as well as the baseline explainability methods we compare against.

6.4.1 Data Sets

We use three high-dimensional multivariate time series data sets: two HPC system telemetry data sets and a motion classification data set.

For all data sets, we normalize the data such that each time series is between 0 and 1 across the training set. We use the same normalization parameters for the test set. We use normalized data to train classifiers, and provide normalized data to the explainability methods. However, the real values of metrics are meaningful to users (e.g., CPU utilization %), so we provide un-normalized data in the explanations given to users and our figures.

HPAS data set: We use HPAS [Ates et al., 2019b] (Chapter 5) to generate synthetic performance anomalies on HPC applications and collect time series data using LDMS. We run our experiments on Voltrino at Sandia National Laboratories, a 24-node Cray XC30m supercomputer with 2 Intel Xeon E5-2698 v3 processors and 125 GB of memory per node [National Technology and Engineering Solutions of Sandia, LLC., 2020]. We run the Cloverleaf, CoMD, miniAMR, miniGhost, and miniMD from the Mantevo Benchmark Suite [Heroux et al., 2009], proxy applications Kripke [Kunen et al., 2015] and SW4lite [Sjogreen, 2018], and MILC which represents part of the codes written by the MIMD Lattice Computation collaboration [The MIMD Lattice Computation (MILC) Collaboration, 2016]. We run each application on 4 nodes,

with and without anomalies. We use the cpuoccupy, memorybandwidth, cachecopy, memleak, memeater and netoccupy anomalies from HPAS.

Each sample has 839 time series, from the `/proc` file system and Cray network counters. We take a total of 617 samples for our data set, and we divide this into 350 training samples and 267 test samples. One sample corresponds to the data collected from a single node of an application run. After this division, we extract 45 second time windows with 30 second overlaps from each sample.

Cori data set: We collect this data set from Cori [National Energy Research Scientific Computing Center, 2020] to test our explainability method with data from large-scale systems and real applications. The goal of this data set is to use monitoring data to classify applications. Cori is a Cray XC40 supercomputer with 12,076 nodes. We run our applications in compute nodes with 2 16-core Intel Xeon E5-2698 v3 processors and 128 GB of memory. We run 6 applications on 64 nodes for 15-30 minutes. The applications are 3 real applications, LAMMPS [Plimpton, 1995], a classical molecular dynamics code with a focus on materials modeling, QMCPACK [Kim et al., 2018], an open-source continuum quantum Monte Carlo simulation code, HACC [Habib et al., 2013], an open-source code uses N-body techniques to simulate the evolution of the universe; 2 proxy applications, NEKBone and miniAMR from ECP Proxy Apps Suite [Exascale Computing Project, 2020]; and HPCG [Dongarra et al., 2016] benchmark which is used to rank the TOP500 computing systems.

We collect a total of 9216 samples, and we divide this into 7373 training and 1843 test samples. Each sample represents the data collected from a single node of an application run and has 819 time series collected using LDMS from the `/proc` file system and PAPI [Terpstra et al., 2010] counters.

Taxonomist data set: This data set, that we previously released [Ates et al., 2018b], was collected from Voltrino, a Cray XC30m supercomputer, using LDMS.

The data set contains runs of 11 different applications with various input sets and configurations, and the goal is to classify the different applications.

We use all the data, which has 4728 samples. We divide it into 3776 training samples and 952 test samples. Each sample has 563 time series. Each sample represents the data collected from a single node of an application run.

NATOPS data set: This data set is from the motion classification domain, released by Ghouaiel et al. [Ghouaiel et al., 2017, Bagnall et al., 2020]. We chose this data set because of the relatively high number of time series per sample, compared to other time series data sets commonly used in the ML domain.

The NATOPS data contains a total of 24 time series representing the X, Y and Z coordinates of the left and right hand, wrist, thumb and elbows, as captured by a Kinect 2 sensor. The human whose motions are recorded repeats a set of 6 Naval Air Training and Operating Procedures Standardization (NATOPS) motions meaning "I have command," "All clear," "Not clear," "Spread wings," "Fold wings," and "Lock wings." We keep the original training and test set of 180 samples each, with 50 second time windows.

6.4.2 Machine Learning Techniques

We evaluate our explainability techniques by explaining 3 different ML pipelines that represent different HPC frameworks proposed by researchers.

Feature Extraction + Random Forest: This technique represents a commonly used pipeline to classify time series data for failure prediction, diagnose performance variability, or classify applications [Tuncer et al., 2017, Tuncer et al., 2019, Ates et al., 2018a, Klinkenberg et al., 2017, Nie et al., 2018]. It is representative of the automated analytics frameworks we describe in Chapters 3 and 4.

This method is not explainable because the random forests produced can be very complex. For example, the random forest we trained with the HPAS data set had

100 trees and over 50k nodes in total. Operators that try to understand a prediction without explainability methods would have to inspect the decision path through each decision tree to understand the mechanics of the decision, and understanding high-level characteristics such as "how can this misclassification be fixed?" is near-impossible without explainability techniques.

We extract 11 statistical features including the minimum, maximum, mean, standard deviation, skew, kurtosis, 5^{th}, 25^{th}, 50^{th}, 75^{th} and 95^{th} percentiles from each of the time series. Then, we train scikit-learn's random forest classifier based on these features [Pedregosa et al., 2011].

Autoencoder: Borghesi et al. has proposed an autoencoder architecture for anomaly detection using HPC time series data [Borghesi et al., 2019b, Borghesi et al., 2019a]. The autoencoder is trained using only "healthy" data, and it learns a compressed representation of this data. At runtime, data is reconstructed using the autoencoder and the mean error is measured. A high error means the new data deviates from the learned "healthy" data; thus it can be classified as anomalous. We implement the architecture described by Borghesi et al. and use it for our evaluation. In order to convert the mean error, which is a positive real number, to class probabilities between 1 and 0, we subtract the chosen threshold from the error and use the sigmoid function. This autoencoder model is a deep neural network, and deep neural networks are known to be one of the least explainable ML methods [Gunning, 2017].

Feature Extraction + Logistic Regression: The logistic regression classifier is inherently interpretable, so we use this pipeline for sanity checks of our explanations in experiments where we need a ground truth for explanations. For input feature vector x the logistic regression model we use calculates the output y using the formula:

$$y = S(w \cdot x),$$

where $S(z) = \frac{1}{1+e^{-z}}$ is the sigmoid function. Thus, the classifier only learns the weight vector w during training[2]. Furthermore, it is possible to deduce that any feature x_i for which the corresponding weight w_i is zero has no effect on the classification. Similarly, features can be sorted based on their impact on the classifier decision using $|w_i|$. We use the same features as the random forest pipeline.

6.4.3 Baseline Methods

We compare our explainability method with popular explainability methods, LIME [Ribeiro et al., 2016], SHAP [Lundberg and Lee, 2017], as well as Random, which picks a random subset of the metrics as the explanation. We choose these methods because there are no existing methods that can explain multivariate time series models without relying on gradients. We use LIME and SHAP to explain feature-extraction based frameworks, and use them to only explain the classifier that is trained using the features instead of the whole framework.

LIME: LIME stands for local interpretable model-agnostic explanations [Ribeiro et al., 2016]. LIME operates by fitting an interpretable linear model to the classifiers predictions of random data samples. The samples are weighted based on their distance to the test sample, which makes the explanations local. When generating samples, LIME generates samples within the range observed in the training set. In our evaluation, we use the open-source LIME implementation [Ribeiro, 2020].

LIME does not directly apply to time series as it operates by sampling the classifier using randomly generated data. Randomly generating HPC time series data while still obeying the physical constraints in the data as well as maintaining representative behavior over time is a challenging open problem. In our evaluation, we apply LIME to frameworks that perform feature extraction, and LIME interprets the classifier that

[2]Other formulations of logistic regression include a b term such that $y = S(w \cdot x + b)$, but we omit this for better interpretability.

takes features as input.

Another challenge with LIME is that it requires the number of features in the explanation as a user input. Generally, it is hard for users to know how many features in an explanation are adequate. In our experiments, we use the number of metrics in our method's explanation as LIME's input.

SHAP: The Shapley additive explanations (SHAP), presented by Lundberg and Lee [Lundberg and Lee, 2017], propose 3 desirable characteristics of explanations, local accuracy, missingness and consistency. They define additive SHAP values, i.e., the importance values can be summed to arrive at the classification. SHAP operates by calculating feature importance values by using model parameters; however, since we do not have access to model parameters, we use KernelSHAP which estimates SHAP values without using model weights.

We use the open-source KernelSHAP implementation [Lundberg, 2020], which we refer to as SHAP in the remainder of the chapter. SHAP also suffers from one of the limitations of LIME; it is not directly applicable to time series, so we apply SHAP to frameworks that perform feature extraction. SHAP does not require the number of features in the explanation as an input.

6.5 Evaluation

In this section, we evaluate our explainability method and compare it with other explainability methods based on qualitative comparisons and the metrics described in Sec. 6.3.4. We aim to answer several questions: (1) Are the explanations minimal? (2) Are the explanations faithful to the original classifier? (3) Are the explanations robust, or do we get different explanations based on small perturbations of the input? (4) Are the explanations generalizable to different samples? (5) Are the explanations useful in understanding the classifier?

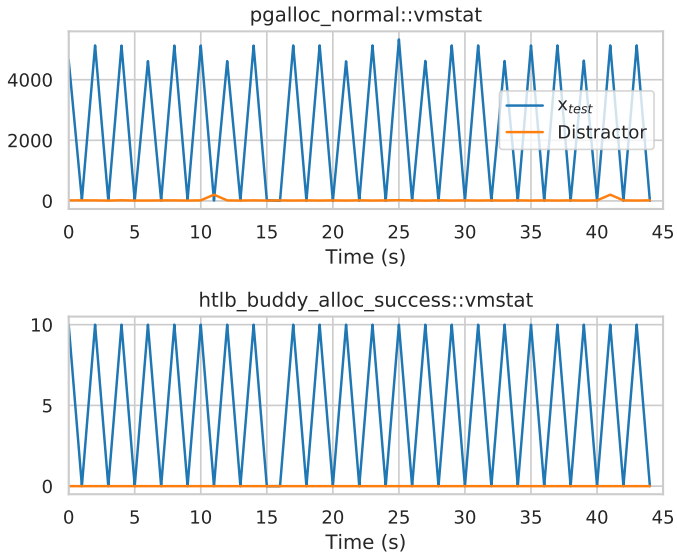


Figure 6·1: The explanation of our method for correctly classified time window with the "memleak" label. Our method provides two metrics as an explanation to change classification label from "memleak" to "healthy." The metrics are shown in the y-axes and the metric names are above the plots. The first metric indicates that the classifier is looking for repeated memory allocations in runs with memory leak. The second metric indicates that the classifier is looking for repeated successful huge page allocations in runs with memory leak. The second metric may indicate that the data set we use is biased and does not include runs with high memory fragmentation.

6.5.1 Qualitative Evaluation

Our first-order evaluation is to use our explanation technique and the baselines to explain a realistic classifier. Similar to the framework described in Chapter 4, we use the random forest classifier with feature extraction and the HPAS dataset, which includes different types of performance anomalies. We choose "memleak" anomaly from HPAS data set, which makes increasing memory allocations without freeing to mimic memory leakage. Our goal is to better understand the classifier's understanding of the "memleak" anomaly. After training the random forest pipeline, we choose a correctly classified time window with the memleak label as x_{test}, and the "healthy" class as the class of interest. We run our method, LIME, and SHAP with the same

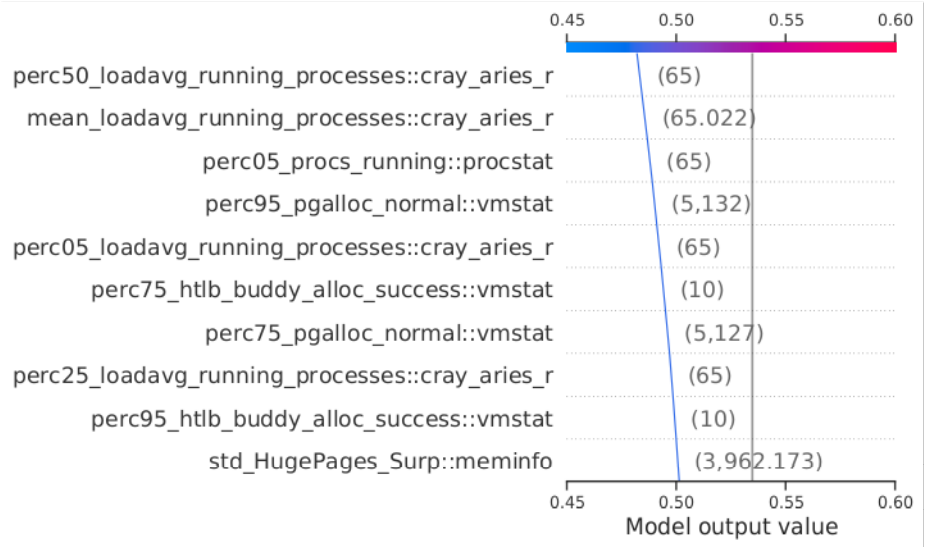


Figure 6·2: The explanation of SHAP for a correctly classified time window with the "memleak" label. SHAP provides 187 features with non-zero SHAP values to explain the characteristics of "memleak" anomaly of which we show the top 10 in the figure. It is very challenging to understand how many features are sufficient for an explanation, whether these features are relevant to "memleak" or other anomalies, or how to interpret the feature values in the explanation.

x_{test} and compare the results.

Our explanation contains two time series, and is shown in Fig. 6·1. The first metric to be substituted is `pgalloc_normal` from `/proc/vmstat`, which is a counter that represents the number of page allocations. Because of our preprocessing, the plot shows the number of page allocations per second. It is immediately clear that the nodes with memory leaks perform many memory allocations and act in a periodic manner.

The second metric in Fig. 6·1 is `htlb_buddy_alloc_successes`, which also belongs to the same time window. This metric shows the number of successful huge page allocations. Memory leaks do not need to cause huge page allocations, since memory leaks in a system with fragmented memory might cause failed huge page allocations. This indicates that our training set is biased towards systems with less

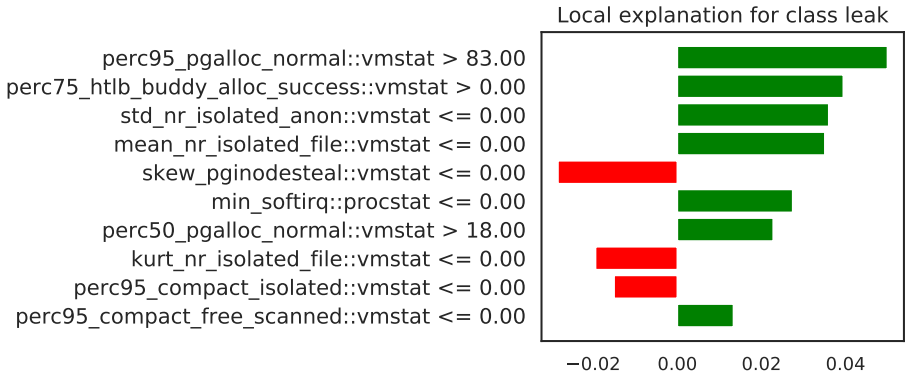


Figure 6·3: The explanation of LIME for correctly classified time window with the "memleak" label. LIME provides features that positively (green) and negatively (red) affect the decision. Although the first two features are derived from the metrics in our explanation, it is not straightforward to interpret the values of the features, especially the negative ones. The number of features is an input from the user.

fragmented memory, most probably because our benchmarks are all short-lived.

The SHAP explanation, in Fig. 6·2, contains 187 features with very similar SHAP values. Even though we can sort the features by importance, it is difficult to decide how many features are sufficient for a good explanation. Also, SHAP provides a single explanation for one sample, regardless of which class we are interested in, so the most important features are features that are used to differentiate this run from other CPU-based anomalies, which may not be relevant if our goal is to understand memory leak characteristics. Finally, it is left to the user to interpret the values of different features, e.g., the 75$^{\text{th}}$ percentile of `pgalloc_normal` was 5,127; however, this does not inform the user of normal values for this metric, or whether it was too high or too low.

The LIME explanation is shown in Fig. 6·3. Green values indicate that the features were used in favor of memory leak, and red values were opposing memory leak. We keep the number of features in the explanation at the default value of 10. The first two features are derived from the metrics in our explanation, and a threshold is given

for the features, e.g., the 95^{th} percentile of page allocations is over 83, which causes this run to be likely to be a memory leak. Interpreting features such as percentiles, standard deviation and thresholds on their values is left to the user. Furthermore, the effect of the red features is unclear, as it is not stated which class the sample would be if it is not labeled as leak.

It is important to note that both LIME and SHAP use randomly generated data for the explanations. In doing so, these methods assume that all of the features are independent variables; however, many features are in fact dependent, e.g., features generated from the same metric. Without knowledge of this, these random data generation methods may test the classifier with synthetic runs that are impossible to get in practice, e.g., synthetic runs where the 75^{th} percentile of the one metric is lower than the 50^{th} percentile of the same metric. Our method does not generate synthetic data, and uses the whole time series instead of just the features, so it is not affected by this.

6.5.2 Comprehensibility

We measure comprehensibility using the number of metrics in the explanation. Our method returns 2 time series for the qualitative evaluation example in Fig. 6·1, and in most cases the number of time series in our explanations is below 3; however, for some challenging cases it can reach up to 10. SHAP returns 187 features in Fig. 6·2, and SHAP explanations typically have hundreds of features for HPC time series data. LIME requires the number of features as an input; however, does not provide any guidelines on how to decide this value.

6.5.3 Faithfulness

We test whether the explainability methods actually reflect the decision process of the models, i.e., whether they are faithful to the model. For every data set, we train

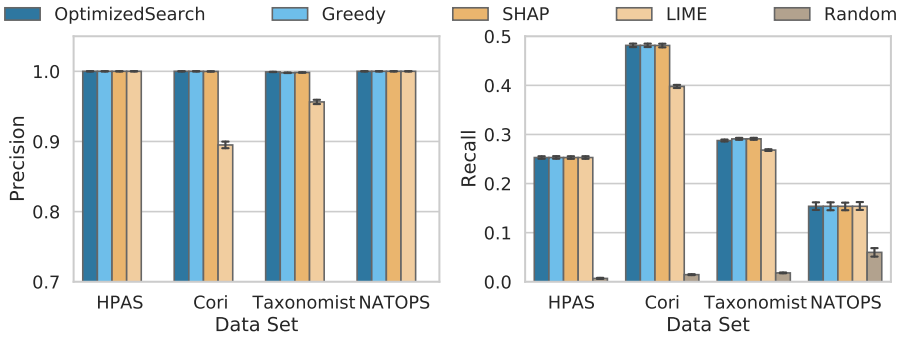


Figure 6·4: Precision and recall of the explanations for a classifier with known feature importances. Our proposed explainability method (OptimizedSearch and Greedy), and SHAP have perfect precision. LIME has lower precision for Cori and Taxonomist data sets, which indicates that although some features have no impact on the classifier decision, they are included in the LIME explanations. The low recall indicates that not every feature is used in every local decision.

a logistic regression model with $L1$ regularization. We change the $L1$ regularization parameter until less than 10 features are used by the classifier. The resulting classifier uses 5 metrics for HPAS and Cori data, 9 for NATOPS and 8 for Taxonomist. Because we know the used features, we can rank the explanations based on precision and recall.

- **Recall:** How many of the metrics used by the classifier are in the explanation?
- **Precision:** How many of the metrics in the explanation are used by the classifier?

We get explanations for each sample in the test set, and show the average precision and recall in Fig. 6·4. In order to not disadvantage other explainability methods, we first run the greedy search method and get the number of metrics in the explanation. Then, we get the same number of metrics from each method. This way, e.g., LIME is not adversely affected by trying to provide 10 features in the explanation even though only 7 are used by the classifier.

The results show that besides random, all explainability methods perform relatively well on faithfulness. Notably, LIME has low precision for the Cori and Taxonomist data sets, which indicates that there may be features in LIME explanations that are

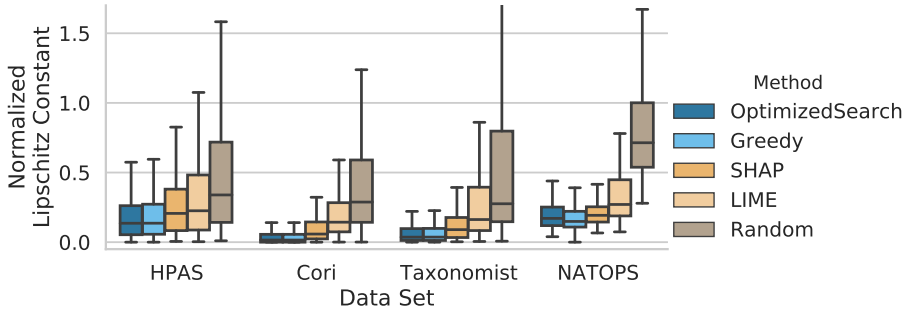


Figure 6·5: Robustness of explanations to changes in the test sample. Our proposed explainability method (OptimizedSearch and Greedy) is the most robust to small changes in the input, resulting in more predictable explanations and better user experience. Lipschitz constant is normalized to be comparable between different data sets, and a lower value indicates better robustness.

actually not used by the classifier at all. This could be due to the randomness in the data sampling stage of LIME.

6.5.4 Robustness

For robustness, we calculate the Lipschitz constant (6.4) for each test sample and show average results in Fig. 6·5. According to the results, our method is the most robust explainability technique. One reason is that our method does not involve random data generation for explanations, which reduces the randomness in the explanations. It is important for explanations to be robust, which ensures that users can trust the ML models and explanations. For the NATOPS data set, the greedy method has better robustness compared to optimized search, because the greedy method inspects every metric before generating an explanation, thus finds the best metric, while the optimized search can stop after finding a suitable explanation even if there can be better solutions.

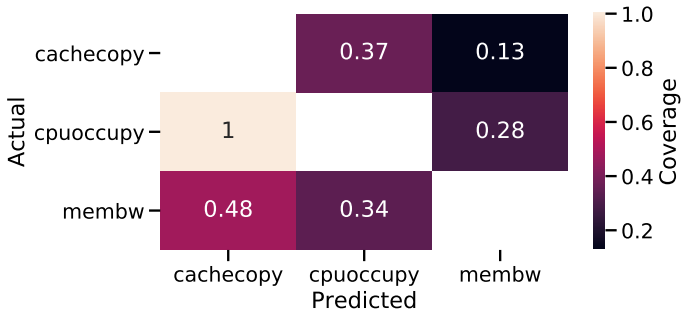


Figure 6·6: The ratio of test samples that our explanations are applicable to, among samples with the same misclassification characteristics. For the cpuoccupy runs that are misclassified as cachecopy, every explanation is applicable to every other sample with the same misclassification.

6.5.5 Generalizability

We test whether our explanations for one x_{test} are generalizable to other samples. We use the HPAS data set and random forest classifier with feature extraction. There are 3 classes that are confused with each other. For each misclassified test instance, we get an explanation and apply the same metric substitutions using the same distractor to other test samples with the same (true class, predicted class) pair.

We report the percentage of misclassifications that the explanation applies to (i.e., successfully flips the prediction for) in Fig. 6·6. According to our results, on average, explanations for one mispredicted sample are applicable to over 40% of similarly mispredicted samples. This shows that users do not need to manually inspect the explanation for every misprediction, and instead they can obtain a general idea of the classifiers error characteristics from a few explanations, which is one of the goals of explainability.

6.5.6 Investigating Misclassifications

As a demonstration, we debug a misclassified sample using our explainability method. This is a typical scenario that would be encountered if ML systems are deployed to

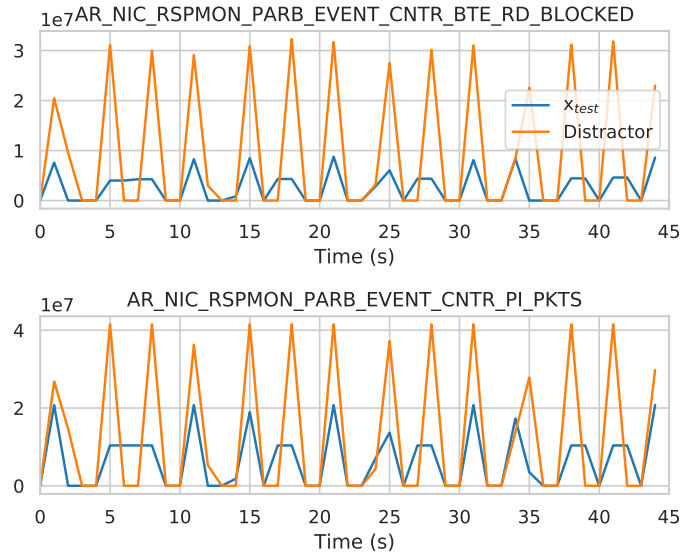


Figure 6·7: Our explanation for a "network" anomaly misclassified as "healthy." The metrics are shown in the y-axes and the metric names are above the plots. The explanation indicates that the anomaly needs higher network traffic to be classified correctly. 4 of the 6 metrics in the explanation are omitted because they appear identical to the metrics shown.

production. We train the autoencoder-based anomaly detection framework [Borghesi et al., 2019b,Borghesi et al., 2019a] using healthy data from the HPAS data set. Among the runs with the network anomaly, the run shown in Fig. 6·7 is misclassified. We explain this misclassification using our explanation method: we choose the misclassified run as x_{test} and the anomalous class as the class of interest (the autoencoder has two classes: anomalous and healthy). We cannot apply the LIME and SHAP baselines here as the autoencoder directly takes time series as input. Recall that LIME and SHAP require tabular data to operate and can not directly use time series data as input.

The explanation includes 6 network metrics from Table 6.1. Metrics 6 and 2 are shown in Fig. 6·7. Field 3 describes if the metric counts the number of flits or packets. Blocked means a flit was available at the input but arbitration caused the selection of

another input. Metrics 1 and 2 count traffic being forwarded by the network interface card (NIC) to the processor, 3 and 4 count the processor memory read and write traffic resulting from requests received over the network, metric 5 counts traffic injected by the NIC into the network, and metric 6 counts reads of processor memory initiated by the NIC's block transfer engine to fetch data included in the put requests it is generating [Cray, 2018].

The explanation indicates that the intensity of the network anomaly in this run needs to be higher, i.e., more network traffic is needed, for this to be classified as a network anomaly. Furthermore, since 6 metrics all need to be modified for the prediction to flip, it can be seen that the autoencoder has learned the parallel behavior of these metrics. For example, if the number of packets is changed independent of the number of flits, the classifier does not change its prediction. It is highly unlikely that randomly generated samples would capture the correlated behavior of these two metrics.

6.6 Conclusion

This chapter investigated explainability for ML frameworks in HPC, particularly focusing on time series data. Such data is widely used in ML-based HPC analysis and management methods that show a lot of promise to improve HPC system performance, efficiency, and resilience. Being explainable is an important requirement for any ML

Table 6.1: Network metric names in explanation. Full names are "AR_NIC_(Field 1)_EVENT_CNTR_(Field 2)_(Field 3)."

Metric	Field 1	Field 2	Field 3
1	RSPMON_PARB	PI	FLITS
2	RSPMON_PARB	PI	PKTS
3	RSPMON_PARB	AMO	FLITS
4	RSPMON_PARB	AMO	PKTS
5	NETMON_ORB	EQ	FLITS
6	RSPMON_PARB	BTE_RD	BLOCKED

framework that seeks widespread adoption.

We defined the counterfactual time series explainability problem and presented a heuristic algorithm that can generate feasible explanations. We also demonstrated the use of our explanation method to explain various frameworks, and compared with other explainability methods. We have shown that our explainability method is the only explainability method that can generate comprehensible explanations for HPC ML frameworks, while having comparable or better faithfulness and robustness compared to existing methods for frameworks that they are both applicable to. We find that counterfactual explanations are more intuitive than baselines for high-dimensional time series data sets, and that using such explanations can yield insights into the models including details on the characteristics the model has learned, and such explanations are helpful to debug biases in the data or misclassifications.

Chapter 7

Automating Instrumentation Decisions to Diagnose Performance Problems in Distributed Applications

Instrumentation, in the form of logs or counters, is the de facto data source engineers use to diagnose performance problems in deployed distributed applications. However, it is difficult to know a priori *where* instrumentation is needed and *what* instrumentation is needed to help diagnose problems that may occur in the future [Zhao et al., 2017, Yuan et al., 2012b, Yuan et al., 2012a, Mace et al., 2015]. Exhaustively recording all possible distributed-application behaviors is infeasible due to the resulting overheads. As a result of these issues, distributed applications often contain lots of embedded (i.e., static) instrumentation, but rarely the right ones in the right locations needed to diagnose a specific problem [Mace et al., 2015, Yuan et al., 2012b]. When new problems occur, they typically cannot be diagnosed quickly because the instrumentation needed to locate their sources is not present.

Diagnosing problems observed in deployment requires customizing instrumentation choices while applications are running. To this end, researchers and practitioners have proposed two sets of complementary techniques: dynamic instrumentation and automated customization of instrumentation. Unfortunately, these techniques do not scale with the increasing complexity of distributed applications such as Ceph [Weil et al., 2006] or OpenStack [Openstack, 2020], since they may require the users to know many details about how the system operates, or are not suited for performance

problems. Dynamic instrumentation allows engineers to insert new instrumentation during runtime in pre-defined [Erlingsson et al., 2011, Mace et al., 2015, Cantrill et al., 2004] or almost arbitrary [Goswami, 2005] locations. Applications that do not support dynamic instrumentation can provide similar functionality by embedding a plethora of static instrumentation that can be selectively enabled during runtime [Vef et al., 2018, Desnoyers, 2020]. Used alone, these methods can result in high diagnosis times because engineers must manually explore the space of possible instrumentation choices in order to locate the source of a problem. Only after doing so, can they identify the problem's root cause and fix it.

To reduce diagnosis times, dynamic or configurable static instrumentation is complemented with statistical techniques that automatically identify the needed instrumentation [Zhao et al., 2017, Arumuga and Liblit, 2010, Zuo et al., 2016, Liblit et al., 2003]. But current automated-instrumentation techniques are not designed for distributed applications, for which identifying the needed instrumentation requires knowledge of which applications processes are involved in processing problematic requests. Existing approaches also focus on correctness and so make assumptions that are not valid for performance problems. For example, Log20 [Zhao et al., 2017] enables instrumentation to help diagnose non-crash correctness problems, such as unanticipated results, by enabling enough instrumentation to identify all unique code paths. But differentiating code paths is neither sufficient nor necessary for performance diagnosis. It is not sufficient because additional instrumentation may be needed to identify where on a unique code path a performance problem lies; it is not necessary for code paths that execute quickly as they do not contain performance problems.

We present the design of *statistical trace-driven automated instrumentation frameworks* (we use the acronym STAIF). STAIFs are a new class of instrumentation framework that are deployed alongside distributed applications. In response to newly-

observed performance problems in the applications they control, they search the space of possible instrumentation choices to dynamically insert or enable the instrumentation that is needed to locate the problem's source. Our design for STAIFs addresses key challenges that must be overcome for STAIFs to be useful. They include: how to architect STAIFs so that they are useful for a variety of distributed applications, how to allow for flexible search strategies for choosing instrumentation to enable, and how to keep STAIFs from enabling too much instrumentation.

STAIFs build on two key observations that make it possible to automatically customize instrumentation for performance problems in distributed applications. First, in many distributed applications, requests that exhibit similar workflows—i.e., that are processed similarly within and among the nodes of a distributed application—should perform similarly [Sambasivan et al., 2011, Sambasivan and Ganger, 2012]. (See Figure 7·1 for two possible request workflows in a simple distributed application.) Thus, if requests that *are expected to* perform similarly *do not do so*, there is something unknown about their workflows. This unknown behavior may be indicative of performance problems, such as unexpected slow code paths being executed, load imbalances (perhaps due to unintended hardware heterogeneity), or contention for shared resources. Localizing the source of the observed variation gives insight into *where* (e.g., in which node, or where in the code base) additional instrumentation is needed to identify the unknown behavior.

Once we have identified *where* instrumentation is needed, focused search strategies–e.g., ones that encode domain knowledge for specific distributed applications or ones based on statistics or machine learning—can then be used to explore *what* instrumentation is needed to explain the variation—i.e., identify the source of the problem. For cases where performance problems manifest as consistently slow requests instead of high variation, a similar process that focuses on identifying dominant

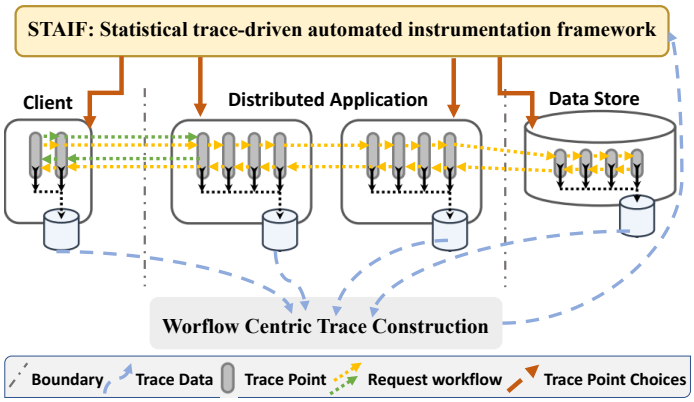


Figure 7·1: Two simple workflows. The figure shows workflows for two requests in a simple distributed application. The first request (green) hits in the in-memory cache, whereas the second request (yellow) requires a storage-node access.

contributors to request response times can be used instead.

The second observation is that recent work on workflow-centric tracing of distributed applications, also called distributed tracing, makes it possible to capture graphs or *traces* of requests' workflows. (Please see Sambasivan et al. [Sambasivan et al., 2016] for a systematization of work in this area.) Workflow-centric tracing works by propagating context (e.g., request IDs) with individual requests as the requests are executed by the distributed application. Records of log points executed by requests are tagged with requests' context (log points that record context are called *trace points*.) Trace points record both a unique name along with arbitrary key/value pairs, which developers use to encode queue lengths, function parameters, or performance counters to record current resource usage.

Asynchronously, a big-data job gathers trace-point records from the machines on which the distributed-applications' processes are running and stitches together ones with related context to create traces (graphs) of requests' workflows. Sampling techniques can be used to keep tracing's overhead low enough (e.g., $< 1\%$) to be used in production, as is done at many companies today [Sigelman et al., 2010, Kaldor et al., 2017, Jaeger, 2020, OpenTracing, 2020].

We have implemented a prototype STAIF, which we call Pythia, and used instantiations of it to diagnose problems in two popular distributed applications, OpenStack [Openstack, 2020] and HDFS [Apache Software Foundation, 2019]. Our implementation can locate problems quickly, while exploring only 30% of the available instrumentation. We also show that parts of our implementation can scale to process instrumentation from production Uber systems. We present the following specific contributions.

1. We show how two statistical measures—performance variation and consistently-high response times—can be combined with workflow-centric tracing to create automated instrumentation frameworks for diagnosing performance problems in distributed applications (§ 7.1).
2. Building on the above observations, we present the design & operation of STAIFs (§ 7.2). Our design is broadly useful for typical request-based distributed systems. We present requirements existing workflow-centric tracing infrastructures must satisfy to be used with STAIFs (§ 7.2.2).
3. We describe two search strategies for customizing instrumentation in response to performance problems (§ 7.3). The first explores the search space of possible instrumentation choices hierarchically by leveraging the caller/callee relationships that are captured by today's open-source tracing infrastructures. The second explores the search space using only happens-before relationships between trace points.
4. We describe our experiences building a prototype STAIF (§ 7.4) and using it to diagnose problems in OpenStack [Openstack, 2020] and HDFS [Apache Software Foundation, 2019] (§ 7.5). We quantify the efficacy of our search algorithms and instrumentation-budgeting mechanisms on these traces as well as those we obtained from Uber.

7.1 Toward STAIFs

This chapter presents statistical trace-driven automated instrumentation frameworks (STAIFs), a novel method for automatically enabling or inserting the instrumentation needed to locate the source of new performance problems in running distributed applications. This automated approach frees application developers to provide the STAIF with a plethora of instrumentation choices as the STAIF will only use the ones it deems most useful.

This section further motivates the need for STAIFs (§ 7.1.1), describes the distributed applications to which we target STAIFs and our assumptions about them (§ 7.1.2), and describes STAIFs' general operation (§ 7.1.3). It concludes by presenting a conceptual example of how STAIFs could help debug a real problem in OpenStack [Openstack, 2020], a distributed-application for managing virtual machines. (§ 7.1.4).

7.1.1 Motivating factors

STAIFs address four inter-related challenges that are a result of distributed applications' scale and complexity.

No perfect one-size-fits-all instrumentation: Past research literature has argued that the instrumentation needed to localize the source of one problem may not be useful for others [Mace et al., 2015, Yuan et al., 2012b, Zhao et al., 2017, Vef et al., 2018]. To illustrate the difficulty of choosing one-size-fits-all instrumentation, Zhao et al. [Zhao et al., 2017] state that Hadoop, HBase, and Zookeeper have been patched over 28,821 times over their lifetimes to add, remove, or modify static log statements embedded in their code. The authors also point out that the 2,105 revisions that modify logs' verbosity levels reflect the tussle between a desire to balance overhead and informativeness of log statements.

How STAIFs address this challenge: STAIFs automatically customize the instrumentation that is active in an application to currently-observed problems.

Data overload: Existing logging or tracing infrastructures capture voluminous amounts of data. For example, Facebook's Canopy workflow-centric tracing infrastructure captures 1.16 GB/s of trace data and even individual traces contain 1000s of trace points [Kaldor et al., 2017]. Problem diagnosis, even when the needed instrumentation is present, can amount to trying to find a needle in a haystack [Rabkin and Katz, 2013].

How STAIFs address this challenge: STAIFs only insert or enable the instrumentation needed for current problems while leaving unneeded instrumentation disabled. This reduces the amount of generated data that engineers must analyze.

Instrumentation search spaces that are too large to explore manually: Assume a distributed application that allows log points (or trace points) to be inserted or enabled at every functions' entry, exit, and exceptional return. (This is similar to the distributed applications used by Mace et al. [Mace et al., 2015] and Erlingsson et al. [Erlingsson et al., 2011].) In this case, the instrumentation search space is a function of the number of functions in the applications' code base, the number of nodes on which the application executes, and the number of low-level parameters that could be exposed within each trace point. In our experience even relatively small distributed applications, such as OpenStack [Openstack, 2020], can have search spaces consisting of 1000s of trace points.

How STAIFs address this challenge: STAIFs automatically explore the instrumentation search space, reducing the amount of time needed to find the instrumentation useful for a problem.

High diagnosis times: Today, diagnosis can take hours or days and over 50% of engineers' time is spent debugging [O'Dell, 2017].

How STAIFs address this challenge: We argue that the challenges described above are significant causes of high diagnosis times. By addressing them, STAIFs will reduce the time needed to diagnose and fix a problem.

7.1.2 Target applications

We target our STAIF design to common request-based distributed applications, such as web servers, databases, and storage systems. These applications have the following properties. *First*, they are comprised of a set of processes that cooperate to handle requests they receive from clients. Each request's processing forms a workflow consisting of the order of work done within and among the processes involved in its execution. Processes may be logically grouped into services and service components to reflect a clean breakdown of functionality. They may be deployed on physical machines, virtual machines, or containers (which we interchangeably call *nodes*).

Second, the amount of work our target distributed applications perform on behalf of individual requests is bounded and, as such, their performance can be characterized by requests' response times. *Third*, requests' response times are (mostly) influenced by the code paths (workflows) executed on behalf of requests, parameters specified within requests, configuration parameters, resource availability, and resource contention.

Requirements: To be used with STAIFs, distributed applications must already support or be modified to support workflow-centric tracing. Examples of distributed applications that already support tracing include CockroachDB [CockroachDB, 2019], HDFS [Apache Software Foundation, 2019], OpenStack [Openstack, 2020, Mirantis OSProfiler, 2020], Ceph [Weil et al., 2006,Blkkn, 2020], and those at the major cloud infrastructure providers [Sigelman et al., 2010,Kaldor et al., 2017,Jaeger, 2020]. This list is growing rapidly due to practitioners' push toward richly-instrumented applications and the open-source community's efforts to standardize tracing APIs [OpenTracing, 2020, OpenTelemetry, 2020].

Adding tracing to an application that does not currently support it involves modifying its source code to propagate context with requests. It also involves modifying logging points to record context (thus transforming them into trace points) or adding new trace points natively. Distributed applications for which some of its services, but not all, can be modified to support tracing can also be used with STAIFs. Legacy distributed applications whose source code cannot be modified can be used with STAIFs if they exhibit simple 1:1 relationships between inputs and outputs and are deployed using frameworks that support tracing (e.g., Istio [Istio, 2020]).

7.1.3 General operation

STAIFs will operate in a continuous cycle as shown in Figure 7·2. Each iteration of the cycle refines hypotheses of the instrumentation needed to explain high performance variation or high response times in the distributed application a given STAIF controls. Since requests' performance only depends on their critical paths, STAIFs only focus on them when making instrumentation decisions. At the beginning of time, a STAIF takes as input: 1) an instrumentation search space, 2) initial, very coarse-grained expectations of which distributed-application requests should perform similarly, which STAIFs will refine in every cycle, and 3) workflow skeletons, which are traces created with a set of trace points are already enabled/inserted in the distributed application.

The *instrumentation search space* lists the instrumentation the STAIF can control. *Initial expectations* are of the form "expect all requests of the same type to perform similarly," which is a coarse-grained starting point applicable to many distributed applications. STAIFs will iteratively refine this expectation every cycle so that it becomes more specific (e.g., initially expect that all READ requests will perform similarly, then expect that all READ requests that hit in the same caches will perform similarly.) Engineers specify initial expectations by listing all possible request types (i.e., internal and external entry points to the system), which must be uniquely named.

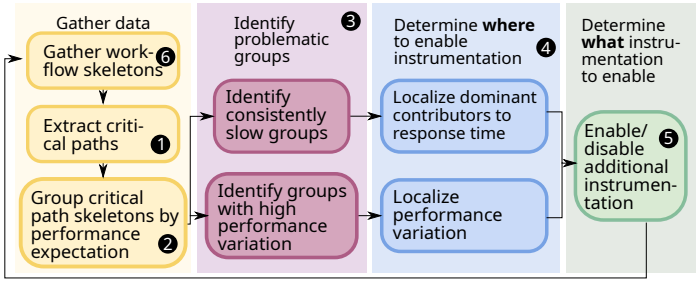


Figure 7·2: STAIFs' continuous cycle of operation.

For example, in HDFS, request types would be READ, WRITE, LIST and so on. These initial expectations could also be mined automatically from already-collected traces.

Workflow skeletons reflect the basic instrumentation that must be enabled to measure request response times, extract critical paths during runtime, and match critical paths to the expectations of similar performance. Concretely, this means that skeletons are created with trace points demarcating request types (e.g., in their names), request start and return to caller, the start of concurrent activity and any synchronization.

A STAIF's cycle of operation is as follows. In the first step of the cycle, the STAIF extracts workflow skeletons' critical paths. In the *second step*, the STAIF groups requests' critical-paths according to the expectations of which ones should perform similarly. This can be done by matching the request types listed in the expectations to trace-point names. Groups are annotated with distributions of overall request latency and performance variances. Each group also maintains a single representative critical-path skeleton which is annotated with detailed response-time distributions between each trace point.

In the *third step*, the STAIF examines the response-time distributions of requests assigned to different groups. It identifies problematic groups, which are groups containing requests that either exhibit high performance variation or which are consistently very slow (i.e., have high response times with low variation). Groups

with high performance variation are identified via user-defined thresholds on variance, informed by SLAs or desired levels of predictability. They could also be identified as groups that account for a large amount of unexplained variability in the overall response-time distribution of all recently-observed requests. Consistently-slow groups are ones not identified as high-variance groups and whose response times fall above a user-specified percentile of the overall response-time distribution (e.g., over the 99^{th} percentile).

In the *fourth step*, the STAIF explores where to enable instrumentation for problematic groups. To do so, it localizes dominant contributors to performance variation or high response times within problematic groups' critical-path representatives. In the *fifth step*, it identifies what instrumentation to enable in problematic areas. For example, one strategy is to enable more granular trace points hierarchically (e.g., first trace points services of a distributed application, then those in components within services). Between rounds of enabling more granular trace points, the strategy could also explore which key/value pairs, such as queue object sizes, that could be enabled within trace points explain variation and permanently enable those as well.

To allow a variety of distributed-applications, STAIFs use pluggable search strategies to both localize dominant contributors and search for what instrumentation to enable. This allows different applications to use custom strategies that encode application-specific knowledge (e.g., which high-variance areas to prioritize).

In the *sixth step*, the STAIF enables or inserts the selected instrumentation and gathers new traces that include it. The STAIF post processes trace points with enabled key/value pairs to add the key to the trace-point name. If the value is numerical, it also adds a bin range the value falls in.

In subsequent iterations, STAIFs use the expectation that requests with identical request types and structures (i.e., those that execute the same trace points in the

same order) should exhibit similar performance. So, it groups requests of the same type whose critical paths execute the same currently-enabled trace points in the same order into the same group.

Concurrently with the above steps, the STAIF preserves representative traces of slow requests, which engineers can examine at any time during their diagnosis efforts. It also explores what instrumentation could be disabled. Engineers can influence future cycles by modifying expectations (e.g., to ignore variance in locations where it is unavoidable [Sambasivan and Ganger, 2012]) or by specifically asking certain trace points to be enabled or disabled.

7.1.4 How STAIFs could aid diagnosis

We illustrate the potential of our STAIF design by discussing a hypothetical scenario of how it could help diagnose problems in OpenStack [Openstack, 2020], a distributed-application for managing clouds. We assume a hierarchical search strategy that iteratively enables more granular instrumentation in problematic areas.

OpenStack is run on high-capacity machines, but SERVER CREATE requests still finish creating new OpenStack VMs very slowly. The root cause is that the number of concurrent servers that can be created within OpenStack's Nova Service is limited to ten by default. Additional SERVER CREATE requests are forced to wait on a semaphore.

For this problem, the STAIF deployed with OpenStack will initially bin critical-path skeletons into groups based on their type (e.g., SERVER CREATE, DELETE). It will identify groups containing CREATE requests as exhibiting high performance variation. This is because CREATEs received during periods of low concurrency will execute immediately, whereas others will have to wait varying amounts of time for the semaphore.

The STAIF will localize the variation to the Nova service and then gradually enable more granular trace points within it (e.g., API-level functions, public functions, and

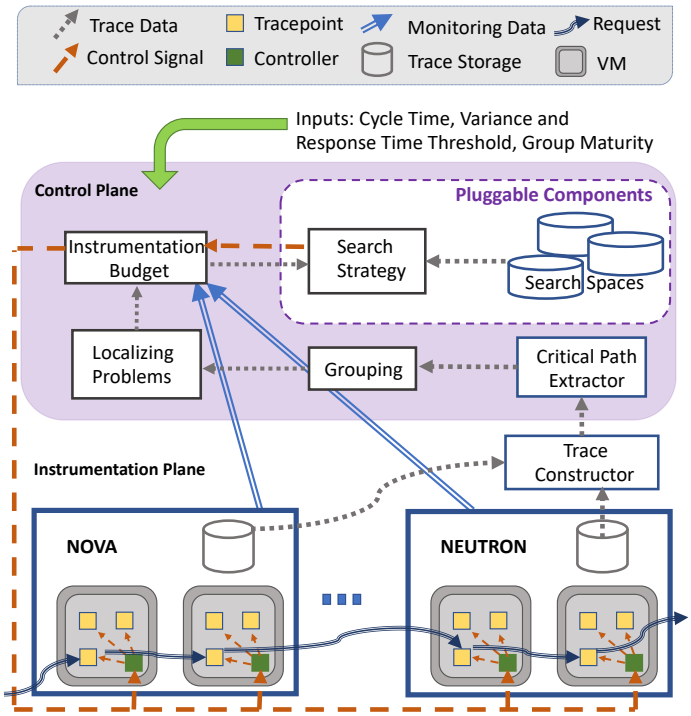


Figure 7·3: STAIF design. In this diagram, a STAIF is deployed to control instrumentation in OpenStack.

then private functions). It will eventually enable trace points within the function that contains the semaphore. It will find that the queue length variable that could be exposed within these trace points explains the observed variation. This will give engineers a strong starting point to identify the problem's root cause.

7.2 STAIF Design

Figure 7·3 shows our design for STAIFs, which is composed of a *control plane* and an *instrumentation plane*. The components in the control plane form the crux of a STAIF's functionality. A STAIF that controls a distributed application has controller agents on each application node that only send and receive signals, while the control plane can be deployed on one or more dedicated nodes, depending on the scale of the application. All data- and compute-intensive parts of the design are parallelizable,

and can be deployed as a big data job.

The control plane and the instrumentation plane interact in a limited and well-defined way. The instrumentation plane provides: (1) monitoring information about the trace collection capacity, and (2) workflow-centric traces in the form of directed acyclic graphs (DAGs), to the control plane. The nodes of these graphs represent events and the edges represent happens-before relationships. There can also be additional information in the graphs such as host names, function arguments, queue lengths, request types. Following the cyclical operation described in § 7.1.3, at each cycle the control plane analyzes the traces and sends control signals to the instrumentation plane. This compartmentalized design allows different workflow-centric tracing infrastructures that are already used in distributed applications to be modified to provide the functionality that STAIFs require.

The major components of the design are the grouping and localization components which identify the performance problems in the system, the instrumentation budget which decides how many trace points to enable or disable, the search space and search strategies which decide which trace points to enable, and the instrumentation plane which collects the traces.

7.2.1 Key Input Parameters

Although STAIFs are automated frameworks, differences in the application being instrumented, deployment scenarios, desired instrumentation level and available instrumentation budget necessitate the configuration of STAIFs before operation. The key inputs that STAIF requires are listed below:

Group maturity: STAIFs wait until enough data is accumulated for a group, before using it to make decisions. This threshold can be specified as the number of traces to collect, or in terms of statistical significance tests.

Cycle time: This is the minimum amount of time a STAIF waits to make

```
class SearchStrategy:
  def __init__(
      self,
      search_spaces,
      tracepoints, # Enabled tracepoints
      **kwargs, # Other settings
  ):
    ...

  def search(self, group, budget):
    # Perform search
    assert len(tracepoints_to_enable) <= budget
    return tracepoints_to_enable
```

Figure 7·4: Pseudocode for a search strategy in Python. The `tracepoints` object provides read access to a shared data structure storing the enabled trace points.

instrumentation decisions. For groups that have accumulated enough trace data it allows coherent decisions to be made about which ones should be prioritized.

Search strategy: Our pluggable design allows any object that implements the API in Fig. 7·4 can be used as the search strategy, which can be tailored for specific applications.

Variance and response-time threshold: STAIFs do not try to localize problems if the scale of the problem is insignificant to the users. We define this by requiring a threshold on overall variance and response time of requests, e.g., latency larger than 30 seconds, or top 95^{th} percentile of all request variances.

Fine-tuning instrumentation: Although not necessary, our design allows for the users to permanently enable/disable instrumentation points, and ignore certain requests or parts of requests even if they have high variance/latency. These requests can be described using key-value pairs (e.g., test vs. production requests) or specific edges in traces.

Table 7.1: STAIF requirements from the instrumentation plane and whether other workflow-centric tracing instrumentation frameworks meet these requirements. We implement all of these requirements for both OSProfiler, which is similar to OpenTelemetry, and for XTrace.

Requirement	Open-Telemetry	XTrace	Pivot-Tracing	Canopy	Zipkin
Concurrency & synchronization	N[1]	Y	Y	Y	N
Controllable	N	N	Y	N	N

7.2.2 Instrumentation Plane Requirements

The instrumentation plane is responsible for collecting traces from the distributed application. The major components of the instrumentation plane are context propagation, code instrumentation (trace points), and trace construction. STAIFs have two requirements from the instrumentation plane: the ability to collect traces that can be used to obtain correct critical paths, and the ability for trace points to be enabled/disabled. However, instrumentation frameworks can affect the overhead and results of STAIFs by differing on how they meet these requirements, shown in Table 7.1. We describe our requirements from the instrumentation plane below:

Correct representation of concurrency and synchronization: For accurate extraction of request workflows and critical paths, concurrency and synchronization points should be represented in the traces collected. Waiting events should not be recorded, or specifically annotated to prevent including them in the critical paths.

Controllable trace points: For instrumentation to be controllable by STAIF, there needs to be a mechanism to enable and disable instrumentation. This requires each trace point to have a unique name, which STAIF can address them by. The overhead of disabled trace points should also be minimal to prevent negative impact on performance.

[1]OpenTelemetry provides the FollowsFrom edges to annotate concurrency and synchronization; however, happens-before relationships may not be captured without disciplined manual tracing.

Performance: When the system is under high load, the instrumentation plane may either contend with the distributed application, or drop trace records. We expect trace collection to be done off the critical path to prevent interference and expect trace records to be dropped under high load.

7.2.3 Grouping, Localization and Search

At each cycle during operation, STAIFs use the **grouping** component to group critical paths according to their performance expectations, generalize from individual traces to overall system performance. Then, the **localization** component filters and prioritizes the groups according to whether they represent performance problems or not. The problematic groups are then used by **search strategies** to make enabling decisions.

STAIFs group critical paths that have the same ordering of trace points and important key/value pairs in the same bins. As a STAIF enables more instrumentation, either more groups are formed or existing groups become more detailed, while disabling instrumentation may cause groups to merge.

The localization component identifies which groups are problematic. To localize problems, a STAIF first filters normal or insignificant groups, e.g., groups that have latency and variance values below user-provided thresholds, or groups that have too few traces. These thresholds depend on the system being monitored, and restrictive thresholds would make a STAIF conservative about enabling instrumentation. However, a too-low, or non-existent threshold would not cause STAIFs to enable too much instrumentation because of the instrumentation budget.

Among the remaining groups, STAIF identifies groups that either exhibit high variance or which are consistently very slow. The threshold for deciding if a group is consistently slow or high variance is determined based on user inputs. Groups are sorted based on how much they are above the threshold and most problematic groups are explored first.

Various search strategies can be used within STAIF to decide which trace points to enable for a problematic group. More details on possible search strategies and the structure of the search space is in § 7.3.

7.2.4 Instrumentation Budget

In order to limit the storage used by the traces and make sure that STAIFs can collect complete traces in the next cycle without any dropped trace records, STAIFs use the instrumentation budget. At each cycle, the budget component decides how many trace points to enable or disable. The decisions are made based on the load on the system and user-provided limits.

To decide how many trace points to enable, the budget component uses monitoring signals collected from all distributed application nodes. These signals are tertiary, indicating whether (0) the node is healthy, (1) the node's trace buffers are almost full and that it may drop trace records, or (2) the node is dropping trace records. The budget will stop enabling of trace points when enough nodes are sending positive signals, and start disabling trace points if the nodes are dropping trace records, to keep a stable input of complete traces. The maximum number of trace points to enable/disable per cycle are provided by the user.

7.2.5 Disabling Instrumentation

Two mechanisms are used to disable trace points. First, asynchronous to the main cycle, STAIFs periodically perform **garbage collection** regardless of the budget decisions, which disables trace points that are not observed in any critical paths of problematic groups. These trace points may have been enabled during past problems or for different workloads/request distributions that are no longer valid.

The second disabling process is executed if the budget component decides to actively disable trace points. We prioritize enabled trace points based on how much

the group variances would increase if we disabled these trace points. To do this, we process representative paths from the current groups and determine which groups would merge as a result of disabling, and how much this merging would increase the variance. The trace points that would not cause merging, or those that do not increase variance are disabled until the desired number of trace points are disabled.

7.2.6 Limitations

Time-scale of problems: Transient problems that disappear in less than one cycle length can be detected, but further instrumentation can not be enabled in time. On the opposite scale, problems that cause fast requests to consistently slow down over many cycles or suddenly slow down due to e.g., configuration changes can be detected; however the additional instrumentation can only show major components to request latency, since enabling instrumentation on healthy/fast requests and comparing with slow requests would not be possible anymore.

Level of instrumentation: STAIFs inherently rely on the available instrumentation, so if a problem occurs in a component that is not instrumented, i.e., black-boxes, the problem can only be narrowed-down to the black-box boundaries. In order to maximize STAIF's performance, instrumentation should be added with good practices. We recommend instrumenting performance-critical functionality, queue lengths and locks, or other typical sources of performance problems. Higher-level languages and tools can be used to aid the developers in adding instrumentation. For example, in OpenStack which is written in Python, all methods of a class or metaclass which is deemed to be performance critical can be traced using one line of code.

Concurrency and synchronization: STAIFs rely on correct context propagation and instrumentation of concurrency/synchronization to learn critical paths. Knowing all concurrency/synchronization points can be hard, but dependencies between tasks can also be inferred using many observations without explicit annotations [Chow

et al., 2014, Mann et al., 2011]; however, once inferred, these dependencies must then be incorporated to each trace.

Inter-group interference: STAIFs inspect the performance of a single group before enabling instrumentation; however, the same instrumentation point may be executing in other groups as well. This may cause requests that would have been in the same group to be in different groups in later cycles. As more requests arrive, STAIF can still collect enough instrumentation from each group to make decisions if those groups become problematic. One way to lower this slow-down effect, which we implement for OpenStack, is to enable trace points per request type, by propagating the request type with the request.

Anti-targets: STAIFs are not targeted to streaming frameworks, such as timely dataflow [Murray et al., 2013], for which the concept of a request is ill-defined. If used with data-intensive frameworks, such as MapReduce [Dean and Ghemawat, 2004], STAIF's instrumentation choices will reflect the underlying data-distribution characteristics.

7.3 Search Space and Search Strategies

At each cycle, after localizing performance problems into groups of critical paths skeletons, STAIFs can work with one of multiple pluggable search strategies in order to decide which trace points to enable. The search strategies use a data structure we call the search space, which represents instrumentation that could be controlled within a distributed application. Search spaces are optimized for specific search strategies. As a result, a single distributed application can have multiple search spaces, each optimized for different search strategies. The search spaces can be provided by the application, or learned by STAIFs during an offline profiling phase. The search strategies are provided a problematic group, and the number of trace points to enable. They also have access

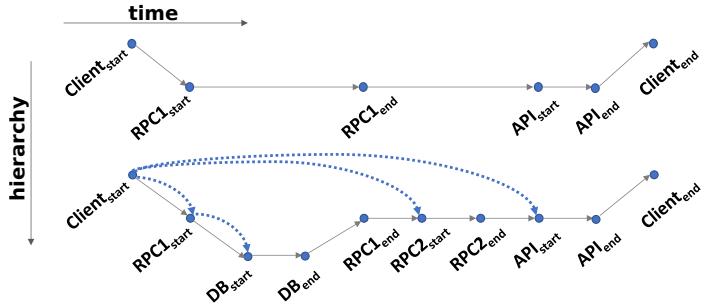


Figure 7·5: An enriched workflow skeleton (top) and the search space (bottom). DB and RPC2 trace points are disabled. The events lower in the hierarchy are drawn lower in the y-axis. The blue dashed edges in the bottom graph from Client to RPC1, RPC2 etc. represent the hierarchy and are specializations added by the hierarchical search strategy.

to the list of enabled trace points and the search space, as shown in Fig. 7·4. This is to let search strategies be aware of already enabled trace points and avoid choosing them as part of the enabling decisions.

In this section, we explore a search space and search strategy combination that takes advantage of the hierarchical relationships captured in widely-available infrastructures. To explore the value of the hierarchy, we also consider a flat search strategy that ignores these relationships.

7.3.1 Hierarchical Search Space

The search space represents all instrumentation that is controllable, and the context in which the instrumentation is expected to appear. The specific behavior that we want to answer using the search space is: (1) Which instrumentation points are executed during which request types? (2) What are the happens-before relationships between the instrumentation points? and (3) what are the boundaries between the instrumentation that this search space contains and other possible search spaces? Besides these three questions, the search space also contains "shortcuts" that can be taken during search that depend on the search strategy used. These may include

information such as the hierarchical caller/callee relationships between instrumentation points, or previously observed performance problems and trace point sets that were used to debug them.

The search space is structured as a collection of unique workflow paths that can be observed in the distributed system. The nodes of the search space are different trace points representing different events, and there are two types of edges, one represents happens-before relationships and the second type are optional edges that search strategies may add to facilitate faster searching. In Fig. 7·5, STAIFs can enable more instrumentation in either the database (DB) or the second remote procedure call (RPC2) based on the search strategy used.

Some applications have traces with many concurrency and synchronization points in the traces, e.g., during HDFS writes, each 64 KB packet is acknowledged by 4 threads, which results in 4^n paths for n packets [Wang, 2011]. In these cases, even though the traces contain many paths, these paths represent repeated behavior, so it is enough for the search space to contain paths extracted from smaller write requests, while keeping track of which trace points are repeating for use during matching.

Hierarchy: Capturing hierarchical relationships in the search space allows us to explore search-spaces and search strategies that make use of the hierarchy to guide instrumentation decisions. Current popular tracing infrastructures already expose hierarchies via the concept of *spans*, which are semantically-meaningful intervals that can contain other spans [OpenTracing, 2020, Sigelman et al., 2010]. Spans are useful to diagnose performance problems since span latencies are more meaningful than edge latencies between trace points. Example spans may include executions of distributed-application services, API calls, kernel system calls, or functions.

Spans typically have a hierarchical caller/callee relationships between them. We define the hierarchy between spans strictly using happens-before relationships. In Fig. 7·5,

the hierarchical relationships are shown using dashed lines, e.g., since $RPC1_{start}$ happens before DB_{start} and DB_{end} happens before $RPC1_{end}$, DB is a child of $RPC1$. This means that we can gain information about DB's performance by only enabling $RPC1$, i.e., if there is a performance problem in DB, $RPC1$'s performance will also be affected.

7.3.2 Matching Paths to the Search Space

At runtime, when the problem is localized to a group, we **match critical paths to the search space** to find the relevant path. Matching involves a trade-off between generalization and specificity. If it is too strict, e.g., requires exact matches for the paths, STAIFs would extensively rely on the traces that we have collected during offline profiling, thus it might not be flexible enough to follow the behavior of the system during runtime. On the other hand, too permissive matching may result in too many matches and start exploration of spurious instrumentation choices.

To match a path to the search space, we iterate the paths in the search space concurrently with the critical path. For any node in the critical path, we iterate the search space paths until an identical node is found. More formally, the critical path "`ABC`" is transformed into the regular expression "`.*A.*B.*C.*`". This matching operation has a time complexity of $\mathcal{O}(n)$ for a search space with n events, thus does not depend on the length of the traces collected at runtime. For all the matching paths, we prioritize the paths which have been observed more frequently during offline profiling. If the most frequent matching path offers no viable trace points to enable, we can move on to less frequent paths.

Missing trace points: If a trace point is enabled, but not contained in the collected path, this information could be used for finer-grained matching, e.g., as the regular expression "`[^ABC]*A[^ABC]*B[^ABC]*C[^ABC]*`" for a critical path "`ABC`" with the trace points `A`, `B` and `C` enabled. To account for possible loss of trace

records or imperfect offline profiling, we do not penalize missing trace points while matching.

7.3.3 Hierarchical Search

Distributed systems are typically composed of a hierarchy of different services, each with their own components, and each component running on multiple nodes. Within a single process, a hierarchy of threads, classes and methods can be found. One possible search strategy is to leverage this intrinsic hierarchy of distributed systems. The hierarchical search strategy explores a hierarchy of spans, i.e., semantically-meaningful intervals, top-down. First enabling lowest granularity and most general spans, and at each cycle enabling more and more granular spans to narrow down performance problems.

In order to operate, the hierarchical search strategy localizes problems within the groups before inspecting the hierarchy. A problem is localized by calculating a variation or latency value for each edge in the group. Then, edges are sorted based on their variance or latency and enabling is done starting with the most problematic edge.

For the problematic edge, we find the "common context" of its endpoints, and enable all children of this common context. For example, in Fig. 7·5, for the 5 different edges, if the high variance edge is edge 1, 3 or 5, we enable all children of the client span, because that is the lowest common context. For edges 2 and 4, we enable all children of RPC1 and API respectively. If the budget is not enough to enable all children, we prioritize them based on their variance history. We use the curved "shortcut" edges in Fig. 7·5 to quickly find the children of the common context.

Key-value pairs are also collected as part of trace points, and thus they may be enabled by the search strategy. We use canonical correlation analysis (CCA) [Hotelling, 1936] to find correlations between the enabled key-value pairs and overall request

latency. The useful variables can then be incorporated into the grouping, by grouping based on bins of variable values, e.g., requests with queue length < 10 and queue length ≥ 10.

7.3.4 Flat Search

In distributed systems where the hierarchy is very weak because of the code design or instrumentation choices, e.g., if there are hundreds of children of a single span, the flat search strategy can be used. This strategy does not rely on the hierarchy, and instead only uses the happens before relationships to find which trace points to enable. It matches the problem groups to the search space, and finds the closest matching critical paths. After that, it enables trace points that divide the most problematic edge equally, based on the budget. At the next cycle, if the problem can be localized further, only one of these new edges will be the problem edge, and the search strategy can continue by splitting that edge. This strategy with a budget of 1 is similar to a binary search: at each cycle, the problem edge is divided into 2 and the problematic half is investigated in the next cycle.

7.3.5 Search Strategy Optimizations

Our pluggable design enables such that there can be many more search strategies. One example is to add fast-forward edges after a problem is diagnosed by engineers. Later, when STAIFs start making the same instrumentation choices, the fast-forward edges can be used to re-enable the same trace points that were used to initially diagnose the problem.

One optimization to the hierarchical method would be to skip levels of hierarchy when searching, e.g., levels that are composed of very thin wrapper functions. Which layers to skip can be decided based on performance values measured.

If more than one edge in a group is problematic, we compare the absolute variance

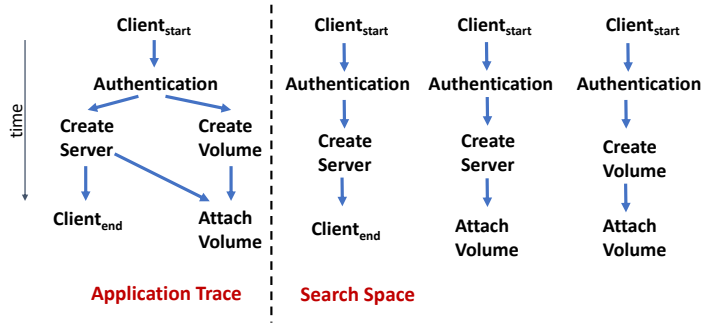


Figure 7·6: A simplified SERVER CREATE **trace and the corresponding paths in the search space.** Depending on the latencies of create server, create volume and attach volume, either of the three observed paths may become critical paths.

of edges when prioritizing. An alternative could be to give higher priority to edges deeper in the call graph since system activities that are nested deeper are more specific in scope and are expected to have less variation.

7.4 Implementation

We implemented a prototype STAIF, which we call *Pythia*, in 6000 lines of Rust, and tested it on OpenStack and HDFS. OpenStack [Openstack, 2020] is an open-source distributed application for managing clouds, written in Python. HDFS [Apache Software Foundation, 2019] is a distributed storage system, written in Java. We discuss our implementation choices for the prototype below.

Constructing the search space: The search space is mined from the distributed system by enabling all possible instrumentation and collecting traces while running exhaustive workloads. In production scenarios, data collection and search space generation can be done offline, for example during tests.

We construct the search space from traces by first extracting all paths from a trace. In Fig. 7·6, a trace of server creation and the corresponding search space paths are shown. The edges represent happens-before relationships, thus the fan-outs and

fan-ins represent concurrency and synchronization. There are three paths in this trace, and based on the runtime performance, e.g., if CREATE VOLUME has a problem and takes too long, any of the three paths can be the critical path. Thus, we keep all three paths in the search space. Care must be taken to keep the hierarchy information correctly while extracting these paths. For example, CREATE SERVER is a child of CLIENT, while CREATE VOLUME is not, even though they seem equivalent in the right 2 paths of the figure.

Deploying Pythia: We have Pythia agents at each application node and deploy the control plane and parts of the instrumentation plane on a separate node. The agents collect monitoring metrics and trace records, and accept requests from the controller node. The processing of the agents is minimized to reduce interference with the application.

The trace construction and critical path extraction is the most compute intensive part of Pythia; therefore it is done using a thread pool on the controller node. Each new request ID is assigned to one of the threads in the thread pool, which keeps track of the new trace records until the request is completed, and puts the critical path into a shared buffer.

At each cycle, the control thread reads the completed critical paths from the shared buffer and the monitoring signals are directly read from the agents. The remaining Pythia components, including the grouping, localization, budget, search space and search strategies are executed on the control plane. These can be parallelized further; however, it was not necessary for our implementation scale.

7.5 Evaluation

In this section, we evaluate our prototype implementation of Pythia using two distributed applications, OpenStack and HDFS, and traces we have obtained from Uber.

Table 7.2: Search space statistics. The results show that Pythia's search space and matching can be used for all three systems.

System	# of Unique Trace Points	# of Unique / Overall Paths in Search Space	Min / Mean / Max Path Length	Search Space Size (MB / # Trace Points)	Construction Time (s)	Time to Match (µs)
OpenStack	296	46 / 561	8 / 515.1 / 911	4.8 / 23694	0.7	730
Uber	4194	2815 / 27196	2 / 10.3 / 2060	3.8 / 28945	13.3	499
HDFS	111	1032 / 78832	208 / 513.1 / 573	54.0 / 529468	470.6	7123

We try to answer the following questions: Can the search space represent the instrumentation in various systems? Is Pythia capable of automatically and quickly localizing the source of problems in real distributed applications? Which search strategies work best for different problems? Are Pythia's instrumentation budget mechanisms effective?

7.5.1 Setup

We test Pythia using three different distributed systems that each use a different tracing framework.

OpenStack: For our experiments, we use an open-source distributed application for managing clouds, OpenStack. OpenStack is written in Python and is composed of many services for managing compute nodes, images, networking or reservations. We choose OpenStack because it is a popular application initially developed by NASA and commonly used by the industry as well as academia.

We use OpenStack version Stein, set up on 3 nodes with Ubuntu 18.04, Linux Kernel 4.15.0 on CloudLab [Duplyakin et al., 2019]. Each node has 8-core X-Gene AppliedMicro 64-bit ARM CPUs and 64 GB of memory. Even though OpenStack

can support much larger installations, we set up a small OpenStack cluster because most nodes in large-scale installations are compute nodes and the OpenStack activity on compute nodes is limited. We configure OpenStack to have two compute nodes with nova-compute services on them, and a single controller node with the remaining OpenStack services.

OpenStack has built-in tracing using OSProfiler, which we augment to include log points into the traces, and add the capability to enable/disable trace points. 22 of the 296 trace points in the search space originate from log points. Enabling/disabling is done by keeping a separate file for each trace point that is written to by Pythia agents. OSProfiler uses spans to record traces, which lets us use the hierarchical relationship between spans.

Uber: We obtained traces from Uber Technologies collected from a single region. These traces are representative of real-world workloads at companies. The traces contain samples of production and test workloads that were executing in a single geographic region when the traces were collected. Requests were randomly sampled to reduce tracing overhead. We processed a total of 100k events from 18k requests.

HDFS: We use the distributed storage application, HDFS, written in Java, to test Pythia. We chose HDFS because (1) it is a commonly used distributed application, (2) it has a highly parallel design where each write request is written to multiple data nodes, and (3) the tracing is very detailed, and event-based instead of span-based like our other two systems and contains no hierarchy.

We use HDFS version 2.7.2 set up on Ubuntu 16.04 nodes with Linux Kernel 4.4.0 on CloudLab [Duplyakin et al., 2019] using 8-core Intel Xeon D-1548 CPUs with Hyperthreading enabled and 64 GB of memory per node. Our HDFS setup uses one name node and three data nodes for three-way duplication of data.

HDFS is traced using X-trace [Fonseca et al., 2007], which we augment to add the

capability to enable/disable trace points using a shared file similar to OSProfiler. We use X-trace's own back end and trace construction method.

Choosing Pythia inputs: The major inputs we decide on are the cycle time, variance and response time threshold, group maturity, and the budget, as described in § 7.2. The cycle time is chosen to be 30 seconds for HDFS and 2 minutes for OpenStack. This is chosen to allow for Pythia to collect enough traces in one cycle. HDFS requests are fast, one read request takes 1 second to complete, while an OpenStack VM creation can take 1 minute or longer. For variance and response time, we do not set a threshold and set Pythia to always investigate the largest problem in the system. For group maturity, we wait until a group has 5 requests before making a decision using it. We set a budget of maximum 3 trace points to enable per cycle. We set no limit on any other resource, except for § 7.5.4.

7.5.2 Search Space and Matching Scalability

Methodology: In order to construct the search space from distributed applications, we enable all the available instrumentation and run a test workload that performs different types of requests, both concurrently and sequentially. For the Uber traces, we use all traces to construct the search space. To measure matching performance, we use the paths that were used to create the search space.

Results: Table 7.2 shows the statistics about the search spaces. The overall results based on the ratio of unique to overall paths, search space size and time to match show that our search space can adequately represent all three applications. The number of unique trace points in the search space and the path lengths vary significantly based on the system. However, in all cases, approximately only 10% of the paths observed are unique over all paths in the traces, indicating significant amount of repeated behavior that Pythia can exploit.

The Uber results show that our search space and matching method can accom-

Table 7.3: Performance problems we use to evaluate Pythia.

Problem	Application	Description	Type
1	OpenStack	Missing index on a performance-critical database column	Configuration
2	OpenStack	Inefficient implementation causes simple commands to take a long time	Code slowdown
3	OpenStack	Too low limit on simultaneous server creations throttles performance	Contention
4	OpenStack	Synthetically injected worst-case problem for hierarchical search	Synthetic
5	OpenStack	Synthetically injected worst-case problem for flat search	Synthetic
6	HDFS	Inefficient implementation of Java Runtime Library function	Code slowdown
7	HDFS	Synthetically injected worst-case problem for flat search	Synthetic

modate more trace points. The matching time is the shortest among all systems. If developers were to add more instrumentation, this would cause no problems for our implementation.

The time to match is measured for single-threaded matching, even though matching is embarrassingly parallel over the paths in the search space so it can be sped up. The HDFS search space takes 7 ms to find the matching paths, the longest among the systems. This is because of the very high number of concurrency and synchronization that result in many paths even though the number of unique trace points are low. The matching duration is a lower limit on Pythia's cycle duration, as new traces need to be processed before any decisions can be made.

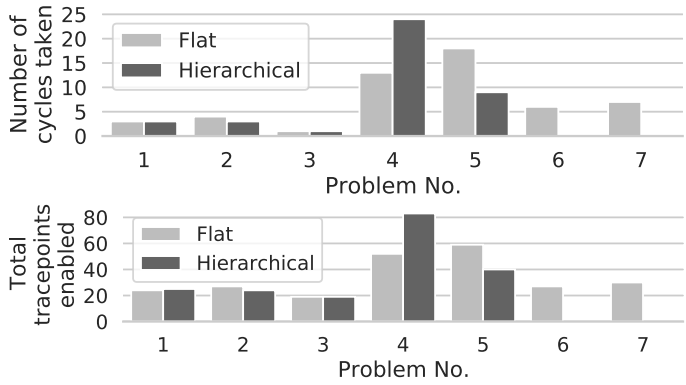


Figure 7·7: The number of cycles taken and trace points enabled by Pythia to locate the problems in Table 7.3.

7.5.3 Case Studies of Real and Synthetic Problems

Methodology: In this section we use Pythia to diagnose real world performance problems collected from bug reports and our explorations of the performance characteristics of different applications. We supplement these problems with synthetic performance problems that we inject into the application code to exercise Pythia further, by sleeping for a random amount of time. The maximum time to sleep is chosen to be half of the average request latency. Table 7.3 provides an overview of the problems we use Pythia with. For each problem, we choose the two trace points closest to the problem as the targets, and run Pythia until it enables the target edge.

In some cases, the systems do not initially have enough instrumentation to diagnose the problem. In this case, we localize the variance/latency as much as possible using the existing instrumentation and manually add more instrumentation in the relevant functions/classes. We run Pythia using all the added instrumentation to get the numbers used in our results.

For the workload, in OpenStack we use a mixture of reservation, server and floating IP create, list and delete requests, and the nova usage-list request. For HDFS, we use a workload composed of read requests.

Results: The cycles taken by Pythia before it enables the target edge are given

in Fig. 7·7. We can only run the flat search strategy for HDFS because the traces in HDFS contain no hierarchy. The results show that for our selection of real problems, Pythia can find the problem within 10 cycles with either search strategy, and always finds the problem with less than 30% of the trace points enabled.

Bug 1: Missing database index (high latency) [Laski, 2013]: The reservations handled by OpenStack are stored in a database, and the database schema was lacking an index in the UUID column. In the OpenStack Stein version that we are using, reservations are handled by the Blazar service. To re-create the problem, we remove the index on the column LEASE_ID in the RESERVATIONS table. We also populate the reservations table before starting Pythia.

As we execute our workload, the number of reservations in the database increase and the LEASE-SHOW requests get slower. This is caused by the sequential scan of the database for the lease ID. The sequential scan can be replaced by a faster index scan, by adding an index to this column, which keeps the lease IDs in a BTree.

With the hierarchical search strategy, Pythia first enables the skeletons for each request type, and as the LEASE-SHOW requests have the highest latency, it hierarchically enables more instrumentation in this request. In the traces, edges with low latency include authentication and network-latency related edges. Edges measuring accesses to the RESERVATIONS table have the highest latency. After this, the operator can add an index to the necessary column to see if the problem is resolved. When we add the index, the average latency for the LEASE-SHOW requests drop from 25 seconds to 16 seconds.

Bug 2: Inefficient implementation (high latency) [Messina, 2015]: The NOVA USAGE-LIST command returns the usage of each tenant over a time period. It takes 57 seconds for the request to return for an OpenStack installation which had 5900 VMs, which makes this request among the slowest requests of OpenStack. Pythia

then enables more instrumentation for this request.

Among the trace points that Pythia enables are 6 repeated GET requests to `/compute/v2.1/os-simple-tenant-usage` that make up for all but the last 500 ms of the request latency. These requests return all the records for individual servers that have been created in the given time period in batches, as well as a summary, which is the only thing that is printed to the user. As Pythia digs in further, the individual function which executes the database query and creates the summary is also enabled. It can be seen in this function that a group of server records are queried, and the time for each of them is calculated separately and then aggregated within Python code. The resolution involves either not returning each server record to the user, which are then discarded, or to perform the aggregation step in a database query, and get only the summary results from the database.

Bug 3: Low concurrent server creation limit (high variance) [Ates et al., 2019a]: If we issue a high number of concurrent server create requests (20), the highest variance edge for the server create request becomes an edge between two nodes in the initial skeleton, as it also represents concurrency. Thus, Pythia immediately enables it as part of the skeleton. This edge corresponds to 7 lines of code, 4 of which are comments. In the remaining 3 lines, a semaphore (NOVA.COMPUTE.MANAGER.COMPUTE-MANAGER._BUILD_SEMAPHORE) is acquired. The trace point also records the number of holders of this lock, and CCA shows that this new variable correlates the most with overall request latency, with a correlation value of 0.85 and a P-value of 10^{-5}. Inspecting the initialization of this semaphore shows that the configuration option MAX_CONCURRENT_BUILDS indicates the number of simultaneous VM creations within a single host, explaining the root cause of the high variance in simultaneous VM creations.

Bug 6: Java Runtime Bug (high variance) [Chung, 2016]: In every read

request of our HDFS workload, the highest variance edge is in the initialization part of the client, and represents a single call to the JAVA.LANG.REFLECT function PROXY.NEWPROXYINSTANCE. Since the Java library contains no tracing instrumentation, Pythia can not localize the problem further; however, we found a patch to improve the performance of this function for our OpenJDK version [Chung, 2016].

Bugs 4, 5, 7: Synthetic problems: In order to investigate the trade-off between the Flat and the Hierarchical search strategies, we inject synthetic problems to worst-case locations. We choose the deepest edge in the hierarchy of OpenStack for problem #4, which is the worst case for the hierarchical search, and an edge as far from the initial skeleton points of OpenStack and HDFS as possible for problems #5 and #7, respectively, which is one of the worst cases for the flat approach, since the initial problem edge has many trace points that can be enabled within. Even in these cases, both search strategies find the problem within 25 cycles, and by enabling less than 80, or 27% of the trace points.

7.5.4 Evaluation of Instrumentation Budget

Methodology: In order to demonstrate the budget, we run Pythia with OpenStack and our workload mixture. We set a limit of 50 MB for the trace point buffers on each OpenStack node. At second 3250, we manually delete old keys from the trace storage, reduce the workload intensity and completely remove some request types from the workload.

Results: Pythia steadily enables instrumentation in high-variance areas until the instrumentation budget is reached, after which it stops enabling. During this period, the memory usage does not significantly increase, and trace records are not dropped, and Pythia keeps processing the incoming traces. After we manually reduce the workload and free memory, Pythia starts enabling more trace points. The garbage collector can also be seen disabling trace points after second 4000. Pythia can adapt

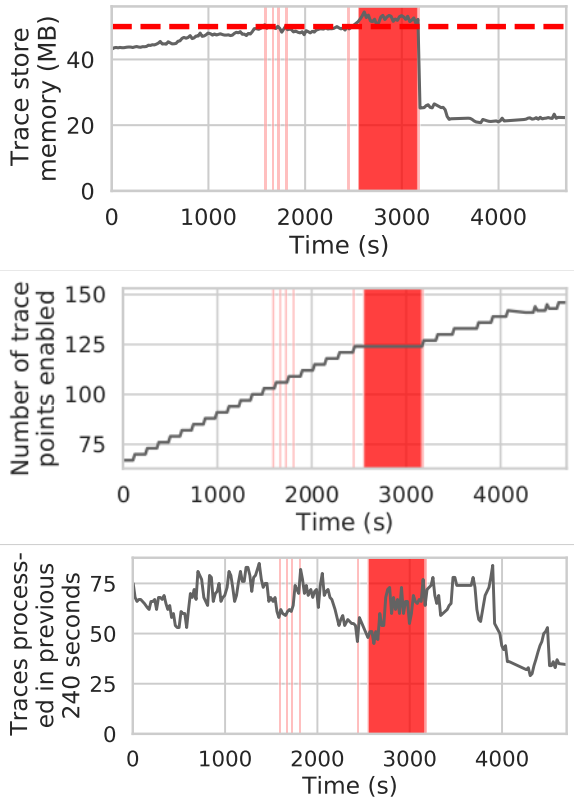


Figure 7·8: Behavior of Pythia under heavy load. When the trace store memory of 50MB is exceeded (shown in red), Pythia does not enable more instrumentation until the memory usage is reduced. This allows for a steady number of traces to be processed.

to workload changes, and can manage to stay under the instrumentation budget.

7.6 Related Work

There is a rich history of work on improving instrumentation quality. We survey existing techniques below.

Dynamic instrumentation: Much work has focused on allowing logs or customizable probes to be inserted in applications at arbitrary or predetermined locations [Erlingsson et al., 2011, Mace et al., 2015, Cantrill et al., 2004]. These tools provide a powerful basis on which to build automated approaches to customizing

instrumentation. PivotTracing [Mace et al., 2015] is designed to correlate monitoring data collected for specific requests across the processes involved in their workflows. It could be a perfect fit on which to build STAIFs if the query language it exposes is extended to support STAIFs' specific use case.

Automatically customizing instrumentation during runtime: Several past approaches explore how to customize instrumentation to diagnose correctness problems observed in individual processes [Arumuga and Liblit, 2010, Zhao et al., 2017, Liblit et al., 2003, Zuo et al., 2016]. For example, the approach described in Arumuga et al. [Arumuga and Liblit, 2010] proposes a method that automatically enables instrumentation to diagnose correctness problems in single processes. Log20 [Zhao et al., 2017] helps diagnose non fail-stop correctness problems by automatically identifying where log statements must be placed to differentiate code paths. Zuo et al. iteratively refine what instrumentation choice should be enabled to explain failures [Zuo et al., 2016]. STAIFs differ from these approaches in that they focus on performance problems and distributed applications.

Enhancing instrumentation offline: Many existing techniques aim to enhance instrumentation offline before applications are run [Yuan et al., 2012a, Yuan et al., 2012b]. For example, Yuan et al. identify what extra information is needed in existing logs to help diagnose failures [Yuan et al., 2012b]. In another paper, Yuan et al. [Yuan et al., 2012a] survey create a tool that automatically inserts instrumentation at key locations determined by an analysis of common problems types. STAIFs complement these methods by helping diagnose new, unanticipated performance problems during runtime.

Other approaches: Log2 [Ding et al., 2015] decides what already-collected log statements are useful for performance problems and discards the rest. Its method is similar to our approach for disabling least-useful instrumentation. But, it does not

consider what instrumentation is needed in the first place.

7.7 Conclusion

It is difficult to know where instrumentation must already be enabled to help debug performance problems that may occur in the future. This difficulty results in wasted developer time trying to debug distributed applications, which is often spent identifying and adding useful instrumentation to systems.

This chapter presents the design of statistical trace-driven instrumentation frameworks (STAIFs), which automatically explore instrumentation choices to enable useful instrumentation for ongoing performance problems. We demonstrated the efficacy of our prototype STAIF implementation, Pythia, by using it to control instrumentation in OpenStack [Openstack, 2020] and HDFS [Apache Software Foundation, 2019] for real and synthetic problems. We have shown that Pythia can locate performance problems by enabling 30% of the available instrumentation in the worst case.

Chapter 8

Conclusion and Future Work

We have presented our thesis, that automated analytics methods can be used to automate the initial steps in addressing problems in distributed systems. The problems we target include performance variations, performance bugs and users executing unwanted applications. To support our thesis, we have designed and evaluated 3 different automated analytics frameworks, and have shown that they outperform the state-of-the-art in their goals. We have also identified two major roadblocks preventing adoption of automated analytics frameworks and proposed tools to help overcome them.

8.1 Future Work

Large-scale distributed systems analytics is not a solved problem. The automated analytics methods we propose are far from being used in production systems, and there is ongoing research on improving automated methods. Furthermore, any of the problems we address about distributed systems complexity and scale grow over time, with the systems' scale. In this section, we discuss possible avenues of future research.

8.1.1 Next Steps on Production Systems

Almost all of the results presented on this dissertation were collected from test platforms, using data collected from synthetic performance problems, and typically using benchmarks and proxy applications. However, most of the challenges that

are addressed by these systems manifest at the scale and complexity of the largest production systems. Therefore, it is important to deploy or evaluate these tools on production systems.

In the past years, our anomaly diagnosis work has been partially demonstrated in production systems [Brandt et al., 2018a, Brandt et al., 2018b]. This has lead to the identification of important challenges including the difficulty of collecting training data, and the lack of explainability of the existing models. Further trials in this type of deployments on production systems are sure to lead to identification of more research challenges in this area.

Another important challenge we identified while working with production systems data is the highly imbalanced nature of the application data collected from supercomputers. In our experience, most of the system time is spent on a few applications and there is a long tail of applications that run only a few times. It is an open question how our application detection framework can scale to support this kind of data, or whether any imbalanced data techniques are required.

Rapidly changing software versions in production systems may have effects on multiple of the proposed frameworks. It is an open question how often Taxonomist needs to be re-trained with newer software versions. Similarly, the search space of STAIFs would also need to be updated when the software is updated; however, ways of quickly updating the search space or having more flexible search spaces is a topic of future research.

8.1.2 Workflow-Centric Tracing

Instrumentation Plane: We have presented a general control plane that can work with different instrumentation plane providers. However, there is still room for improvement in this area. Cross-layer instrumentation that can incorporate instrumentation from the kernel as well as other stack layers is one improvement that would allow

Pythia to work with a wider range of problems. Reducing the overhead of disabled trace points is also something that is necessary to be able to deploy Pythia. The overhead is very dependent on the instrumentation plane and the features that the programming language provides, so for some applications disabled trace points can have zero overhead, but not for many.

Currently the enabling and disabling of trace points can be done independently for each request type. However, there can be situations where we only want to enable trace points for one group, or based on other parameters of the trace. It is an open question whether such an implementation is possible, and what the exact benefits will be for Pythia.

Search Strategies: We have shown two search strategies that work with our search space. However, more research on search spaces and strategies can be done to use more advanced algorithms like machine learning or reinforcement learning to enable and disable trace points, or to perform search.

8.1.3 Multivariate Time Series Explainability

Minimizing probabilities: Our problem statement maximizes the prediction probability for the target class. For binary classification, this is equivalent to minimizing the probability for the other class, but for multi-class classification, there can be differences between minimizing and maximizing. We found that in practice both false positives and true positives can be explained by maximizing one class' probability, but a similar explainability method could be designed to minimize class probabilities and get new explanations.

Approximation algorithms: We have presented a heuristic algorithm; however, we have not provided any bounds on the optimality of our algorithm. Finding approximation algorithms with provable bounds is an open problem, as such algorithms may yield better explanations in a shorter time.

Explainable models: During our experiments, we conducted a preliminary experiment using CORELS, which is an interpretable model that learns rule lists [Angelino et al., 2017]. We trained CORELS using the HPAS data set to classify the time windows between the 5 anomalies and the healthy class. Regardless of our experimentation with different configuration options, the model in our experiments took over 24 hours to train and the resulting rule list classified every time window as healthy regardless of the input features. Challenges in this direction include designing inherently explainable models that can give good accuracy for HPC time series data.

Production systems: The challenges faced when deploying ML frameworks on HPC systems can lead to important research questions, and it has been shown that user studies are critical for evaluating explainability methods [Poursabzi-Sangdeh et al., 2018]. We hope to work with administrators and deploy HPC ML frameworks with our explainability method to production systems and observe how administrators use the frameworks and explanations, and find the strengths and weaknesses of different approaches.

8.1.4 Beyond Large-Scale Distributed Systems

The automated analytics methods we proposed target large-scale systems like supercomputers or distributed applications. An unexplored area is whether the proposed frameworks are applicable to other types of systems. For example, Internet-of-Things devices may be monitored and similar automated methods can be used to monitor system health and diagnose problems. These devices have similarities with large-scale distributed systems, as they are composed of identical hardware and software, so the similarities can be exploited to design machine learning frameworks.

One specific tool that is applicable beyond the scope if was designed for is our explainability method. There are other scientific domains, such as weather and earthquake prediction, or medicine, where ML is used on high-dimensional multivariate

time series. To the best of our knowledge, our work is the first work targeting the explainability of such frameworks, and we would like to see our method applied to other domains.

Appendix A

Proof for NP-Hardness of the Counterfactual Time Series Explainability Problem

In order to prove the NP-hardness of the counterfactual time series explainability problem defined in Sec. 6.2, we first consider a simplified problem of explaining a binary classifier f by finding a minimum A such that $f(x') = 1$ for a fixed x_{dist}. The proof we use is similar to the proof by Karlsson et al. [Karlsson et al., 2018].

Lemma 1. *Given a random forest classifier f that takes m time series of length 1 as input, a test sample x_{test}, and a distractor x_{dist}, the problem of finding a minimum set of substitutions A such that $f(x') = 1$ is* NP*-hard.*

Proof. We consider the hitting set problem as follows: Given a collection of sets $\Sigma = \{S_1, S_2, \ldots, S_n \subseteq U\}$, find the smallest subset $H \subseteq U$ which hits every set in Σ. The hitting set problem is NP-hard [Karp, 1972]. We enumerate U such that each elements maps to a number between 0 and m, $U = [0, m]$.

Assume there is an algorithm to solve our problem that runs in polynomial time. We construct a special case of our problem that can be used to solve the hitting set problem described above. Assume the time series are all of length 1 and can have values 0 or 1. Construct a random forest classifier $\mathcal{R}(x) = y : \mathbb{R}^{m \times 1} \to \mathbb{R}$ of n trees $\mathcal{R} = \{T_1, T_2, \ldots, T_n\}$, $m = |U|$. Each tree T_i then classifies the multivariate time series as class 1 if any time series of the corresponding subset $S_i \subseteq [0, m]$ is 1, and classifies as class 0 otherwise. $\mathcal{R}(x)$ thus returns the ratio of trees that classify as 1, or the ratio of sets that are covered. For x_{test}, we use a multivariate time series of all 0s (which will be classified as class 0), and as the distractor x_{dist} we use all 1s, classified as class 1.

Our algorithm finds a minimum set of substitutions A such that $\mathcal{R}(x') = 1$. We can transform A to the solution of the hitting set problem H by adding the j^{th} element of U to H if $A_{jj} = 1$. Thus, $H \subseteq U$ has minimum size and hits each subset S_j, i.e., $H \cap S_j \neq \emptyset$ for all $j \in [0, n]$. Thus, we can use our algorithm to solve the hitting set problem. Since the hitting set problem in NP-hard, the existence of such a polynomial-time algorithm is unlikely. □

Theorem 1. *Given classifier f, class of interest c, test sample x_{test}, and the training set X for the classifier, the* counterfactual time series explainability problem *of finding x_{dist} and A that maximize $f_c(x')$ is* NP*-hard.*

Proof. Lemma 1 shows the NP-hardness for a special case of our problem with random forest classifiers, binary time series, binary classification and without the added problem of choosing a distractor x_{dist}. Based on this, the general case with real-valued time series and more complicated classifiers is also NP-hard. □

References

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 *SC14: International Conference for High Performance Computing, Networking, Storage and Analysis*, 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.

Alvarez Melis, D. and Jaakkola, T. (2018). Towards robust interpretability with self-explaining neural networks. In Bengio, S., Wallach, H., Larochelle, H., Grauman, K., Cesa-Bianchi, N., and Garnett, R., editors, *Advances in Neural Information Processing Systems 31*, pages 7775–7784. Curran Associates, Inc.

Alvarez-Melis, D. and Jaakkola, T. S. (2018). On the robustness of interpretability methods. arXiv:1806.08049 [cs.LG].

Alverson, B., Froese, E., Kaplan, L., and Roweth, D. (2012). Cray XC series network. `https://www.cray.com/sites/default/files/resources/CrayXCNetwork.pdf`.

Angelino, E., Larus-Stone, N., Alabi, D., Seltzer, M., and Rudin, C. (2017). Learning certifiably optimal rule lists. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '17, page 35–44, New York, NY, USA. Association for Computing Machinery.

Apache Software Foundation (2019). HDFS architecture. https://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html.

Arumuga, P. N. and Liblit, B. (2010). Adaptive bug isolation. In *International Conference on Software Engineering*, pages 255–264, New York, New York, USA. ACM Press.

Arya, V., Bellamy, R. K. E., Chen, P.-Y., Dhurandhar, A., Hind, M., Hoffman, S. C., Houde, S., Liao, Q. V., Luss, R., Mojsilović, A., Mourad, S., Pedemonte, P., Raghavendra, R., Richards, J., Sattigeri, P., Shanmugam, K., Singh, M., Varshney, K. R., Wei, D., and Zhang, Y. (2019). One explanation does not fit all: A toolkit and taxonomy of AI explainability techniques. arXiv: 1909.03012 [cs.AI].

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*, pages 440–453.

Assaf, R. and Schumann, A. (2019). Explainable deep neural networks for multivariate time series predictions. In *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI-19*, pages 6488–6490. International Joint Conferences on Artificial Intelligence Organization.

Ates, E., Aksar, B., Leung, V. J., and Coskun, A. K. (2020a). Artifact for Counterfactual Explanations for Machine Learning on Multivariate HPC Time Series Data. [Software] https://doi.org/10.5281/zenodo.3762951.

Ates, E., Aksar, B., Leung, V. J., and Coskun, A. K. (2020b). Artifact for Counterfactual Explanations for Machine Learning on Multivariate HPC Time Series Data. [Dataset] https://doi.org/10.5281/zenodo.3760027.

Ates, E., Aksar, B., Leung, V. J., and Coskun, A. K. (2020c). Counterfactual explanations for machine learning on multivariate HPC time series data. under review.

Ates, E., Sturmann, L., Toslali, M., Krieger, O., Megginson, R., Coskun, A. K., and Sambasivan, R. R. (2019a). An automated, cross-layer instrumentation framework for diagnosing performance problems in distributed applications. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '19, page 165–170, New York, NY, USA. Association for Computing Machinery.

Ates, E., Toslali, M., Coskun, A. K., and Sambasivan, R. R. (2020d). Automating instrumentation decisions to diagnose performance problems in distributed applications. under review.

Ates, E., Tuncer, O., Turk, A., Brandt, J., Leung, V. J., Egele, M., and Coskun, A. K. (2018a). Taxonomist: Application detection through rich monitoring data. In Aldinucci, M., Padovani, L., and Torquati, M., editors, *Euro-Par 2018: Parallel Processing*, Cham. Springer International Publishing.

Ates, E., Tuncer, O., Turk, A., Leung, V. J., Brandt, J., Egele, M., and Coskun, A. K. (2018b). Artifact for Taxonomist: Application Detection through Rich Monitoring Data. `https://doi.org/10.6084/m9.figshare.6384248.v1`.

Ates, E., Zhang, Y., Aksar, B., Brandt, J., Leung, V. J., Egele, M., and Coskun, A. K. (2019b). HPAS: An HPC performance anomaly suite for reproducing performance variations. In *Proceedings of the 48th International Conference on Parallel Processing*, number 40 in ICPP 2019, New York, NY, USA. Association for Computing Machinery.

Auweter, A., Bode, A., Brehm, M., Brochard, L., Hammer, N., Huber, H., Panda, R., Thomas, F., and Wilde, T. (2014). A case study of energy aware scheduling on supermuc. In Kunkel, J. M., Ludwig, T., and Meuer, H. W., editors, *Supercomputing*, pages 394–409, Cham. Springer International Publishing.

Bagnall, A., Lines, J., Vickers, W., and Keogh, E. (2020). The UEA & UCR time series classification repository. `www.timeseriesclassification.com`.

Bailey, D. H., Barszcz, E., Barton, J. T., Browning, D. S., Carter, R. L., Dagum, L., Fatoohi, R. A. an d 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.

Bartolini, A., Borghesi, A., Libri, A., Beneventi, F., Gregori, D., Tinti, S., Gianfreda, C., and Altoè, P. (2018). The D.A.V.I.D.E. big-data-powered fine-grain power and performance monitoring support. In *Proceedings of the 15th ACM International Conference on Computing Frontiers*, CF '18, page 303–308, New York, NY, USA. Association for Computing Machinery.

Bhatele, A., Jain, N., Livnat, Y., Pascucci, V., and Bremer, P. (2016). Analyzing network health and congestion in dragonfly-based supercomputers. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 93–102.

Bhatele, A., Mohror, K., Langer, S. H., and Isaacs, K. E. (2013). There goes the neighborhood: performance degradation due to nearby jobs. In *ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '13. IEEE Computer Society. LLNL-CONF-635776.

Bhuyan, M. H., Bhattacharyya, D. K., and Kalita, J. K. (2011). Nado: network anomaly detection using outlier approach. *Proceedings of the International Conference on Communication, Computing & Security*, pages 531–536.

Blkkn (2020). Tracing ceph with BLKKN. `http://docs.ceph.com/docs/master/dev/blkin/`.

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.

Borghesi, A., Bartolini, A., Lombardi, M., Milano, M., and Benini, L. (2019a). Anomaly detection using autoencoders in high performance computing systems. *Proceedings of the AAAI Conference on Artificial Intelligence*, 33:9428–9433.

Borghesi, A., Bartolini, A., Lombardi, M., Milano, M., and Benini, L. (2019b). A semisupervised autoencoder-based approach for anomaly detection in high performance computing systems. *Engineering Applications of Artificial Intelligence*, 85:634–644.

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 User's Group*. `https://www.osti.gov/servlets/purl/1248686`.

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.

Brandt, J. M., Gentile, A. C., Cook, J. E., Allan, B. A., Cook, J., Aaziz, O., Tucker, T., Nichamon, N., Taerat, N., Ates, E., Tuncer, O., Egele, M., Turk, A., and Coskun, A. (2018a). Runtime HPC system and application performance assessment and diagnostics. `https://www.osti.gov/servlets/purl/1500155`.

Brandt, J. M., Gentile, A. C., Hammond, S. D., Cook, J., Allan, B. A., Tucker, T., Naksinehaboon, N., Taerat, N., Cook, J., Aaziz, O., Ates, E., Tuncer, O., Egele, M., Turk, A., Coskun, A., Izadpanah, R., and Dechev, D. (2018b). Application performance insights via system monitoring. `https://www.osti.gov/servlets/purl/1532642`.

Breiman, L. (2001a). Random forests. *Machine Learning*, 45(1):5–32.

Breiman, L. (2001b). Statistical modeling: The two cultures. *Statistical Science*, 16(3):199–215.

Breiman, L. (2017). *Classification and regression trees*. Routledge.

Byrne, A., Ates, E., Turk, A., Pchelin, V., Duri, S. S., Nadgowda, S., Isci, C., and Coskun, A. (2020). Praxi: Cloud software discovery that learns from practice. *IEEE Transactions on Cloud Computing*, pages 1–1.

Cantrill, B., Shapiro, M. W., and Leventhal, A. H. (2004). Dynamic instrumentation of production systems. In *ATC '04: Proceedings of the 2004 USENIX Annual Technical Conference*.

CCI-MOC (2020). MOC public code repository for kilo-puppet sensu modules. `https://github.com/CCI-MOC/kilo-puppet/tree/liberty/sensu`. Accessed: July 9, 2020.

Chandrasekaran, K., Karp, R., Moreno-Centeno, E., and Vempala, S. (2011). Algorithms for implicit hitting set problems. In *Proceedings of the Twenty-Second Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '11, page 614–629, USA. Society for Industrial and Applied Mathematics.

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):2379–2390.

Chow, M., Meisner, D., Flinn, J., Peek, D., and Wenisch, T. F. (2014). The mystery machine: end-to-end performance analysis of large-scale internet services. In *OSDI' 14: Proceedings of the 11th USENIX conference on Operating Systems Design and Implementation*.

Chung, M. (2016). (proxy) examine performance of dynamic proxy creation. `https://bugs.launchpad.net/nova/+bug/1481262`.

Cisco (2017). Cisco bug: Csctf52095 - manually flushing os cache during load impacts server. `https://quickview.cloudapps.cisco.com/quickview/bug/CSCtf52095`.

Cockcroft, A. (2016). Microservices workshop: Why, what and how to get there. `https://www.slideshare.net/adriancockcroft/microservices-workshop-craft-conference`.

CockroachDB (2019). `https://www.cockroachlabs.com/`.

Combs, J., Nazor, J., Thysell, R., Santiago, F., Hardwick, M., Olson, L., Rivoire, S., Hsu, C.-H., and Poole, S. W. (2014). Power signatures of high-performance computing workloads. In *Proceedings of the 2nd International Workshop on Energy Efficient Supercomputing*, E2SC '14, pages 70–78, Piscataway, NJ, USA. IEEE Press.

Cook, B., Kurth, T., Austin, B., Williams, S., and Deslippe, J. (2017). Performance variability on Xeon Phi. In *High Performance Computing*, pages 419–429, Cham. Springer International Publishing.

Cray (2018). Aries hardware counters (s-0045). Technical report, Cray. `https://pubs.cray.com/bundle/Aries_Hardware_Counters_S-0045-40/page/About_Aries_Hardware_Counter_S-0045.html`.

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

Dart, E., Rotman, L., Tierney, B., Hester, M., and Zurawski, J. (2013). The science DMZ: A network design pattern for data-intensive science. In *SC'13*, pages 1–10.

De Assis, M. V. O., Rodrigues, J. J. P. C., and Proenca, M. L. (2013). A novel anomaly detection system based on seven-dimensional flow analysis. *IEEE Global Telecommunications Conference (GLOBECOM)*, pages 735–740.

Dean, J. and Ghemawat, S. (2004). Mapreduce: Simplified data processing on large clusters. In *OSDI'04: Sixth Symposium on Operating System Design and Implementation*, pages 137–150, San Francisco, CA.

Delimitrou, C. and Kozyrakis, C. (2013). iBench: Quantifying interference for datacenter applications. In *IEEE International Symposium on Workload Characterization (IISWC)*, pages 23–33.

DeMasi, O., Samak, T., and Bailey, D. H. (2013). Identifying HPC codes via performance logs and machine learning. In *Proceedings of the First Workshop on Changing Landscapes in HPC Security*, pages 23–30, New York, NY, USA. ACM.

Desnoyers, M. (2020). Using the linux kernel tracepoints. `https://www.kernel.org/doc/Documentation/trace/tracepoints.txt`.

Dhurandhar, A., Chen, P.-Y., Luss, R., Tu, C.-C., Ting, P., Shanmugam, K., and Das, P. (2018). Explanations based on the missing: Towards contrastive explanations with pertinent negatives. In Bengio, S., Wallach, H., Larochelle, H., Grauman, K., Cesa-Bianchi, N., and Garnett, R., editors, *Advances in Neural Information Processing Systems 31*, pages 592–603. Curran Associates, Inc.

Ding, R., Zhou, H., Lou, J.-G., Zhang, H., Lin, Q., Fu, Q., Zhang, D., and Xie, T. (2015). Log^2: A cost-aware logging mechanism for performance diagnosis. In *ATC '15: Proceedings of the 2015 USENIX Annual Technical Conference*.

Dongarra, J., Heroux, M. A., and Luszczek, P. (2016). A new metric for ranking high-performance computing systems. *National Science Review*, 3(1):30–35.

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 *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, pages 155–164.

Duplyakin, D., Ricci, R., Maricq, A., Wong, G., Duerig, J., Eide, E., Stoller, L., Hibler, M., Johnson, D., Webb, K., Akella, A., Wang, K., Ricart, G., Landweber, L., Elliott, C., Zink, M., Cecchet, E., Kar, S., and Mishra, P. (2019). The design and operation of CloudLab. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, pages 1–14.

Egele, M., Woo, M., Chapman, P., and Brumley, D. (2014). Blanket execution: Dynamic similarity testing for program binaries and components. In *23rd USENIX Security Symposium*, pages 303–317, San Diego, CA. USENIX Association.

Endrei, M., Jin, C., Dinh, M. N., Abramson, D., Poxon, H., DeRose, L., and de Supinski, B. R. (2018). Energy efficiency modeling of parallel applications. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, SC '18. IEEE Press.

Erlingsson, U., Peinado, M., Peter, S., and Budiu, M. (2011). Fay: extensible distributed tracing from kernels to clusters. In *SOSP '11: Proceedings of the 23nd ACM Symposium on Operating Systems Principles*.

Evans, J. J., Groop, W. D., and Hood, C. S. (2003). Exploring the relationship between parallel application run-time and network performance in clusters. In *Annual IEEE International Conference on Local Computer Networks*, pages 538–547.

Exascale Computing Project (2020). ECP proxy applications. `https://proxyapps.exascaleproject.org/`.

Fonseca, R., Porter, G., Katz, R. H., Shenker, S., and Stoica, I. (2007). X-Trace: a pervasive network tracing framework. In *NSDI '07: Proceedings of the 4th USENIX Symposium on Networked Systems Design and Implementation*.

Ganglia (2020). Ganglia monitoring system. `ganglia.info`. Accessed: July 9, 2020.

Gee, A. H., Garcia-Olano, D., Ghosh, J., and Paydarfar, D. (2019). Explaining deep classification of time-series data with learned prototypes. arXiv:1904.08935 [cs.LG].

Ghouaiel, N., Marteau, P.-F., and Dupont, M. (2017). Continuous pattern detection and recognition in stream - a benchmark for online gesture recognition. *International Journal of Applied Pattern Recognition*, 4(2).

Giannerini, S. (2012). The quest for nonlinearity in time series. *Handbook of Statistics: Time Series*, 30:43–63.

Goswami, S. (2005). An introduction to KProbes. `https://lwn.net/Articles/132196/`.

Goyal, Y., Wu, Z., Ernst, J., Batra, D., Parikh, D., and Lee, S. (2019). Counterfactual visual explanations. In Chaudhuri, K. and Salakhutdinov, R., editors, *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, pages 2376–2384, Long Beach, California, USA. PMLR.

Grant, R. E., Pedretti, K. T., and Gentile, A. (2015). Overtime: A tool for analyzing performance variation due to network interference. In *Proceedings of the 3rd Workshop on Exascale MPI*, ExaMPI '15, pages 4:1–4:10, New York, NY, USA. ACM.

Guan, Q., Fu, S., DeBardeleben, N., and Blanchard, S. (2013). Exploring time and frequency domains for accurate and automated anomaly detection in cloud computing systems. In *2013 IEEE 19th Pacific Rim International Symposium on Dependable Computing*, pages 196–205.

Gunning, D. (2017). Explainable artificial intelligence (XAI). Technical report, DARPA. `https://www.darpa.mil/attachments/XAIProgramUpdate.pdf`.

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*.

Habib, S., Morozov, V., Frontiere, N., Finkel, H., Pope, A., and Heitmann, K. (2013). Hacc: Extreme scaling and performance across diverse architectures. In *SC '13: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 1–10.

Hall, P. (2019). On the art and science of machine learning explanations. arXiv:1810.02909 [stat.ML].

Hayes, G. (2019). mlrose: Machine Learning, Randomized Optimization and SEarch package for Python. https://github.com/gkhayes/mlrose. Accessed: July 9, 2020.

Heroux, M. A., Doerfler, D. W., Crozier, P. S., Willenbring, J. M., Edwards, H. C., Williams, A. l., Rajan, M., Keiter, E. R., Thornquist, H. K., and Numrich, R. W. (2009). Improving performance via mini-applications. Technical Report SAND2009-5574, Sandia National Laboratories. https://mantevo.github.io/pdfs/MantevoOverview.pdf.

Hoefler, T. and Belli, R. (2015). Scientific benchmarking of parallel computing systems: Twelve ways to tell the masses when reporting performance results. In *SC*, pages 73:1–73:12.

Hotelling, H. (1936). Relations between two sets of variates. *Biometrika*, 28(3/4):321–377.

Hsu, C.-W., Chang, C.-C., and Lin, C.-J. (2003). A practical guide to support vector classification. Technical report. https://www.csie.ntu.edu.tw/~cjlin/papers/guide/guide.pdf.

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.

IEEE and The Open Group (2018). POSIX Standard: dd. http://pubs.opengroup.org/onlinepubs/9699919799/utilities/dd.html.

Inadomi, Y., Patki, T., Inoue, K., Aoyagi, M., Rountree, B., Schulz, M., Lowenthal, D., Wada, Y., Fukazawa, K., Ueda, M., Kondo, M., and Miyoshi, I. (2015). Analyzing and mitigating the impact of manufacturing variability in power-constrained supercomputing. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '15, New York, NY, USA. Association for Computing Machinery.

Istio (2020). Istio: Connect, secure, control, and observe services. https://istio.io.

Jaeger (2020). Jaeger: open-source, end-to-end distributed tracing. https://www.jaegertracing.io.

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, 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*, pages 1–10.

Kaldor, J., Mace, J., Bejda, M., Gao, E., Kuropatwa, W., O'Neill, J., Ong, K. W., Schaller, B., Shan, P., Viscomi, B., Venkataraman, V., Veeraraghavan, K., and Song, Y. J. (2017). Canopy: An end-to-end performance tracing and analysis system. In *SOSP '17: Proceedings of the 26th Symposium on Operating Systems Principles*.

Karlsson, I., Rebane, J., Papapetrou, P., and Gionis, A. (2018). Explainable time series tweaking via irreversible and reversible temporal transformations. In *2018 IEEE International Conference on Data Mining (ICDM)*, pages 207–216.

Karp, R. M. (1972). *Reducibility among Combinatorial Problems*, pages 85–103. Springer US, Boston, MA.

Kasick, M. P., Tan, J., Gandhi, R., and Narasimhan, P. (2010). Black-box problem diagnosis in parallel file systems. In *Proceedings of the 8th USENIX Conference on File and Storage Technologies*, FAST'10, pages 4–4, Berkeley, CA, USA. USENIX Association.

Keahey, K., Riteau, P., Stanzione, D., Cockerill, T., Mambretti, J., Rad, P., and Ruth, P. (2018). Chameleon: a scalable production testbed for computer science research. In *Contemporary High Performance Computing: From Petascale toward Exascale*, volume 3 of *Chapman & Hall/CRC Computational Science*, chapter 5.

Kim, J., Baczewski, A. D., Beaudet, T. D., Benali, A., Bennett, M. C., Berrill, M. A., Blunt, N. S., Borda, E. J. L., Casula, M., Ceperley, D. M., Chiesa, S., Clark, B. K., Clay, R. C., Delaney, K. T., Dewing, M., Esler, K. P., Hao, H., Heinonen, O., Kent, P. R. C., Krogel, J. T., Kylänpää, I., Li, Y. W., Lopez, M. G., Luo, Y., Malone, F. D., Martin, R. M., Mathuriya, A., McMinis, J., Melton, C. A., Mitas, L., Morales, M. A., Neuscamman, E., Parker, W. D., Flores, S. D. P., Romero, N. A., Rubenstein, B. M., Shea, J. A. R., Shin, H., Shulenburger, L., Tillack, A. F., Townsend, J. P., Tubman, N. M., Goetz, B. V. D., Vincent, J. E., Yang, D. C., Yang, Y., Zhang, S., and Zhao, L. (2018). QMCPACK: an open sourceab initioquantum Monte Carlo package for the electronic structure of atoms, molecules and solids. *Journal of Physics: Condensed Matter*, 30(19):195901.

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

Koh, P. W. and Liang, P. (2017). Understanding black-box predictions via influence functions. In Precup, D. and Teh, Y. W., editors, *Proceedings of the 34th International Conference on Machine Learning*, volume 70 of *Proceedings of Machine Learning Research*, pages 1885–1894, International Convention Centre, Sydney, Australia. PMLR.

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

Kuo, C., Shah, A., Nomura, A., Matsuoka, S., and Wolf, F. (2014). How file access patterns influence interference among cluster applications. In *CLUSTER*, pages 185–193.

Lan, Z., Zheng, Z., and Li, Y. (2010). Toward automated anomaly identification in large-scale systems. *IEEE Transactions on Parallel and Distributed Systems*, 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 SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 1939–1947.

Laski, A. (2013). missing index on reservations.uuid. `https://bugs.launchpad.net/nova/+bug/1203872`.

Lawrence Livermore National Laboratory (2018). IOR benchmark application. `https://github.com/hpc/ior`.

Leung, V. J., Phillips, C. A., Bender, M. A., and Bunde, D. P. (2003). Algorithmic support for commodity-based parallel computing systems. Technical Report SAND2003-3702, Sandia National Laboratories. `https://prod-ng.sandia.gov/techlib-noauth/access-control.cgi/2003/033702.pdf`.

Liblit, B., Aiken, A., Zheng, A. X., and Jordan, M. I. (2003). Bug isolation via remote program sampling. In *PLDI '03: Programming Language Design and Implementation*. ACM.

Lipton, Z. C. (2018). The mythos of model interpretability. *Queue*, 16(3):31–57.

Lundberg, S. (2020). A game theoretic approach to explain the output of any machine learning model. `https://github.com/slundberg/shap`. Accessed: July 9, 2020.

Lundberg, S. M. and Lee, S.-I. (2017). A unified approach to interpreting model predictions. In Guyon, I., Luxburg, U. V., Bengio, S., Wallach, H., Fergus, R., Vishwanathan, S., and Garnett, R., editors, *Advances in Neural Information Processing Systems 30*, pages 4765–4774. Curran Associates, Inc.

Ma, C., Teo, Y. M., March, V., Xiong, N., Pop, I. R., He, Y. X., and See, S. (2009). An approach for matching communication patterns in parallel applications. In *IEEE International Symposium on Parallel Distributed Processing*, pages 1–12.

Mace, J., Roelke, R., and Fonseca, R. (2015). Pivot Tracing: dynamic causal monitoring for distributed systems. In *SOSP '15: Proceedings of the 25th Symposium on Operating Systems Principles*.

Mann, G., Sandler, M., Krushevskaja, D., Guha, S., and Even-dar, E. (2011). Modeling the parallel execution of black-box services. In *Proceedings of the 3rd USENIX Workshop on Hot Topics in Cloud Computing*.

Marathe, A., Zhang, Y., Blanks, G., Kumbhare, N., Abdulla, G., and Rountree, B. (2017). An empirical survey of performance and energy efficiency variation on intel processors. In *Proceedings of the 5th International Workshop on Energy Efficient Supercomputing*, E2SC'17, New York, NY, USA. Association for Computing Machinery.

Maricq, A., Duplyakin, D., Jimenez, I., Maltzahn, C., Stutsman, R., and Ricci, R. (2018). Taming performance variability. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 409–425, Carlsbad, CA. USENIX Association.

Martens, D. and Provost, F. (2014). Explaining data-driven document classifications. *MIS Quarterly*, 38(1):73–100.

Mass Open Cloud (2020). Massachusetts Open Cloud (MOC). `http://massopen.cloud`. Accessed: July 9, 2020.

McCalpin, J. D. (1995). Memory bandwidth and machine balance in current high performance computers. *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, pages 19–25.

McCalpin, J. D. (2018). HPL and DGEMM performance variability on the Xeon Platinum 8160 processor. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, SC '18. IEEE Press.

McCurdy, C. and Vetter, J. (2010). Memphis: Finding and fixing numa-related performance problems on multi-core platforms. In *2010 IEEE International Symposium on Performance Analysis of Systems Software (ISPASS)*, pages 87–96.

Mehrotra, P., Djomehri, J., Heistand, S., Hood, R., Jin, H., Lazanoff, A., Saini, S., and Biswas, R. (2012). Performance evaluation of Amazon EC2 for NASA HPC applications. In *Proceedings of the 3rd Workshop on Scientific Cloud Computing*, ScienceCloud'12, page 41–50, New York, NY, USA. Association for Computing Machinery.

Message Passing Interface Forum (1994). MPI: A Message-Passing Interface Standard. Technical report, Knoxville, TN, USA. `http://mpi-forum.org/docs/mpi-1.0/mpi-10.ps`.

Messina, A. (2015). "Nova usage" taking too much time with many VMs in database. `https://bugs.launchpad.net/nova/+bug/1481262`.

Miller, T. (2019). Explanation in artificial intelligence: Insights from the social sciences. *Artificial Intelligence*, 267:1–38.

Mirantis OSProfiler (2020). OSProfiler. `https://docs.openstack.org/osprofiler/latest/`.

Mothilal, R. K., Sharma, A., and Tan, C. (2020). Explaining machine learning classifiers through diverse counterfactual explanations. In *Proceedings of the 2020 Conference on Fairness, Accountability, and Transparency*, FAT* '20, page 607–617, New York, NY, USA. Association for Computing Machinery.

Murdoch, W. J., Singh, C., Kumbier, K., Abbasi-Asl, R., and Yu, B. (2019). Definitions, methods, and applications in interpretable machine learning. *Proceedings of the National Academy of Sciences*, 116(44):22071–22080.

Murray, D. G., McSherry, F., Isaacs, R., Isard, M., Barham, P., and Abadi, M. (2013). Naiad: A Timely Dataflow System. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 439–455, New York, NY, USA. ACM.

Nagios (2020). The industry standard in IT infrastructure monitoring. `www.nagios.org`. Accessed: July 9, 2020.

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 ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 2029–2038.

National Energy Research Scientific Computing Center (2020). Cori. `https://docs.nersc.gov/systems/cori/`. Accessed: July 9, 2020.

National Technology and Engineering Solutions of Sandia, LLC. (2020). Advanced systems technology test beds. https://www.sandia.gov/asc/computational_systems/HAAPS.html. Accessed: July 9, 2020.

NERSC (2016). Number of NERSC users and projects through the years. www.nersc.gov/about/nersc-usage-and-user-demographics/number-of-nersc-users-and-projects-through-the-years/. Accessed January 4, 2018.

Netti, A., Kiziltan, Z., Babaoglu, O., Sîrbu, A., Bartolini, A., and Borghesi, A. (2019). FINJ: A fault injection tool for HPC systems. In Mencagli, G., B. Heras, D., Cardellini, V., Casalicchio, E., Jeannot, E., Wolf, F., Salis, A., Schifanella, C., Manumachu, R. R., Ricci, L., Beccuti, M., Antonelli, L., Garcia Sanchez, J. D., and Scott, S. L., editors, *Euro-Par 2018: Parallel Processing Workshops*, pages 800–812, Cham. Springer International Publishing.

Nie, B., Xue, J., Gupta, S., Patel, T., Engelmann, C., Smirni, E., and Tiwari, D. (2018). Machine learning models for GPU error prediction in a large scale HPC system. In *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 95–106.

O'Dell, D. H. (2017). The Debugging Mindset. *ACM Queue*, 15(1):50.

Office of Inspector General (2014). Semiannual report to congress. https://www.nsf.gov/pubs/2014/oig14002/oig14002.pdf.

Openstack (2020). Openstack web site. https://www.openstack.org.

OpenTelemetry (2020). OpenTelemetry website. http://opentelemetry.io/.

OpenTracing (2020). OpenTracing website. http://opentracing.io/.

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. *International Conference on Cloud Computing and Services Science*, 1.

O'Dell, D. H. (2017). The debugging mindset. *Queue*, 15(1):71–90.

Palczewska, A., Palczewski, J., Marchese Robinson, R., and Neagu, D. (2014). Interpreting random forest classification models using a feature contribution method. *Advances in Intelligent Systems and Computing*, page 193–218.

Panda, D. K. et al. (2018). OSU Micro-Benchmarks. http://mvapich.cse.ohio-state.edu/benchmarks/.

PeacLab (2019). HPAS: HPC performance anomaly suite. https://github.com/peaclab/hpas.

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

Peisert, S. (2010). Fingerprinting communication and computation on HPC machines. *Lawrence Berkeley National Laboratory*. `http://doi.org/10.2172/983323`.

Peisert, S., Potok, T. E., and Jones, T. (2015). ASCR cybersecurity for scientific computing integrity - research pathways and ideas workshop. `http://doi.org/10.2172/1236181`.

Plimpton, S. (1995). Fast parallel algorithms for short-range molecular dynamics. *Journal of Computational Physics*, 117(1):1–19.

Poursabzi-Sangdeh, F., Goldstein, D. G., Hofman, J. M., Vaughan, J. W., and Wallach, H. (2018). Manipulating and measuring model interpretability. arXiv:1802.07810 [cs.AI].

Prometheus (2020). Monitoring system & time series database. `www.prometheus.io`. Accessed: July 9, 2020.

Quinlan, J. R. (1986). Induction of decision trees. *Machine Learning*, 1(1):81–106.

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

RedLock CSI Team (2018). Lessons from the cryptojacking attack at Tesla. Technical report.

Rencher, A. C. and Christensen, W. F. (2012). *Multivariate Regression*, chapter 10, pages 339–383. John Wiley & Sons, Ltd.

Ribeiro, M. T., Singh, S., and Guestrin, C. (2016). "Why should I trust you?": Explaining the predictions of any classifier. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16, page 1135–1144, New York, NY, USA. Association for Computing Machinery.

Ribeiro, M. T. C. (2020). Lime: Explaining the predictions of any machine learning classifier. `https://github.com/marcotcr/lime`. Accessed: July 9, 2020.

Rosenberg, E. (2018). Nuclear scientists logged on to one of Russia's most secure computers — to mine bitcoin. *The Washington Post*.

Russell, S. and Norvig, P. (2009). *Artificial Intelligence: A Modern Approach*. Prentice Hall Press, USA, 3rd edition.

Sambasivan, R. R. and Ganger, G. R. (2012). Automated diagnosis without predictability is a recipe for failure. In *Proceedings of the 4th USENIX conference on Hot Topics in Cloud Computing*, pages 21–21. USENIX Association.

Sambasivan, R. R., Shafer, I., Mace, J., Sigelman, B. H., Fonseca, R., and Ganger, G. R. (2016). Principled workflow-centric tracing of distributed systems. In *SoCC '16: Proceedings of the Seventh Symposium on Cloud Computing*.

Sambasivan, R. R., Zheng, A. X., De Rosa, M., Krevat, E., Whitman, S., Stroucken, M., Wang, W., Xu, L., and Ganger, G. R. (2011). Diagnosing performance changes by comparing request flows. In *NSDI'11: Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*.

Sato, K., Ahn, D. H., Laguna, I., Lee, G. L., Schulz, M., and Chambreau, C. M. (2017). Noise injection techniques to expose subtle and unintended message races. In *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '17, page 89–101, New York, NY, USA. Association for Computing Machinery.

Schilz, M., Belak, J., Bhatele, A., Bremer, P.-T., Bronevetsky, G., Casas, M., Gamblin, T., Isaacs, K., Laguna, I., Levine, J., Pascucci, V., Richards, D., and Rountree, B. (2014). Performance analysis techniques for the exascale co-design process. *Advances in Parallel Computing*, 25:19–32.

Schlegel, U., Arnout, H., El-Assady, M., Oelke, D., and Keim, D. A. (2019). Towards a rigorous evaluation of XAI methods on time series. In *2019 IEEE/CVF International Conference on Computer Vision Workshop (ICCVW)*, pages 4197–4201.

Schmidt, P. and Biessmann, F. (2019). Quantifying interpretability and trust in machine learning systems. arXiv:1901.08558 [cs.LG].

Sefraoui, O., Aissaoui, M., and Eleuldj, M. (2012). OpenStack: toward an open-source solution for cloud computing. *International Journal of Computer Applications*, 55(3):38–42.

Sigelman, B. H., Barroso, L. A., Burrows, M., Stephenson, P., Plakal, M., Beaver, D., Jaspan, S., and Shanbhag, C. (2010). Dapper, a large-scale distributed systems tracing infrastructure. Technical Report dapper-2010-1, Google.

Sjogreen, B. (2018). Sw4 final report for icoe. Technical report, Lawrence Livermore National Laboratory (LLNL), Livermore, CA. `https://www.osti.gov/servlets/purl/1476865`.

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

Skinner, D., Wright, N., Fuerlinger, K., Yelick, K., and Snavely, A. (2009). Integrated performance monitoring IPM. `ipm-hpc.sourceforge.net/`. Accessed January 15, 2018.

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 Appllications*, pages 129–173.

Snodgrass, R. (1988). A relational approach to monitoring complex systems. *ACM Trans. Comput. Syst.*, 6(2):157–195.

Sterling, T., Becker, D. J., Savarese, D., Dorband, J. E., Ranawake, U. A., and Packer, C. V. (1995). Beowulf: A parallel workstation for scientific computation. In *Proceedings of the 24th International Conference on Parallel Processing*, pages 11–14.

Taerat, N., Brandt, J., Gentile, A., Wong, M., and Leangsuksun, C. (2011). Baler: deterministic, lossless log message clustering tool. *Computer Science - Research and Development*, 26(3):285.

Tanenbaum, A. S. and Van Steen, M. (2007). *Distributed systems: principles and paradigms*. Prentice-Hall.

Terpstra, D., Jagode, H., You, H., and Dongarra, J. (2010). Collecting performance data with papi-c. In Müller, M. S., Resch, M. M., Schulz, A., and Nagel, W. E., editors, *Tools for High Performance Computing 2009*, page 157–173.

The MIMD Lattice Computation (MILC) Collaboration (2016). MILC benchmark application. `http://www.physics.utah.edu/~detar/milc/`.

Thebe, O., Bunde, D. P., and Leung, V. J. (2009). Scheduling restartable jobs with short test runs. In Frachtenberg, E. and Schwiegelshohn, U., editors, *Job Scheduling Strategies for Parallel Processing*, pages 116–137. Springer Berlin Heidelberg.

Tjoa, E. and Guan, C. (2019). A survey on explainable artificial intelligence (XAI): Towards medical XAI. arXiv:1907.07374 [cs.AI].

Tuncer, O., Ates, E., Zhang, Y., Turk, A., Brandt, J., Leung, V. J., Egele, M., and Coskun, A. K. (2017). Diagnosing performance variations in HPC applications using machine learning. In Kunkel, J. M., Yokota, R., Balaji, P., and Keyes, D., editors, *High Performance Computing*, pages 355–373, Cham. Springer International Publishing.

Tuncer, O., Ates, E., Zhang, Y., Turk, A., Brandt, J., Leung, V. J., Egele, M., and Coskun, A. K. (2019). Online diagnosis of performance variation in HPC systems using machine learning. *IEEE Transactions on Parallel and Distributed Systems*, 30(4):883–896.

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*.

Ueda, Y. and Nakatani, T. (2010). Performance variations of two open-source cloud platforms. In *IEEE International Symposium on Workload Characterization (IISWC'10)*, pages 1–10.

Vef, M.-A., Tarasov, V., Hildebrand, D., and Brinkmann, A. (2018). Challenges and solutions for tracing storage systems: A case study with spectrum scale. *ACM Transactions on Storage (TOS)*, 14(2):18–24.

Wachter, S., Mittelstadt, B., and Russell, C. (2017). Counterfactual explanations without opening the black box: Automated decisions and the GDPR. *Harvard Journal of Law & Technology*, 31:841.

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, W. (2011). End-to-end tracing in HDFS. Master's thesis, Carnegie Mellon University Pittsburgh, PA. `http://reports-archive.adm.cs.cmu.edu/anon/2011/CMU-CS-11-120.pdf`.

Wang, X., Smith, K., and Hyndman, R. (2006). Characteristic-based clustering for time series data. *Data Mining and Knowledge Discovery*, 13(3):335–364.

Weil, S. A., Brandt, S. A., Miller, E. L., Long, D. D. E., and Maltzahn, C. (2006). Ceph: a scalable, high-performance distributed file system. In *OSDI '06: Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation*.

Whalen, S., Peisert, S., and Bishop, M. (2013). Multiclass classification of distributed memory parallel computations. *Pattern Recognition Letters*, 34(3):322 – 329.

Wheelwright, S., Makridakis, S., and Hyndman, R. J. (1998). *Forecasting: methods and applications*. John Wiley & Sons.

Xiong, Q., Ates, E., Herbordt, M. C., and Coskun, A. K. (2018). Tangram: Colocating HPC applications with oversubscription. In *IEEE High Performance Extreme Computing Conference*, pages 1–7.

Yang, C.-T., Lai, K.-C., and Tung, H.-Y. (2011). On construction of a well-balanced allocation strategy for heterogeneous multi-cluster computing environments. *The Journal of Supercomputing*, 56(3):270–299.

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., Park, S., Huang, P., Liu, Y., Lee, M. M., Tang, X., Zhou, Y., and Savage, S. (2012a). Be conservative: enhancing failure diagnosis with proactive logging. In *OSDI' 12: Proceedings of the 10th conferences on Operating Systems Design & Implementation*.

Yuan, D., Zheng, J., Park, S., Zhou, Y., and Savage, S. (2012b). Improving software diagnosability via log enhancement. *ACM SIGPLAN Notices*, 47(4):3–14.

Zcash Electric Coin Company (2016). Zcash open source miner challenge. `www.zcashminers.org`. Accessed: July 9, 2020.

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*, pages 2–5.

Zhao, X., Rodrigues, K., Luo, Y., Stumm, M., Yuan, D., and Zhou, Y. (2017). Log20: Fully automated optimal placement of log printing statements under specified overhead threshold. In *SOSP '17: Proceedings of the 26th Symposium on Operating Systems Principles*.

Zipkin Foundation (2020). OpenZipkin: A distributed tracing system. `http://zipkin.io/`.

Zuo, Z., Fang, L., Khoo, S.-C., Xu, G., Lu, S., Fang, L., Khoo, S.-C., and Xu, G. (2016). Low-overhead and fully automated statistical debugging with abstraction refinement. In *OOPSLA '16: Proceedings of the ACM international conference on Object oriented programming systems languages and applications*.

CURRICULUM VITAE