Diagnosing Performance Variations in HPC Applications Using Machine Learning

Ozan Tuncer[1(✉)], Emre Ates[1], Yijia Zhang[1], Ata Turk[1], Jim Brandt[2], Vitus J. Leung[2], Manuel Egele[1], and Ayse K. Coskun[1]

[1] Boston University, Boston, MA, USA
{otuncer,ates,zhangyj,ataturk,megele,acoskun}@bu.edu
[2] Sandia National Laboratories, Albuquerque, NM, USA
{brandt,vjleung}@sandia.gov

Abstract. With the growing complexity and scale of high performance computing (HPC) systems, application performance variation has become a significant challenge in efficient and resilient system management. Application performance variation can be caused by resource contention as well as software- and firmware-related problems, and can lead to premature job termination, reduced performance, and wasted compute platform resources. To effectively alleviate this problem, system administrators must detect and identify the anomalies that are responsible for performance variation and take preventive actions. However, diagnosing anomalies is often a difficult task given the vast amount of noisy and high-dimensional data being collected via a variety of system monitoring infrastructures.

In this paper, we present a novel framework that uses machine learning to automatically diagnose previously encountered performance anomalies in HPC systems. Our framework leverages resource usage and performance counter data collected during application runs. We first convert the collected time series data into statistical features that retain application characteristics to significantly reduce the computational overhead of our technique. We then use machine learning algorithms to learn anomaly characteristics from this historical data and to identify the types of anomalies observed while running applications. We evaluate our framework both on an HPC cluster and on a public cloud, and demonstrate that our approach outperforms current state-of-the-art techniques in detecting anomalies, reaching an *F-score* over 0.97.

1 Introduction

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 [12,28,31]. In addition to leading to

J.M. Kunkel et al. (Eds.): ISC High Performance 2017, LNCS 10266, pp. 355–373, 2017.
DOI: 10.1007/978-3-319-58667-0_19

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 [16], firmware bugs [1], memory leaks [6], CPU throttling for thermal control [15], and resource contention [12,18,28]. 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 system logs along with 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 [7]. 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 study, we describe a machine learning framework that can automatically detect compute nodes that have exhibited known performance anomalies and also diagnose 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 [33]. 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. Our specific contributions are:

- An easy-to-compute and fast statistical feature extraction approach that significantly reduces the amount of data required for performance analysis at runtime, while retaining relevant information for anomaly detection.
- A novel low-overhead method based on machine learning algorithms that can automatically detect and identify the anomalies that cause performance variations. We demonstrate that our approach outperforms the state-of-the-art techniques on identifying anomalies on two fundamentally different platforms: a CRAY XC30m HPC cluster and a public cloud.

The rest of the paper is organized as follows: Sect. 2 provides an overview of related work, Sect. 3 describes our machine-learning-based anomaly detection framework, Sect. 4 describes the state-of-the-art algorithms that we implement as baselines, Sect. 5 explains our experimental methodology, and Sect. 6 provides our experimental findings. Finally, we conclude in Sect. 7.

2 Related Work

Analysis of performance anomalies in large scale computing systems is a widely studied topic in the literature [25,32]. Some monitoring systems utilize raw data directly to define thresholds on monitored metrics to trigger alarms that

warn system administrators about possible performance impasses [13]. Such approaches do not provide root cause analysis, and manually defining thresholds for root cause analysis requires expert knowledge and is hard to maintain.

A critical problem in automated anomaly diagnosis based on application performance and resource usage is the overwhelming volume of data monitored at runtime [25]. Time series analysis methods such as correlation and dynamic time warping [10] incur unacceptable computational overhead when used with high dimensional data. Various dimensionality reduction techniques such as principal component analysis (PCA) have been used to address this problem [20,23,27]. However, techniques focused on reducing the dataset can sometimes eliminate features that are useful for anomaly detection (see Sect. 6.1).

Another way of addressing the data volume problem is to generate fingerprints (i.e., signatures) by transforming monitored data. Bodik et al. [14] use quantiles of measured values (e.g., 95^{th} percentile) at different time epochs to summarize the collected metric time series in a fingerprint. They further reduce this data using logistic regression to eliminate metrics that are irrelevant for anomaly detection. We use Bodik et al.'s technique as a baseline and demonstrate that our approach has superior anomaly detection accuracy (See Sect. 6).

Anomaly detection is typically orthogonal to dimensionality reduction techniques. Researchers have used statistical techniques and machine learning algorithms (i.e., either alone or after dimension reduction) for detecting and classifying specific subsystem anomalies such as high network congestion [11], poor file system performance [26], temperature-related issues [9], or out-of-memory errors [16]. These techniques can detect specific anomalies with high precision, but existing methods do not provide a generic framework to detect and classify anomalies occurring in compute nodes.

Most of the related work on automated anomaly detection use low-dimensional data collected via coarse-grained monitoring tools (e.g., 1 min or greater sampling period). Several researchers have demonstrated that a detailed view on how platform resources are being utilized using finer-grained monitoring (e.g., sampling every second) can provide better insight into application behavior and can be leveraged to more effectively discover anomalies. For this purpose, they use manually selected examples of power- and thermal-issues [15] as well as file system congestion and runaway memory demands [6]. These studies do not propose an automated method to discover these problems.

A large number of studies focus on anomaly detection via log file analysis (e.g., [19,21,24]). In this work, we use application resource usage and performance characteristics to detect anomalies instead of relying on system logs; hence, our work is orthogonal to log-file based anomaly detection approaches.

To the best of our knowledge, our work is the first to address the anomaly detection and classification problem using an automated framework in conjunction with fine-grained monitoring tools in HPC systems. By leveraging statistical features that are useful for time series clustering, our anomaly detection method is able to diagnose anomalies more accurately than other known approaches.

3 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. We apply our anomaly diagnosis offline after application runs are complete. In future work, online or periodic anomaly detection can be performed by extending our framework. In the next subsections, we explain the details of our proposed approach on feature extraction and machine learning.

3.1 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 [7]. 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 Sect. 5.1, are not specific to any monitoring infrastructure and the proposed framework can be coupled with different HPC monitoring systems (e.g., [2,5,7]). 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 [35]:
  - *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 [36].
- *Linearity* is a measure of how well a time series can be forecasted with traditional linear models [22].
- *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 Sect. 6.5.

3.2 Anomaly Diagnosis Using Machine Learning

Our machine-learning-based anomaly diagnosis approach is depicted in Fig. 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 anomaly and when we inject a synthetic anomaly to one of the nodes (see Sect. 5.2 for details on injected anomalies). When an application finishes executing, we compute statistical features using the metrics collected from individual nodes as described in Sect. 3.1. 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.

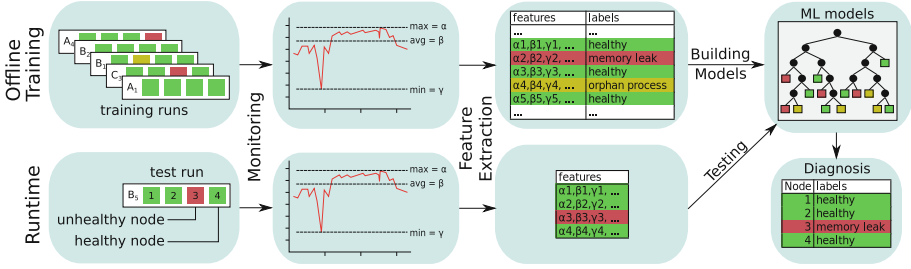


Fig. 1. Overall system architecture. Machine learning models built offline are used for classifying observations at runtime.

4 Baseline Methods

We implemented two state-of-the art methods as baselines of comparisons: the statistical approach proposed by Lan et al. [27] (referred as "ST-Lan"), and the fingerprinting approach of Bodik et al. [14] (referred as "FP-Bodik").

4.1 ST-Lan [27]

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 FP-Bodik [14]

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.

5 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.

5.1 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 [33] 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) [7]. 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)
- 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 MW datacenter dedicated for research purposes [3].

In MOC, we use virtual machines (VMs) managed by OpenStack [30], 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 [34]. Every 5 s, 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 [4].

5.2 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:

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

- *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 [6].
- *Orphan processes:* When a job terminates incorrectly, it may result in orphan processes that continue using system resources such as memory and CPU [16,17].
- *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 2.2 GHz [32].

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 125 ms every 250 ms.

5.3 Applications

In order to test our system with a variety of applications, we use the NAS Parallel Benchmarks (NPB) [8]. We pick five NPB applications (bt, cg, ft, lu and sp), with which we can obtain feasible running times (10–15 min) 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.

5.4 Implementation Details

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

During pre-processing, 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 [29] for the implementations of these algorithms.

ST-Lan: This algorithm uses the first $N = 3$ independent components determined 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. 2. $N = 7$ and

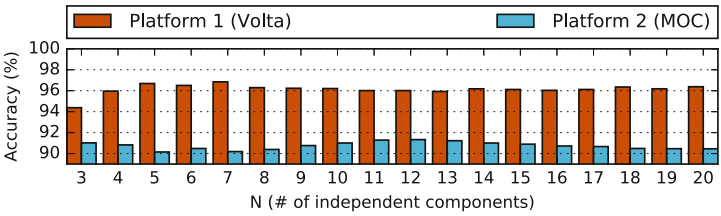


Fig. 2. Classification accuracy of ST-Lan w.r.t. number of independent components used in the algorithm for the two platforms used in this study.

$N = 12$ provide the highest accuracy on Volta and MOC, respectively. We settled on $N = 10$ as it provides a good middleground 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 an 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 [14], Bodik et al. select the epoch length as 15 min 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 s.

6 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:

$$precision_{C_i} = (\#\text{ of correct predictions})_{C_i} / (\#\text{ of predictions})_{C_i} \quad (1)$$

$$recall_{C_i} = (\#\text{ of correct predictions})_{C_i} / (\#\text{ of elements})_{C_i} \quad (2)$$

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

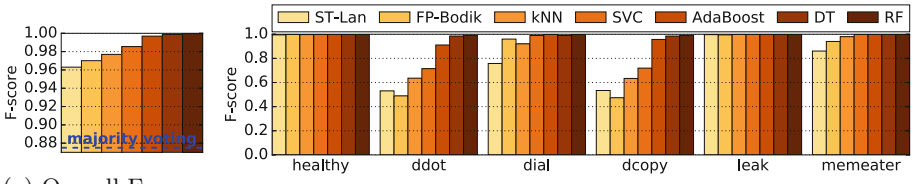
that class in our data set. The harmonic mean of these weighted average values is the overall F-score.

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, 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.

6.1 Anomaly Detection and Classification

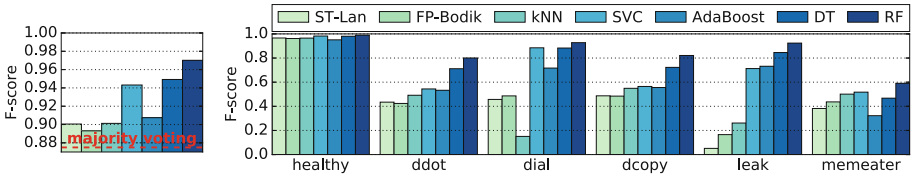
Figures 3 and 4 show the effectiveness of the anomaly detection approaches in terms of overall and per-anomaly F-scores in Volta and MOC environments, respectively. Note that half of our application runs use 4 healthy nodes and the other half use 3 healthy nodes and a single unhealthy node. Hence, the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



(a) Overall F-score

(b) F-scores for each anomaly

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



(a) Overall F-score

(b) F-scores for each anomaly

Fig. 4. F-scores for anomaly classification in MOC.

Table 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 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)

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. 3 demonstrates the importance 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 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 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 3 and 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 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

Table 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. 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

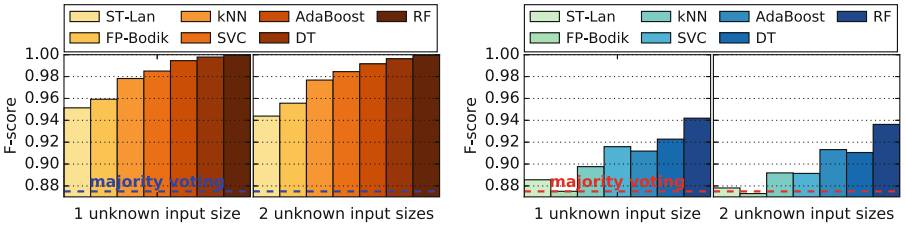
period by increasing it to 5 s; however, the impact on classification accuracy is negligible. We believe that the reduction in accuracy in MOC mainly stems from the noise in the virtualized environment, caused by the interference due to VM consolidation and migration.

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.

6.2 Classification with Unknown Application Input Sizes

In a real-world scenario, we expect to encounter application input sizes other than those used during training. This 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 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.



(a) Volta (b) MOC

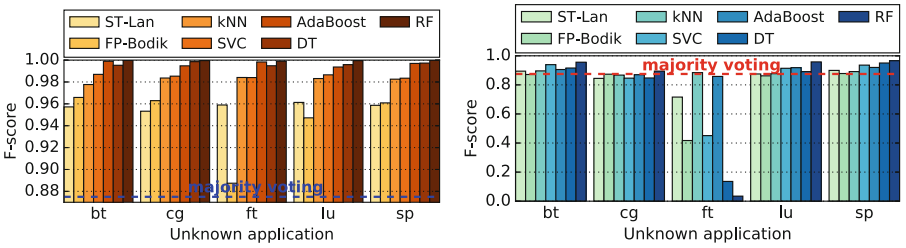
Fig. 5. Overall F-score when the training data excludes one or two input sizes and the testing is done using only the excluded input sizes

6.3 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.

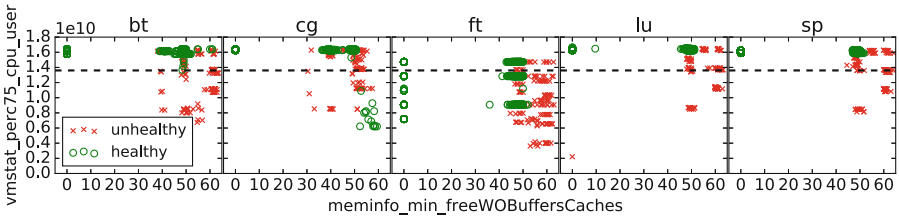
Figure 6 shows the overall F-score of the detection algorithms for each unknown application. The most prominent result in the figure is that most algorithms have very poor classification accuracy in MOC when the unknown application is *ft*. Figure 7a illustrates how *ft* is different than other applications 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 of metrics is significantly larger. Hence, DT is able to find more reliable features to classify healthy runs as depicted in Fig. 7b.

Figure 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

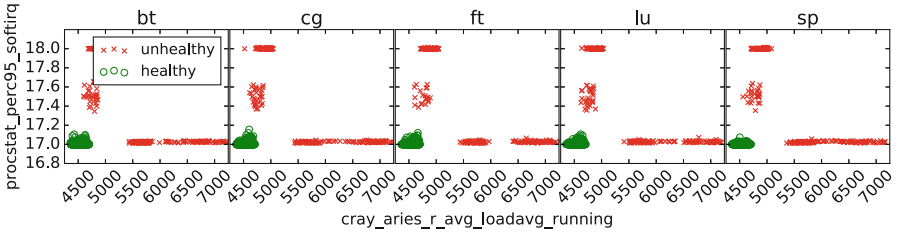


(a) Volta (b) MOC

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



(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.

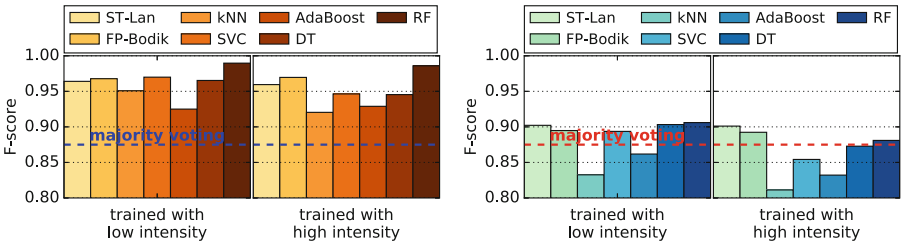
Fig. 7. The scatter plots of the datasets for the most important two features used by DT to classify healthy data.

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.

6.4 Classification with Unknown Anomaly Intensities

In this section, 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 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 low anomaly intensity, the algorithms misclassify some of the 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.



(a) Volta (b) MOC

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

6.5 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 s sampling period, takes 10.1 s on average using a single thread. This translates into 11 ms 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 $11/48 = 0.23$ ms computational overhead per second (0.02%) on Volta servers. Performing classification with trained machine learning algorithms takes approximately 10 ms 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.

7 Conclusion and Future Work

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. In this paper, we present an automated, low-overhead, and highly-accurate framework using machine learning for detection and identification of anomalies in HPC systems. 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.

In this work, we have focused on a subset of NPB applications, for which we have observed mostly flat profiles. As future work, we will explore runtime detection of anomalies considering applications that contain substantial variations in their resource usage. We are also planning to embed our solutions within the LDMS monitoring framework and evaluate our approach with a wider set of real-life applications.

Acknowledgments. This work has been partially funded by Sandia National Laboratories. Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

References

1. Cisco bug: Csctf52095 - manually flushing os cache during load impacts server. https://quickview.cloudapps.cisco.com/quickview/bug/CSCtf52095
2. Ganglia. ganglia.info
3. Massachusetts Open Cloud (MOC). http://info.massopencloud.org
4. MOC public code repository for kilo-puppet sensu modules. https://github.com/CCI-MOC/kilo-puppet/tree/liberty/sensu. Accessed 27 Oct 2016
5. Nagios. www.nagios.org
6. Agelastos, A., Allan, B., Brandt, J., Gentile, A., Lefantzi, S., Monk, S., Ogden, J., Rajan, M., Stevenson, J.: Toward rapid understanding of production HPC applications and systems. In: IEEE International Conference on Cluster Computing, pp. 464–473, September 2015
7. Agelastos, A., et al.: The lightweight distributed metric service: a scalable infrastructure for continuous monitoring of large scale computing systems and applications. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC), pp. 154–165, November 2014
8. Bailey, D.H., et al.: The NAS parallel benchmarks - summary and preliminary results. In: Proceedings of the ACM/IEEE Conference on Supercomputing, pp. 158–165, August 1991
9. Baseman, E., Blanchard, S., Debardeleben, N., Bonnie, A., Morrow, A.: Interpretable anomaly detection for monitoring of high performance computing systems. In: Outlier Definition, Detection, and Description on Demand Workshop at ACM SIGKDD, San Francisco, August 2016
10. Berndt, D.J., Clifford, J.: Using dynamic time warping to find patterns in time series. In: Proceedings of the 3rd International Conference on Knowledge Discovery and Data Mining, vol. 10, pp. 359–370, August 1994
11. Bhatele, A., Titus, A.R., Thiagarajan, J.J., Jain, N., Gamblin, T., Bremer, P.T., Schulz, M., Kale, L.V.: Identifying the culprits behind network congestion. In: IEEE International Parallel and Distributed Processing Symposium (IPDPS), pp. 113–122, May 2015
12. Bhatele, A., Mohror, K., Langer, S.H., Isaacs, K.E.: There goes the neighborhood: performance degradation due to nearby jobs. In: SC, pp. 41:1–41:12, November 2013

13. Bodík, P., Fox, A., Jordan, M.I., Patterson, D., Banerjee, A., Jagannathan, R., Su, T., Tenginakai, S., Turner, B., Ingalls, J.: Advanced tools for operators at amazon.com. In: Proceedings of the First International Conference on Hot Topics in Autonomic Computing, June 2006
14. Bodik, P., Goldszmidt, M., Fox, A., Woodard, D.B., Andersen, H.: Fingerprinting the datacenter: automated classification of performance crises. In: Proceedings of the 5th European Conference on Computer Systems, pp. 111–124 (2010)
15. Brandt, J., et al.: Enabling advanced operational analysis through multi-subsystem data integration on trinity. In: Proceedings of the Cray User's Group (2015)
16. Brandt, J., Chen, F., De Sapio, V., Gentile, A., Mayo, J., Pebay, P., Roe, D., Thompson, D., Wong, M.: 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, pp. 2–7, June 2010
17. Brandt, J., Gentile, A., Mayo, J., Pébay, P., Roe, D., Thompson, D., Wong, M.: 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, pp. 7–14, June 2009
18. Dorier, M., Antoniu, G., Ross, R., Kimpe, D., Ibrahim, S.: Calciom: mitigating I/O interference in HPC systems through cross-application coordination. In: IPDPS, pp. 155–164, May 2014
19. Fronza, I., Sillitti, A., Succi, G., Terho, M., Vlasenko, J.: Failure prediction based on log files using random indexing and support vector machines. J. Syst. Softw. **86**(1), 2–11 (2013)
20. Fu, S.: Performance metric selection for autonomic anomaly detection on cloud computing systems. In: IEEE Global Telecommunications Conference, pp. 1–5, December 2011
21. Gainaru, A., Cappello, F., Snir, M., Kramer, W.: Fault prediction under the microscope: a closer look into HPC systems. In: SC, pp. 77:1–77:11, November 2012
22. Giannerini, S.: The quest for nonlinearity in time series. In: Handbook of Statistics: Time Series, vol. 30, pp. 43–63 (2012)
23. Guan, Q., Fu, S.: Adaptive anomaly identification by exploring metric subspace in cloud computing infrastructures. In: IEEE 32nd International Symposium on Reliable Distributed Systems, pp. 205–214, September 2013
24. Heien, E., LaPine, D., Kondo, D., Kramer, B., Gainaru, A., Cappello, F.: Modeling and tolerating heterogeneous failures in large parallel systems. In: SC, pp. 1–11, November 2011
25. Ibidunmoye, O., Hernández-Rodriguez, F., Elmroth, E.: Performance anomaly detection and bottleneck identification. ACM Comput. Surv. **48**(1), 4:1–4:35 (2015)
26. Kasick, M.P., Gandhi, R., Narasimhan, P.: Behavior-based problem localization for parallel file systems. In: Proceedings of the 6th Workshop on Hot Topics in System Dependability, October 2010
27. Lan, Z., Zheng, Z., Li, Y.: Toward automated anomaly identification in large-scale systems. IEEE Trans. Parallel Distrib. Syst. **21**(2), 174–187 (2010)
28. Leung, V.J., Phillips, C.A., Bender, M.A., Bunde, D.P.: Algorithmic support for commodity-based parallel computing systems. Technical report SAND2003-3702, Sandia National Laboratories (2003)
29. Pedregosa, F., et al.: Scikit-learn: machine learning in Python. J. Mach. Learn. Res. **12**, 2825–2830 (2011)
30. Sefraoui, O., Aissaoui, M., Eleuldj, M.: OpenStack: toward an open-source solution for cloud computing. Int. J. Comput. Appl. **55**(3), 38–42 (2012)

31. Skinner, D., Kramer, W.: Understanding the causes of performance variability in HPC workloads. In: IEEE International Symposium on Workload Characterization, pp. 137–149, October 2005
32. Snir, M., et al.: Addressing failures in exascale computing. Int. J. High Perform. Comput. Appl. **28**, 129–173 (2014)
33. Sterling, T., Becker, D.J., Savarese, D., Dorband, J.E., Ranawake, U.A., Packer, C.V.: Beowulf: a parallel workstation for scientific computation. In: Proceedings of the 24th International Conference on Parallel Processing, pp. 11–14 (1995)
34. Turk, A., Chen, H., Tuncer, O., Li, H., Li, Q., Krieger, O., Coskun, A.K.: Seeing into a public cloud: monitoring the Massachusetts open cloud. In: USENIX Workshop on Cool Topics on Sustainable Data Centers, March 2016
35. Wang, X., Smith, K., Hyndman, R.: Characteristic-based clustering for time series data. Data Min. Knowl. Disc. **13**(3), 335–364 (2006)
36. Wheelwright, S., Makridakis, S., Hyndman, R.J.: Forecasting: Methods and Applications. Wiley, Hoboken (1998)