Online Diagnosis of Performance Variation in HPC Systems Using Machine Learning

Ozan Tuncer, Emre Ates, Yijia Zhang, Ata Turk,
Jim Brandt, Vitus J. Leung, Manuel Egele, and Ayse K. Coskun

Abstract—As the size and complexity of high performance computing (HPC) systems grow in line with advancements in hardware and software technology, HPC systems increasingly suffer from performance variations due to shared resource contention as well as software- and hardware-related problems. Such performance variations can lead to failures and inefficiencies, which impact the cost and resilience of HPC systems. To minimize the impact of performance variations, one must quickly and accurately detect and diagnose the anomalies that cause the variations and take mitigating actions. However, it is difficult to identify anomalies based on the voluminous, high-dimensional, and noisy data collected by system monitoring infrastructures.

This paper presents a novel machine learning based framework to automatically diagnose performance anomalies at runtime. Our framework leverages historical resource usage data to extract signatures of previously-observed anomalies. We first convert collected time series data into easy-to-compute statistical features. We then identify the features that are required to detect anomalies, and extract the signatures of these anomalies. At runtime, we use these signatures to diagnose anomalies with negligible overhead. We evaluate our framework using experiments on a real-world HPC supercomputer and demonstrate that our approach successfully identifies 98% of injected anomalies and consistently outperforms existing anomaly diagnosis techniques.

Index Terms—High performance computing, anomaly detection, machine learning, performance variation

1 Introduction

Extreme-scale computing is essential for many engineering and scientific research applications. These applications suffer from significant performance variations reaching up to 100% difference between the best and worst completion times with the same input data [1], [2]. Performance variations can be caused by hardware- and software-related *anomalies* such as orphan processes left over from previous jobs [3], firmware bugs [4], memory leaks [5], CPU throttling for thermal control [6], reduced CPU frequency due to hardware problems [7], and shared resource contention [8], [9]. In addition to performance degradation, these anomalies can also lead to premature job terminations [3]. The unpredictability caused by anomalies, combined with the growing size and complexity of high performance computing (HPC) systems, makes efficient system management challenging. In fact, unpredictability is a major roadblock towards the design of exascale HPC systems.

Detection and diagnosis of anomalies have traditionally relied on the experience and expertise of human operators. By continuously monitoring and analyzing system logs, performance counters, and application resource usage patterns, HPC operators can assess system health and identify the root causes of performance variations. In today's HPC systems, this process would translate into manual examination of billions of data points per day [10]. As system size and complexity grows, such manual processing becomes increasingly time-consuming and error-prone. Hence, automated anomaly diagnosis is essential for efficient operation of future HPC systems.

While a number of techniques have been proposed for detecting anomalies that cause performance variations in HPC systems [11], [12], these techniques still rely on human operators to discover the root causes of the anomalies, leading to delayed mitigation and wasted compute resources. An effective way of decreasing the delay between anomalies and remedies is to automate the *diagnosis* of anomalies [13], which paves the way for automated mitigation.

In this paper, we propose a framework to automatically detect the compute nodes suffering from previously observed anomalies and identify the type of the anomaly independent of the applications running on the compute nodes. Our framework detects and diagnoses anomalies by applying machine learning algorithms on resource usage and performance metrics (e.g., number of network packets received), which are already collected in many HPC systems. Compared to our earlier work [14], our new framework diagnoses anomalies online, i.e., without having to wait for an application to complete its execution.

We evaluate our framework on a Cray XC30m supercomputer using multiple benchmark suites and synthetic anomalies that are designed to mimic real-life scenarios. We demonstrate that our approach effectively identifies 98% of the injected anomalies while leading to only 0.08% false anomaly alarms with an *F-score* over 0.99, while the *F-score*s of other state-of-the-art techniques remain below 0.94. As shown in our evaluation, our approach successfully diagnoses anomalies even when running unknown applications that are not used during training. Our specific contributions are as follows:

- *Ozan Tuncer, Emre Ates, Yijia Zhang, Ata Turk, Manuel Egele, and Ayse K. Coskun are with the Electrical and Computer Engineering Department, Boston University, Boston, MA, 02215.*
  E-mail: {otuncer,ates,zhangyj,ataturk,megele,acoskun}@bu.edu
- *Jim Brandt and Vitus J. Leung are with Sandia National Laboratories.*
  E-mail: {brandt,vjleung}@sandia.gov

- We design and build an application-agnostic, low-overhead, online anomaly detection framework that enables automatic detection and diagnosis of previously-observed anomalies that contribute to performance variations. Using experiments on a Cray XC30m supercomputer, we demonstrate that our approach consistently outperforms state-of-the-art techniques on diagnosing anomalies.
- We introduce an easy-to-compute online statistical feature extraction and selection approach that identifies the features required to detect target anomalies and reduces the computational overhead of online anomaly diagnosis by more than 50%.

The remainder of the paper starts with an overview of the related work. Then, Sec. 3 introduces our anomaly diagnosis framework. In Sec. 4, we explain our experimental methodology. Section 5 presents our results, and we conclude in Sec. 6.

2 Related Work

In the last decade, there has been growing interest in building automatic performance anomaly detection tools for cloud or HPC systems [15]. 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 [16], [17]. 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 [18], auto-regressive integrated moving average [19], spectral Kalman filter [19], and Holt-Winters forecasting [20]. While such methods successfully detect anomalous behavior, they are not designed to identify the type of the anomalies (i.e., diagnosis). Moreover, these methods can 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 [21], DBSCAN [22], isolation forest [23], hierarchical clustering [24], k-means clustering [25], and kernel density estimation [20], as well as supervised learning algorithms such as support vector machines (SVM) [26], k-nearest-neighbors (kNN) [18], [27], random forest [28], and Bayesian classifier [29]. Although these studies can detect anomalies on HPC and cloud systems with high accuracy, only a few studies aim at diagnosing the anomaly types [13]. As we show in our evaluation (Sec. 5), 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 [11], previous works on HPC anomaly detection have also studied features such as correlation coefficients [30], Shannon entropy [31], and mutual information gain [24]. Some approaches also explored advanced feature extraction methods including principal component analysis (PCA) [12], [27], independent component analysis (ICA) [27], [29], and wavelet-transformation [32], [33]. In Sec. 5, we demonstrate that combining statistical feature extraction with anomaly-aware feature selection results in superior anomaly indicators compared to the features selected using statistical techniques such as ICA.

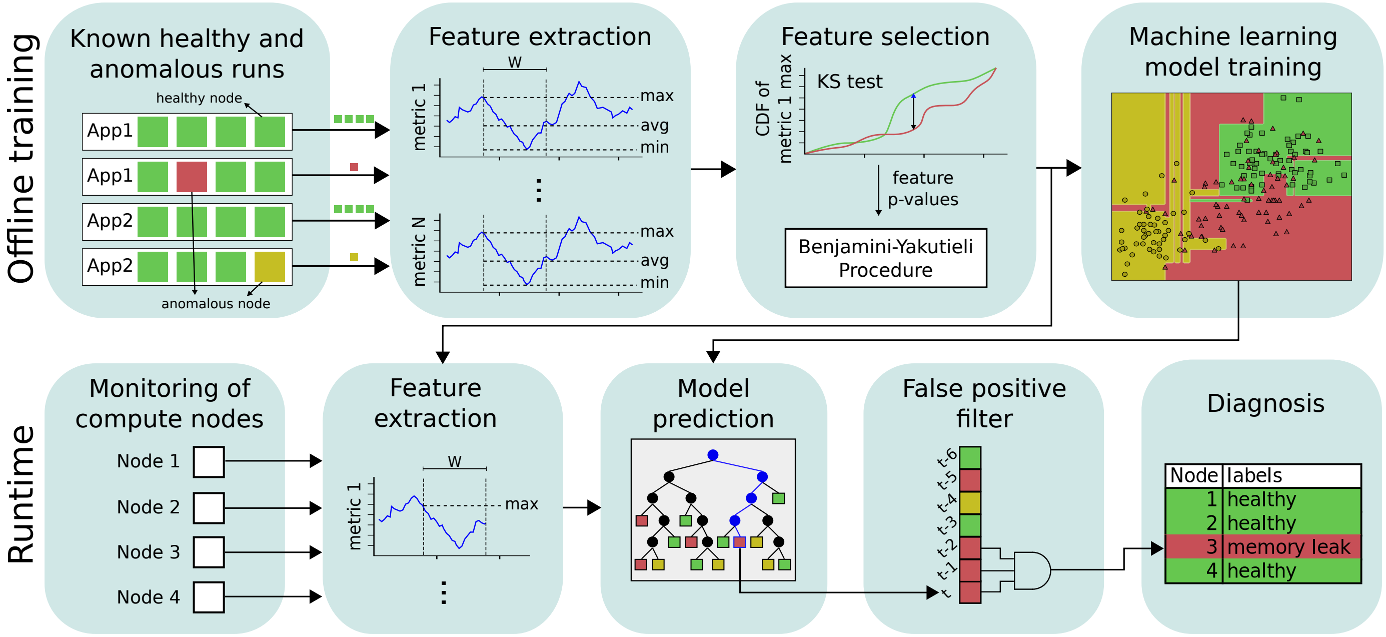
Failure detection and diagnosis on large scale computing systems is related to performance anomaly detection as these two fields share some common interests and technologies [7]. Nguyen et al. proposed a method to pinpoint faulty components by analyzing the propagation of the faults [34]. Similarly, a diagnostic tool called PerfAugur has been developed to help administrators trace the cause of an anomaly by finding common attributes that predicate an anomaly [35]. These studies focus on fault detection, whereas our goal is to find the causes of performance variation, which do not necessarily lead to failures.

In our recent work [14], we designed an automated anomaly detection framework that diagnoses anomalies in contrast to solely detecting anomalies. However, this framework detects anomalies once an application completes its execution, which can cause significant delays in anomaly diagnosis especially for long-running applications. Moreover, our earlier work did not have a solution for identifying features that are required for anomaly detection, which is necessary for low-overhead analysis at runtime.

Our work 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, has negligible computational overhead, and can diagnose anomalies at runtime. In Sec. 5, we show that our technique consistently outperforms the state-of-the-art methods in diagnosing performance anomalies in HPC systems.

3 Anomaly Detection and Diagnosis

Our goal is to quickly and accurately detect whether a compute node is *anomalous* (i.e., experiencing anomalous behavior) and classify the *type* of the anomaly (e.g., network contention or memory leak) at runtime, independent of the application that is running on the compute node. We target anomalies that are caused by applications or system software/hardware such as orphan processes, memory



Node	labels
1	healthy
2	healthy
3	memory leak
4	healthy

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

leaks, and shared resource contention. Once an anomaly is detected, mitigative measures can be taken promptly by administrators, users, or automated system management mechanisms.

To detect and classify anomalies, we introduce an automated approach based on machine learning. Figure 1 shows an overview of our framework. We leverage historical resource usage and performance data that are collected from healthy and anomalous nodes to learn the signatures of target anomalies. As an HPC application can run on multiple nodes in parallel, if any of these nodes is anomalous, the entire application's resource usage patterns may be affected. Hence, if a job running on multiple nodes suffers from an anomaly, we include only the anomalous node in the training set and discard the remaining nodes' data.

Using the data collected from known healthy and anomalous runs, we extract and identify the statistical features that are useful to detect target anomalies, and generate concise anomaly signatures using machine learning algorithms. At runtime, we monitor the compute nodes, extract the features that are identified during training, and compare these features with the anomaly signatures. The remainder of this section explains these steps in detail.

3.1 Monitoring and Feature Extraction

We leverage data that are already periodically collected in HPC systems to assess system health. These data typically consist of time series of resource usage and performance metrics such as CPU utilization, number of network packets received, and power consumption. Our framework does not depend on a specific set of collected metrics and can be coupled with a variety of HPC monitoring tools such as Ganglia [36] and LDMS [10].

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

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

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

In each measurement time step, the features are calculated in at most $O(\log W)$ computational complexity for a sliding window size of W. With a constant and small W, this enables us to generate features at runtime with negligible overhead. The value of W is determined offline based on the target anomalies and the target system (see Sec. 5.1 for details). While using a large window size typically makes it easier to detect anomalies, it delays anomaly detection.

3.2 Feature Selection

HPC system monitoring infrastructures may collect hundreds of resource usage and performance metrics per compute node [10]. Performing anomaly diagnosis based on only a subset of these metrics and calculating only the statistical features that are useful for diagnosis can save significant computational overhead. Furthermore, feature selection can improve the accuracy of classification algorithms (see Sec. 5.2).

During training, we first generate all features, and then identify the features that are useful for anomaly detection using the *Kolmogorov-Smirnov* (KS) test [37] together with the *Benjamini-Yakutieli* procedure [38]. This methodology has been shown to be successful for selection of time series features for regression and binary classification [39].

For a given feature that is extracted from an anomalous node, the KS test compares the cumulative distribution function (CDF) of that feature with the CDF of the same feature when running the same application without any anomaly. Based on the number of data points in the CDFs and the maximum distance between the CDFs, the KS test provides a *p-value*. Here, a small p-value indicates that there is a high statistical difference between the two CDFs, which, in this case, is caused by the presence of an anomaly.

The p-values corresponding to all features are then sorted and passed to the Benjamini-Yakutieli procedure. This procedure determines a p-value threshold and discards the features with p-values higher than this threshold. This threshold is calculated probabilistically based on a given *expected false discovery rate* (FDR), which is the proportion of useful features among all discarded features. As discussed in Sec. 5.2 in detail, the FDR parameter has negligible impact on the number of features selected in our dataset.

The methodology described above has been proposed for binary classification [39]. We adapt this methodology for anomaly diagnosis via multi-class classification as follows: We select the useful features for each anomaly-application pair that exists in the training data. We then use the union of the selected features to train machine learning models. This way, even if a feature is useful only to detect a specific anomaly when running a specific application, that feature is included in our analysis. Since this approach looks at every feature independently, we might filter out features whose combination (or joint CDF) is an indicator of an anomaly. However, our results indicate that this is not the case in practice as the feature selection only results in accuracy improvement (see Sec. 5). A *feature cross* (i.e., multiplying every feature with every other feature) before feature selection could be used in cases where this feature selection procedure eliminates important feature pairs.

3.3 Model Training

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

With training data from a diverse set of applications that represent the expected workload characteristics in the target HPC system, machine learning algorithms extract the signatures of anomalies independent of the applications running on the compute nodes. This allows us to identify previously observed anomaly signatures on compute nodes even when the nodes are running an *unknown* application that is not used during training. Hence, our framework does not need any application-related information from workload managers such as Slurm[1].

We focus on tree-based machine learning models since our earlier studies demonstrated that tree-based models are superior for detecting anomalies in HPC systems compared to SVM or kNN [14]. Tree-based algorithms are inherently multi-class classifiers, and can perform classification based on the behavior of combinations of features. Distance based classifiers such as kNN are often misled by our data since we have many features in our dataset, and they are from very different domains, e.g., the 75th percentile of CPU utilization and the standard deviation of incoming packets per second. Moreover, tree-based algorithms, in general, generate easy-to-understand models that lend themselves to scrutinization by domain experts. Other models such as neural networks typically provide low observability into their decision process and reasoning, and require significantly more training samples compared to tree based algorithms. Tree-based models also allow us to further reduce the set of features that needs to be generated as decision trees typically do not use all the available features to split the training set. Hence, at runtime, we calculate only the features that are selected by both the feature selection phase and the learning algorithms, further reducing the computational overhead of our framework.

We focus on three tree-based machine learning models: decision tree, random forest and adaptive boosting (AdaBoost). Decision trees [40] are constructed by selecting the feature threshold that best splits the training set, which is then placed in the top of the tree, dividing the training set into two subsets. This process is repeated recursively with the resulting subsets of the training data until each subset belongs to a single class. During testing, the decision tree compares the given features with the selected thresholds starting from the top of the tree until a leaf is found. The predicted class is the class of the leaf node. Random forest [41] is an ensemble of decision trees that are constructed using random subsets of the features. Majority voting is used to get the final prediction from the ensemble of trees. AdaBoost [42] is another type of tree ensemble that trains each tree with examples that the previous trees misclassify.

3.4 Runtime Anomaly Diagnosis

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

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

To increase robustness against false alarms, we do not raise an alarm when a node is predicted as anomalous based on data collected from a single sliding window. Instead, we consider an anomaly prediction as valid only if the same prediction persists for C consecutive sliding windows. Otherwise, we label the node as healthy. Increasing the

[1] https://slurm.schedmd.com/

parameter C, which we refer to as the *confidence threshold*, decreases the number of false anomaly alarms while delaying the detection time. Similar to W, the value of C depends on the characteristics of the target anomalies as well as the target system. Hence, C should be selected empirically based on preliminary experiments (see Sec. 5.3 for details).

4 Experimental Methodology

To evaluate the efficacy of our framework, we run controlled experiments on an HPC testbed. We mimic anomalies observed in this testbed by running synthetic programs simultaneously with various HPC applications, and diagnose the anomalies using our framework and selected baseline techniques. This section presents the details on our target HPC testbed as well as the synthetic anomalies and the applications we use.

4.1 Target HPC System

We perform our experiments on Volta, a Cray XC30m testbed supercomputer located at Sandia National Laboratories. Volta consists of 52 compute nodes, organized in 13 fully connected switches with 4 nodes per switch. Each node has 64GB of memory and two sockets, each with an Intel Xeon E5-2695 v2 CPU with 12 2-way hyper-threaded cores.

Volta is monitored by the Lightweight Distributed Metric Service (LDMS) [10]. At every second, LDMS collects 721 metrics in five categories as described below:

- Memory metrics (e.g., the amount of 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 and inactive pages; read and write counts)
- Cray performance counters (e.g., power consumption, dirty, write-back counters; received/transmitted bytes/packets)
- Aries network interface controller counters (e.g., received/transmitted packets, flits, blocked packets)

Out of the collected 721 metrics, we discard 158 metrics that are constant during all of our experiments.

4.2 Applications

To evaluate our framework using a diverse set of resource usage characteristics, we use benchmark applications with which we can obtain 10-15 minute running times using three different input configurations. Such running times occur frequently in supercomputers [43].

Table 1 presents the applications we use in our evaluation. The NAS Parallel Benchmarks (NPB) are widely used by the HPC community as a representative set of HPC applications. The Mantevo Benchmark Suite is developed by Sandia National Laboratories as proxy applications for performance and scaling experiments. These applications mimic the computational cores of various scientific workloads. In addition, we use kripke, which is another proxy application developed by Lawrence Livermore National Laboratory for HPC system performance analysis. All applications in Table 1 use MPI for inter-process and inter-node communication.

TABLE 1
Applications used in evaluation

Benchmark	Application	# of MPI ranks	Description
NAS Parallel Benchmarks [44]	bt	169	Block tri-diagonal solver
	cg	128	Conjugate gradient
	ft	128	3D fast Fourier transform
	lu	192	Gauss-Seidel solver
	mg	128	Multi-grid on meshes
	sp	169	Scalar penta-diagonal solver
Mantevo Benchmark Suite [45]	miniMD	192	Molecular dynamics
	CoMD	192	Molecular dynamics
	miniGhost	192	Partial differential equations
	miniAMR	192	Stencil calculation
Other	kripke [46]	192	Particle transport

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

In addition to the 4-node application runs, we experiment with 32-node runs with four applications (kripke, miniMD, miniAMR, and miniGhost), with which we can obtain 10-15 minute running times for two input configurations. Using these 32-node runs, we show that our framework can diagnose anomalies when running large applications after being trained using small applications (see Sec. 5.9).

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

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

Before generating features from an application run, we remove the first and last 30 seconds of the collected time series data to strip out the initialization and termination phases of the application. Note that the choice of 30 seconds is based on these particular applications and the configuration parameters used.

4.3 Synthetic Anomalies

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

TABLE 2
Target performance anomalies

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

As shown in Table 2, we experiment with three types of anomalies. Orphan processes typically result from incorrect job termination. These processes continue to use system resources until their execution is completed or until they are forcefully killed by an external signal [3], [47]. Out-of-memory problems occur due to memory leaks or insufficient available memory on the compute nodes [5]. Finally, contention of shared resources such as network links can significantly degrade performance [8].

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

1) *dcopy* allocates two equally sized matrices of type `double`, fills one matrix with a number, and copies it to the other matrix repeatedly, simulating CPU and cache interference. After 10^9 write operations, the matrix size is changed to cycle between 0.9, 5 and 10 times the sizes of each cache level. The anomaly intensity scales the sizes of these matrices.
2) *dial* stresses a single CPU core by repeatedly generating random floating-point numbers and performing arithmetic operations. The intensity sets the utilization of this process: Every 0.25 seconds, the program sleeps for $0.25 \times (1 - \text{intensity})$ seconds.
3) *leak* allocates a $(200 \times \text{intensity})$KB `char` array, fills the array with characters, sleeps for two seconds, and repeats the process. The allocated memory is never released, leading to a memory leak. After 10 iterations, the program restarts to avoid crashing the main program by consuming all available memory.
4) *memeater* allocates a $(360 \times \text{intensity})$KB `int` array and fills the array with random integers. It periodically increases the size of the array using `realloc` and fills in new elements. After 10 iterations, the program sleeps for 120 seconds and restarts.
5) *linkclog* adds a delay before MPI messages sent out of and into the selected anomalous node by wrapping MPI functions. The duration of this delay is proportional to the message size and the anomaly intensity. We emulate network contention using message delays rather than using external jobs with heavy communication. This is because Volta's size and flattened butterfly network topology with fully-connected routers prevent us from clogging the network links with external jobs.

Using synthetic anomalies with different characteristics allows us to study various aspects of online anomaly diagnosis such as window size selection, feature selection, and sensitivity against anomaly intensities. Our method, in principle, applies to other types of anomalies as well; however, depending on the anomaly, the framework will likely select a different feature set to generate anomaly signatures and perform diagnosis.

We use two anomaly scenarios: (1) persistent anomalies, where an anomaly program executes during the entire application run, and (2) random-offset anomalies, where an anomaly starts at a randomly selected time while the application is running. In the latter scenario, each application has two randomly selected anomaly start times.

4.4 Baseline Methods

We have implemented two state-of-the-art algorithms for HPC anomaly detection as baselines to compare with our work. These algorithms are an independent-component-analysis-based approach developed by Lan et al. [27] (referred to as "ICA-Lan") and a threshold-based fingerprinting approach by Bodik et al. [13] (referred to as "FP-Bodik").

4.4.1 ICA-Lan

ICA-Lan [27] relies on Independent Component Analysis (ICA) [48] to detect anomalies on HPC systems. During training, ICA-Lan first normalizes the collected time series metrics and applies ICA to extract the *independent features* of the data. Each of these features is a linear combination of the time series data of all metrics within a sliding window, where the coefficients are determined by ICA. ICA-Lan then constructs *feature vectors* composed of the top m independent features for each node and each sliding window.

To test for anomalies in an input sliding window, ICA-Lan constructs a feature vector with the coefficients used during training, and compares the Euclidean distance between the new feature vector and all the feature vectors generated during training. If the new feature vector is an outlier, the input is marked as anomalous. In the original paper [27], ICA-Lan can tell only whether a node is anomalous or not without classifying the anomaly. In our implementation, we generalize this method by replacing their distance-based outlier detector with a kNN (k Nearest Neighbor) classifier.

The original implementation of Lan et al. chooses the top $m = 3$ components. However, because we collect many more metrics compared to Lan et al.'s experiments, the top $m = 3$ are not sufficient to capture the important features and results in poor prediction performance. Hence, we use $m = 10$ as it provides good diagnosis performance in general in our experimental environment [14].

4.4.2 FP-Bodik

FP-Bodik [13] uses common statistical features and relies on thresholding to compress the collected time series data into feature vectors called *fingerprints*. Specifically, FP-Bodik first selects the metrics that are important for anomaly detection using logistic regression with L1 regularization. FP-Bodik then calculates the 25th, 50th, and 95th percentiles of the selected time series metrics for each node and each sliding time series window. Then, a healthy range of these percentiles are identified from the healthy runs in the training phase. Next, each percentile is further replaced by a tripartite value (-1, 0, or 1) that represents whether the observed value is below, within, or beyond the healthy range, respectively. FP-Bodik constructs fingerprints that are composed of all the tripartite values for each node and each monitoring window. Finally, the fingerprints are compared to each other, and a fingerprint is marked as anomalous whenever its closest labeled fingerprint (in terms of L2 distance) belongs to an anomaly class.

4.5 Implementation Details

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

While training the classifiers in our evaluation, we use periodic time series windows instead of sliding windows. This reduces the training times at the expense of a negligible decrease in the detection accuracy, allowing us to perform an extensive evaluation using hundreds of distinct training sets within feasible duration. In a production environment, a similar approach can be taken to tune the input parameters of our framework (W and C). However, to achieve the maximum detection accuracy, the final training should be performed using sliding windows.

4.6 Evaluation

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

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

We calculate the following statistical measures to assess how well the anomaly detection techniques distinguish healthy and anomalous time windows from each other:

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

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

- *Precision*: The fraction of the number of windows correctly predicted with an anomaly type to the number of all predictions with the same anomaly type.
- *Recall*: The fraction of the number of windows correctly predicted with an anomaly type to the number of windows with the same anomaly type.
- *F-Score*: The harmonic mean of precision and recall.
- *Overall F-Score*: The F-score calculated using the weighted averages of precision and recall, where the precision and recall of each class is weighted by the number of instances of that class. A naïve classifier that marks every window as healthy would achieve an overall F-score of 0.87 as approximately 87% of our data set consists of healthy windows.

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

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

5 Results

In this section, we first describe our parameter selection for sliding time series window size (W), the false discovery rate (FDR) during feature selection, and the confidence threshold (C). We then compare different anomaly detection and diagnosis methodologies in terms of classification accuracy and robustness against unknown applications and anomaly intensities that are not encountered during training. We

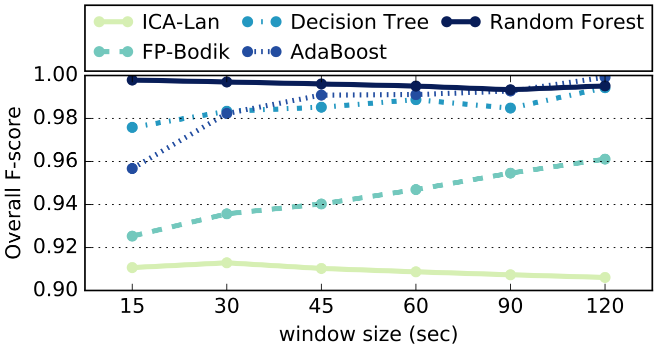


Fig. 2. The impact of window size on the overall F-score. The classification is less effective with small window sizes where a window cannot capture the patterns in the time series adequately.

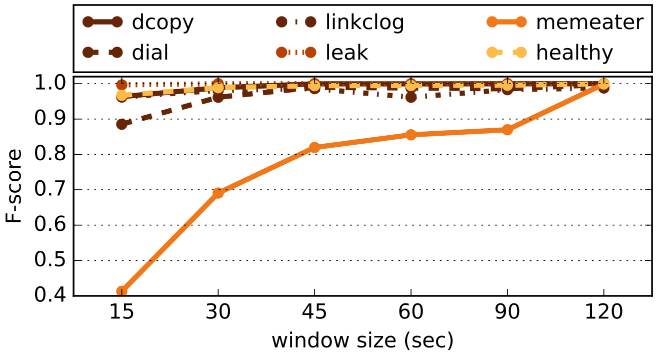


Fig. 3. The impact of window size on the per-class F-scores of the AdaBoost classifier. The F-score of memeater anomaly decreases significantly as window size get smaller.

demonstrate the generality of our framework by diagnosing anomalies during large application runs. Finally, we study the anomaly detection delay and the resource requirements of different diagnosis methodologies.

5.1 Window Size Selection

As discussed in Sec. 3.1, the size of the time series window that is used to generate statistical features may affect the efficacy of anomaly detection. While using a large window size allows capturing longer time series signatures in a window, it delays the anomaly detection.

Figure 2 shows the impact of window size on the overall F-score of baseline algorithms as well as our proposed framework with three different classifiers. The results presented in the figure are obtained using all features (i.e., without the feature selection step).

While the impact of window size on the overall F-score is below 3%, the F-scores of most classifiers tend to decrease with decreasing window size as small windows cannot capture the time series behavior adequately. The impact of the window size depends on the anomaly characteristics. This can be seen in Fig. 3, which depicts the per-class F-scores of the AdaBoost classifier for different window sizes. The F-score for the *memeater* anomaly decreases significantly with the decreasing window size. This is because the behavior of application runs with *memeater* is very similar to the healthy application behavior during *memeater*'s sleep phase, which is 120 seconds. Hence, as the window size gets smaller, more windows occur entirely within *memeater*'s sleep phase both in the training and the testing set, confusing the classifier on *memeater*'s signature. The reduction in the F-score of the healthy class due to this confusion is less significant than that of the *memeater* class because our dataset has 42 times more healthy windows than windows with *memeater*.

The window size in our framework needs to be determined based on the nature of the target anomalies and the system monitoring infrastructure. Based on the results in Figures 2 and 3, we conclude that a 45-second window size is a reasonable choice to accurately detect our target anomalies while keeping the detection delay low. For the rest of the paper, we use a window size (W) of 45 seconds.

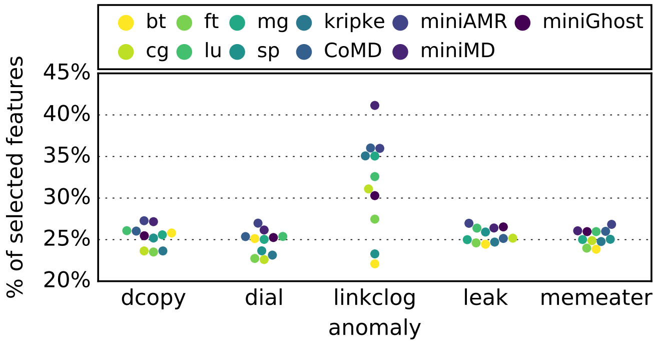


Fig. 4. The percentage of features selected by our feature selection method (Sec. 3.2) for different application-anomaly pairs. Except for the *linkclog* anomaly, less than 28% of the features are useful for anomaly detection. 41% of the features are selected for detecting linkclog.

5.2 Feature Selection

Feature selection is highly beneficial for reducing the computational overhead needed during online feature generation. Our framework's feature selection methodology identifies 43-44% of the 6193 features we generate as useful features to identify our target anomalies for an *expected False Discovery Rate* (FDR) range between 0.01% and 10%. As the FDR parameter has a negligible impact on the number of selected features, we simply use FDR $= 1\%$.

Figure 4 shows the percentage of selected features for each application-anomaly pair. While less than 28% of the features are identified as useful for detecting *dcopy*, *dial*, *leak*, and *memeater* anomalies, up to 41% of the features can be used as indicators of *linkclog*. For the applications where the *linkclog* anomaly is detrimental for performance such as miniMD, more features are marked as useful. This is because the resource usage patterns of such applications change significantly when suffering from *linkclog*. On the other hand, applications such as *bt* and *sp* are not affected by *linkclog* as they either have a light communication load or use non-blocking MPI communication. Fewer features are marked as useful for such applications.

Using a reduced feature set can also improve the effectiveness of certain machine learning algorithms. For example, random forest contains decision trees that are trained using a subset of randomly selected features. In the absence of irrelevant features, the effectiveness of random forest slightly increases as shown in Table 3. The effectiveness of other learning algorithms such as decision tree and AdaBoost are not impacted by feature selection as feature selection is embedded in these algorithms.

TABLE 3
The impact of feature selection on anomaly detection. The effectiveness of random forest improves when using only the selected features.

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

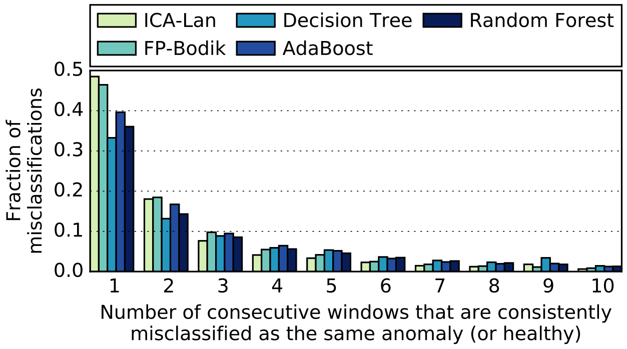


Fig. 5. Distribution of consecutive misclassifications. Most misclassifications do not persist for more than a few consecutive windows.

5.3 The Impact of Confidence Threshold

As shown in the Fig. 5, the majority of the misclassifications persist only for a few consecutive windows in all classifiers. To reduce false anomaly alarms, we filter out the non-persistent misclassifications using a confidence threshold, C. A prediction is considered as valid only if it persists for C consecutive windows.

Figure 6 shows the change in the false anomaly alarm rate and the anomaly miss rate when using various confidence thresholds. Using $C = 5$ reduces the false alarm rate by 23-44% when using the machine learning algorithms and by 25-69% when using the baseline anomaly detection algorithms. On the other hand, the anomaly miss rate increase by 15-30% and 6-34% when using the machine learning algorithms and the baselines, respectively. To keep the anomaly detection delay low while decreasing the false alarm rate in all classifiers, we use a confidence threshold of $C = 5$ for the rest of the paper.

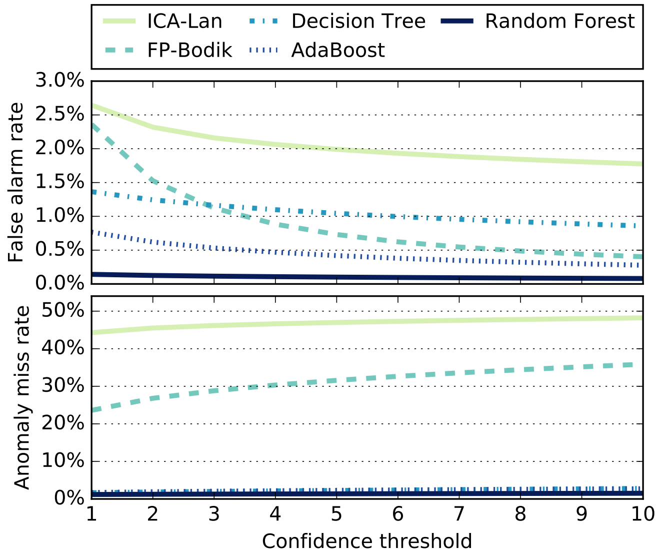
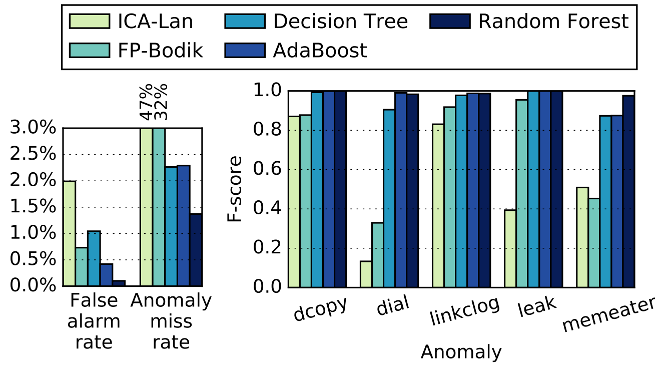


Fig. 6. The impact of confidence threshold on the false alarm rate. Filtering non-persistent anomaly predictions using a confidence threshold reduces the false alarm rate while increasing anomaly miss rate.



(a) Detection

(b) Diagnosis

Fig. 7. Anomaly detection and diagnosis statistics of various classifiers using 5-fold stratified cross-validation. Random forest correctly identifies 98% of the anomalies while leading to only 0.08% false anomaly alarms.

5.4 Anomaly Detection and Classification

Figure 7 presents the false positive and negative rates for anomaly detection as well as F-scores for the anomaly types we study for the 5-fold stratified cross validation. Our proposed machine learning based framework consistently outperforms the baseline techniques in terms of anomaly miss rate and F-scores. While FP-Bodik can achieve fewer false alarms, it misses nearly a third of the anomalies. The best overall performance is achieved using random forest, which misses only 1.7% of the anomalous windows and classifies the target anomalies nearly ideally. It raises false anomaly alarms only for 0.08% of the healthy windows, which can be decreased further by adjusting the C parameter at the expense of slightly increased anomaly miss rate and delayed diagnosis. As decision tree is a building block of AdaBoost and random forest, it is simpler and underperforms AdaBoost and random forest as expected.

The F-scores corresponding to *memeater* are lower than those for other anomalies as the classifiers tend to mispredict *memeater* as healthy (and vice versa) during the sleep phase of *memeater*, where its behavior is similar to healthy application runs. Due to the randomized feature selection in random forest, random forest also uses the features that are not the primary anomaly indicators but are still helpful on anomaly detection. Thus, random forest is more robust against noise in the data, and can still detect *memeater* where other classifiers are unsuccessful. Also note that the *memeater* anomaly degrades performance only by up to 10% while the *dcopy*, *dial*, and *linkclog* anomalies can degrade application performance by up to 300%. Hence, the detection of *memeater* is harder but is also less critical compared to other anomalies.

As seen in Fig. 7, FP-Bodik misses nearly a third of the anomalous windows. This is because even though FP-Bodik performs metric selection through L1 regularization, it gives equal importance to all the selected metrics. However, metrics should be weighted for robust anomaly detection. For instance, network-related metrics are more important for detecting network contention than memory-related metrics. Tree-based machine learning algorithms address this problem by prioritizing certain features through putting them closer to the root of a decision tree. Another reason for FP-Bodik's poor anomaly miss rate is that FP-Bodik only uses 25th, 50th, and 95th percentiles in the time series data. Other statistics such as variance and skew are also needed to capture more complex patterns in the time series.

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

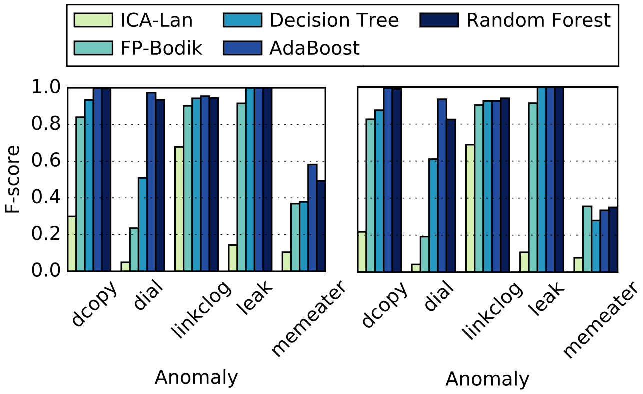
The most important ICA-Lan metrics are those with the highest absolute weight in the first ten independent components. In our models, the most important ICA-Lan metrics are the time series of idle time spent in various CPU cores. Idle CPU core time is indeed independent from other metrics in our data as some of our applications do not use all the available CPU cores in a node (see Sec. 4.2), and the decision on which cores are used by an application is governed by the operating system of the compute nodes.

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

5.5 Classification with Unknown Input Configurations

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

Figure 8 shows F-scores for each anomaly obtained during our unknown input configuration study. Except for the *memeater* anomaly, our approach can diagnose anomalies with over 0.8 F-score even when the behavior of the applications are different than that observed during training due to the unknown input configurations. The F-scores tend to decrease when more input configurations are unknown. There are two reasons for the decreasing F-scores: First, removing certain input configurations from the training set reduces the training set size, resulting in a less detailed modeling of the anomaly signatures. Second, the behavior of an application with an unknown input configuration may be similar to an anomaly, making diagnosis more difficult. One such example is the *memeater* anomaly, where healthy application runs with certain unknown input configurations are predicted as *memeater* by the classifiers.



(a) One unknown input configuration

(b) Two unknown input configurations

Fig. 8. Anomaly diagnosis statistics when the training data excludes certain *unknown* input configurations for each application and the testing is done using only the excluded input configurations.

5.6 Classification with Unknown Applications

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

Figure 9 presents the anomaly detection results when we repeat this procedure where each applications is selected once as the unknown application. With the AdaBoost and random forest classifiers, the proposed framework achieves over 0.94 overall F-score on the average, while the average F-score of ICA-Lan is below 0.65. FP-Bodik achieves a similar F-score but misses 44% of the anomalous windows when unknown applications are running.

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

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

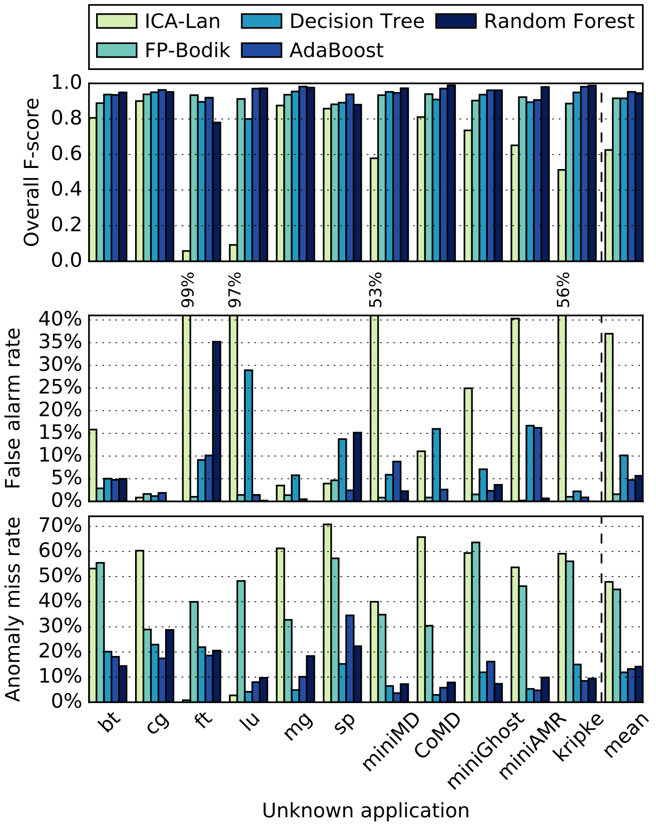


Fig. 9. Anomaly detection and diagnosis statistics when the training data excludes one application and the testing is done using only the excluded *unknown* application. With the random forest classifier, the proposed framework is robust against unknown applications, achieving over 0.94 F-score and below 6% false alarm rate on the average.

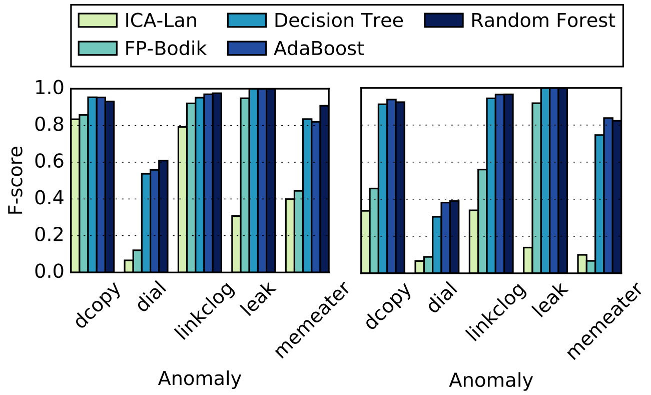
5.7 Classification with Unknown Anomaly Intensities

We also study how the diagnosis effectiveness is impacted by unknown anomaly intensities where we use distinct anomaly intensities during training and the remaining intensities during testing. Figure 10 shows the F-scores for different anomalies with unknown intensities. High F-scores indicate that the anomaly signatures do not change significantly when their intensity changes, and our proposed framework can successfully diagnose anomalies. For example, the memory usage gradually increases with the *leak* anomaly in all intensities. Hence, *leak* can be detected based on the skew in the time series of the allocated memory size.

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

5.8 Diagnosing Anomalies with Low Intensities

We study anomaly diagnosis effectiveness when the anomalies have $1/10^{\text{th}}$ of the intensities we have used so far. In this subsection, we train the framework with the anomaly intensities 20%, 50%, and 100%, and test with the anomaly intensities 2%, 5%, and 10%. Figure 11 shows the resulting per-anomaly F-scores. In *dcopy*, *leak*, and *memeater*, the intensity determines the size of the memory used in the anomaly program. With low anomaly intensities, random forest diagnoses the signatures of these anomalies with F-scores above 0.98. In *linkclog*, the intensity determines the delay injected into the MPI communication functions, which is detected even with low intensities. In *dial*, the intensity sets the utilization of the anomaly program. As the intensity drops, the impact of *dial* on the application performance and resource usage patterns decreases, making it harder and also less critical to detect the anomaly.



(a) One unknown anomaly intensity

(b) Two unknown anomaly intensities

Fig. 10. Anomaly diagnosis statistics when the training data excludes certain *unknown* anomaly intensities and the testing is done using only the excluded anomaly intensity.

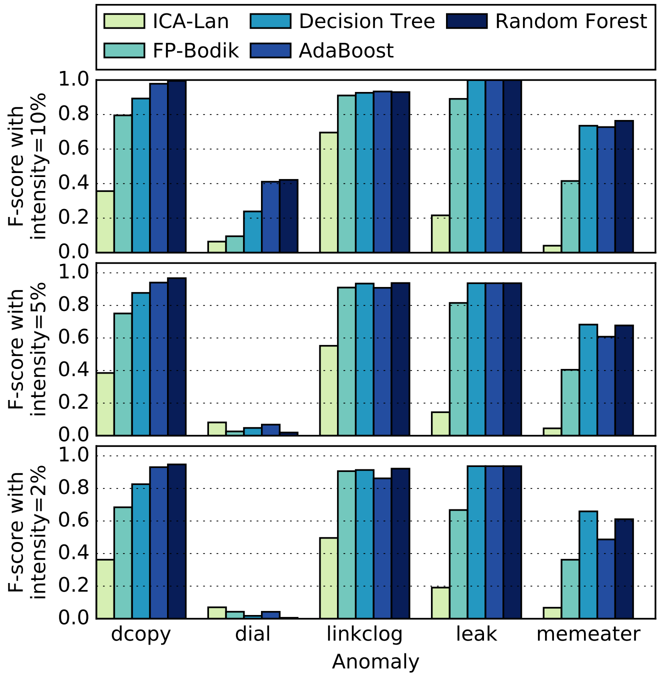
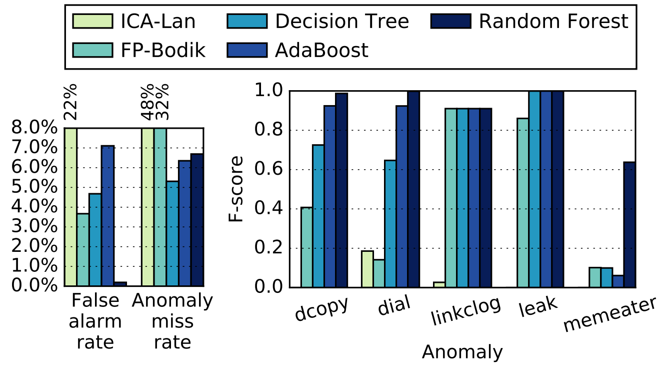


Fig. 11. Anomaly diagnosis statistics when the models are trained with anomaly intensities 20%, 50%, and 100%, and tested with intensities 2%, 5%, and 10%. Most anomaly signatures are detected when the intensity is lowered. In the *dial* anomaly, the intensity sets the utilization of the synthetic anomaly program, making it less impactful on the application performance and also harder to detect.



(a) Detection

(b) Diagnosis

Fig. 12. Anomaly detection and diagnosis statistics when the models are trained with 4-node application runs and tested with 32-node runs. The results are very similar to those with unknown input configurations as the 32-node runs use input configurations that are not used for training.

5.9 Classification with Large Applications

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

Figure 12 shows the detection and diagnosis statistics when running 32-node applications. As our framework checks individual nodes for anomalies and does not depend on how many nodes are being used in parallel by an application, the application size has minimal impact on the F-scores. The decrease in the F-scores is mainly because large application runs use input configurations that are unknown to the trained models. Hence, the F-scores in Figure 12 are similar to those in Figure 8, where the classifiers are trained with certain input configurations and tested with the remaining ones.

5.10 Detection Delay

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

Figure 13 shows the diagnosis delay observed for each anomaly and each classifier as well as the fraction of the anomalies that are not detected until the application execution finished. Random forest achieves only five to ten seconds of delay while diagnosing linkclog and memeater. This delay is due to the fact that the proposed framework requires $C = 5$ consecutive windows to be diagnosed with the same anomaly before accepting that diagnosis. The delay is larger for other anomalies, where the diagnosis is performed with statistics that slowly change as the 45-second sliding window moves (e.g., mean and variance).

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

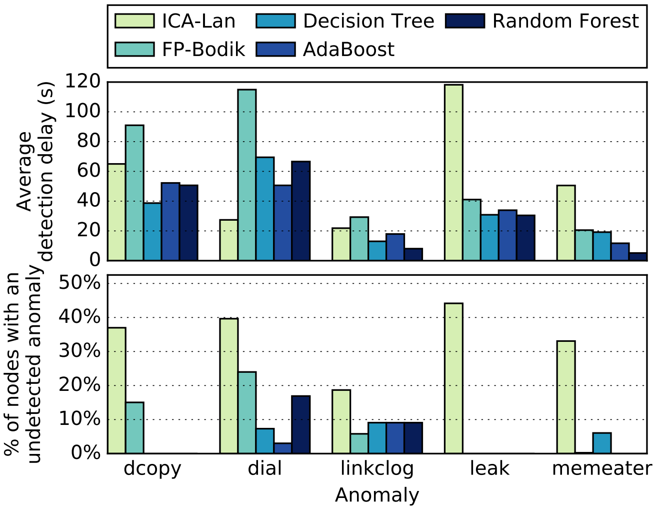


Fig. 13. Anomaly detection delay and percentage of nodes with undetected anomalies when anomalies start at a random time while the application is running.

TABLE 4
Single-threaded computational overhead of model training and anomaly detection.

	Training time for 8 billion time series data points		Testing time per sliding window per node	
	Feature generation & selection	Model training	Feature generation	Model prediction
ICA-Lan	10 days	30 s	62 ms	0.03 ms
FP-Bodik	5 days	10 mins	31 ms	148 ms
Decision tree	5 days	7 mins	4 ms	0.01 ms
AdaBoost	5 days	138 mins	13 ms	0.1 ms
Random forest	5 days	6 mins	13 ms	0.03 ms

5.11 Overhead

We assume that the node resource usage data is already being collected from the target nodes for the purposes of checking system health and diagnostics. State-of-the-art monitoring tools such as LDMS collect data with a sampling period of one second while using less than 0.1% of a core on most compute nodes [10].

Table 4 presents the average training and testing time of the baseline techniques as well as our framework with different machine learning algorithms when using a single thread on an Intel Xeon E5-2680 processor. As seen in Table 4, all approaches require the longest time for feature generation and selection. Note that feature generation and selection are embarrassingly-parallel processes and are conducted only once during training.

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

Detecting and diagnosing anomalies in a single sliding window of a single node with AdaBoost and random forest takes approximately 13ms using a single thread, which is negligible given that the window period is one second. Decision tree achieves the smallest feature generation overhead because it uses only a third of the features selected by our feature selection method (Sec. 3.2), whereas AdaBoost and random forest use nearly all selected features.

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

6 Conclusions and Future Work

Automatic and online detection and diagnosis of performance anomalies in HPC platforms have become increasingly important in today's growing HPC systems for robust and efficient operation. In this paper, we proposed an online anomaly diagnosis framework, which leverages the already-monitored performance and resource usage data to learn and detect the signatures of previously-observed anomalies. We evaluated our framework using experiments on a real-world HPC supercomputer and demonstrate that our approach effectively identifies 98% of the synthetic anomalies while leading to only 0.08% false anomaly alarms, consistently outperforming the state-of-the-art for anomaly diagnosis. We also showed that our approach learns the anomaly signatures independent of the executing applications, enabling anomaly diagnosis even when running applications that are not seen during training.

In this paper, we have focused on detecting signatures of synthetic anomalies. In future work, we will focus on designing elaborate synthetic anomalies that closely mimic the anomalies experienced by HPC system users based on user complaints and feedback. Furthermore, we will integrate our framework with the Lightweight Distributed Monitoring Service to diagnose anomalies while users are running production HPC applications.

Acknowledgments

This work has been partially funded by Sandia National Laboratories. Sandia National Laboratories is a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC., a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energy's National Nuclear Security Administration under Contract DE-NA0003525. This paper describes objective technical results and analysis. Any subjective views or opinions that might be expressed in the paper do not necessarily represent the views of the U.S. Department of Energy or the United States Government.

References

[1] V. J. Leung, C. A. Phillips, M. A. Bender, and D. P. Bunde, "Algorithmic support for commodity-based parallel computing systems," Sandia National Laboratories, Tech. Rep. SAND2003-3702, 2003.

[2] D. Skinner and W. Kramer, "Understanding the causes of performance variability in HPC workloads," in *IEEE International Symposium on Workload Characterization*, Oct 2005, pp. 137–149.

[3] J. Brandt, F. Chen, V. De Sapio, A. Gentile, J. Mayo *et al.*, "Quantifying effectiveness of failure prediction and response in HPC systems: Methodology and example," in *International Conference on Dependable Systems and Networks Workshops*, Jun 2010, pp. 2–7.

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

[5] A. Agelastos, B. Allan, J. Brandt, A. Gentile, S. Lefantzi *et al.*, "Toward rapid understanding of production HPC applications and systems," in *IEEE International Conference on Cluster Computing (CLUSTER)*, Sept 2015, pp. 464–473.

[6] J. Brandt, D. DeBonis, A. Gentile, J. Lujan, C. Martin *et al.*, "Enabling advanced operational analysis through multi-subsystem data integration on trinity," *Proc. Cray Users Group*, 2015.

[7] M. Snir, R. W. Wisniewski, J. A. Abraham, S. V. Adve, S. Bagchi *et al.*, "Addressing failures in exascale computing," *Int. J. High Perform. Comput. Appl.*, pp. 129–173, May 2014.

[8] A. Bhatele, K. Mohror, S. H. Langer, and K. E. Isaacs, "There goes the neighborhood: Performance degradation due to nearby jobs," in *SC*, Nov 2013, pp. 41:1–41:12.

[9] M. Dorier, G. Antoniu, R. Ross, D. Kimpe, and S. Ibrahim, "Calciom: Mitigating I/O interference in HPC systems through cross-application coordination," in *IPDPS*, May 2014, pp. 155–164.

[10] A. Agelastos, B. Allan, J. Brandt, P. Cassella, J. Enos *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)*, Nov 2014, pp. 154–165.

[11] J. Klinkenberg, C. Terboven, S. Lankes, and M. S. Muller, "Data mining-based analysis of hpc center operations," *CLUSTER*, pp. 766–773, 2017.

[12] L. Yu and Z. Lan, "A scalable, non-parametric method for detecting performance anomaly in large scale computing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 7, pp. 1902–1914, 2016.

[13] P. Bodik, M. Goldszmidt, A. Fox, D. B. Woodard, and H. Andersen, "Fingerprinting the datacenter: Automated classification of performance crises," in *Proceedings of the 5th European Conference on Computer Systems*, 2010, pp. 111–124.

[14] O. Tuncer, E. Ates, Y. Zhang, A. Turk, J. Brandt *et al.*, "Diagnosing performance variations in hpc applications using machine learning," in *International Supercomputing Conference (ISC-HPC)*, 2017, pp. 355–373.

[15] O. Ibidunmoye, F. Hernández-Rodriguez, and E. Elmroth, "Performance anomaly detection and bottleneck identification," *ACM Computing Surveys*, vol. 48, no. 1, pp. 1–35, 2015.

[16] R. Ahad, E. Chan, and A. Santos, "Toward autonomic cloud: Automatic anomaly detection and resolution," *Proceedings - 2015 International Conference on Cloud and Autonomic Computing, ICCAC 2015*, pp. 200–203, 2015.

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

[18] S. Jin, Z. Zhang, K. Chakrabarty, and X. Gu, "Accurate anomaly detection using correlation-based time-series analysis in a core router system," *IEEE International Test Conference*, pp. 1–10, 2016.

[19] N. Laptev, S. Amizadeh, and I. Flint, "Generic and scalable framework for automated time-series anomaly detection," *Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pp. 1939–1947, 2015.

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

[21] V. Nair, A. Raul, S. Khanduja, S. Sundararajan, S. Keerthi *et al.*, "Learning a hierarchical monitoring system for detecting and diagnosing service issues," *Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pp. 2029–2038, 2015.

[22] X. Zhang, F. Meng, P. Chen, and J. Xu, "Taskinsight : A fine-grained performace anomaly detection and problem locating system," *IEEE International Conference on Cloud Computing*, pp. 2–5, 2016.

[23] L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin *et al.*, "Hpctoolkit: Tools for performance analysis of optimized parallel programs," *Concurrency Computation Practice and Experience*, vol. 22, no. 6, pp. 685–701, 2010.

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

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

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

[27] Z. Lan, Z. Zheng, and Y. Li, "Toward automated anomaly identification in large-scale systems," *IEEE Trans. Parallel Distrib. Syst.*, vol. 21, no. 2, pp. 174–187, Feb. 2010.

[28] B. Arzani and G. Outhred, "Taking the blame game out of data centers operations with netpoirot," *ACM Conference of the Special Interest Group on Data Communication*, pp. 440–453, 2016.

[29] G. Wang, J. Yang, and R. Li, "An anomaly detection framework based on ica and bayesian classification for iaas platforms," *KSII Transactions on Internet and Information Systems (TIIS)*, vol. 10, no. 8, pp. 3865–3883, 2016.

[30] X. Chen, X. He, H. Guo, and Y. Wang, "Design and evaluation of an online anomaly detector for distributed storage systems," *Journal of Software*, vol. 6, no. 12, pp. 2379–2390, 2011.

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

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

[33] D. O'Shea, V. C. Emeakaroha, J. Pendlebury, N. Cafferkey, J. P. Morrison, and T. Lynn, "A wavelet-inspired anomaly detection framework for cloud platforms," *International Conference on Cloud Computing and Services Science*, vol. 1, April 2016.

[34] H. Nguyen, Z. Shen, Y. Tan, and X. Gu, "Fchain: Toward black-box online fault localization for cloud systems," *International Conference on Distributed Computing Systems*, pp. 21–30, 2013.

[35] S. Roy, A. C. König, I. Dvorkin, and M. Kumar, "Perfaugur: Robust diagnostics for performance anomalies in cloud services," in *Proceedings of the International Conference on Data Engineering*, 2015, pp. 1167–1178.

[36] Ganglia. [Online]. Available: ganglia.info

[37] F. J. Massey Jr, "The kolmogorov-smirnov test for goodness of fit," *Journal of the American statistical Association*, vol. 46, no. 253, pp. 68–78, 1951.

[38] Y. Benjamini and D. Yekutieli, "The control of the false discovery rate in multiple testing under dependency," *Annals of Statistics*, vol. 29, pp. 1165–1188, 2001.

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

[40] L. Breiman, *Classification and regression trees*. Routledge, 2017.

[41] L. Breiman, "Random forests," *Machine Learning*, vol. 45, no. 1, pp. 5–32, 2001.

[42] Y. Freund and R. E. Schapire, "A decision-theoretic generalization of on-line learning and an application to boosting," *Journal of Computer and System Sciences*, vol. 55, no. 1, pp. 119 – 139, 1997.

[43] D. G. Feitelson, D. Tsafrir, and D. Krakov, "Experience with using the parallel workloads archive," *Journal of Parallel and Distributed Computing*, vol. 74, no. 10, pp. 2967–2982, 2014.

[44] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter *et al.*, "The NAS parallel benchmarks - summary and preliminary results," in *Proceedings of the ACM/IEEE Conference on Supercomputing*, Aug 1991, pp. 158–165.

[45] M. A. Heroux, D. W. Doerfler, P. S. Crozier, J. M. Willenbring, H. C. Edwards *et al.*, "Improving performance via mini-applications," Sandia National Laboratories, Tech. Rep. SAND2009-5574, 2009.

[46] A. Kunen, T. Bailey, and P. Brown, "Kripke-a massively parallel transport mini-app," Lawrence Livermore National Laboratory (LLNL), Livermore, CA, Tech. Rep., 2015.

[47] J. Brandt, A. Gentile, J. Mayo, P. Pébay, D. Roe *et al.*, "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*, Jun 2009, pp. 7–14.

[48] P. Comon, "Independent component analysis, a new concept?" *Signal Processing*, no. 36, pp. 287–314, 1992.

Ozan Tuncer is a Ph.D. candidate at the Department of Electrical and Computer Engineering of Boston University. He received his B.S. degree in Electrical and Electronics Engineering from the Middle East Technical University, Turkey. His research interests include data center power and thermal management, workload management for high performance computing, and data analytics for cloud system management.

Emre Ates is a Ph.D. candidate in the Department of Electrical and Computer Engineering of Boston University. He received his B.Sc. degree in Electrical and Electronics Engineering with honors from the Middle East Technical University, Ankara, Turkey. His current research interests include management and monitoring of large-scale computing systems.

Yijia Zhang is a PhD candidate in the Department of Electrical and Computer Engineering of Boston University. He received his B.S. degree in Department of Physics from the Peking University, Beijing, China. His current research interests include performance optimization of high performance computing.

Ata Turk is a Research Scientist in the Electrical Computer Engineering Department of Boston University (BU). He leads the healthcare and bigdata analytics groups in the Massachusetts Open Cloud project. His research interests include bigdata analytics and combinatorial optimization for performance, energy, and cost improvements in cloud computing applications. Prior to joining BU, Dr. Turk was a postdoctoral researcher at Yahoo Labs.

Jim Brandt is a Distinguished Member of Technical Staff at Sandia National Laboratories in Albuquerque, New Mexico, where he leads research in HPC monitoring and analysis.

Vitus Leung is a Principal Member of Technical Staff at Sandia National Laboratories in Albuquerque, New Mexico, where he leads research in distributed memory resource management. He has won R&D 100, US Patent, and Federal-Laboratory-Consortium Excellence-in-Technology-Transfer Awards for work in this area. He is a Senior Member of the ACM and has been a Member of Technical Staff at Bell Laboratories in Holmdel, New Jersey and a Regents Dissertation Fellow at the University of California.

Manuel Egele is an assistant professor in the Electrical and Computer Engineering Department of Boston University (BU). He is the head of the Boston University Security Lab where his research focuses on practical security of commodity and mobile systems. Dr. Egele is a member of the IEEE and the ACM.

Ayse K. Coskun is an associate professor at the Department of Electrical and Computer Engineering, Boston University (BU). She was with Sun Microsystems (now Oracle), San Diego, prior to her current position at BU. Her research interests include energy-efficient computing, 3-D stack architectures, computer architecture, and embedded systems and software. She received the M.S. and Ph.D. degrees in Computer Science and Engineering from the University of California, San Diego.