Live Forensics for HPC Systems: A Case Study on Distributed Storage Systems

Saurabh Jha*, Shengkun Cui*, Subho S. Banerjee*, Tianyin Xu*,
Jeremy Enos*†, Mike Showerman*†, Zbigniew T. Kalbarczyk*, and Ravishankar K. Iyer*†
*University of Illinois at Urbana-Champaign, Urbana-Champaign, IL 61801, USA.
†National Center for Supercomputing Applications (NCSA), Urbana, IL 61801, USA.

Abstract—**Large-scale high-performance computing systems frequently experience a wide range of failure modes, such as reliability failures (e.g., hang or crash), and resource overload-related failures (e.g., congestion collapse), impacting systems and applications. Despite the adverse effects of these failures, current systems do not provide methodologies for proactively detecting, localizing, and diagnosing failures. We present Kaleidoscope, a near real-time failure detection and diagnosis framework, consisting of of hierarchical domain-guided machine learning models that identify the failing components, the corresponding failure mode, and point to the most likely cause indicative of the failure in near real-time (within one minute of failure occurrence). Kaleidoscope has been deployed on Blue Waters supercomputer and evaluated with more than two years of production telemetry data. Our evaluation shows that Kaleidoscope successfully localized 99.3% and pinpointed the root causes of 95.8% of 843 real-world production issues, with less than 0.01% runtime overhead.**

I. Introduction

Large-scale high-performance storage systems frequently experience a wide range of *failure modes* [1]–[4], including reliability failures (e.g., hang or crash) and resource overload-related failures (e.g., congestion collapse [5]). The net effects of these failures on systems and applications are often indistinguishable in terms of impact, and their mitigation strategies can vary significantly (e.g., throttling for congestion, or restart for a hung process). The inability to mitigate failures early enough can impact a single component (e.g., a data server), enable propagation of the failure across multiple interconnected components, or even cause a whole system outage, thereby adversely impacting application performance and resilience [6]–[12]. Thus, there is an need for not only detecting the failure, but also identification of the failure mode in real-time. As we show using a real-world failure scenario from the Blue Waters supercomputer's storage system (refer §II-A), a reliability failure can be construed as a performance problem and vice versa.

To address above problems, we propose Kaleidoscope, a system that uses machine learning (ML) to detect a failure, identify the failure mode, and diagnose the failure cause by using existing monitoring data in near real-time. Moreover, we have demonstrated Kaleidoscope and its scalability on Blue Waters, which is the largest university-based high-performance computing (HPC) system in the world, in terms of both compute and storage nodes. We focus on high-performance storage systems because they have the most failures and lost compute hours[1]. For example, in 2018, NCSA reported that storage-related failures have accounted for 64.4% (i.e., >32 million core hours) of total lost core hours on a yearly basis. Further, the problem is expected to be worse in emerging and future exascale systems, with even lower mean time between failures and higher-impact service outages, because of increasing system scale, heterogeneity, and complexity [13], [14].

Why Machine Learning? Kaleidoscope uses multi-modal telemetry data from numerous monitors that provide system-wide temporal and spatial information on performance and reliability. The monitors either actively poll the system components [15], [16] (e.g., with pings/heartbeats), or passively aggregate performance and reliability measurements [15], [17], [17]–[19] (e.g., based on server load). The problem with telemetry data is that they are often noisy due to asynchronous collection [18], [19], failure propagation [4], [10], [20], and non-determinism in the system (e.g., in adaptive routing and load balancing) [21]–[23]. Therefore, when analyzed in isolation, telemetry data of a single modality may lead to misdiagnoses, i.e., false positives (e.g., in the case of failure propagation) and false negatives (e.g., in the case of partial failures). Moreover, the vast amounts of available telemetry data (on the order of TBs per day [24]) lead to cognitive overload of system managers [25]. They cannot keep up with the incoming data for identifying and debugging failure issues, significantly delaying the identification and mitigation of the failure.[2] To address those problems, Kaleidoscope uses ML methods that use domain-guided methods to accurately estimate the system state in the presence of noisy data, thereby detecting failures and identifying the failure mode and failure cause.

While existing approaches are useful [26]–[38], they have significant drawbacks because they do not (i) jointly address reliability failures and resource-overload-related failures; (ii) focus on detecting and identifying failures and their failure mode in storage systems (except [30], which focuses on distinguishing network vs storage failures, and [15], [17], which mostly focuses on offline diagnosis); and (iii) deal with the difficulty of collecting/labeling training data, especially for rare failure scenarios in production settings [32], [33].

Our Approach. Kaleidoscope is a near real-time failure detection and diagnosis framework. It consists of hierarchical domain-guided interpretable ML models: (i) a *failure localization model* for identifying component failures (e.g., failures of

[1] In this paper, we identify the failures at the granularity of storage clients, network path to storage, storage servers, and RAID devices.

[2] For example, as we will show in §VII-C, a partial failure of an I/O load balancer on Blue Waters, which was impacting application performance by as much as 25%, remained undetected for several weeks.

compute nodes, load balancers, the network, storage servers, and RAID devices), and (ii) a *failure diagnosis model* for identifying the failure mode of a system component as either a resource-overload-related failure or a reliability failure.

The **failure localization model** uses ML and *I/O path-tracing data* to estimate the *failure state* of the storage components. I/O path-tracing data provide information on the route taken by the request (from the storage client on the compute node to the disk on the storage server) and the availability of the components on the route. The model incorporates the insight that the success of multiple I/O probes (e.g., a write I/O request) indicates that the components on the request path are healthy with a high probability. Each measurement in the I/O path-tracing data provides information on only a subset of storage components. Hence, the model jointly analyzes the I/O path-tracing data from multiple probes, and infers the probability of component failures.

To address the problem of noisy and multi-modal telemetry data and their joint analysis, our ML model uses the probabilistic graphical model (PGM) formalism to express the statistical dependence between the system components and the path-tracing data. Here, the failure state of each component is modeled as a *hidden* random variable; the path availability (i.e., the probability that an I/O request will complete successfully) is modeled as *observed* random variables; and the statistical dependence among random variables is derived using the design and implementation details of path-tracing monitors, the storage system, and the system topology. The proposed ML model is based on the insight that (i) even though individual path-tracing measurements might be noisy, (ii) groups of different measurements that are related to one another can be jointly considered to reduce the noise and estimate the failure state of the components, and (iii) the underlying statistical relationships between the storage components and the telemetry data can be used to correct for noise. We derive those statistical relationships by using the system topology and the paths taken by the I/O requests.

Although PGMs require less data for training and inference (compared to current approaches [32], [33]), dynamic collection of path-tracing data can be expensive due to intrusive instrumentation and data collection, which can interfere with application performance. To address that problem, Kaleidoscope uses *Store Pings*, a set of low-cost and low-latency probing monitors that not only probe a disk from a client by using an I/O request and record the response time (similar to ioping [39]), but also, unlike ioping, provide a mechanism for pinning (i.e., enforcing the use) of specific components on the I/O request path (e.g., a disk, or data servers).

It is hard to distinguish between different failure modes because of limited observability, measurement noise, and failure propagation effects (described in §II-B). Notwithstanding, we have demonstrated that the proposed **failure diagnosis model**, which is a domain-informed statistical model, is able to accurately identify the failure mode and the likely causes (as discussed in §V-B) by using (i) components' telemetry data, which include performance metrics and RAS logs, and (ii) the failure state estimated using the failure localization model. The failure diagnosis model uses the Local Outlier Factor [40], an unsupervised anomaly detection method, which answers the question, "Which modality of the telemetry data (among RAS logs and performance metrics) best explains why one component is flagged as failed, while others are marked as healthy by the failure localization model?" The proposed model indicates the failure mode of the failed component as *either* a reliability failure (i.e., an error logs), or a resource-overload-related failure (i.e., a performance metric).

Results. We have implemented and deployed Kaleidoscope on the Cray Sonexion [41] high-performance distributed storage system of Blue Waters, a petascale supercomputer at the National Center for Supercomputing Applications at the University of Illinois at Urbana-Champaign. Cray Sonexion uses the Lustre file system [42], which is used by more than 70 of the top 100 supercomputers [43] and is offered by cloud service vendors. The key results are as follows.

1) *High accuracy*: We used 843 production issues that were identified and resolved by the Blue Waters operators as the ground truth. Kaleidoscope correctly localized the component failures across all failure modes and resource overloads for 99.3% of the cases and accurately diagnosed the failure cause for 95.8% of the cases by pointing to the most likely failure cause and it distinguished between reliability failures and resource overloads/contention within 5–10 minutes of the failure incident. Moreover, Kaleidoscope found additional failures that were not present in the ground truth data, i.e., had not previously been identified.
2) *Low rate of false positives*: With respect to false positives, Kaleidoscope outperforms by 100× the state of the art regression-based failure localization model, NetBouncer [26] customized for cloud networks, which focuses *only* on identifying partial and fail-stop failures and not on resource-overload-related failures and diagnosis.
3) *Low overhead*: The overhead introduced by Kaleidoscope is less than 0.01% of the system's peak I/O throughput.
4) *Long-term characterization*: Kaleidoscope was used to improve our understanding of storage-related failures by characterizing two years of production data.

II. Background and Motivation

A. Blue Waters Storage Design

We describe the Cray Sonexion storage subsystem of Blue Waters and introduce our terminologies. Cray Sonexion is designed for large-scale HPC systems with I/O-intensive workloads, such as machine learning and large simulations. It's deployment on Blue Waters consists of 6 management servers, 6 metadata servers (MS), 420 data servers (DS), and 582 I/O load-balancers (LNET nodes). The storage servers in Cray Sonexion are connected via an internal Infiniband network (storage network). LNET nodes connect 28,000+ computing nodes (i.e., clients) on Cray Gemini interconnection network (compute network) to storage network. Cray Sonexion uses the Lustre parallel distributed file system to manage 36 PB of disk space across 17,280 HDD disk devices. The disks are arranged in a grid RAID [44] and are referred to as *object storage devices* (OSDs). Each storage server is attached to one or more OSDs. Lustre offers high-availability and failover features. In Lustre, data servers are arranged as active-active pair to

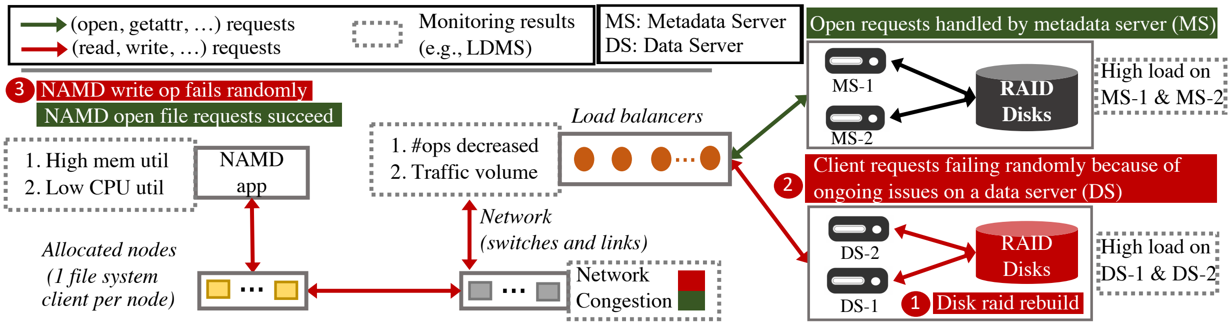

(open, getattr, …) requests
(read, write, …) requests
Monitoring results (e.g., LDMS)
MS: Metadata Server
DS: Data Server
Open requests handled by metadata server (MS)
MS-1
MS-2
RAID Disks
High load on MS-1 & MS-2
3 NAMD write op fails randomly
NAMD open file requests succeed
Load balancers
1. High mem util
2. Low CPU util
NAMD app
1. #ops decreased
2. Traffic volume
2 Client requests failing randomly because of ongoing issues on a data server (DS)
Network (switches and links)
Allocated nodes (1 file system client per node)
Network Congestion
DS-2
DS-1
RAID Disks
High load on DS-1 & DS-2
1 Disk raid rebuild

Figure 1: Propagation of I/O failure and challenges in identifying, localizing, and disambiguating the causes of I/O failures.

achieve load balancing and high availability for connected OSDs, whereas metadata servers are arranged as active-passive pair for connected OSDs. The computing nodes are diskless: all I/O operations go by RPC to the LNET nodes, and the LNET nodes forward the request to the storage servers.

B. Motivating Failure Scenario

We describe a real-world failure scenario (see Fig. 1) which frequently occurs in the distributed storage system the of Blue Waters supercomputer to illustrate the difficulty of identifying the root cause of an application failure/slowdown using telemetry data. The telemetry data obtained during this failure scenario capture the following partial views:

1) *Storage view.* In this failure scenario, the telemetry data indicated high load and increasing service time on a pair of data servers. These data servers eventually hang and lead to unavailability of the files stored in these data servers. At the same time, other data servers (not shown in the figure) do not show symptoms of high load.
2) *Application view.* In this failure scenario, the NAMD [45] application issues *open* and *write* I/O requests. They are handled via FS clients (kernel modules) on each compute node. To write to the file, the FS client first *opens* the file and gets the file handler by accessing the *metadata server*, and then uses this file handler to directly *write* to the file on disk via the corresponding *data servers*. However, in this case, the *write* request fails because of the FS client request timeout, despite the successful completion of the *open* request. The request failure causes the applications to fail. From the point of view of the application, the FS clients were partially failing.

Both views hint at a problem in the system, they are not sufficient for detecting and diagnosing the failure. The real cause of the failure was deeply hidden in the server logs. The analysis of the server logs revealed that a disk failure in the storage device (OSD in Lustre) was the real cause of the storage server and application failure. The failure of the disk triggers a RAID disk rebuild, which in turn decreases the effective I/O bandwidth available to two data servers (DS-1 and DS-2). The decrease in bandwidth causes an increase in the service time of I/O requests, which, in turn increases the load on data servers DS-1 and DS-2, which, in turn leads to server hang and unavailability of the files, ultimately causing application to fail. Intuitively, it can be seen from the failure scenario example that the failure mitigation will depend on both the failure location and mode. Overall, we find that the telemetry data, when analyzed in isolation and as illustrated in Fig. 1, provide outcomes and results that in general seem conflicting, even to experts. For example, the telemetry data on the application hint at high memory utilization, whereas telemetry data on the data server can hint at high load.

C. Challenges

The failure scenario above highlights multiple challenges:

Dataset heterogeneity & Fusion. Large-scale HPC systems produce vast amounts of telemetry data (at application, network, and storage layers) by using multiple monitors across the system stack. These datasets are highly heterogeneous in nature (e.g., sampling frequency of monitors), and provides only partial observability into the system (i.e., storage and application levels). Thus, highlighting the need to jointly analyze datasets to avoid conflicting outcomes.

Data labelling and rare failures. There are challenges in both labeling the failure data, and acquiring them. This problem exacerbates due to a long tail of one-off, unique failures that are previously unknown and hard to anticipate based on historical data (discussed in §VII-C).

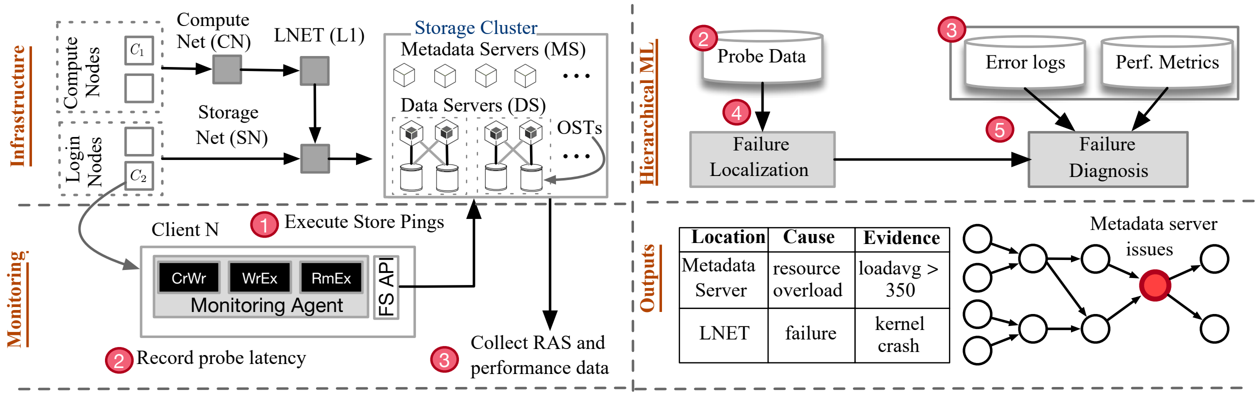
Measurement uncertainty, noise, & propagation effects, emanated from (i) timing issues in asynchronous measurement and data collection intervals, (ii) non-determinism due to path redundancy and randomness in routing, and (iii) failure propagation leading to variability and noise in measurements.

Timeliness of analytics. Minimal number of monitors must be placed strategically across the system (i) to provide spatial and temporal observability, and (ii) to reduce data and time required to perform analytics.

Those challenges make it difficult (i) to identify the failing component, and (ii) to discern the failure modes. That leaves system operators with no option but to comb through multiple monitoring dashboards to form their conclusions about failures based on their experience, and that significantly increases the response time for mitigating the impact of failure (upto 4–8 hours), leading to unexpected outages and impact. This is untenable for future exascale systems that would require realtime failure detection, diagnosis and mitigation.

III. Kaleidoscope Overview

Fig. 2 shows the design of Kaleidoscope. The "Infrastructure" part (upper left) shows a simplified diagram of Blue Waters storage system (described in §VII). The "Monitoring" part (lower left) shows the telemetry data collected from the system across the stack (described in §IV). The "Hierarchical ML" part (upper right, described in §V) shows the interconnected



Location	Cause	Evidence
Metadata Server	resource overload	loadavg > 350
LNET	failure	kernel crash

Figure 2: An overview of Kaleidoscope design and implementation.

ML models that provide failure localization (i.e., identifying the failed component), and diagnosis capabilities (i.e., identifying the failure mode and pointing to the anomalous telemetry data indicative of the failure). The "Outputs" part (lower right) provides an interpretable set of results and dashboards that can be used by the system managers (described in §VI).

Kaleidoscope addresses the challenges of identifying failing components and discerning the failure modes described in §II-C via the following approaches:

1) *Fusing heterogeneous telemetry data for increased observability.* Kaleidoscope uses telemetry data from across the system, capturing both the system and application views, to increase spatial and temporal observability. The fusion and comprehensive analysis of the data enable accurate detection of both resource overload and reliability failures.
2) *Hierarchical probabilistic ML models for dealing with data uncertainty and noises.* Kaleidoscope uses hierarchical probabilistic ML models that use domain knowledge to model measurement noises and failure propagation effects. The hierarchical ML models enable data analysis at different granularities and time scales.
3) *Unsupervised ML models for dealing with insufficient samples and rare failures.* Kaleidoscope uses unsupervised ML models and leverages domain knowledge on the system design and architecture to alleviate the challenges of (i) labeling the failures, and (ii) acquiring training data on rare failures, especially on rare one-off failures.
4) *Low-cost automation for timely analytics.* The use of unsupervised methods alleviates the need for costly training and re-training of models. Store Pings (refer to §IV-A) are low-cost monitor to provide observability into storage.

IV. Monitors & Telemetry Data

A. End-to-end Probing Monitors

Kaleidoscope uses end-to-end I/O probing monitors (see ❶ & ❷ in Fig. 2), to collect path-tracing telemetry data. that provide observability into the health of each component on the path. For example, a successful probe from A to B through C and D reveals that all the components (A, B, C and D) are healthy; if the probe fails, it means that at least one component on the path is experiencing a failure. *A probe is marked as successful when it completes within a pre-specified time limit (i.e., meets its service-level objective); otherwise it is marked as failed.* Although distributed path-tracing tools exist (e.g., Zipkin [46] and Uber Jaeger [47] for microservices, and Darshan [48] for HPC I/O, dynamic collection of tracing data can be hugely costly. Moreover, the available tools only provide application views and fail to provide observability into the storage infrastructure view, which is critical, as we show in §II-B. Hence, we created Store Pings, which are low-cost probing monitors that not only probe a disk from a client by means of I/O requests (similar to ioping [39]) and record the response time, but also provide a mechanism for pinning the path of the I/O requests to a disk through specific load balancers and servers. The pinning of the path eliminates the need for tracing of the request, and thereby reduces the overhead of data collection on path availability. While Store Pings are analogous to the ICMP-based network ping (which provides visibility only into the network), the two are significantly different. Specifically, Store Pings are designed for storage systems and provide visibility across the entire system stack, which includes compute, network/interconnect, and storage subsystems. Since, Store Pings generate an I/O probing request of fixed size, an **I/O failure** occurs when the I/O completion time is higher than or equal to one second. We use one second as SLO because 99% of the Store Ping probes on Blue Waters completes within one second.

Path Pinning. Store Pings provide path-pinning capabilities by leveraging Lustre's file system support for pinning of a file on a specific object storage device (and hence the data server),[3] thereby eliminating the need to modify Lustre to support path pinning. Since the metadata server has all the data chunk information, an I/O request to the file uniquely identifies both the OSD and the data server. It also prunes the number of possible paths that can be taken by the I/O request (from the client to the OSD). For example, a Store Ping executing on a compute node (which is a storage client) and accessing data on an OSD can use only 4 load balancers (LNETs) instead of all the LNETs (of which there are more than 500) in the system. Although pinning of all the components (e.g., pinning of I/O requests to a particular LNET) on the path is desirable, it is unnecessary and would require changes in the proprietary software and hardware in the compute and storage system to support deterministic routing. We leverage probabilistic models to handle non deterministic paths (§V).

[3] Store Pings executes independently of other applications. It creates and operates on its own set of files to achieve the monitoring goals.

Increased Observability. The API of a Store Ping is: store_ping(ost, *io_op, kwargs), where *io_op is a function pointer to an I/O operation, and kwargs is the argument of *io_op. Store Pings use direct I/O requests to avoid any caching effect, which ensures that each I/O request traverses all the way from the clients to the disks on the data servers. We designed three types of Store Pings, CrWr, WrEx, and RmEx, which correspond to three different I/O requests: (i) CrWr, which creates and writes a new file; (ii) WrEx, which writes to an existing file; and (iii) RmEx, which removes an existing file. CrWr and RmEx test the functionality of the metadata servers, whereas WrEx tests the functionality of the data servers (and, correspondingly, RAID disks). For example, a CrWr requires two different back-end operations to complete: (i) creation of a file by a metadata server on a random data server (and the corresponding RAID disks) and addition of the file entry to the metadata index, and (ii) opening and writing of a file on the data server (and the corresponding RAID disks). The payload of a write request is only 64 bytes. Together, the three types of Store Pings test all the storage subsystems (which include storage clients located on compute nodes, network interconnections, storage servers, and RAID devices) that are involved in ensuring successful I/O operations.

Placement. Store Pings are strategically placed in the system to provide both spatial and temporal differential observability in near real-time. Store Pings generate probing requests continuously at regular intervals to measure the availability and performance of storage components. Note that Store Pings should be enabled only on a subset of clients to reduce the overhead of the Store Pings and their impact on existing I/O requests, while providing complete spatial observability.

Selecting the number of Store Pings and their placement can be formulated as a constraint optimization problem. The subsets of components that can be tested together are limited by the set of I/O paths, which are in turn limited by the topology, probing mechanism, and I/O request routing protocols.

We use *Boolean network tomography principle* to solve the constraint optimization problem of selecting the number of Store Ping monitors and their placement [49]. Specifically, the placement of monitors[4] in Kaleidoscope is guided by the *sufficient identifiability condition* (discussed in [49], [50]), which states that in a topology graph G of a system (in this case the Lustre storage system) consisting of both monitor and non-monitor nodes, any set of up to k failed components is identifiable if for any non-monitor $v \in G$ and failure set F with $|F| \leq k$ such that $v \notin F$, there is a measurement path going through v but no node in F. In other words, there must exist a set of I/O paths that can be used by Store Pings to uniquely identify the failure-state of each component and detect up to k concurrent failures. This is also referred as *spatial differential observability* and allows us to handle redundancies as long as the condition is met. [49] provides set of rules and algorithms to meet sufficient identifiability condition to identify number of monitoring nodes and their placement for any arbitrary storage system. We omit the detailed discussion because of lack of space. Blue Waters's system managers not only want to identify failures of storage components but also failures of service nodes and login nodes. We place Store Ping monitors on storage clients that (i) have different system stacks (e.g., kernel versions), (ii) are physically located on different networks, and (iii) execute different services (e.g., scheduling, user login, and data moving). Specifically, we place monitors on all the service nodes that provide scheduling and other capabilities (64 nodes); import/export (I/E) nodes (25 nodes) that move bulk data into and out of the storage system; and login nodes (4 nodes), which launch applications. The I/E nodes and login nodes are on the storage network, whereas the service nodes are on the proprietary compute network fabric. This placement scheme meets both the production requirements (given by system managers) and theoretical requirements (from network tomography principle).

Probing Plan. At any given time, Store Pings are executed from (i) all login nodes, (ii) 1 out of 64 service nodes chosen randomly, and (iii) 1 out of 25 I/E nodes chosen randomly. That probing plan satisfies our minimal probing plan for inferring storage system health, while providing reliability for the monitoring infrastructure; if a client failure occurs, another client can be chosen as a monitor. Store Pings are executed every minute for each OSD, data server, and metadata server. That results in 72 CrWr and 72 RmEx (from 6 clients to 6 metadata servers and 6 OSDs) and 5,184 WrEx (from 6 clients to 432 data servers and 432 OSDs) requests per minute.

B. *Component Logs*

Kaleidoscope uses a comprehensive monitoring system (similar to the monitoring system described in [31], [51]) to collect performance measurements and RAS (reliability, availability, and serviceability) logs for each system component (including compute nodes, load balancers, network switches, and storage servers) in real-time (see ③ in Fig. 2). We use the Light-weight Distributed Metric Service (LDMS) [18], a data-aggregation tool, to collect performance measurements (e.g., loadavg, memory utilization, disk latency) for compute nodes, load-balancers (LNETs) and switches. We use ISC (the Integrated System Console) [52] to collect performance measurements on storage components (e.g., disks, and servers), LDMS data, and RAS logs on a centralized server.

V. Hierarchical Machine Learning Models

Kaleidoscope uses hierarchical domain-guided unsupervised ML models to provide live forensics capabilities. These hierarchical ML-models include: (i) failure localization model (for identifying the failed nodes), and (ii) failure diagnosis model (for identifying the failure mode of the failed node).

A. *Failure Localization Model*

Kaleidoscope uses a *failure localization* model (see ④ in Fig. 2) for identifying component(s) that are failed or overloaded, and thus are leading to I/O failures. Kaleidoscope uses telemetry data obtained from Store Ping monitors for that purpose. However, Store Ping measurements are noisy (due to

[4] In the network tomography formalism, both the ends of the probing path is referred as monitors. However, in this work, we designate a storage component as a monitor only if it executes Store Ping. A Store Ping path starts at a storage client and ends at an object storage device (OSD). OSDs that do not execute Store Ping are not referred as monitors.

asynchronous data collection, adaptive routing/load-balancing, and failure propagation among others) and provide partial view (i.e., the measurements only provide information on a subset of the system components). These challenges are hard to deal with traditional threshold or voting-based methods which often lead to over-counting and misdiagnosis [26]. Therefore, we model these noise/uncertainties in the telemetry data as well as provide a formalism to fuse these partial views.

We use probabilistic graphical model (PGM) formalism, in particular the factor graph (FG) model [53], to jointly analyze and fuse the telemetry dataset from all the Store Pings monitors placed on the system, while accounting for the noise and related uncertainties. PGMs specify the relationships between the random variables using a graphical structure, where a node represents a random variable, and an edge represents the statistical relationship between random variables. This graphical structure allows PGMs to capture complex conditional independence between the random variables (i.e., domain knowledge), specified in a human interpretable manner. Using such domain knowledge in turn reduces both the data requirements (compared to supervised machine learning methods [26], [30], [33]) as well as inference time. The proposed PGM model is based on the insight that even though individual Store Ping measurements might be noisy, groups of different Store Ping measurements that are related to one another can be jointly considered to reduce the measurement errors, all while estimating the failure state of the components. Kaleidoscope uses the most general form of PGMs called factor graphs (FGs), which is a generalized formalism for specifying and computing inference on PGMs. In our FG model, the *failure state* of each component (which is *hidden*) on a path and its corresponding path availability (which is *observed* using Store Ping telemetry data) are specified as random variables, and the functional as well as statistical relationship between hidden and observed variables as (in terms of path) factor functions. An inference on the FG model allows Kaleidoscope to estimate failure state of each component, and explain the observed telemetry data. This determination of the failure state localizes failed components in the system.

Formalism. We define the *health*, and hence the failure state, of a component as as random variable, $X_i^{(t)}$, whose value captures the probability of a component i successfully serving an I/O request at time t. We use the shorthand X_i for $X_i^{(t)}$, as the variable changes at every time step.[5] In the absence of measurements, X_i is derived from a prior beta distribution[6], i.e., $X_i \sim Beta(\alpha, \beta)$, where α and β determine the shape of the distribution. At any time step, α and β are updated based on the inference at the previous time step (described later in the 'inference' paragraph).

Store Ping-based monitoring provides *reachability* measurements between a client C_i and an OSD OSD_j. We use a random variable $Y_{\langle C_i, OSD_j \rangle}$ to denote the number of successful Store Pings between C_i and OSD_j in the interval $(t-1, t]$. We model $Y_{\langle C_i, OSD_j \rangle}$'s prior using a binomial distribution, $Y_{\langle C_i, OSD_j \rangle} \sim Binomial(A_{\langle C_i, OSD_j \rangle}, N)$, where $A_{\langle C_i, OSD_j \rangle}$ denotes the reachability from the C_i to OSD_j, and N denotes the total number of Store Pings issued from the C_i to OSD_j. We use binomial distribution because it allows us to compute the probability of observing a specified number of "successes" (in this case, number of successful Store pings between C_i and OSD_j), which we observe through our telemetry data.

We use the domain knowledge of underlying statistical relationships between the telemetry data and the components' health to calculate $A_{\langle C_i, OSD_j \rangle}$. These statistical relationships are based on the understanding of system topology and I/O request path. For example, lets consider the case when the exact routing information of an I/O request is available using a path tracing tool. Using the route information, we could have determined $A_{\langle C_i, OSD_j \rangle}$ solely by the product of individual component's health (i.e., all components on the path must work for the request to be successful): $A_{\langle C_i, OSD_j \rangle} = \prod_{i \in \mathbb{P}(\langle C_i, OSD_j \rangle)} X_i$. Where $\mathbb{P}(\langle C_i, OSD_j \rangle)$ denotes the path between C_i and OSD_j.

Recall from §IV-A, that collecting path-tracing data is expensive. Hence, we must model the redundancies (e.g., high availability pairs and failover) and non-determinism to calculate $A_{\langle C_i, OSD_j \rangle}$. In our system, a Store Ping destined for an OSD may take a different path among several possible paths depending on the load and routing policies. For the sake of clarity, we illustrate the procedure to model redundancies by modeling the path of I/O request through a high-availability data server. We use the same methodology to model other redundancies (e.g., load-balancers and network paths). An I/O request to an OSD can be routed through one of the two data servers connected to it. Hence, a destination OSD is not reachable if both data servers (DS-1 and DS-2) connected to it are unavailable or the OSD itself is not available. R_{OSD_i}, the probability of an I/O request completing successfully from a load balancer to an OSD_i, is given by:

$$R_{OSD_i} = (1 - (1 - X_{DS_1}) \cdot (1 - X_{DS_2})) \cdot X_{OSD_i}$$

where X_{DS_1} and X_{DS_2} denote the health of data servers in the HA pair associated with the OSD (denoted by OSD_i). In the equation, $1 - X_{DS_1}$ and $1 - X_{DS_2}$ determine the probability distributions of the DS_1 and DS_2 to be *failed* respectively, and their product determines the probability distribution that both will be in a failed state. That probability distribution, when multiplied by the probability distribution of the OSD's health, gives the reachability of the OSD from one of the data servers. As shown in Fig. 3 (bottom half), the $A_{\langle C_i, OSD_j \rangle}$ between client C_i and OSD_j is given by:

$$A_{\langle C_i, OSD_j \rangle} = X_{C_i} \cdot X_{L_a} \cdot X_{CN_b} \cdot X_{SN_c} \cdot X_{MS_d} \cdot R_{OSD_j},$$

where C_i, L_*, CN_*, SN_*, MS_*, and OSD_j stand for *client*, *LNET*, *compute network*, *storage network*, *metadata Server*, and *object storage device* respectively, as shown in Fig. 3. Here, the path availability $A_{\langle C_i, OSD_j \rangle}$ only models the non-determinism associated with load balancing on the data server.

[5] All the variables defined below are time variant, however we use the same shorthand to simplify the description.

[6] Beta distributions are: (1) continuous distribution which models the success of an event (here an I/O request) and (2) commonly used as a conjugate prior for Bernoulli and Binomial random variables (which we use in our model). Moreover, use of conjugate priors drastically reduces the computation time for inference [54].

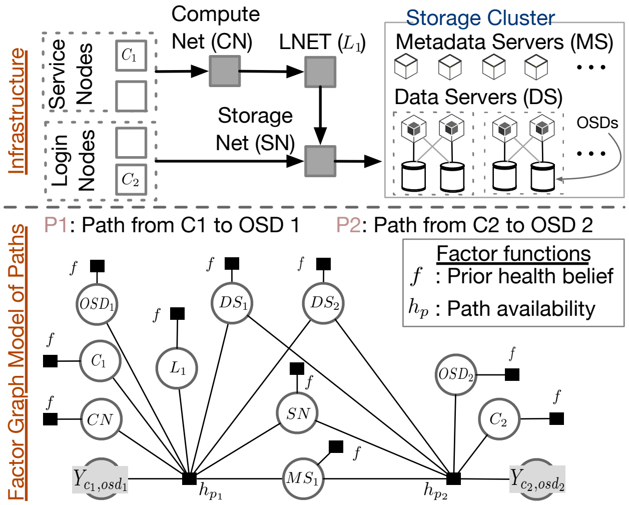


Figure 3: The FG model for failure localization. Non-shaded circles represent hidden random variables, and shaded circles represent observed random variables (measurements).[7]

We follow a similar approach to derive $A_{\langle C_i, OSD_j \rangle}$ for our system. Moreover, Kaleidoscope models the temporal evolution by using estimated component health parameters from previous inferences and uses the uncertainty to quantify the confidence in the inference results.

The model described above can be represented using a factor graph (FG) that models the relationship between different random variables (shown as circles in Fig. 3) and *functional relationships* known as factor functions (shown as dark boxes). The relationships between random variables are extracted from the system topology diagram, which can be derived from the reference manuals or mined using tracing tools.

Fig. 3 shows a part of the FG that models (i) the health of components that lie on the path of $\langle C_1, OSD_1 \rangle$ and $\langle C_2, OSD_2 \rangle$, and the path availability for these components. The components OSD_1, OSD_2, DS_1, and DS_2 form a high-availability (HA) pair (i.e., a I/O request to a particular OSD in the pair can be served by either of the data servers). The circles in the FG represent random variables (e.g., a component's health). The factor functions, represented by squares, encapsulate the relationships among the random variables. The singleton factor functions f_i encapsulate the prior belief on the health of the component (which is known from a previous time step or from training time), which is given by the beta distribution (described above). The multivariate factor function $h_{\langle C_i, OSD_j \rangle}$ models the number of successful Store Pings on a path, given by the binomial distribution (described above).

Inference. With the factor graph model, we can calculate the health of each component X_i in the system. The expected health of a component i can be estimated as $E[X_1, X_2, X_3, ...|Y_{p_1}, Y_{p_2}, Y_{p_3}, ...]$. Observations $(Y_{p_1}, Y_{p_2}, Y_{p_3}, ...)$ and the prior belief on the health of components (α and β for each X_i) are needed at time step T_j. Y_{p_i} is measured as the number of observed successful Store Pings during a specified interval, and α and β are obtained from the inference result at the previous time step, and at time zero initialized to 0.5 (i.e., there is no prior information of the components being either healthy or failed.). Intuitively, the inference procedure biases the prior belief of the model on the failure states of the components using the telemetry data (i.e., Store Ping probing data) obtained in the current time step. Kaleidoscope solves the inference task by using the Monte Carlo Markov Chain algorithm [55], a technique for estimating the expectation of a statistic from a complex distribution (in this case, $E[X_1, X_2, X_3, ...|Y_{P_1}, Y_{P_2}, Y_{P_3}, ...]$) by generating a large number of samples from the model and directly estimating the statistic. We also quantify the confidence in the inference results and use it to reduce the false positives. Our model declares a component to be failed *only* when the confidence in the inference is more than 75%. Failure localization model is implemented using PyMC3, a Python-based probabilistic programming language [56]. It uses samples collected over five minutes, i.e., the results of 26,640 I/O requests, for inference.

Training. Note that training is not explicitly required for the proposed model. However, it can help bootstrap the model before deployment. One key advantage of using probabilistic models like FGs is that training of such models can be reduced to inference on the model parameters (i.e., estimating the parameters of the used probabilistic distributions). In the case of a parametric FG that parameterizes certain statistical relationships (as in our model), we set up the training problem just like the inference problem to pick the set of parameters that can explain a data trace generated by the system.

B. Failure Diagnosis Model

The *failure diagnosis model* (see 5 in Fig. 2) leverages (i) components' telemetry data, which include performance metrics and RAS logs, and (ii) the failure state estimated with the failure localization model, to understand the likely cause of the failure. It uses the insight that a failed component behaves significantly differently from its healthy counterparts. For example, telemetry data obtained from a failed data server may reveal high load (e.g., high memory utilization) or an error (e.g., process crash), whereas the telemetry data of the healthy data servers will not reveal any such failures.

We use that insight to formulate the failure-diagnosing problem as an explainability problem that can be phrased as a conditional question: "Which modality of the telemetry data (amongst RAS logs and performance metrics) best explain the reason why one component is flagged as failed while others to be marked as healthy by the failure localization model?"

The failure diagnosis model answers that conditional question by statistically comparing the measurements of the failed component and the healthy components by using an unsupervised ML-based anomaly detection method that selects a measurement that best distinguishes the failed components from the healthy ones. If there has been a reliability failure (e.g., kernel crash), it will point to error logs, and if there has been a resource-overload-related failure, it will point to a performance metric, such as high server load. Note that the conditional question is fundamentally different from the non-conditional question "Which modality of the telemetry data

[7] Only paths from $\langle C_1, OSD_1 \rangle$ and $\langle C_2, OSD_2 \rangle$ are shown, for clarity. Redundancies and network components have also been removed for clarity.

are anomalous across all components?" The non-conditional question usually suffers from noises (e.g., each component produces hundreds if not thousands of error logs that may not be relevant to diagnosing the failure [37]), making it challenging to precisely distinguish anomaly from normal behavior. In other words, the conditional question eliminates the noise in the first place. For example, we should not flag a data server as failed just because its utilization is higher than the other servers. However, if the *failure localization model* identifies the server as failed and high load is the only factor that differs the failed data server from the other healthy data servers, then the failure of the failed data server is most likely due to high load.

Diagnosing Reliability Failures. Kaleidoscope attributes and diagnoses reliability failures based on log analysis. Working with the vendor and national labs, we have curated a library of regular-expression patterns to filter error logs that are indicative of reliability failures (e.g., kernel dump). Currently, our library consists of 184 regular expression patterns. In the absence of such a library, we could use existing log pattern mining tools (e.g., Baler [57]) to automatically create a library of regular-expression patterns from existing logs, and then filter the patterns based on their severity level, i.e., by using patterns of a severity level of 4 (warning) and above.

Kaleidoscope filters RAS logs of storage components by using the library of aforementioned regular-expression patterns (§IV-B). The error logs generated by the failed/failing components are compared to the error logs of healthy components, $\delta = L_{\text{UO}} - \bigcup_{i \in HO} L_i$, where L represents the log set, and UO and HO represent failed/failing and healthy components, respectively. If $\delta \neq \varnothing$, then δ is provided as evidence, and the failed status is attributed to component failures.

Diagnosing Resource Overload and Contention. Kaleidoscope attributes and diagnoses resource overload/contention based on the following telemetry data: (i) the server performance metrics (e.g., loadavg), which captures the load of a server at 5-minute intervals; (ii) the RAID device performance metrics (e.g., await time, which captures the average disk service time (in milliseconds)), and taken by a disk device to serve an I/O request; and (iii) the network performance counters (e.g., stall).

Kaleidoscope compares the performances of storage components of similar types (e.g., data servers) by using the local outlier factor (LOF) algorithm [40]. The LOF is based on the concept of *local density*, where locality is given by k-nearest neighbors and the density is estimated by the distance to the neighbors. By comparing the local density of a target with the local densities of its neighbors, Kaleidoscope identifies regions with similar densities, and pinpoints outliers that have a substantially lower density than their neighbors in terms of performance metric values. Using the LOF algorithm, we calculate LOF score using the aforementioned telemetry data for each component indicating the similarity/dissimilarity of the component to other components in terms of its performance. Using that score, we can ask the aforementioned conditional question. If we find that the failed component has a score of 1.0 (i.e., the performance is similar to that of other components), then there is no reason to believe that the component failure was caused by a resource overload/contention problem.

Table I: Effectiveness (measured by true positives) of Kaleidoscope's triage and root-cause analysis.

Localization	**True Positive**	**False Negative**	**Total**
	837 (99.3%)	6 (0.7%)	843
Diagnosis	**Correct Diagnosis**	**Misdiagnosis**	**Total**
Reliability Failure	340 (98.3%)	6 (1.7%)	346
Overload/Contention	468 (94.2%)	29 (5.8%)	497

We chose LOF because storage components within a homogeneous group could have different modes of operations that are not indicative of anomalies. For example, we found normal states in which k data servers had a low loadavg (less than 10) and $N - k$ data servers had a high loadavg (larger than 64). However, if there is one data server with a loadavg significantly higher than that of the rest, it indicates an anomaly, and such behavior is effectively captured by LOF. In Kaleidoscope, we use a configuration named LOF_r and declare a component to have "resource overload/contention" if the LOF value of the failed component is LOF_r times larger than the max LOF value of a healthy component. (The default value of LOF_r is 1.5.)

We use the outlier-based method to ask the conditional question for their simplicity and effectiveness. Our approach is very similar to that of, and inspired by, Distalyzer [36]. However, Distalyzer is only suited to offline diagnostics as it does not provide a methodology for identifying/labeling failed components because it assumes that such a label is already available. Thanks to Kaleidoscope's hierarchical approach, it is possible to integrate more sophisticated statistical methods and log analysis methodologies [31], [37], [58].

Training and Inference. Failure diagnosis is completely unsupervised, and therefore does not require any training. However, the method requires a library of regular-expression patterns that is created in the offline mode through manual methods (using vendor support) or automatic methods (using statistical learning techniques such as clustering [57], [59]). Failure diagnosis is implemented in Python.

VI. Evaluation

We have deployed Store Ping monitors on Cray Sonexion for two years and Kaleidoscope's live forensics on Cray Sonexion for more than three months. However, to comprehensively evaluate the effectiveness of Kaleidoscope's live forensics, we fed the two years of monitoring data collected by Store Ping monitors *retrospectively*. The evaluation is based on 843 production issues resolved by the Cray Sonexion operators over the two-year span. Each of the 843 issues has a corresponding report after manual investigation. We use the dataset as the *ground truth* to measure the true positives and false negatives. We also quantify the false positives by inspecting 100 randomly selected issues from the issues reported by Kaleidoscope.

A. Effectiveness

Kaleidoscope observed 26,596 I/O failure events in total (25,427 resource overloads and 1,169 reliability failures). The number is significantly higher than the 843 production issues. This is because many of the I/O failure events are transient and short-cycled and thus does not lead to production issues.

In Cray Sonexion, operators use the following two policies to identify important I/O failure events for manual investigation:

1) certain class of failures are auto-fixed by the system within one minute of occurrence (e.g., network recovery to route out bad links). Kaleidoscope finds out these cases and stops alarms by monitoring recovery events.
2) resource overload/contention events are often transient in nature and a mitigation action is triggered only when the condition continues for more than 30 minutes. Fig. 4 shows the histogram of the duration of these I/O failures.

Applying the above two policies on the results generated by Kaleidoscope reduces I/O failure events from 26,596 to 1,525. We evaluated the effectiveness of Kaleidoscope regarding its accuracy of both localizing the failed components and diagnosing their root causes. Table I summarizes the results.

Localization accuracy. Kaleidoscope was able to localize the failed components (caused by either reliability failures or resource overload/contention) for 99.3% of the production issues (837 out of 843). Only six out of 843 production issues were not detected by Kaleidoscope. We read the report and found that none of the six issues had any impact on the I/O completion time. All six issues belonged to disk drive failures. Those failures were recorded and flagged for repairs to avoid RAID failures. Kaleidoscope additionally detected 688 events. We refrained from labeling these additional events as false positives because there was no evidence supporting that these were not actual issues. On the contrary, we found that many of the performance issues either went unnoticed because the system was not monitored adequately (such as no dedicated monitoring for disk load), or were ignored because there was no automatic alerting mechanism to take remediation action on the events in time.

Diagnosis accuracy. Among the 843 production issues, 346 were caused by reliability failures and 497 were caused by resource overload. Applying the same heuristic on Kaleidoscope output as used by the operators (described above), we found that Kaleidoscope reported 340 reliability failures and 468 overloads, which accounts for 98.3% of reliability failures and 94.2% of the resource overload/contention issues from the list of production issues (see Table I). Kaleidoscope additionally detected 22 reliability failures and 558 resource overload issues. We had managed to manually validate 100 of those resource overload issues detected by Kaleidoscope and they were indeed true.

Kaleidoscope presented error logs or performance metric to the operator for further investigation. Kaleidoscope diagnosis module missed 35 production issues: (i) 6 issues were missed by localization module, and (ii) 29 resource overload issues coincidentally had random noise in the logs, which confused Kaleidoscope.

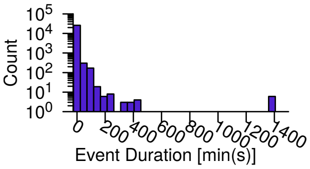


Figure 4: Histogram of failure duration.

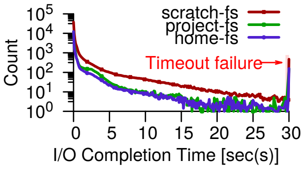


Figure 5: WrEx measured latency on three file partitions.

Table II: Comparing failure localization in Kaleidoscope and NetBouncer using 6 months of production data consisting of 186 issues.

	True Positive	False Negative	Alarms
Kaleidoscope	184	2	4892
NetBouncer	110	76	116,072

False Alarms & Misdiagnosis. It was challenging to measure false positives (FP) due to the lack of ground truth dataset—an I/O failure detected by Kaleidoscope but not being resolved could come from non-technical reasons (e.g., low priority jobs).

To statistically estimate the FP rate, we randomly selected 100 failures identified by Kaleidoscope (referred as Kaleidoscope events): 50 tagged with "reliability failures" and 50 tagged with "resource overload/contention." Kaleidoscope's failure localization model was able to localize *all* true cases of failures correctly. However, Kaleidoscope's failure diagnosis model misdiagnosed the root cause of four (out of 100) cases.

B. Baseline Comparison

Kaleidoscope is the first (to our knowledge) system that supports real-time forensics for peta-scale storage systems. In our work, we compare Kaleidoscope with NetBoucner [26]. We choose NetBouncer because it significantly outperformed existing failure localization methods designed for large-scale networks [60]–[62] and was tested on a real deployment.

Table II shows the localization accuracy of Kaleidoscope and NetBouncer [26], the state-of-the-art failure localization method. Our implementation was reviewed by the author(s) of NetBouncer.

NetBouncer has 110 true positives (out of 186 true positive cases found in 6 months of our retrospective data), i.e., it misses 76 true cases that were captured by Kaleidoscope. NetBouncer's missing those issues because it is incapable of modeling 1) non-determinism due to path redundancy and 2) temporal evolution of the component state, which is modeled by Kaleidoscope as discussed in §V-A. Furthermore, Kaleidoscope reports a total number of 4,892 events, far less than the number reported by NetBouncer. Given that self-recovered failures and overload condition less than 30 minutes can be filtered out, we can reduce the alarms to 412 (instead of 4,892) and 92,000 (instead of 116,072) respectively. The significant difference in the results of NetBouncer and Kaleidoscope is due to NetBouncer's inability of distinguishing I/O failure events as reliability failures or overload/contention.

C. Monitoring Overhead

We used the IOR benchmark [63] to measure the monitoring overhead in a worst-case scenario. The measurement used stress testing to max out the throughput offered by Cray Sonexion. IOR was running on 4,320 compute nodes during

Table III: Impact of 100 Store Ping monitors running at 30 second interval on IOR benchmark [63]. The mean value of I/O throughput without Kaleidoscope is normalized to 100. The off configuration is shared across both 100 and 6 montiors.

Kaleidoscope	**100 monitors**		**6 monitors**	
	Mean	*Std*	*Mean*	*Std*
Off	100	0.15	100	0.15
On	97.58	0.32	99.99	0.12

this measurement. Table III shows the monitoring overhead introduced by Store Pings when (i) 100 monitors were running at 30 second interval and (ii) 6 monitors were running at one-minute interval. Recall from §IV-A, we need 6 monitors for our probing plan to provide sufficient measurements, and we show result for 100 monitors to show the scalability of our solution. Store Pings decreased mean throughput *only* by less than 0.01% in Cray Sonexion. However, scaling to 100 monitors and increasing the frequency by 2× would decreases the throughput by less than 2.42%. Note that the average throughput in production is significantly below the peak throughput under the stress test. We also measured the time difference between the launch of Store Pings for a given interval and found that all Store Pings were launched within 10 seconds and 98.4% were launched within 3 seconds.

VII. Operational Experience

Our interaction with Cray Sonexion's operators shows that Kaleidoscope help them understand the tail latency and performance variation in near real time. Operators can detect performance regression by comparing the measurements from different points of time. Fig. 5 shows the latency measurement histogram for the WrEx Store Pings (RmEx and CrWr are omitted for clarity). We can see that 99% of WrEx completed within one second (Service Level Objectives or SLO), and only 0.14% failed with timeout. Furthermore, the operators use Kaleidoscope to characterize storage-related failures in Blue Waters. Such fine-grained characterization is not possible before the deployment of Kaleidoscope as previously deployed methods lacked joint analysis methods for identification, and disambiguation of failures. While previous work [15] has characterized I/O failures, to the best of our knowledge, this is the first study which considers the impact of both reliability and resource-overload failures on I/O request completion time.

A. I/O Failures Caused by Reliability Failures

Kaleidoscope finds that the most common symptom of reliability failures is performance degradation that leads to I/O failures; only a very small percentage (0.057% of 346 failures (Table I)) of reliability failures caused system-wide outages. For example, disk failure is tolerated by the RAID array which uses RAID resync on hot-spare disks to protect the RAID array from future failures. Such a resync or periodic scrubbing of a RAID array takes away a certain amount of bandwidth for an extended period of time, ranging from 4–12 hours, which increases completion time of I/O requests. As shown in Fig. 6, I/O requests during reliability failures increase the average completion time of I/O requests by up to 52.7× compared to the average I/O completion time in failure-free scenarios; the 99th percentile of I/O request completion times is 31 seconds.

B. I/O Failures Caused by Resource Overloads

Kaleidoscope reveals that resource overloads frequently lead to I/O failures. We used *disk service time* (await), returned by iostat, as a metric of the load on disk devices. await measures the average end-to-end time for a request including device queuing and the time to service the I/O request on the disk device. await is different from I/O completion time, which includes the traversal time between the client and the disk. Fig. 7 shows a histogram of disk service time (await) returned by iostat using an event-driven measurement (triggered only when loadavg exceeds 50). Such anomalies occur frequently. We found 14,081 such unique events by clustering the per-disk continuous data points in time with *service times longer than one second.* **Excessive I/O.** Excessive I/O requests create high load on the server and lead to disk-level contention, causing performance and stability issues. Fig. 8 shows a histogram of the duration of excessive I/O requests by applications to the metadata server. The duration of high I/O requests are generally small (lasting less than 10 seconds); however, there is a long tail of applications that send high I/O requests for hours. In one case, an application caused high load on the metadata server by opening and closing 75,000+ million files in 4 hours, leading to 20,000+ I/O requests per second. During that event, loadavg increased from 60 to 350 with the 50th and 99th percentile duration being 12 and 227 minutes, respectively.

High load. The increase of I/O request completion time has a strong correlation with the load on storage servers. High load conditions are caused by a flood of I/O requests on a storage server by either one application (e.g., extreme I/O), or multiple applications competing for resources. Fig. 9 shows the histogram of load across all servers. It shows the average and 99th percentile completion time of I/O requests at different load values of the storage servers. Overall, we can see a strong correlation between an increase in load and the completion time of I/O requests. At high load (loadavg of 350), the average and 99th percentile I/O request completion time increases to 1 second and 10 seconds, respectively.

C. Identifying One-off Failures

Kaleidoscope found many one-off, unique failures that do not have a common pattern and are previously unknown. Such failures can hardly be anticipated based on historical datasets. Kaleidoscope found four such failures per month on average. The following describes one of such failures. LNET nodes serve as bridge between computing nodes and storage servers. A request from a client to an OSD (a RAID disk device) can be served by any of 4 LNET nodes. For any pair of ⟨client, OSD⟩, the group of 4 LNET nodes are fixed and chosen in round-robin when routing a request. In a rare failure incident, LNET had partially failed, but the failure was not detected as Cray Sonexion uses heartbeats to detect failures. The partial failure caused LNET to drop requests passing through it, causing I/O failures. The I/O bandwidth (in MB/sec) for the applications served by the failed LNET node decreased by 25+% for

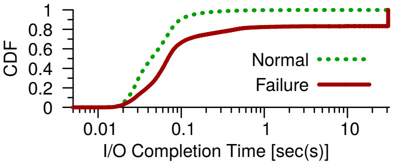


Figure 6: CDF of I/O request completion time under reliability failures ("Faliure") and no failures ("Normal").

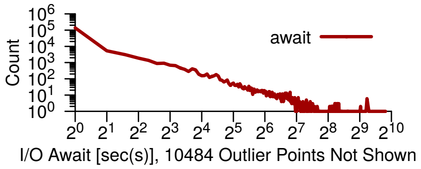


Figure 7: Histogram of disk service time as a load metric. The figure shows that overload is frequent in Cray Sonexion.

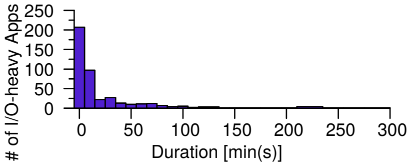


Figure 8: Histogram of duration of high I/O requests per application on a metadata server. The tail shows extreme I/O.

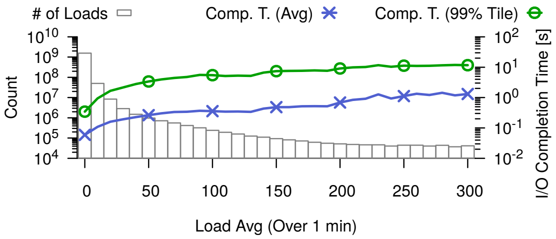


Figure 9: Correlation between load (i.e., loadavg) and latency. ("Comp. T." is the completion time of I/O requests.)

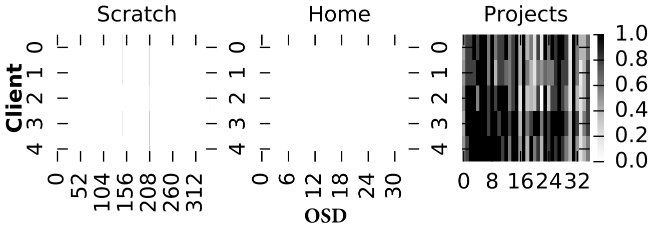


Figure 10: Issue on Scratch OSD 208 and Projects MDS.

multiple hours. Upon investigation, it was found that the LNET had suffered a software error that caused it to drop I/O requests for weeks. Using Kaleidoscope, we detected the failure in <5 minutes.

VIII. Discussion and Limitations

A. *Interpretability of ML models*

Researchers provide diverse and sometimes non-overlapping motivations for interpretability, and offer myriad notions of what attributes render models and results interpretable [64]. Below, we discuss two aspects of this general interpretability problem in the context of Kaleidoscope.

Model Interpretability. The proposed hierarchical unsupervised ML models will significantly enhance interpretability, and hence wide-spread adoption/deployment of Kaleidoscope.

1) We use models that inherently capture *all* the system modeling assumptions. For example, Kaleidoscope through Factor Graphs (FG), a probabilistic graphical model (PGM) formalism, encodes that an I/O request failure occurs only if one or more components on the I/O request path fail.
2) Our model incorporates aspects of the storage system, i.e., topology and storage system architecture details directly into the graphical structure of the PGM. This allows the overall hierarchical model to be constructed directly from domain knowledge without requiring any pre-labeled training data (in contrast to supervised methods like deep neural networks).

Kaleidoscope can be extended to different system topologies and storage system architectures (described later in §VIII-B). Kaleidoscope automatically creates/changes the ML models with appropriate parameters using the system topology and file system I/O protocols (encoded through I/O request paths), which can be provided as an input. For example, Kaleidoscope will automatically add additional node(s) to the FG model to capture the failure state of newly added component(s) in the system. Similarly, if the I/O protocols change (i.e., the path taken by an I/O operations change), Kaleidoscope will automatically change the the factor functions to reflect new I/O paths.

Result Interpretability. System managers of Blue Waters have created several monitoring dashboards [51] to visualize live data in multiple ways. As we highlight in the §I, these analyses process vast amounts of telemetry data leading to cognitive overload of the system managers. Kaleidoscope strives to reduce this cognitive overload by providing intuitive charts and summaries of the telemetry data to quickly identify and understand the failure location and the failure mode (i.e., reasoning behind the ML output). An example of such a chart providing evidence of failure localization inference is shown in Fig. 10. The inference pointed to the existence of two concurrent failures: (i) a load issue on scratch data server 208 and (ii) an outage of projects file system metadata server. Fig. 10 uses a heatmap to depict a failure impact on clients (as an evidence). Each cell in the heatmap shows the ratio of operations that took longer than 1 second to the total number of operations issued during 5 minutes interval by a given client (y-axis) to each data server from Scratch, Home, and Projects Lustre partitions (x-axis), with darker color means higher ratio. Clients 0, 1, and 2 are the login nodes on Ethernet network, client 3 gives an aggregated view of all 25 Import/Export nodes on Infiniband network, and client 4 provides an aggregated view of all 64 service nodes on compute network. As seen from the figure, scratch data server 208 and project metadata server are behaving anomalously compared to rest of the cluster. Thus, Kaleidoscope, in addition to detecting and diagnosing failures, provides *significant value* in directly summarizing and visualizing the relevant evidence for a detected failure. Without Kaleidoscope the system managers will have to monitor a large number of failures-modes and failed-components across every instant of time.

B. *Generalizing Kaleidoscope to other systems.*

Kaleidoscope is not tied to a specific storage architecture. Kaleidoscope uses (i) Store Ping monitoring data for failure localization, and (ii) performance metrics and RAS logs for failure diagnosis. Performance metrics and RAS logs are already available on all storage systems. However, Store Pings must be deployed on the storage system for running Kaleidoscope. The goal of Store Ping is to test all components of the storage system such as load balancers, network, metadata servers, and object storage servers/devices (OSDs) using native storage-system operations (such as read, write, remove, etc.). Store Ping achieves this goal by pinning the files strategically (discussed in §IV-A) onto OSDs such that the health of each of these components can be inferred. Fortunately, such support is available for all popular POSIX-compliant HPC storage systems such as Ceph [65], Gluster [66] and GPFS [67]. Let us consider Ceph. Storage cluster clients in Ceph use the CRUSH (controlled replication under scalable hashing [68]) algorithm to efficiently compute information about data location, instead of having to depend on a central lookup table (e.g., in the case of Lustre clients use MDS for file lookup). We can use CRUSH (via crushtool) to get the mapping rules, and use those to place the files to specific OSDs (and in doing so invoke MDS operations). Finally, recall from above that the ML models used by Kaleidoscope are not tied to specific system topology and storage protocols.

C. Dealing with large number of alarms

There is a trade-off between detecting failures quickly and generating too many alarms (due to transient failures and micro-bursts). This is a fundamental limitation of any failure detection algorithm (and it is not tied to ML). Hence, to reduce the overhead (§VI-C), Kaleidoscope on Blue Waters is configured to collect datasets at 60s intervals. Therefore, Kaleidoscope cannot detect micro-burst performance anomalies [69] and transient failures that are shorter than dataset collection interval (60s). Detecting transient failures/ micro-bursts is an active area of research as it allows designers to craft load-balancing and quality-of-service techniques.

In this work, we report all failures irrespective of their duration (except for the failures that are shorter than dataset collection interval and cannot be detected by Kaleidoscope). Hence, Kaleidoscope reported >26,000 failures, which is much greater than production issues. We observe more alarms than production issues because many of the I/O failure events are transient and short-lived and thus does not lead to production issues. In particular, most of the short-lived issues are related to resource overload problems (caused by one or more applications), which are filtered using heuristics discussed in §VI-A. Moreover, as we show in §VII, Kaleidoscope caught many failures that went unnoticed in the production for several weeks despite all the existing monitoring tools.

IX. Related Work

Kaleidoscope is built upon the wealth literature on failure detection and localization [16], [26], [30], [31], [70]–[77]. Kaleidoscope is more than a failure detector. It not only detects and localizes the failing component, but also reveals the probable causes by pinpointing the error logs or performance metrics. As discussed in §I, the capability of jointly localizing and discerning the failure mode is critically important to devise the right recovery strategies. To the best of our knowledge, no existing solution provides such a capability.

Kaleidoscope is the first effort for designing a hierarchical domain-driven ML-based realtime failure detection and diagnosis framework that leverages vast amounts of heterogeneous telemetry data for large-scale high-performance storage systems. Kaleidoscope's failure detection and diagnosis capabilities are fundamentally different from prior work that applies statistical or machine learning using system telemetry data: (i) prior solutions are data hungry requiring big data for training (e.g., [32], [33]), (ii) prior work supports either (a) anomaly detection [34], [35], (b) failure localization [26], [27], [29]–[31], or (c) failure diagnosis only [36], [37]. (iii) no prior solution handles uncertainty in telemetry data and in the system.

Kaleidoscope *proactively* detects, localizes, and diagnoses I/O timeout and slowness before the applications being affected. It uses active measurements from Store Ping monitors to support ML-based failure detection and diagnosis. It is different from passive or reactive approaches [30], [73], [78]. This requires very low monitoring overhead, i.e., Kaleidoscope has to run on a small subset of client nodes and cannot probe every single path deterministically. While active probing is a well-established technique for failure detection [16], [26], Kaleidoscope solves those key challenge by effectively modeling non-determinism and uncertainty in the distributed systems as discussed in §III. As a result, Kaleidoscope reduces the number of probes by orders of magnitude compared with existing methods [16], [26], [30], [77]. In fact, active measurements are applied in limited context for storage subsystems (e.g., TOKIO [15], [79]). However, these probes have high overhead, and hence are executed once in a day.

X. Conclusion

This paper advocates the need for identifying and diagnosing resource overload and reliability failures jointly to effectively coordinate recovery strategy. We build Kaleidoscope and deploy it on a petascale production system to disambiguate component failures from resource overload/contention issues.

Acknowledgments

We thank W.T.C. Kramer, A. Gentile, J. Brandt, K. Saboo, V. Yogatheesan, K. Atchley, and J. Applequist for their insightful comments on the early drafts of this manuscript. This research was supported in part by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, under Award Number 2015-02674. This work is partially supported by the National Science Foundation (NSF) under Grant No 2029049; a Sandia National Laboratories contract number 1951381[8]; by the IBM-ILLINOIS Center for Cognitive Computing Systems Research (C3SR), a research collaboration that is part of the IBM AI Horizon Network; by Intel and NVIDIA through equipment donations. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF, Sandia National Laboratories, NVIDIA or Intel. The first author is also supported by the 2020 IBM PhD fellowship.

This research is part of the Blue Waters sustained-petascale computing project, which is supported by the National Science Foundation (awards OCI-0725070 and ACI-1238993) the State of Illinois, and as of December, 2019, the National Geospatial-Intelligence Agency. Blue Waters is a joint effort of the University of Illinois at Urbana-Champaign and its National Center for Supercomputing Applications.

[8] 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.

References

[1] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr, "Basic concepts and taxonomy of dependable and secure computing," *IEEE Trans. Dependable Secur. Comput.*, vol. 1, no. 1, pp. 11–33, Jan. 2004.

[2] C. Di Martino, F. Baccanico, J. Fullop, W. Kramer, Z. Kalbarczyk, and R. Iyer, "Lessons learned from the analysis of system failures at petascale: The case of Blue Waters," in *Proc. of 44th Annual IEEE/IFIP Int. Conf. on Dependable Systems and Networks (DSN)*, 2014.

[3] H. S. Gunawi, R. O. Suminto, R. Sears, C. Golliher, S. Sundararaman, X. Lin, T. Emami, W. Sheng, N. Bidokhti, C. McCaffrey, G. Grider, P. M. Fields, K. Harms, R. B. Ross, A. Jacobson, R. Ricci, K. Webb, P. Alvaro, H. B. Runesha, M. Hao, and H. Li, "Fail-Slow at Scale: Evidence of Hardware Performance Faults in Large Production Systems," in *Proceedings of the 16th USENIX Conference on File and Storage Technologies (FAST'18)*, Oakland, CA, USA, Feb. 2018.

[4] B. Schroeder and G. Gibson, "A large-scale study of failures in high-performance computing systems," *IEEE Transactions on Dependable and Secure Computing*, vol. 7, no. 4, pp. 337–350, 2010.

[5] S. Floyd, "Congestion control principles," BCP 41, RFC 2914, September, Tech. Rep., 2000.

[6] S. Jha, A. Patke, J. Brandt, A. Gentile, B. Lim, M. Showerman, G. Bauer, L. Kaplan, Z. Kalbarczyk, W. Kramer, and R. Iyer, "Measuring congestion in high-performance datacenter interconnects," in *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. Santa Clara, CA: USENIX Association, Feb. 2020, pp. 37–57. [Online]. Available: https://www.usenix.org/conference/nsdi20/presentation/jha

[7] S. Chunduri, T. Groves, P. Mendygral, B. Austin, J. Balma, K. Kandalla, K. Kumaran, G. Lockwood, S. Parker, S. Warren, N. Wichmann, and N. Wright, "Gpcnet: Designing a benchmark suite for inducing and measuring contention in hpc networks," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '19. New York, NY, USA: Association for Computing Machinery, 2019. [Online]. Available: https://doi.org/10.1145/3295500.3356215

[8] M. Mubarak, P. Carns, J. Jenkins, J. K. Li, N. Jain, S. Snyder, R. Ross, C. D. Carothers, A. Bhatele, and K.-L. Ma, "Quantifying I/O and communication traffic interference on dragonfly networks equipped with burst buffers," in *Cluster Computing, 2017 IEEE Int'l Conf. on*. IEEE, 2017, pp. 204–215.

[9] X. Yang, J. Jenkins, M. Mubarak, R. B. Ross, and Z. Lan, "Watch out for the bully! job interference study on dragonfly network," in *SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2016, pp. 750–760.

[10] C. Di Martino, W. Kramer, Z. Kalbarczyk, and R. Iyer, "Measuring and understanding extreme-scale application resilience: A field study of 5,000,000 HPC application runs," in *Dependable Systems and Networks (DSN), Annual IEEE/IFIP International Conference on*, 2015, pp. 25–36.

[11] B. Schroeder and G. A. Gibson, "A large-scale study of failures in high-performance computing systems," *IEEE Transactions on Dependable and Secure Computing*, vol. 7, no. 4, pp. 337–350, 2010.

[12] T. Hauer, P. Hoffmann, J. Lunney, D. Ardelean, and A. Diwan, "Meaningful availability," in *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. Santa Clara, CA: USENIX Association, Feb. 2020, pp. 545–557. [Online]. Available: https://www.usenix.org/conference/nsdi20/presentation/hauer

[13] M. Snir, W. D. Gropp, and P. Kogge, "Exascale research: preparing for the post-moore era," 2011.

[14] N. Baker, F. Alexander, T. Bremer, A. Hagberg, Y. Kevrekidis, H. Najm, M. Parashar, A. Patra, J. Sethian, S. Wild, K. Willcox, and S. Lee, "Workshop report on basic research needs for scientific machine learning: Core technologies for artificial intelligence," 2 2019.

[15] G. K. Lockwood, N. J. Wright, S. Snyder, P. Carns, G. Brown, and K. Harms, "Tokio on clusterstor: Connecting standard tools to enable holistic i/o performance analysis," 2018.

[16] C. Guo, L. Yuan, D. Xiang, Y. Dang, R. Huang, D. Maltz, Z. Liu, V. Wang, B. Pang, H. Chen, Z.-W. Lin, and V. Kurien, "Pingmesh: A Large-Scale System for Data Center Network Latency Measurement and Analysis," in *Proceedings of the 2015 ACM SIGCOMM Conference (SIGCOMM'15)*, London, United Kingdom, Aug. 2015.

[17] B. Yang, X. Ji, X. Ma, X. Wang, T. Zhang, X. Zhu, N. El-Sayed, H. Lan, Y. Yang, J. Zhai, W. Liu, and W. Xue, "End-to-end i/o monitoring on a leading supercomputer," in *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. Boston, MA: USENIX Association, Feb. 2019, pp. 379–394. [Online]. Available: https://www.usenix.org/conference/nsdi19/presentation/yang

[18] A. Agelastos, B. Allan, J. Brandt, P. Cassella, J. Enos, J. Fullop, A. Gentile, S. Monk, N. Naksinehaboon, J. Ogden *et al.*, "The lightweight distributed metric service: a scalable infrastructure for continuous monitoring of large scale computing systems and applications," in *SC'14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2014, pp. 154–165.

[19] "splunk," https://www.splunk.com.

[20] J. B. Dugan, S. J. Bavuso, and M. A. Boyd, "Dynamic fault-tree models for fault-tolerant computer systems," *IEEE Transactions on reliability*, vol. 41, no. 3, pp. 363–377, 1992.

[21] A. Gulati, C. Kumar, I. Ahmad, and K. Kumar, "Basil: Automated io load balancing across storage devices." in *Fast*, vol. 10, 2010, pp. 13–13.

[22] R. Alverson, D. Roweth, and L. Kaplan, "The gemini system interconnect," in *2010 18th IEEE Symposium on High Performance Interconnects*. IEEE, 2010, pp. 83–87.

[23] B. Alverson, E. Froese, L. Kaplan, and D. Roweth, "Cray xc series network," *Cray Inc., White Paper WP-Aries01-1112*, 2012.

[24] S. Jha, M. Showerman, A. Saxton, J. Enos, G. Bauer, B. Bode, J. Brandt, A. Gentile, Z. Kalbarczyk, R. K. Iyer, and W. Kramer, "Holistic measurement-driven system assessment," in *Proceedings of the ACM International Conference on Supercomputing*, Sep. 2019.

[25] "operational data analytics," https://sc19.supercomputing.org/proceedings/bof/bof_pages/bof137.html, 2019.

[26] C. Tan, Z. Jin, C. Guo, T. Zhang, H. Wu, K. Deng, D. Bi, and D. Xiang, "NetBouncer: Active Device and Link Failure Localization in Data Center Networks," in *Proceedings of the 16th USENIX Symposium on Networked Systems Design and Implementation (NSDI'19)*, Boston, MA, USA, Feb. 2019.

[27] P. Bahl, R. Chandra, A. Greenberg, S. Kandula, D. A. Maltz, and M. Zhang, "Towards highly reliable enterprise network services via inference of multi-level dependencies," in *Proceedings of the 2007 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, ser. SIGCOMM '07. New York, NY, USA: Association for Computing Machinery, 2007, p. 13–24. [Online]. Available: https://doi.org/10.1145/1282380.1282383

[28] W. M., H. S., S. K., G. T., and T. M., "Canario: Sounding the alarm on io-related performance degradation," in *Proceedings of the 34th IEEE International Parallel and Distributed Processing Symposium, IPDPS,*, 2020.

[29] B. Arzani, S. Ciraci, L. Chamon, Y. Zhu, H. H. Liu, J. Padhye, B. T. Loo, and G. Outhred, "007: Democratically finding the cause of packet drops," in *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. Renton, WA: USENIX Association, Apr. 2018, pp. 419–435. [Online]. Available: https://www.usenix.org/conference/nsdi18/presentation/arzani

[30] Q. Zhang, G. Yu, C. Guo, Y. Dang, N. Swanson, X. Yang, R. Yao, , M. Chintalapati, A. Krishnamurthy, and T. Anderson, "Deepview: Virtual Disk Failure Diagnosis and Pattern Detection for Azure," in *Proceedings of the 15th USENIX Symposium on Networked Systems Design and Implementation (NSDI'18)*, Renton, WA, USA, Apr. 2018.

[31] B. Yang, X. Ji, X. Ma, X. Wang, T. Zhang, X. Zhu, N. El-Sayed, H. Lan, Y. Yang, J. Zhai, W. Liu, and W. Xue, "End-to-end I/O Monitoring on a Leading Supercomputer," in *Proceedings of the 16th USENIX Symposium on Networked Systems Design and Implementation (NSDI'19)*, Boston, MA, USA, Feb. 2019.

[32] A. Das, F. Mueller, P. Hargrove, E. Roman, and S. Baden, "Doomsday: predicting which node will fail when on supercomputers," in *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2018, pp. 108–121.

[33] M. Du, F. Li, G. Zheng, and V. Srikumar, "Deeplog: Anomaly detection and diagnosis from system logs through deep learning," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 1285–1298.

[34] W. Xu, L. Huang, A. Fox, D. Patterson, and M. I. Jordan, "Detecting large-scale system problems by mining console logs," in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, 2009, pp. 117–132.

[35] M. Y. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. Brewer, "Pinpoint: Problem determination in large, dynamic internet services," in *Proceedings International Conference on Dependable Systems and Networks*. IEEE, 2002, pp. 595–604.

[36] K. Nagaraj, C. Killian, and J. Neville, "Structured comparative analysis of systems logs to diagnose performance problems," in *Presented as*

part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12). San Jose, CA: USENIX, 2012, pp. 353–366. [Online]. Available: https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/nagaraj

[37] ——, "Structured Comparative Analysis of Systems Logs to Diagnose Performance Problems," in *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI'12)*, San Jose, CA, USA, Apr. 2012.

[38] O. Tuncer, E. Ates, Y. Zhang, A. Turk, J. Brandt, V. J. Leung, M. Egele, and A. K. Coskun, "Diagnosing performance variations in hpc applications using machine learning," in *High Performance Computing*, J. M. Kunkel, R. Yokota, P. Balaji, and D. Keyes, Eds. Cham: Springer International Publishing, 2017, pp. 355–373.

[39] K. Khlebnikov and K. Kolyshkin, "ioping: simple disk I/0 latency measuring tool," https://github.com/koct9i/ioping.

[40] M. M. Breunig, H.-P. Kriegel, R. T. Ng, and J. Sander, "LOF: Identifying Density-based Local Outliers," in *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data (SIGMOD'00)*, Dallas, Texas, USA, May 2000.

[41] T. Sherman, "Cray sonexion™ data storage system."

[42] "Lustre filesystem," http://lustre.org/, Accessed: 2019-02-06.

[43] S. Oral and F. Baetke, "LUSTRE Community BOF: Lustre in HPC and Emerging Data Markets: Roadmap, Features and Challenges," https://sc18.supercomputing.org/proceedings/bof/bof_pages/bof176.html.

[44] M. Holland and G. Gibson, "Parity Declustering for Continuous Operation in Redundant Disk Arrays," in *Proceedings of the 5th International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS-V)*, Boston, MA, USA, Oct. 1992.

[45] Theoretical and Computational Biophysic Group, University of Illinois at Urbana-Champaign, "NAMD v. 2.9," http://www.ks.uiuc.edu/Development/Download/download.cgi?PackageName=NAMD, 2014.

[46] "Zipkin," https://zipkin.io.

[47] Y. Shkuro, "Evolving Distributed Tracing at Uber Engineering," https://eng.uber.com/distributed-tracing/.

[48] P. Carns, K. Harms, W. Allcock, C. Bacon, S. Lang, R. Latham, and R. Ross, "Understanding and improving computational science storage access through continuous characterization," *ACM Trans. Storage*, vol. 7, no. 3, Oct. 2011. [Online]. Available: https://doi.org/10.1145/2027066.2027068

[49] L. Ma, T. He, A. Swami, D. Towsley, and K. K. Leung, "Network capability in localizing node failures via end-to-end path measurements," *IEEE/ACM transactions on networking*, vol. 25, no. 1, pp. 434–450, 2016.

[50] L. Ma, T. He, A. Swami, D. Towsley, K. K. Leung, and J. Lowe, "Node Failure Localization via Network Tomography," in *Proceedings of the 2014 Conference on Internet Measurement Conference*, ser. IMC '14. New York, NY, USA: ACM, 2014, pp. 195–208. [Online]. Available: http://doi.acm.org/10.1145/2663716.2663723

[51] B. D. Semeraro, R. Sisneros, J. Fullop, and G. H. Bauer, "It takes a village: Monitoring the blue waters supercomputer," in *Proceedings of the 2014 IEEE International Conference on Cluster Computing (CLUSTER'14)*, Madrid, Spain, Sep 2014.

[52] M. Butler, "blue waters super system," https://www.globusworld.org/files/2010/02/SuperSystem-BW-.pdf.

[53] D. Koller, N. Friedman, and F. Bach, *Probabilistic graphical models: principles and techniques*. MIT press, 2009.

[54] P. Diaconis, D. Ylvisaker *et al.*, "Conjugate priors for exponential families," *The Annals of statistics*, vol. 7, no. 2, pp. 269–281, 1979.

[55] R. M. Neal, "Probabilistic inference using markov chain monte carlo methods," 1993.

[56] J. Salvatier, T. V. Wiecki, and C. Fonnesbeck, "Probabilistic programming in python using pymc3," *PeerJ Computer Science*, vol. 2, p. e55, Apr. 2016. [Online]. Available: https://doi.org/10.7717/peerj-cs.55

[57] N. Taerat, J. Brandt, A. Gentile, M. Wong, and C. Leangsuksun, "Baler: deterministic, lossless log message clustering tool," *Computer Science-Research and Development*, vol. 26, no. 3-4, p. 285, 2011.

[58] F. Mahdisoltani, I. Stefanovici, and B. Schroeder, "Improving Storage System Reliability with Proactive Error Prediction," in *Proceedings of the 2017 USENIX Annual Technical Conference (USENIX ATC'17)*, San Jose, CA, July 2017.

[59] R. Vaarandi, "A data clustering algorithm for mining patterns from event logs," in *Proceedings of the 3rd IEEE Workshop on IP Operations & Management (IPOM 2003)(IEEE Cat. No. 03EX764)*. IEEE, 2003, pp. 119–126.

[60] Y. Peng, J. Yang, C. Wu, C. Guo, C. Hu, and Z. Li, "deTector: a Topology-aware Monitoring System for Data Center Networks," in *Proceedings of the 2017 USENIX Annual Technical Conference (USENIX ATC'17)*, San Jose, CA, July 2017.

[61] H. Herodotou, B. Ding, S. Balakrishnan, G. Outhred, and P. Fitter, "Scalable Near Real-Time Failure Localization of Data Center Networks," in *Proceedings of the 20th ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD'14)*, New York City, NY, USA, Aug. 2014.

[62] D. Ghita, H. Nguyen, M. Kurant, K. Argyraki, and P. Thiran, "Netscope: Practical Network Loss Tomography," in *Proceedings of 2010 IEEE Conference on Computer Communications (INFOCOM'10)*, San Diego, CA, USA, Mar. 2010.

[63] "Parallel file system I/O Benchmark," https://github.com/LLNL/ior.

[64] Z. C. Lipton, "The mythos of model interpretability," *Queue*, vol. 16, no. 3, pp. 31–57, 2018.

[65] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, and C. Maltzahn, "Ceph: A Scalable, High-Performance Distributed File System," in *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI'06)*, Seattle, WA, November 2016.

[66] E. B. Boyer, M. C. Broomfield, and T. A. Perrotti, "Glusterfs one storage server to rule them all," 7 2012.

[67] F. Schmuck and R. Haskin, "GPFS: A Shared-Disk File System for Large Computing Clusters," in *Proceedings of the 1st USENIX Conference on File and Storage Technologies (FAST'02)*, Monterey, California, USA, Jan. 2002.

[68] S. A. Weil, S. A. Brandt, E. L. Miller, and C. Maltzahn, "Crush: Controlled, scalable, decentralized placement of replicated data," in *SC'06: Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*. IEEE, 2006, pp. 31–31.

[69] Q. Zhang, V. Liu, H. Zeng, and A. Krishnamurthy, "High-resolution measurement of data center microbursts," in *Proceedings of the 2017 Internet Measurement Conference*, ser. IMC '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 78–85. [Online]. Available: https://doi.org/10.1145/3131365.3131375

[70] J. B. Leners, H. Wu, W.-L. Hung, M. K. Aguilera, and M. Walfish, "Detecting Failures in Distributed Systems with the Falcon Spy Network," in *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP'11)*, Cascais, Portugal, Oct. 2011.

[71] J. B. Leners, T. Gupta, M. K. Aguilera, and M. Walfish, "Taming Uncertainty in Distributed Systems with Help from the Network," in *Proceedings of the 10th European Conference on Computer Systems (EuroSys'15)*, Bordeaux, France, Apr. 2015.

[72] ——, "Improving availability in distributed systems with failure informers," in *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI'13)*, Lombard, IL, USA, Apr. 2013.

[73] P. Huang, C. Guo, J. R. Lorch, L. Zhou, and Y. Dang, "Capturing and Enhancing In Situ System Observability for Failure Detection," in *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI'18)*, Carlsbad, CA, USA, Oct. 2018.

[74] N. Hayashibara, X. Défago, R. Yared, and T. Katayama, "The ϕ Accrual Failure Detector," in *Proceedings of the 23rd IEEE International Symposium on Reliable Distributed Systems (SDRS'04)*, Florianópolis, Brazil, Oct. 2004.

[75] T. D. Chandra and S. Toueg, "Unreliable Failure Detectors for Reliable Distributed Systems," *Journal of the ACM*, vol. 43, no. 2, pp. 225–267, Mar. 1996.

[76] M. K. Aguilera, G. L. Lann, and S. Toueg, "On the Impact of Fast Failure Detectors on Real-Time Fault-Tolerant Systems," in *Proceedings of the 16th International Symposium on Distributed Computing (DISC'02)*, Toulouse, France, Oct. 2002.

[77] B. Arzani, S. Ciraci, L. Chamon, Y. Zhu, H. Liu, J. Padhye, B. T. Loo, and G. Outhred, "007: Democratically Finding the Cause of Packet Drops," in *Proceedings of the 15th USENIX Symposium on Networked Systems Design and Implementation (NSDI'18)*, Renton, WA, USA, Apr. 2018.

[78] A. Roy, H. Zeng, J. Bagga, and A. C. Snoeren, "Passive Realtime Datacenter Fault Detection and Localization," in *Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI'17)*, Boston, MA, Mar. 2017.

[79] G. K. Lockwood, S. Snyder, T. Wang, S. Byna, P. Carns, and N. J. Wright, "A year in the life of a parallel file system," in *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*, 2018, pp. 931–943.