Exploring Void Search for Fault Detection on Extreme Scale Systems

Eduardo Berrocal*, Li Yu*, Sean Wallace*, Michael E. Papka†‡, and Zhiling Lan*

*Illinois Institute of Technology, Chicago, IL, USA
†Argonne National Laboratory, Argonne, IL, USA
‡Northern Illinois University, DeKalb, IL, USA
eberroca@iit.edu, lyu17@iit.edu, swallac6@iit.edu, papka@anl.gov, lan@iit.edu

Abstract—**Mean Time Between Failures (MTBF), now calculated in days or hours, is expected to drop to minutes on exascale machines. The advancement of resilience technologies greatly depends on a deeper understanding of faults arising from hardware and software components. This understanding has the potential to help us build better fault tolerance technologies. For instance, it has been proved that combining checkpointing and failure prediction leads to longer checkpoint intervals, which in turn leads to fewer total checkpoints. In this paper we present a new approach for fault detection based on the Void Search (VS) algorithm. VS is used primarily in astrophysics for finding areas of space that have a very low density of galaxies. We evaluate our algorithm using real environmental logs from Mira Blue Gene/Q supercomputer at Argonne National Laboratory. Our experiments show that our approach can detect almost all faults (i.e., sensitivity close to 1) with a low false positive rate (i.e., specificity values above 0.7). We also compare our algorithm with a number of existing detection algorithms, and find that ours outperforms all of them.**

Index Terms—**Reliability, Void Search, Fault Detection, Blue Gene/Q, Environmental Data**

I. Introduction

Mean Time Between Failures (MTBF), now calculated in days or hours, is expected to drop to minutes in exascale machines [4] [2]. In the past, a number of technologies have been presented to improve fault tolerance (FT) of large-scale systems, and new resilience techniques are emerging to address new challenges posed by extreme-scale computing [5], [6], [7], [8], [9]. *The advancement of resilience technologies, however, greatly depends on a deeper understanding of faults* arising from hardware/software components. According to literature [1], faults are defined as hardware defects or software flaws. They can cause system components to change from a normal state to an error state, and the use of components in an error state can lead to failures. In this work, we define the process of identifying when a fault has occurred and pinpointing its location as *fault detection.* In particular, our work focuses on detecting the faults that are likely to lead to failures.

Fault detection is critical to confine faults, avoid or limit their propagation, and recover quickly from them. For example, failure prevention methods (e.g., preemptive process migration [19]) rely on good fault detection to avoid impending failure; the checkpoint operation cost could be substantially reduced when instructed as to when and where to perform checkpoints.

Numerous works have been devoted to fault detection on large-scale systems [10], [20], [21], [22]. However, the totality of existing studies are based on RAS (Reliability, Availability and Serviceability) logs[1]. These logs are simple and limited since they are designed to be read and understood by humans (e.g. system administrators). RAS logs are composed of messages generated by a centralized monitor process, which queries the different components at a given frequency and may raise a warning if some value is above a given threshold. For example, the time to complete an I/O operation is too long or CPU temperature is too high. Or, it may raise an error if a component is not functioning correctly or not working altogether. By looking at one value at a time, however, RAS logs may miss important patterns hidden in the data.

Environmental logs, on the other hand, are numerical values directly read from the hardware sensors spread all over the system such as fan speed, CPU temperature, input voltage, and so on. Since every new generation of supercomputing systems come with better hardware sensors and profiling capabilities, it is of great importance to understand environmental data especially with respect to resilience. *In this study, we show that using environmental log data to detect hardware faults (and hence predict potential failures) is not only possible, but it can actually outperform RAS logs-based methods.*

We present *a new approach based on the Void Search (VS) algorithm.* VS has been studied extensively in the field of astrophysics [15], [16], [17], [18], primarily in the context of searching for regions of space with a low density of galaxies. Unlike traditional data mining algorithms which are focused on extracting patterns from existing data points, VS looks for patterns of empty space – or low density regions – and uses these patterns to detect anomalies in the data. Generally, VS is very suitable for situations where faults, or outliers, are scarce (or nonexistent) in the training data in comparison to "normal" data (in other words, when outliers are very difficult to characterize). We propose a VS algorithm for hardware fault detection using environmental data.

We evaluate our design using real logs from the Mira super-

[1]RAS logs are also called event logs.

computer, a 48-rack IBM Blue Gene/Q at Argonne National Laboratory [25]. Our experiments show that VS can detect almost all faults (i.e., sensitivity close to 1) with a low false positive rate (i.e., specificity values above 0.7). The result is better than those obtained by existing detection studies based on RAS logs [10], [21], [22], indicating that the combined use of environmental log and our VS algorithm has real potential to improve fault detection. Furthermore, we analyze and compare our design with other outlier detection algorithms based on classifiers such as Naive Bayes (NB), Support Vector Machines (SVMs) and Artificial Neural Networks (ANNs) as well as K-means, a clustering based method. We find that our VS based design outperforms all other detection algorithms in terms of detection accuracy. The proposed work is the first of its kind and therefore will open a new research direction in the area of fault detection on extreme scale systems.

The rest of this paper is organized as follows. Section II gives an overview of VS algorithms, the Mira supercomputer, and information related to its environmental logs. Section III presents the design of our VS-based algorithm. The results for all the experiments can be found in Section IV. Section V presents a discussion about our findings. Finally, Sections VI and VII conclude with related work and a summary, respectively.

II. Background

A. Void Search Algorithms

Void Search (VS) algorithms are a family of methods aimed at finding regions of empty – or low density – space for a given set of data points. These empty regions are known as voids, and in astrophysics they are essential for studying galaxy formation and the structure of the cosmic web [15]. For us, voids are essentially patterns of feature space for low density data regions. This contrasts sharply with traditional algorithms which search for patterns of feature space for existing data points. Voids can be of great interest in the case where one of the data classes in a two-class learning problem is hard to characterize.

Formulation of the problem is quite simple: given a space of N dimensions, the mission of a VS algorithm is to find regions with a very low density of points[2]. Figure 1 shows an example of voids for a given set of randomly generated points in a 2-dimensional space. In this particular example, only voids with zero density are shown.

The definition of a void, however, can change depending on the nature of the data. For example in [16], the authors partition the space into a grid of cells of equal size and declare every cell that meets the density criteria as a starting void. Furthermore, they merge voids only if its centers are closer to each other than they are to any existing data point. This definition avoids the creation of tunnels between voids, which in turn helps approximate voids by large spherical or elliptical shapes. This makes sense when working with the structure of the universe, since it has been observed

[2]Definition of low density depends on the particular algorithm.

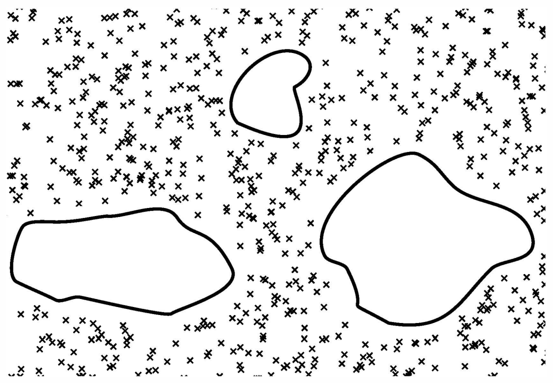


Fig. 1: Three voids in a given set of randomly generated points.

that voids tend to become more spherical over time due to gravitational instabilities causing the collapse of high density regions [26]. Another definition can be found in [17] and [15], where void centers are defined as very low density points, and “walls” between voids are defined as regions surpassing a density threshold relative to its surrounding voids’ centers. This definition tends to fit very well for data that spreads like a web of walls fencing low density regions (similar in structure to a Voronoi tessellation).

Independently of how we build voids, outliers can simply be discovered by checking whether they fall into one of the voids. It is clear that the more stable a void is[3], the more accurate the algorithm will be as time progresses. Our observations indicate that environmental data tends to be very stable under normal conditions. For illustration, consider Figure 2, where we show temperature readings for a period of one month in a fault-free node board. As the figure shows, temperatures stay fairly stable between 25 and 35 degrees Celsius throughout the period. Nevertheless, it is possible to imagine a scenario where voids are adapted over time given major changes in the data.

B. Mira Supercomputer

Mira has a theoretical peak of 10,066.3 TFlop/s, and a Linpack performance of 8,586.6 TFlop/s. Each of the 48 racks has two midplanes. Each midplane is composed of 16 node boards each having 32 compute cards. The compute card is composed of one chip module and 16 GB of DDR3 memory. Finally, the computing chip has a single 18-core PowerPC A2 processor [24]. Out of these 18 cores, 16 are for applications, one is for system software, and one core is left inactive. Each core has four hardware threads. Thus, BlueGene/Q has a total of 1,024 nodes per rack, or 16,384 cores per rack.

The machine is built with a number of hardware sensors placed on multiple components all over the machine. Depending on the sensor, the type of the data collected can be temperature, coolant flow and pressure, fan speed, voltage, or current. Sensor data is gathered and stored in IBM DB2

[3]A stable void is a void whose density value does not change drastically when new data is used to build voids.

TABLE I: An example of environmental information

location	time	inletFlowRate	outletFlowRate	coolantPressure	inletCoolantTemp	ambientHumidity	diffPressure	...
R1A-L	2012-09-01 00:03:03	2680	2681	3949	1779	3144	1582	...
R0F-L	2012-09-01 00:03:03	2417	2414	4049	1786	4718	1369	...
R10-L	2012-09-01 00:38:10	2673	2668	3985	1784	3329	1406	...

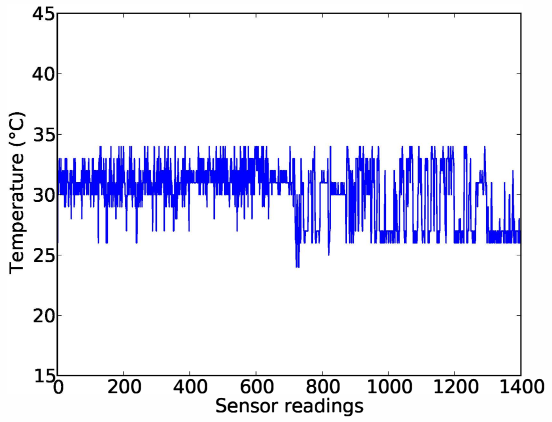


Fig. 2: Temperature readings for a period of one month in a non-faulty node board.

relational database with a given periodicity of about 4 minutes on average (although it can be configured to be anywhere between 1 and 30 minutes).

Table I shows an example of sensor data for the coolant system in Mira. Location is a human readable unique identifier, and all sensor data are strictly numerical. Since the coolant system is installed with at a rack granularity, locations here only represent racks (R10-L is the coolant system for rack number 10). In general, sensors can be located in a variety of components such as service cards, node boards, I/O boards, coolant systems, bulk power modules, and optical modules. The Midplane Monitor and Control System (MMCS) organizes sensor information in different tables based on the component which the sensor applies to. There are tables for the service card, the node board, the I/O board, the coolant monitor (as shown in table I), and so on. More information about location IDs and environmental data in Blue Gene/Q can be found in [41].

Since components in the system already form a natural hierarchy, it is possible to think about sensors as forming a hierarchy too. This is useful when trying to consider which sensors can provide pragmatic information for potential faults in a particular component. For instance, in the case of the Blue Gene/Q, if the power to a rack fails, all 1,024 nodes in that rack are going to go down, inevitably experiencing a failure.

Hardware faults are extracted from the RAS logs, which in Blue Gene/Q are categorized into informational (INFO), warnings (WARN) and fatals (FATAL). Only FATAL events are severe faults that presumably lead to failures [41]. For the purpose of this study, only FATAL events affecting nodes are considered as faults.

III. Detection Algorithm

A. Overview

The first step in the design of our algorithm is to determine an appropriate granularity for data analysis. For example, we considered analyzing the whole machine at once trying to make our algorithm detect faults for the entire system. However, this approach proved infeasible as the number of features became unmanageable. If we think of sensors as the features for learning, the whole machine has hundreds of thousands of features. Another problem with this centralized method is the difficulty of locating faults. Predicting fault location is key in order to make preventive actions (such as migrating processes or doing local checkpoints) effective in extreme scale systems.

Instead, we analyze each hardware component independently using only sensors that are relevant to that component. Since detection is performed per component, it is possible to know the locations of the detected faults, (which can lead to future failures) immediately. Furthermore, our algorithm is node card[4] centric, caring only about faults that can affect node cards directly or indirectly. For example, in this study a link card fault is only relevant if it makes a node (or a group of nodes) unable to communicate with other nodes. In other words, if node faults happens as a result of a link card fault.

Sensors relevant to each node are selected based on the hierarchy of components inside the system. Sensors from the rack's bulk power and coolant environment, the link card connected to the node, the node board where the node is inserted, and the actual sensors in the node card are compiled, making a total of 180 sensors. Although this is much better than hundreds of thousands, a reduction of dimensionality is still needed since void search's runtime is hardly impacted by the number of dimensions (see Section IV-C).

The first pre-processing task needed for our analysis is to define the periodicity of the data (how often we examine the data for fault detection). In our case, periodicity is set to 40 minutes, which ensures that we are getting data from all the sensors (the maximum sensor periodicity is 30 minutes in Blue Gene/Q) in each reading; sensors with a higher frequency get all their values throughout the 40 minutes period averaged.

To reduce dimensionality we use principal component analysis (PCA) [27], a well known and widely used tool in data analysis. PCA transforms the features of a given data

[4]We use node card and node interchangeably.

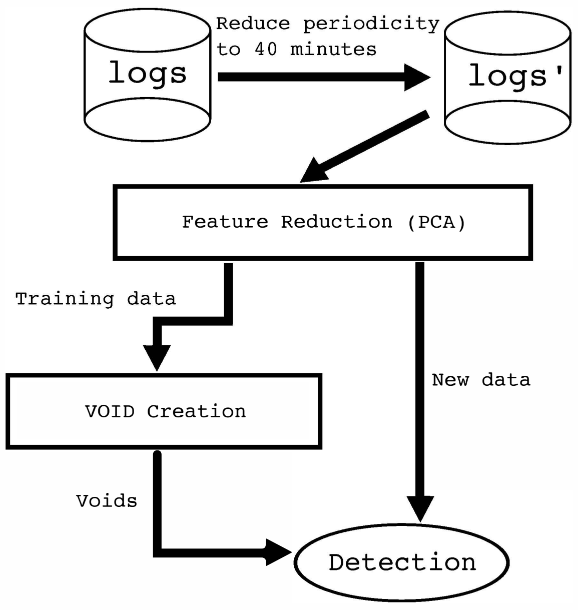


Fig. 3: High level overview of our Void Searching fault detection approach.

set to a new set of uncorrelated features called principal components which are a linear combination of the original ones. PCA quantifies the importance that each component has in describing the variability of the data, allowing us to use fewer dimensions by picking the most relevant components, while still preserving all the important patterns in the data. PCA, however, has some limitations, such as the impossibility to project the patterns found in fewer dimensions (such as voids) back to the original dimensions. Even so, this is really not a problem for us because we are only interested in knowing if a particular node is going to fail; it is not of concern as to why that particular node is going to fail. Root cause analysis can always be performed post-mortem with the untransformed log data if necessary.

A high level overview of our approach is presented in Figure 3. Our approach is composed of three main steps: step 1 is data pre-processing, particularly feature reduction; step 2 is void construction to build voids for each node; step 3 is to detect faults every 40 minutes.

B. Void Construction

Algorithm 1 shows how we create voids for a given training set D in an n dimensional space, where the training data corresponds to points in time with no faults. Here δ represents the longitude of each cell's side in the grid of cells C. Line 2 initializes V, which will be the returned set of voids. In line 3 the limits L of the grid of cells are calculated. The limits are the minimum and maximum values of every dimension for which we have a data point, adjusted so every side of the grid has a natural number of cells. This is equivalent to drawing an enclosing rectangle around a cloud of data points in a two dimensional space.

Algorithm 1 Void creation algorithm

```
1:  procedure CREATEVOIDS($D, n, \delta$)
2:      $V \leftarrow \{\}$
3:      $L \leftarrow$ calculateLimits($D, n$)
4:      $C \leftarrow$ createCells($L, n, \delta$)
5:      for $i \leftarrow 0, |C|$ do
6:          $I \leftarrow False$
7:          for $j \leftarrow 0, |D|$ do
8:              if datumInside($C[i], \delta, n, D[j]$) then
9:                  $I \leftarrow True$
10:                 break
11:             end if
12:         end for
13:         if $I = False$ then
14:             $V \leftarrow V \cup C[i]$
15:         end if
16:     end for
17:     return $V$
18: end procedure
```

Line 4 creates the grid of cells, given L, n and δ. Realize that this operation is not needed, since we can calculate the cell's coordinates given the cell number i, the limits L and the longitude of each cell's side δ, and we can compute the total number of cells very easily too. However, we think it makes the representation more clear if we show the creation of the cells explicitly. To represent a cell in this case it is enough to store its center's coordinates.

From line 5 to 16, the algorithm iterates over all the cells. For every cell $C[i]$, we check if any point in D is inside the cell (lines 7 to 12). If it is, we skip it and go to the next one. If no points are inside the cell we add it to V as a void. It is possible to see that our algorithm is an extreme case of VS where density is zero. Finally, all the voids V are returned in line 17.

The complexity of the algorithm depends on the number of cells (which depends on n and δ) and the number of data points. For simplicity, consider that all sides of the grid have the same size M. The complexity, then, is $\Theta(\lceil M/\delta \rceil^n |D|)$, which is exponential with respect to the number of dimensions n (this is the reason why we reduce dimensionality through PCA). Both the parameters δ and n control a trade-off between runtime and accuracy of the algorithm. A smaller δ means more cells (which also means higher accuracy), but at a larger computational cost. However, choosing a very small δ may produce overfitting (voids end up taking too much of the empty space surrounding the training data) causing too many false positives. Likewise, a larger n produces better results too, but only up to a point. Recall that PCA sorts components by their importance in describing the variability of the data. Hence, every new dimension that we add has an ever decreasing value at an exponentially increasing computational cost. In the end, choosing appropriate values for n and δ depends on the nature of the input data and should be set experimentally.

One thing that is worth noting about Algorithm 1 is its simplicity with regard to parallelization as it is embarrassingly parallel with respect to the outer loop. Although in distributed memory (such as MPI) D still needs to be broadcasted to all processes, it is usually a small amount of data. For our Blue Gene/Q environmental logs, $|D|$ is close to 4.5K points for a node in a period of four months. Supposing 3 dimensions, $|D|$ is around 50KB (or 100KB for 64bits).

Another interesting observation, since our algorithm treats each node as independent, is that it is able to scale linearly with the size of the supercomputer. We will show in Section IV that it is feasible to make every node compute its own voids while having a centralized approach to perform detection using Algorithm 2.

Algorithm 2 Outlier detection algorithm

```
1: procedure DETECT($d, V, n, \delta$)
2:     for $i \leftarrow 0, |V|$ do
3:         if datumInside($V[i], \delta, n, d$) then
4:             return $True$
5:         end if
6:     end for
7:     return $False$
8: end procedure
```

C. Detection

Once the void set V is calculated, we can detect faults using Algorithm 2. Here d represents a given environmental data, which is expected to be previously projected to the new n principal components. If we find that d fits into any of the voids, then *True* is returned indicating that a fault occurs. This signifies that a failure is likely to happen in the next 40 minutes and that proactive measures are needed for this node. The runtime of Algorithm 2 is linear in the number of voids, which in turn is bounded by the number of cells $O(\lceil M/\delta \rceil^n)$. In practice we find that, for $n \leq 3$, the algorithm using Blue Gene/Q environmental data always runs under 4 seconds in any average modern CPU (see Section IV-C).

IV. Experimental Results

We evaluate our detection design by using a four-month RAS and environmental log collected from the Mira supercomputer from September 1st to December 31st of 2012. Since we wanted to have nodes with enough faults to produce results that could help us understand the potentiality of environmental logs with respect to fault detection, only the nodes with four or more faults are used. In total, we evaluate 4325 hardware faults from 362 nodes during a period of 4 months.

We have conducted three sets of experiments: first, we assessed the detection accuracy of our design, second, we evaluated runtime, and finally, we compared VS with other outlier detection algortihms.

All of the evaluations, with the exception of the clustering algorithm (discussed later), are done using 10-fold cross validation, which is a widely used technique to evaluate a particular algorithm with a given set of labeled data (data from which we know the class). It first randomizes the order of the data. Then, it runs the algorithm 10 times, splitting the data into 10 (almost) identical folds, using nine different folds for training and one for testing in every run. When the 10 runs are completed, it calculates the average of ten runs to compute the desired metrics.

A. Metrics

We use three metrics for evaluation: *sensitivity*, *specificity*, and *S-measure*. Sensitivity represents the rate of total faults that were predicted among all faults in the data. This metric is also commonly known as *recall*, and it can be computed as follows:

$$sensitivity = \frac{Tp}{Tp + Fn},$$

where Tp and Fn are the true positives (real faults that were detected) and false negatives (real faults that were missed) respectively. Specificity, on the other hand, represents the rate of total non-faults that were predicted among all non-faults in the data. This metric can also be called *negative recall*, and it is computed this way:

$$specificity = \frac{Tn}{Tn + Fp},$$

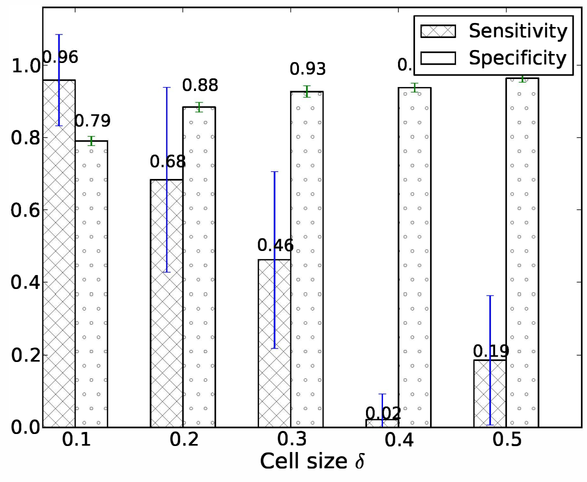
where Tn and Fp are the true negatives (real non-faults that were predicted as such) and false positives (real non-faults that were detected as faults incorrectly) respectively.

In addition, we introduce a new metric called *S-measure* (inspired by the existing F-measure) to compare the results of our VS with that of other algorithms. This metric combines sensitivity and specificity given they have an equal weight in the analysis. It is defined as follows:

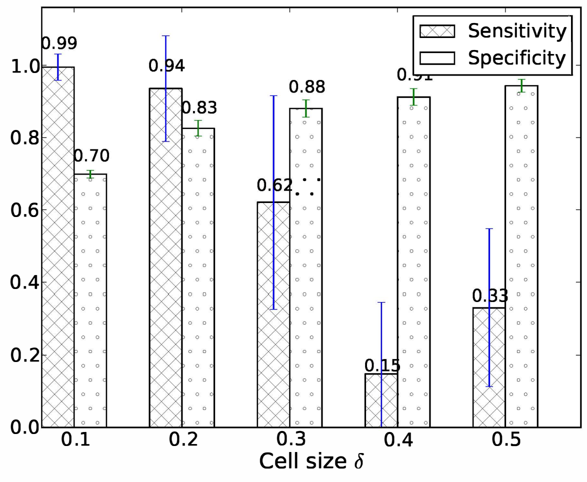
$$S\text{-}measure = 2 \times \frac{Sensitivity \times specificity}{Sensitivity + specificity}$$

B. Detection Accuracy of our VS Algorithm

In the first set of experiments, we assess our VS based algorithm under various parameter settings. The results are presented in Figure 4. The two graphs presented correspond to the same experiments using different numbers of dimensions – 2 dimensions for (a) and 3 dimensions for (b). Both graphs evaluate VS by changing the δ parameter from 0.1 to 0.5. In both cases sensitivity increases as δ decreases, reaching values greater than 0.9 for $\delta <= 0.1$ in (a) and for $\delta <= 0.2$ in (b). However, we also see that specificity tends to decrease as cell sides decrease. This is expected since smaller cells do ultimately produce some overfitting as number of voids increases. This results in more "normal data" falling into voids than it would do with a bigger δ.



(a) $n = 2$



(b) $n = 3$

Fig. 4: Results for our VS based algorithm using 10-fold cross validation over a period of four months. The results shown are averaged across the 361 faultiest nodes. Two plots are presented, representing the cases after feature reduction to two and three dimensions, respectively.

Fortunately, specificity never gets below 0.7 in the case of 3 dimensions and 0.79 in the case of 2 dimensions, which means that, at most, preventative actions are performed 30% of the time they should not have. Although this may seem like a lot at first, these predictions are done for every node separately. Under a hierarchical checkpointing scheme, for example, checkpoints to the local SSD disk – or even to a neighboring node's SSD disk – can be done in seconds. Nevertheless, [14] and [29] prove that, when combining checkpointing and prediction, checkpointing algorithms benefit the most when the salient characteristic of a predictor is its sensitivity (recall). Missing a failure is much more expensive than having some false alarms. In other words, *it is better to be safe than sorry* [14]!

C. Runtime of Our VS Algorithm

We evaluate the run time of our C implementation of VS for different values of n and δ in the learning and prediction phases of the algorithm. Figure 5 presents execution times for the learning (void construction) phase for one node, running on a single thread on a common CPU (Intel i7 at 3.20GHz). Note that time is presented in a logarithmic scale.

Figure 5 makes clear the huge impact that dimensionality has in the performance of VS. In our case, $n = 3$ is almost two orders of magnitude slower than $n = 2$. If we take the prediction results from Figure 4, we can see that results for $(\delta = 0.1, n = 2)$ are similar to those for $(\delta = 0.1, n = 3)$ and $(\delta = 0.2, n = 3)$, even though the former takes around 20 seconds to build the voids while the latter take 40 and 5 minutes respectively. Nevertheless, and considering that learning only takes place every few months, even minutes can not be considered a huge impact. It is possible to think of a scenario where every node computes its voids independently in minutes and send them over to a central repository for future predictions.

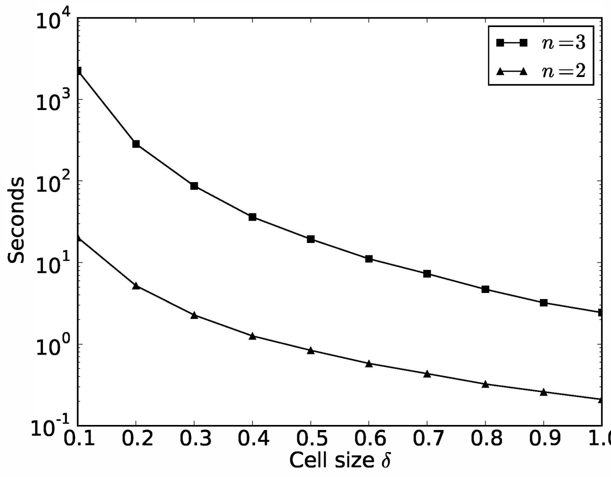


Fig. 5: Run time (in seconds) for the sequential VS algorithm under different values of n and δ on a 3.20Ghz CPU.

It is worth noting that the detection process is fast as mentioned in Section III-C, less than 4 seconds for this 4-month log using the parameter combination $(\delta = 0.1, n = 3)$. Considering that nodes' predictions are independent from one another, using a good high throughput computing (HTC) engine in a modest "data analysis cluster" running 1024 processes [5], it would be possible to detect faults for all the nodes in a 48K node machine (such as Mira) in under 3 minutes. Again, this is considering that we use the most expensive parameter

[5] 64 nodes in the case of nodes with 16 sockets.

combination. For $(\delta = 0.1, n = 2)$, prediction time per node is always below 100 ms.

D. Comparison with Other Algorithms

We conducted another set of experiments in order to evaluate how other popular outlier detection algorithms could perform using environmental data for fault detection. More specifically, we evaluate two classes of algorithms: classifier based and cluster based. For classifier based, we run Naive Bayes (NB), Support Vector Machines (SVMs) and Artificial Neural Networks (ANNs). For a representation of clustering methods, we use the simple K-means algorithm.

NB is a very simple probabilistic classifier based on the Bayes' theorem. It assumes that all features are completely independent from one another given the class variable Y. Moreover, NB calculates the posterior probabilities $P(X_j|Y = y_i)$ of features X_j given the training data for each class y_i. New data can be classified computing the probability for each class and picking the biggest among them.

SVMs are classification algorithms that aim to find linear decision boundaries which maximize the distance to the closest points from each class. This linear boundary can be extended to a non-linear one with the so called "kernel trick", which uses kernels to compare data points in a different feature space where linear boundaries correspond to non-linear ones in the original feature space [35]. In outlier detection problems, SVMs are used with just one class, in such a way that a boundary is constructed for "normal data" only. When a new point is evaluated, it is considered an outlier if it falls outside of this created boundary. Different kernels can be used to build this boundary. The kernel we use in these experiments is the radial basis function (RBF) kernel, which allows SVMs to learn complex regions of feature space [34]. We set the kernel parameters μ and γ experimentally, trying different values and picking the ones that produced the best prediction.

ANNs are graphical models inspired by the functionality of the brain, where nodes are connected to other nodes analogous to how neurons are connected in the brain. Nodes in ANNs combine values from multiple inputs and produce an output by a defined function (such as sigmoid or a step function). This output can be transferred to other nodes in the next layer, or to the output of the network [35]. An algorithm learns a neural network by adjusting the weights of the inputs to the different neurons in the graph so as to best fit the training data. In this paper, we use a multi-layer preceptron (MLP) ANN with one hidden layer.

The last algorithm we compare is simple K-means. In simple K-means, cluster's centers (the means) are first chosen at random. Each point, then, belongs to the cluster whose mean is the closest to the point. After this step, the means of every cluster are re-calculated and points are re-assigned to the cluster with the closest mean. This procedure is repeated iteratively until the clusters no longer change [39]. We use K-means with two clusters $(k = 2)$, and evaluate it with the *classes to clusters* evaluation. Under this evaluation, the algorithm first creates clusters with unlabeled data, and then each cluster is labeled with one of the classes based on the greater number of instances of the class within each cluster. Once clusters are labeled, we can compute metrics by counting how many instances of each class fall in each different cluster.

The results for these experiments are shown in Table II. For comparison purposes, VS is also shown for two different combinations of parameters n and δ. These combinations correspond to the best achieved accuracy from each value of n. Our results indicate that VS outperforms all of the other algorithms by having the highest S-measure values. Although we find $(\delta = 0.2, n = 3)$ to be the winner given its very good specificity, in the end it all boils down to how much overfitting a system is willing to tolerate in order to have a little bit of extra sensitivity, or a substantial reduction in execution time.

One interesting discovery is that both SVM and K-means are able to effectively discover all faults in the data $(sensitivity = 1)$. In the case of K-means, this is an indication that faults do cluster together in the feature space (since they all fall in the same cluster), and that a pattern can actually be learned. Both cases, however, produce too much overfitting. SVM leaves too many non-faults outside the boundary while, K-means, which tends to create clusters of approximately the same size, produces too many false positives. This is due to the fact that the number of instances of one class is much larger than the number of instances of the other. Nevertheless, both approaches can be used under a scenario where false positives do not produce excess overhead.

Finally, we can see that both NB and ANN are not appropriate choices. In the case of NB, sensitivity is too low to be considered a viable option. As for ANN, the algorithm is unable to detect any hardware faults at all (it classifies all data as non-fault).

V. Discussion

We have already seen in Section IV-C that a VS based algorithm can be quite expensive computationally if the parameters are not chosen carefully. The good news – as we already know – is that we do not need to always use the most expensive combination of parameters in order to get the best results. In fact, values of δ (i.e., cell size) too small may produce overfitting, which affects algorithm accuracy significantly; or values of n (i.e., grid dimensionality) too large may affect performance substantially while adding little in return.

However, one may still make the case that 40 minutes per node for void creation, or 4 seconds per node for fault detection, is still very high if we consider that large-scale systems have tens of thousands of nodes and are expected to have hundreds of thousands in the next generations of machines. Although we have already proposed scenarios where such times may be acceptable (i.e., learning is done every few months and detection can be efficiently parallelized), our implementation is by no means optimized as much as it could be. For example, data can be indexed efficiently using point coordinates. When we are checking if a cell should be a void, we can query only for points whose coordinates are inside the boundaries of the cell (instead of going through all the dataset).

TABLE II: Comparative study of our VS based algorithm with other detection algorithms

	NB	SVM-RBF ($\mu = 0.1, \gamma = 10^{-3}$)	ANN	K-MEANS ($k = 2$)	VS ($\delta = 0.2, n = 3$)	VS ($\delta = 0.1, n = 2$)
Sensitivity	0.493	1	0	1	0.935	0.959
Specificity	0.857	0.458	1	0.516	0.825	0.791
S-measure	0.625	0.628	0	0.681	0.877	0.867

Another possible optimization is to run both procedures on a hardware accelerator, such as a GPU, parallelizing the outer loop (i.e., the loop i). Changing the definition of what a void is can also be explored. Joining and approximating voids by large spherical or elliptical shapes could make detection a simple and fast calculation.

Another goal of this work, apart from exploring VS algorithms, was to show the great potential of environmental logs, as opposed to RAS logs, for fault detection. We find two clear advantages: first, by using environmental logs, it is easier to create a decentralized design where localizing failures is straightforward. We do not need to learn complex relationships between different events' types, or combine algorithms with topology information, in order to isolate a particular fault. The second reason is the impressive results of three out of the five outlier detection algorithms analyzed with respect to sensitivity (recall). The state-of-the-art in RAS based failure prediction provides a sensitivity of about 0.5 [12], while we can achieve well above 0.9 using environmental logs.

VI. Related Work

Numerous works have been devoted to the understanding of failures in HPC systems based on log information. For example, Sahoo et al. [20] explored three classes of algorithms in a 350-node IBM cluster: time-series, rule-based, and Bayesian networks; Zheng et al. [21] used a genetic algorithm (GA) to learn rules that can predict failure times as well as failure locations; in [22], rule-based, SVMs and Nearest Neighbor based classifiers are explored for failure prediction on a Blue Gene/L system; Lan et al. [10] proposed a dynamic meta-learning prediction engine to combine the benefit of multiple algorithms to boost accuracy; and Gainaru et al. combines signal-processing concepts and data-mining techniques to predict fault occurrences [12]. Although similar in spirit, our work is different from these studies in that we use environmental data while they focus on RAS logs exclusively. Furthermore, we propose a new algorithm for fault detection (Void Search) that, to the best of our knowledge, has not yet been explored for fault detection in the literature.

In our previous work [43], we presented online data reduction techniques (instance and feature selection) to remove redundant and noisy data in environmental logs. Under that approach, all environmental data available from the whole system is gathered in a centralized buffer and analyzed together. The problem with this approach, however, is that it is not as scalable as the number of hardware sensors (and hence features in the data) grows linearly with the size of the machine. For that reason, in this paper we follow a decentralized path, where environmental logs get analyzed in a component per component basis.

A vast amount of literature has also been devoted to outlier detection algorithms. For instance, NB, and Bayesian networks in general, have been used for outlier detection problems such as network intrusion detection [30], [31], [32] and disease outbreaks detection [33]. SVMs have been proposed for anomaly detection in, among others areas, audio signal data [36] and system call intrusion detection [37], [38]. Multilayer perceptrons has also been proposed for network intrusion detection [40]. Finally, K-means can be found in outlier detection problems like mammogram classification [44] as well as network anomaly detection [45].

VII. Summary

In this paper we have presented a new approach for fault detection based on the VS algorithm. We evaluated it using 4 months of environmental logs from the Mira supercomputer, a 48-rack IBM Blue Gene/Q at Argonne National Laboratory [25]. Our results show that VS can detect almost all faults (i.e., sensitivity close to 1) with a low false positive rate (i.e., specificity values above 0.7). Furthermore, we compared our design with other popular outlier detection algorithms, and showed that VS outperforms all of them by having the best combination of the evaluation metrics sensitivity and specificity. We also evaluated the runtime of VS to show the impact that different values of parameters can have on it. In the case of void construction, which is run every few months, the best combination of parameters only takes 5 minutes for each node. Fault detection never goes above 4 seconds for any combination of parameters.

Looking forward, we plan to continue exploring VS potentiality by analyzing more logs from other new generation petascale systems. Our future agenda also includes trying different feature reduction approaches such as ICA [11], specializing our algorithm to be able to accept voids of density different from zero, and exploring hardware accelerators (such as GPUs) to speed up runtimes by taking advantage of its easy parallelization.

Acknowledgment

This work is supported in part by US National Science Foundation grant CNS-1320125 and Department of Energy contract DE-AC02-06CH11357.

References

[1] A. Avizienis, J. Laprie, B. Randell, and C. Landwehr, "Basic Concepts and Taxonomy of Dependable and Secure Computing," *IEEE Trans. Dependable and Secure Computing,* vol. 1, no. 1, pp. 11-33, Jan. 2004.

[2] ANL report, "Addressing Failures in Exascale Computing," March 2013.

[3] "MPI: A Message-Passing Interface Standard Version 3.0," September 2012. [Online]. Available: http://www.mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf

[4] B. Schroeder and G. Gibson, "Understanding Failure in Petascale Computers," *Journal of Physics Conference Series: SciDAC,* vol. 78, p. 012022, June 2007.

[5] L. Bautista-Gomez, D. Komatitsch, N. Maruyama, S. Tsuboi, F. Capello, and S. Matsuoka, "FTI: high performance fault tolarance interface for hybrid systems," *High Performance Computing, Networking, Storage and Analysis (SC), 2011 International Conference for,* 2011, pages 1-12.

[6] A. Moody, G. Bronevetsky, K. Mohror, and B. R. De Supinski, "Design, modeling, and evaluation of a scalable multi-level checkpointing system," *High Performance Computing, Networking, Storage and Analysis (SC), 2010 International Conference for,* 2010, pages 1-11.

[7] X. Dong, N. Muralimanohar, N. Jouppi, R. Kaufmann, Y. Xie, "Leveraging 3D PCRAM Technologies to Reduce Checkpoint Overhead for Future Exascale Systems," *Supercomputing,* 2009.

[8] D. Ibtesham, D. Arnold, P. G. Bridges, K. B. Ferreira, and R. Brightwell, "On the viability of compression for reducing the overheads of checkpoint/restart-based fault tolerance," *2012 41st International Conference on Parallel Processing,* vol. 0, pp. 148-157, 2012.

[9] T. Z. Islam, K. Mohror, S. Bagchi, A. Moody, B. De Supinski, and R. Eigenmann, "MCRENGINE: A Scalable Checkpointing System Using Data-Aware Aggregation and Compression," *High Performance Computing, Networking, Storage and Analysis (SC), 2012 International Conference for,* 2012.

[10] Z. Lan, J. Gu, Z. Zheng, R. Thakur, and S. Coghlan, "A study of Dynamic Meta-Learning for Failure Prediction in Large-Scale Systems," *Journal of Parallel and Distributed Computing (JPDC),* 2010.

[11] Z. Lan, Z. Zheng, and Y. Li, "Toward Automated Anomaly Identification in Large-Scale Systems," *IEEE Transactions on Parallel and Distributed Systems,* vol. 21, no. 2, 2010.

[12] A. Gainaru, F. Cappello, M. Snir, and W. Kramer, "Fault prediction under the microscope: A closer look into HPC systems," *Supercomputing (SC12),* 2012.

[13] G. Aupy, Y. Robert, F. Vivien, and D. Zaidouni, "Impact of fault prediction on checkpointing strategies," *Technical Report* RR-8023, v1, INRIA, July 2012.

[14] G. Aupy, Y. Robert, F. Vivien, and D. Zaidouni, "Checkpointing algorithms and fault prediction," *Technical Report* RR-8237, INRIA, February 2013.

[15] M. C. Neyrinck, "ZOBOV: a parameter-free void-finding algorithm," *Monthly Notices of the Royal Astronomical Society,* vol. 386, 2007, pp. 2101-2109.

[16] J. Aikio and P. Maehoenen, "A Simple Void-searching Algorithm," *Astrophys. J.,* vol. 497, p. 534, Apr. 1998.

[17] E. Platen, R. Van De Weygaert, and B. J. T. Jones, "A cosmic watershed: the wvf void detection technique," *Monthly Notices of the Royal Astronomical Society,* vol. 380, no. 2, 2007, pp. 551-570.

[18] F. Hoyle and M. S. Vogeley, "Voids in the 2dF Galaxy Redshift Survey," *Astrophys. J.,* vol. 607. pp. 751-764, 2004.

[19] C. Engelmann, G. R. Vallée, T. Naughton, and S. L. Scott, "Proactive Fault Tolerance Using Preemptive Migration," *Parallel, Distributed and Network-based Processing, 2009 17th Euromicro International Conference on,* 2009.

[20] R. K. Sahoo, A. J. Oliner, I. Rish, M. Gupta, J. E. Moreira, S. Ma, "Critical Event Prediction for Proactive Management in Large-scale Computer Clusters," *International conference on Knowledge discovery and data mining,* pp 426-435, 2003.

[21] Z. Zheng, Z. Lan, R. Gupta, S. Coghlan, and P. Beckman, "A Practical Failure Prediction with Location and Lead Time for Blue Gene/P," *Proc. of the 1st Workshop on Fault-Tolerance for HPC at Extreme Scale (FTXS), in conjunction with DSN'10,* 2010.

[22] Y. Liang, Y. Zhang, H. Xiong, R. Shaoo, "Failure prediction in IBM BlueGene/L event logs," *7th International Conference on Data Mining,* 2007.

[23] G. Lakner and B. Knudson, "IBM System Blue Gene Solution: Blue Gene/Q System Administrator," May 2013. [Online]. Available: http://www.redbooks.ibm.com/redbooks/pdfs/sg247869.pdf

[24] R. Haring, M. Ohmacht, T. Fox, M. Gschwind, D. Satterfield, K. Sugavanam, P. Coteus, P. Heidelberger, M. Blumrich, R. Wisniewski, A. Gara, G.-T. Chiu, P. Boyle, N. Chist, and C. Kim, "The IBM Blue Gene/Q compute chip," *Micro, IEEE,* vol. 32, no. 2, pp. 48-60, march-april 2012.

[25] Mira information at Argonne Leadership Computing Facility. [Online]. Available: http://www.alcf.anl.gov/mira

[26] V. Icke and R. van de Weygaert, "Fragmenting the universe," *Astronomy and Astrophysics,* vol. 607, pp. 751-764, 2004.

[27] Hotelling, H., "Analysis of a complex of statistical variables into principal components," *Journal of Educational Psychology,* 24, 417-441, and 498-520, 1933.

[28] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, I. H. Witten, "The WEKA Data Mining Software: An Update," *SIGKDD Explorations,* Volume 11, Issue 1, 2009.

[29] M. S. Bouguerra, A. Gainaru, F. Cappello, L. Bautista Gomez, N. Maruyama and S. Matsuoka, "Improving the computing efficiency of HPC systems using a combination of proactive and preventive checkpointing," *Proceedings of IEEE IPDPS,* 2013.

[30] D. Barbara, J. Couto, S. Jajodia, N. Wu, "Detecting novel network intrusions using Bayes estimators," *Proceedings of the 1st SIAM International Conference on Data Mining,* 2001.

[31] A. A. Sebyala, T. Olukemi, L. Sacks, "Active platform security through intrusion detection using naive Bayesian network for anomaly detection," *Proceedings of the London Communications Symposium,* 2002.

[32] C. Siaterlis, B. Maglaris, "Towards multi-sensor data fusion for dos detection," *Proceesings of the ACM Symposium on Applied Computing. ACM Press,* 439-446, 2004.

[33] W.-K. Wong, A. Moore, G. Cooper, and M. Wagner, "Bayesian network anomaly pattern detection for disease outbreaks," *Proceedings of the 20th International Conference on Machine Learning,* AAAI Press, 808-815, 2003.

[34] V. Chandola, A. Banerjee, and V. Kumar, "Anomaly Detection: A Survery," *ACM Computing Surverys,* Vol. 41, No. 3, Article 15, July 2009.

[35] T. Hastie, R. Tibshirani, and J. Friedman, "The Elements of Statistical Learning," Springer, Second Edition.

[36] M. Davy, S. Godsill, "Detection of abrupt spectral changes using support vector machines, an application to audio signal segmentation," *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing,* 2002.

[37] E. Eskin, A. Arnold, M. Prerau, L. Portnoy, and S. Stolfo, "A geometric framework for unsupervised anomaly detection," *Proceedings of the Conference on Applications of Data Mining in Computer Security,* Kluwer Academics, 78-100.:w

[38] W. Hu, Y. Liao, and V. R. Vemuri, "Robust anomaly detection using support vector machines," *Proceedings of the International Conference on Machine Learning,* 282-289, 2003.

[39] J. A. Hartigan, M. A. Wong, "Algorithm AS 136: A K-Means Clustering Algorithm," *Journal of the Royal Statistical Society: Series C (Applied Statistics),* vol. 28, p100, 1979.

[40] A. K. Ghosh, A. Schwartzbard, and M. Schatz, "Learning program behavior profiles for intrusion detection," *Proceedings of the 1st USENIX Workshop on Intrusion Detection and Network Monitoring,* 51-62, 1999.

[41] G. Lakner and B. Knudson, "IBM System Blue Gene Solution: Blue Gene/Q System Administrator," May 2013. [Online]. Available: http://www.redbooks.ibm.com/redbooks/pdfs/sg247869.pdf

[42] Stampede Supercomputer, [Online]. Available: https://www.tacc.utexas.edu/stampede/.

[43] L. Yu, Z. Zheng, Z. Lan, T. Jones, J. M. Brandt, and A. C. Gentile, "Filtering Log Data: Finding the Needles in the Haystack," *Proc. of DSN'12,* 2012.

[44] K. Thangavel, A. K. Mohideen, "Semi-Supervised K-Means Clustering for Ourlier Detection in Mammogram Classification," *Trendz in information Sciences & Computing (TISC),* 2010.

[45] A. M. Riad, I. Elhenawy, A. Hassan, and N. Awadallah, "Visualize Network Anomaly Detection by Using K-means Clustering Algorithm," *International Journal of Computer Networks & Communications (IJCNC),* vol. 5, no. 5, 2013.