Online Characterization of Buggy Applications Running on the Cloud

Arnamoy Bhattacharyya*, Harsh Singh*, Seyed Ali Jokar Jandaghi* and Cristiana Amza*
*Department of Electrical and Computer Engineering
University of Toronto, Toronto,Canada

***Abstract*—As Cloud platforms are becoming more popular, efficient resource management in these Cloud platforms helps the Cloud provider to deliver better quality of service to its customers. In this paper, we present an online characterization method that can identify potentially *failing* jobs in a Cloud platform by analyzing the jobs' resource usage profile as the job runs. We show that, by tracking the online resource consumption, we can develop a model through which we can predict whether or not a job will have an abnormal termination. We further show, using both real world and synthetic data, that our online tool can raise alarms as early as within the first 1/8th of the potentially failing job's lifetime, with a false negative rate as low as 4%. These alarms can become useful in implementing either one of the following resource-conserving Cloud management techniques: alerting clients early, de-prioritizing jobs that are likely to fail or assigning them less performant resources, deploying or up-regulating diagnostic tools for potentially faulty jobs.**

I. Introduction

Resource usage monitoring of large server farms and High Performance Computing platforms (HPC) is required to maintain the cost effectiveness of such large infrastructure investments by guaranteeing SLAs, high availability, reliability and shorter downtime of services to end users.

Detecting "faulty" programs in a Cloud environment opens up the possibility of more efficient resource management by the Cloud provider. An early identification of a failure can serve for improving resource usage or job diagnosis in the following ways: i) alerting the user about the potential fate of their job, so that the user can help save resources by killing the job themselves, ii) de-prioritizing jobs that are likely to fail or migrating them to a different platform or assigning them less performant resources to run on, and iii) deploying or upregulating diagnostic tools for potentially faulty jobs.

Either or all of these methods are expected to achieve two goals. First, we reduce the number of times a faulty jobs is resubmitted to the Cloud, by providing the user with more comprehensive diagnostic data and second, we selectively provide better quality of service to users with non-faulty jobs.

In this paper we present novel techniques to build online classifiers to predict the fate of a running task as *successful* termination or *buggy* based on run-time metrics. Our technique shows that there is a correlation between the on-line variation in resource usage over the lifetime of a job, and the likelihood of abnormal termination for the job.

We quantify this normal vs. buggy behaviour of a program as a multidimensional signal, where each dimension represents usage time-series of a particular resource. We present and evaluate effectiveness of the trained classifiers using both real-world dataset obtained from Google cluster data [16] and our own generated data using a well-known benchmark suite containing faulty applications – BugBench [18].

We believe it would be useful for the cloud system administrator to monitor and learn over time to determine whether some users may submit a larger fraction of faulty jobs than others.

In summary, in this paper, we answer the following important research questions:

- What are the correlations between the on-line resource consumption monitoring data of a job and the job's fate while running on a Cloud, if any?
- Is it possible to predict a job's fate *early* in its life cycle using its resource consumption statistics?

II. Related Work

Many researchers have studied the statistical properties of the interarrival times of failures (i.e. the elapsed times between failures), to better understand the temporal characteristics of failures [1], [2], [3], [4], [5]. A series of studies have looked into using Reliability, Availability, and Serviceability (RAS) logs that are collected at HPC systems to improve failure prediction techniques [6], [3], [7], [8]. Some of the work that studied failure prediction in HPC systems used event-driven approaches [3], [7]. Alternatively, a period-based approach was studied [9] where different classifiers are periodically explored and evaluated using BlueGene/L RAS logs. A recent work also looks into online failure prediction by parsing logs [24]. Our approach is very lightweight as compared to heavy log analysis and therefore more proactive in an online setting for failure detection.

Some recent studies focused on utilizing failure predictors in improving checkpointing strategies in HPC systems [6], [10]. In a more recent study, Gianaru et al. [6] introduced a hybrid approach for predicting failures in HPC systems that is based on both signal-processing concepts and data-mining techniques. Recent studies that analyzed field data collected from HPC production systems indicated that failures are highly unlikely to be independent, and that both temporal and spatial correlations exist between failures [11], [1], [2], [12], [5], [13].

Another, more recent study from Google by Ford et al. [11] characterizes the availability of data in Google's main storage infrastructure observed over the course of a year. Several studies briefly discussed the effect of environmental issues as

978-3-901882-85-2

part of a more general analysis of HPC failure behaviour [14], [5], [15].

Researchers have come up with online prediction schemes for Google dataset [21], [22], [23]. Our work differs from them in two ways: (1) None of the previous research has presented a comprehensive study using both real world but unknown dataset and a dataset whose behaviour is well understood. (2) Our approach is very lightweight as compared to heavy log analysis or offline modeling.

III. Motivation

Our techniques are based on the fact that there are sections of code in the application that put forth clues in the resource consumption that can help identify the application's fate in case of a faulty run. In this section we present code sections from real world buggy applications and show the corresponding deviation in the resource consumption from a normal run.

Listing 1: Code Snippet from gnu *man* [18].

```
1. static char **
2. get_section_list (void) {
3. int i;
4. char *p;
5. char *end;
6. static char *tmp_section_list[100];

 //some code

7. i = 0;
8. for (p = colon_sep_section_list; ; p = end+1) {
9.     if ((end = strchr (p, ':')) != NULL)
10.     *end = '\0';

11.     tmp_section_list[i++] = my_strdup (p);

12.     if (end == NULL || i+1 == sizeof(
    tmp_section_list))
13.       break;
       }
      //more code
      }

14. static void
15. split (char *string, void (*fn)(char *, int),
    int perrs) {
16. char *p, *q, *r;

17. if (string) {
18.     p = my_strdup(string);
19.     for (q = p; ; ) {
20.       r = index(q, ':');
21.       if (r) {
22.      *r = 0;
23.          fn (q, perrs);
24.          q = r+1;
25.       } else {
26.          fn (q, perrs);
27.          break;
           }
28.     }
29.    free (p);
      }
     }
```

Listing 1 shows buggy code from the GNU *man* application, which is part of the BugBench [18] suite. The get_section_list function appears in lines 973–981 in the file man.c. This function creates a list of sections based on the argument given to the *man* application, where the sections are separated by ':' given in the argument. In function *get_section_list*, the *for* loop in line 8–13 has a wrong exit condition in line 12. The *sizeof(tmp_section_list)* should be *sizeof(tmp_section_list)sizeof(char *)*, otherwise there will be an overflow of the static array *tmp_section_list*, causing the application to crash. The bug will be triggered when the application is called with an argument that has more than 100 ':' in it's name. But fewer number of :'s will not cause the application to fail.

In the *man* application, just before the execution of get_section_list, there is a call to a function called *init_manpath* that performs the initialization work of splitting the argument based on the ':'s present in it by making a call to the function *split*. Listing 1 also shows the relevant part of the *split* function. The for loop at lines 19–28 will cause a spike in the CPU utilization and also memory consumption for a buggy input with hundreds of ':'s in it, therefore giving a hint to a bug detection tool about predicting the failure outcome.

IV. Prediction of Application's Fate

Given the motivation, we develop our application behaviour characterization technique based on the resource consumption statistics collected periodically as the application runs. In this paper we focus mainly on memory related failures and therefore we use memory related resources for consideration.

In the next sections, we describe step-by-step the details of our online characterization methodology.

A. Step 1: Identifying Resources to Profile

Google provides a real-world dataset collected in their datacenter clusters so that researchers can play with it and gain important insights [16]. We use the resource consumption data provided by Google as a standard on what metrics a typical cloud provider can gather for jobs running on their platform. A job in the Google cluster may have three different fates – (1) Successful, (2) Killed or (3) Failed. We explicitly take a look at successful vs killed jobs. A job in the cluster may be either killed by the user of the job or by an inspector in the Google cluster if the job is consuming an unusually higher amount of resources than what is allocated to it. Previous work has looked into the behaviour of *failed* jobs and found out that there is correlation between the failed jobs and their I/O activity [17].

We take a look at explicitly *killed* jobs because we found that many jobs are killed due to having memory issues, probably due to a memory leak in the code or an underestimation by the user of the required memory resource by the task.

The memory and CPU related resource consumption features provided by Google are the following: Canonical Memory Usage (CMU), Cycles Per Instruction (CPI), CPU Utilization (CPU Usage), Memory Accesses Per Instructions (MAI), Maximum memory usage, Total Page Cache: Total Linux page cache (file-backed memory), Unmapped Page Cache (UPC).

For any machine learning algorithms to work, it is useful to filter out redundant features. Not only does it make the

algorithm run faster but also saves the sampling overhead of redundant features. In the next step we identify the redundant features from the above list.

B. Step 2: Filtering out Redundant Statistics

To identify the most significant features that determine the fate of a job, we extract and plot the trace of different resource consumptions related to memory for 100 (randomly selected) *killed* and 100 (randomly selected) *successful* jobs over their lifetime. We find that there exists a high correlation between the following feature pairs:

- MAI and CPI
- Total Page Cache and Unmapped Page cache
- Maximum memory usage and Canonical memory usage

Therefore we can use either of the features in each feature pair for building our machine learning model and discard the rest. We observe the exact same behaviour for the successful jobs as well.

Therefore, we use the following four resource consumption statistics for building our machine learning model and classification: (1) CPI, (2) CPU Utilization, (3) Memory Usage, (4) Total Page Cache.

C. Step 3: Feature Set for Prediction

After discarding the redundant features, we have to find a useful feature set that can be used in online characterization of a job. We use signal processing techniques to extract the feature set from the sampled resource utilization data. The behaviour $\mathbb{B}$ of a job in our case is represented by a n-dimensional signal, where each dimension is representative of a particular resource usage (e.g. CPU or Memory consumption) over the job's lifetime.

$$\mathbb{B} \equiv \{s_1, s_2, \ldots, s_n\} \tag{1}$$

Here n is the number of resources sampled. In our case $n = 4$ as we are using four features as described in the previous section.

Next, we need a way to represent a dimension s_i so that it can be fed to a machine learning classifier. Typically we want a dimension of a signal s_i to be equivalent of a set of *aggregate* metrics f_j^i.

$$s_i \equiv \left\{f_1^i, f_2^i, \ldots, f_n^i\right\} \tag{2}$$

Our job is to identify pairwise matching m between same dimensions s_i and s_j of two signals.

$$m = match\left(s_i, s_j\right) \tag{3}$$

$$m = match\left(\left\langle f_1^i, f_2^i, \ldots, f_n^i\right\rangle, \left\langle f_1^j, f_2^j, \ldots, f_n^j\right\rangle\right) \tag{4}$$

Once we have the aggregate metrics for all the n dimensions, we have a complete representation of the signal. We use the following statistical methods to compute aggregate metrics f_k^i from each resource consumption dimension s_i.

1) Simple Statistics: The *minimum*, *maximum*, *average* and *standard deviation* of the sampled values of the dimension over the job's lifetime.

2) Higher Order Statistics:

- **Skewness** The measure of the asymmetry of the distribution of the signal dimension about its mean. Skewness indicates the symmetry of the probability density function (PDF) of the amplitude of a time series. A time series with an equal number of large and small amplitude values has a skewness of zero.
- **Kurtosis** The measure of the "tailedness" of the signal dimension. Kurtosis measures the peakedness of the PDF of a time series. A kurtosis value close to three indicates a Gaussian-like peakedness.
- **ARIMA Model features** The Autoregressive integrated moving average (ARIMA) model is used to make forecasts from time series data. ARIMA models are generally denoted ARIMA(p, d, q) where parameters p, d, and q are non-negative integers, p is the order of the Autoregressive model, d is the degree of differencing, and q is the order of the Moving-average model.
  We calculate the values p, d and q of the ARIMA model for each signal to extract the behaviour of time series data and use them as features in our prediction.

D. Step 4: A Machine Learning Classifier

We use a binary classifier for predicting the fate of an application. We use supervised learning to train a Support Vector Machine (SVM) [19]. For training, our predictor takes the following tuple for each application under consideration.

$$\iota_i = \langle features_i, fate_i \rangle \tag{5}$$

Where the feature set $feature_i$ can be represented using the following equation.

$$features_i = \{agg_1, agg_2, \ldots, agg_n\} \tag{6}$$

Here each agg_j represents the various statistics calculated for each dimension s_j of that signal and is given by the following equation.

$$agg_j = \langle min_j, max_j, avg_j, std_j, skew_j, kurt_j, p_j, d_j, q_j \rangle \tag{7}$$

Once we have trained the model using the $feature_i$ and the observed fate $fate_i$, we calculate the same $feature_{test}$ for a test case and input it to our prediction model. The prediction model then uses the representation statistics for a given signal to make its prediction $fate_{pred}$.

E. Step 5: Predicting Early

We use the state machine shown in Figure 1 for evaluating the confidence of our online prediction scheme. The prediction p of a task can be in any state from the set of stated S.

$$S = \{initial, weak, stronger, strongest\} \tag{8}$$

We select a window size w of sample datapoints and extract the aggregate features for those datapoints. Then the machine learning model makes a prediction for the window and stores

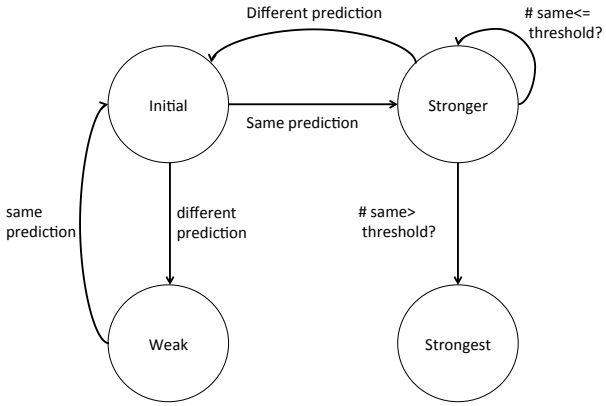

Fig. 1: State machine for online prediction of a task's fate.

the prediction at its initial state. The machine learning model keeps making predictions for subsequent windows and compares the prediction with the last prediction. A match between the current and the last prediction improves the confidence level of the prediction; therefore we move the prediction state to a *stronger* state. Once the state of the prediction reaches the *strongest* state (after making a threshold number of successful predictions), we make our final prediction about the fate of the job.

V. Experimental Evaluation

In this section, we describe the results of predicting a job's fate using a classifier model built using our method. We define our problem as a binary classification problem as we have two states for a job to predict – *buggy* or *successful*. We present the results for both the Google Dataset and BugBench. We run the BugBench application in our private machine that has an Intel(R) Xeon(R) CPU E5-2650 at 2.00GHz processor with 32GB of main memory. For creating buggy and successful runs of BugBench, we use input data provided by the benchmark. We take an average of 5 runs for getting each resource consumption time series. Also we set a sampling rate of 10K instructions using the PAPI [20] performance monitoring tool.

A. Predicting From Whole Lifetime of Jobs

In this section, we present the prediction results using the resource consumptions for the whole lifetime of the job. We gather statistics until completion of a job (in case of a buggy job, until crash) and calculate the aggregate statistics $features_i$.

For building the classification models, we use a python implementation of a binary SVM classifier. We test with different kernel functions for the SVM classifier and have found the *rbf* kernel to be working best. We also use cross-validation and grid search to find the optimal parameter values to be used with the rbf kernel.

First we train the SVM classifier with features from 1000 jobs. The training set contains aggregate metrics of resource consumption for the 1000 jobs (a mixture of *killed* and *successful* jobs) along their whole lifetime. After training we perform predictions on 100 "test" jobs that were not seen during the training phase. The features used for the tested jobs are the same aggregate metrics over their whole lifetime. We notice that the SVM classifier is able to predict 100% of the killed jobs (no false positives). But for the successful jobs, the prediction success rate is 88%. Closer investigation of the data set reveals that for most of the *killed* jobs, the lifetime is long, giving rise to 8000 sampling points in the data stream of resource usage. Therefore, the aggregate metrics collected from them are more information rich. While for most successful jobs, the lifetime is very short (around 20 sampling points) and therefore the classifier does not have enough information to classify the successful jobs. For BugBench applications, we first train our model using a total of 500 runs of four applications with different inputs. The inputs are chosen such that half of them will cause the application to fail and half of them will not. We observe a better prediction accuracy (90%) here than the Google dataset due to successful jobs having enough samples.

B. Predicting early

To determine the earliest time we can predict the fate of a job, after training the SVM classifier, instead of considering the aggregate metrics from the whole lifetime of a job, we compute aggregate metrics for different durations from the submission of a job.

We find that the predictions for the killed jobs reach a *strongest* state at as early as within $\frac{1}{8}$th of the total duration of the job for the Google dataset and within half of the job's duration for the BugBench applications.

C. Considering User and Hardware Information

In a datacenter, we have more information on the users of a job and the machine on which the jobs run. An earlier study on "failed" jobs by Nosayba et al. [17] showed that some users are prone to submit failed jobs. When we include the user ID as one of the features, our machine learning model was able to bring the false negative rate from 12% to 4%. But when we use the machine ID on which the jobs run, the prediction does not improve.

VI. Conclusion

In the paper we present an online characterization technique based on the resource usage data of jobs. With machine learning, we are able to detect the killed jobs with 100% accuracy and no false positives for both real world and synthetic workloads. We propose a mechanism to detect a job's fate early in its life cycle, which might greatly help the cloud administrator. We strongly believe that our work will greatly help users and cloud administrators to filter out harmful jobs in the cloud and save costly resources from being wasted.

REFERENCES

[1] Song Fu and Cheng-Zhong Xu. 2010. Quantifying event correlations for proactive failure management in networked computing systems. J. Parallel Distrib. Comput. 70, 11 (November 2010), 1100-1109.

[2] Song Fu and Cheng-Zhong Xu. Exploring event correlation for failure prediction in coalitions of clusters. In Proc, of SC07, 2007.

[3] Y. Liang, Y. Zhang, M. Jette, Anand Sivasubramaniam, and R. Sahoo. BlueGene/L failure analysis and prediction models. In Proc. of DSN06, 2006.

[4] Ramendra K. Sahoo and Mark S. Squillante. Failure data analysis of a large-scale heterogeneous server environment. In In Proceedings of the 2004 International Conference on Dependable Systems and Networks, pages 772–781, 2004.

[5] B. Schroeder and G. Gibson. A large-scale study of failures in high-performance computing systems. In Proc. of DSN06.

[6] Ana Gainaru, Franck Cappello, Marc Snir, and William Kramer. Fault prediction under the microscope: a closer look into HPC systems. In Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, SC 12, 2012.

[7] R. K. Sahoo, A. J. Oliner, I. Rish, M. Gupta, J. E. Moreira, S. Ma, R. Vilalta, and A. Sivasubramaniam. Critical event prediction for proactive management in large-scale computer clusters. In Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining, KDD 03.

[8] Li Yu, Ziming Zheng, Zhiling Lan, and S. Coghlan. Practical online failure pre- diction for blue gene/p: Period-based vs event-driven. In Dependable Systems and Networks Workshops (DSN-W), 2011 IEEE/IFIP 41st International Conference on, pages 259 –264, june 2011.

[9] Yanyong Zhang and A. Sivasubramaniam. Failure prediction in ibm bluegene/l event logs. In Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on, april 2008.

[10] Eric Heien, Derrick Kondo, Ana Gainaru, Dan LaPine, Bill Kramer, and Franck Cappello. Modeling and tolerating heterogeneous failures in large parallel systems. In Proc, of SC11, 2011.

[11] Daniel Ford, Franc ois Labelle, Florentina I. Popovici, Murray Stokely, Van-Anh Truong, Luiz Barroso, Carrie Grimes, and Sean Quinlan. Availability in globally distributed storage systems. In Proc. of OSDI10, 2010.

[12] R. Ren, X. Fu, J. Zhan, and W. Zhou. LogMaster: Mining Event Correlations in Logs of Large scale Cluster Systems. ArXiv e-prints, March 2010.

[13] T. Thanakornworakij, R. Nassar, C.B. Leangsuksun, and M. Paun. The effect of correlated failure on the reliability of HPC systems. In Proc. of Parallel and Distributed Processing with Applications Workshops (ISPAW), 2011.

[14] E. Pinheiro, W. D. Weber, and L. A. Barroso. Failure trends in a large disk drive population. In Proc. of Usenix FAST 2007.

[15] B. Schroeder, E. Pinheiro, and W.-D. Weber. DRAM errors in the wild: a large- scale field study. In Proc. of SIGMETRICS '09, 2009.

[16] https://github.com/google/cluster-data/blob/master/ClusterData2011_2.md

[17] El-Sayed, N.; Schroeder, B., "Reading between the lines of failure logs: Understanding how HPC systems fail," in Dependable Systems and Networks (DSN), 2013 43rd Annual IEEE/IFIP International Conference on , vol., no., pp.1-12, 24-27 June 2013

[18] S. Lu, Z. Li, F. Qin, L. Tan, P. Zhou, and Y. Zhou. BugBench: A benchmark for evaluating bug detection tools. In Bugs 2005 (Workshop on the Evaluation of Software Defect Detection Tools) on Programming Language Design and Implementation (PLDI) 2005, 2005.

[19] https://en.wikipedia.org/wiki/Support_vector_machine

[20] http://icl.cs.utk.edu/papi/

[21] A. Rosa, L. Y. Chen and W. Binder,"Catching failures of failures at big-data clusters: A two-level neural network approach," 2015 IEEE 23rd International Symposium on Quality of Service (IWQoS), Portland, OR, 2015, pp. 231-236.

[22] A. Rosa, L. Y. Chen and W. Binder, "Predicting and Mitigating Jobs Failures in Big Data Clusters," Cluster, Cloud and Grid Computing (CCGrid), 2015 15th IEEE/ACM International Symposium on, Shenzhen, 2015, pp. 221-230.

[23] X. Chen, C. D. Lu and K. Pattabiraman, "Failure Prediction of Jobs in Compute Clouds: A Google Cluster Case Study," Software Reliability Engineering Workshops (ISSREW), 2014 IEEE International Symposium on, Naples, 2014, pp. 341-346.

[24] T. Kimura, A. Watanabe, T. Toyono and K. Ishibashi, "Proactive failure detection learning generation patterns of large-scale network logs," Network and Service Management (CNSM), 2015 11th International Conference on, Barcelona, 2015, pp. 8-14.