Failure Prediction in the Internet of Things due to Memory Exhaustion

Mohammad Rafiuzzaman, Julien Gascon-Samson, Karthik Pattabiraman, Sathish Gopalakrishnan
Electrical and Computer Engineering, The University of British Columbia (UBC)
{rafiuzzaman,julien.gascon-samson,karthikp,sathish}@ece.ubc.ca

ABSTRACT

We present a technique to predict failures resulting from memory exhaustion in devices built for the modern Internet of Things (IoT). These devices can run general-purpose applications on the network edge for local data processing to reduce latency, bandwidth and infrastructure costs, and to address data safety and privacy concerns. Applications are, however, not optimized for all devices and could result in sudden and unexpected memory exhaustion failures because of limited available memory on those IoT devices. Proactive prediction of such failures, with sufficient lead time, allows for adaptation of the application or its safe termination. Our memory failure prediction technique for applications running on IoT devices uses k-Nearest-Neighbor (kNN) based machine learning models. We have evaluated our technique using two third-party applications and a real-world IoT simulation application on two different IoT platforms and on an Amazon EC2 `t2.micro` instance for both single and multitenancy use cases. Our results indicate that our technique significantly outperforms simpler threshold-based techniques: in our test applications, with 180 seconds of lead time, failures were accurately predicted with 88% recall at 74% precision for a single application failure and 76% recall at 71% precision for multitenancy failure.

CCS CONCEPTS

• **Computer systems organization → Embedded systems**;

KEYWORDS

IoT, failure prediction, memory exhaustion.

ACM Reference Format:
Mohammad Rafiuzzaman, Julien Gascon-Samson, Karthik Pattabiraman, Sathish Gopalakrishnan. 2019. Failure Prediction in the Internet of Things due to Memory Exhaustion. In *The 34th ACM/SIGAPP Symposium on Applied Computing (SAC '19), April 8–12, 2019, Limassol, Cyprus.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3297280.3297311

1 INTRODUCTION

This work is motivated by the need for tasks to execute on a variety of devices that comprise the modern Internet of Things (IoT) (e.g., Raspberry Pi, BeagleBone, mangOH, Banana Pi), and where many of the devices are moderately resource-constrained and adding resource virtualization would hinder their performance.

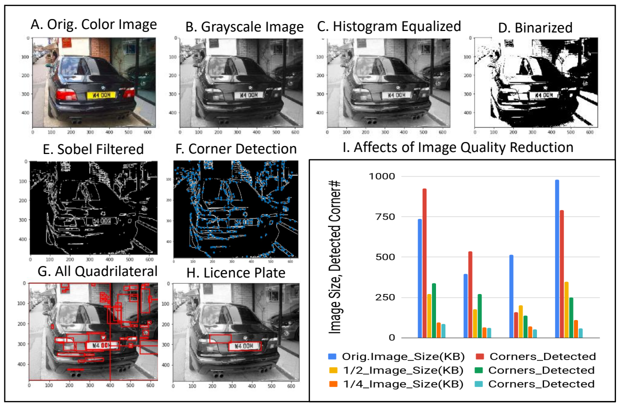

Figure 1: An automatic License Plate Detection (LPD) application. Subplots (A) to (G) shows the image filtering and processing steps to identify the region corresponding to a license plate (H). Subplot (I) shows a decrease in the number of detected corners in (F) if we downsize the image.

Context. The Internet of Things comprises a massive collection of interconnected devices that produce and consume data [19]. In typical cloud-based models, IoT devices send the data that they produce to the cloud, where processing takes place. As the amount of data produced by IoT devices is ever increasing, a purely cloud-centric model will not scale in the future [36]. Moreover, latency-sensitive IoT applications such as augmented reality [22] will benefit from running computations *closer to the edge* [7]; i.e., on or close to the devices that produce and consume the data. As modern IoT devices are becoming more powerful and affordable, they can run general-purpose Operating Systems (OS) such as embedded Linux [1], thereby opening the door to designing and executing complex processing applications written in high-level, abstract languages on these devices. The problem we address in this paper arises from running compute tasks on heterogeneous devices, with limited prior characterization of how a task may behave on a specific platform.

Motivation. Given the hardware and software heterogeneity of the IoT landscape, running high-level applications at edge nodes is challenging. IoT devices can exhibit more stringent and variable resource constraints compared to cloud environments which are more homogeneous. This can lead to sudden resource exhaustion where memory becomes a primary resource bottleneck [30]. Based on our own observations, memory exhaustion in IoT leads to (a) undesirable application response time, (b) sudden application failure,

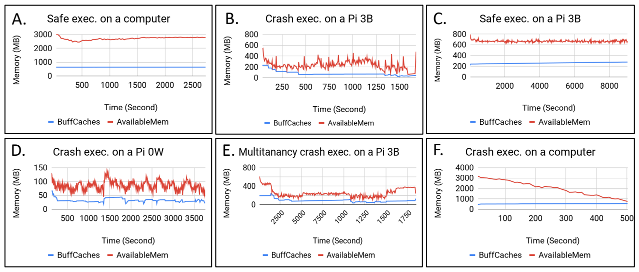

Figure 2: Memory graph of systems (Table 1) running a License Plate Detection task under various circumstances. Here, BuffCaches indicates the reserved system memory and AvailableMem indicates the available system memory

and even (c) sudden OS failure leading to an unresponsive system. We collectively refer to these as *failures*.

As a motivational use case, we study an initial component of an automatic license plate recognition application developed with standard libraries [32] (more details in Section 2.1.1). We ran the resource-intensive License Plate Detection (LPD) task, which identifies the positions of license plates on cars, as shown in Figure 1. The task was executed on two different IoT platforms (i.e., Raspberry Pi 3B and 0W), and on a typical desktop machine that is representative of a node in a data center (Table 1). On the desktop computer, we were able to run the task safely with its default configuration (Figure 2(A)). However, when we ran the same task *at the edge* (i.e., on our IoT devices), we observe different behaviours. With the default configuration, the task exhibited an eventual system crash due to memory outage on the Pi 3B (Figure 2(B)). However, downsizing the frames to half of their original sizes resulted in a safe execution on that same device (Figure 2(C)), but not on the lower-powered Pi 0W (Figure 2(D)). By downsizing the frames even further, the task could eventually run safely on a Pi 0W, at the expense of a decrease in accuracy (i.e., reduced image corner detections as in Figure 1(I)).

The situation becomes more complex in the presence of multitenancy [2]. Hardware advances in modern IoT devices allow them to execute multiple applications simultaneously to get the advantages of multitasking. However, when we placed the LPD task on a device which is already executing another processing task (e.g., video surveillance task, described in Section 2.1.2), both applications eventually crashed (Figure 2(E)), despite being optimized individually. For most of these crashes (both for single and multitenancy), our IoT devices ran out of physical memory and started to consume the remaining pages from their reserved memory, which led to complete system failures as the devices were not able anymore to run their OS critical functions stored in their reserved memories [14]. Note that while our desktop computer test bed was much more powerful (Figure 2(A)), we were able to reach a similar *crash* outcome by inducing artificial processing delays, which led to buffers filling up and consuming large amounts of memory (Figure 2(F)). In that case, however, the OS preemptively killed the task when its virtual memory limit became exceeded, which is not the case for IoT devices due to their limited available memory (Figure 2 (B,D)).

This motivational example (Figure 2) shows that the limited physical memory of IoT devices increases the likelihood of *failures* due to memory exhaustion, which is difficult to detect and differentiate from normal operation and becomes more complicated in the presence of multitenancy [2]. Unlike general purpose computers that have early warning signs (Figure 2(F)) of possible memory exhaustion [6, 9, 10, 28], resource-constrained IoT devices do not reveal such explicit warning signs (Figure 2(B,D,E)), making the memory exhaustion prediction problem challenging.

Table 1: Configurations of devices used in the experiments

	Memory	***Swap Space***	***Processor***
Pi 0W	512MB	100MB	quad-core 1.2 Ghz ARM7
Pi 3B	1GB	100MB	single-core 1 Ghz ARM6
EC2 t2	1GB	N/A	single-core virtual CPU
Computer	4GB	2GB	quad-core 2.4 GHz intel

Memory exhaustion failures can have disastrous consequences, especially for mission-critical applications or for devices being deployed in remote, human inaccessible locations [35]. Proactive actions like periodic checkpointing, task rescheduling, migrating or self-adaptation techniques, can be used to avoid failures. Such avoidance techniques require enough lead time and resources to initiate and complete before the failure actually occurs [15, 34]. Moreover, arbitrary and redundant deployment of such actions (without a significant risk of failure) will result in additional overhead on already resource-limited IoT devices. While there has been some prior work in predicting failures due to memory exhaustion in the cloud or in traditional cluster systems [5, 6, 10, 18], they are either device-specific [10], which is impractical given the heterogeneity of the IoT landscape (e.g., a solution that works for Pi 3B will not work for Pi 0W, as shown in Figure 2 (B-D)), or they only focus on application-level failures (in which the OS does not fail) [5]. In this paper, we tackle the challenge of predicting application and OS memory exhaustion failures for edge applications running on modern IoT devices so that enough lead time is received for mitigation techniques to be applied (e.g., migrating to the cloud).

Contributions. *To the best of our knowledge, we are the first to develop a technique to predict failures due to memory exhaustion in resource-constrained IoT.* We make the following contributions:

(1) Define a systematic approach to identify appropriate system resource parameters indicating the likelihood of memory exhaustion failures in IoT devices (Section 2).
(2) Develop a novel technique called MARK (**M**onitor and **A**nalyze **R**esource **K**eys) to extract, analyze and process such parameters (Section 3).
(3) Introduce simple k-Nearest-Neighbors (k-NN) [29] based classification models to predict memory exhaustion failures in cross-platform heterogeneous modern IoT devices that can predict failures with acceptable recall (Section 4).
(4) Evaluate our models under varying load conditions on two IoT devices and on an Amazon EC2 `t2.micro` instance, with three real-world case studies for both single and multitenancy [2] use cases. For the single use case, considering the video-surveillance application [15], we find that our prediction model can predict failures with a 180 seconds of advance warning time having a 88% recall (i.e., the ability to identify

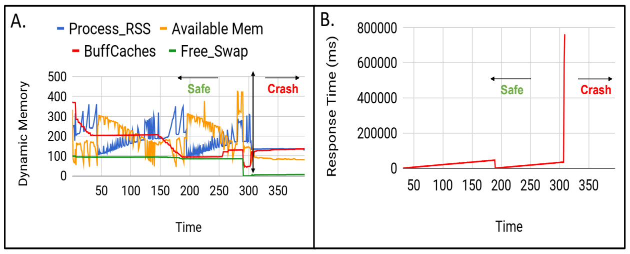

Figure 3: Memory and response time graph of our SensorSim application running on a Pi 0W under memory congestions.

all relevant *crash* failures) at a 74% precision (i.e., the ability to return only relevant *crash* failures). For the multitenancy use case (two concurrent applications), the model can predict failures with 76% recall at a 71% precision (Section 4).

Implications. To avoid the consequences of imminent failures, different mitigation or self-adaptation techniques require different amount of time to perform. For instance, Gascon-Samson et. al.'s ThingsMigrate [15] takes only about 5 seconds to migrate one of our benchmark applications (*Surveillance* – Section 2.1.2) from a Pi 0W to a Pi 3B while preserving all of its states. In such cases, 180 seconds lead time provided by our prediction technique is more than enough to perform the mitigation technique in the form of migration. However, other mitigation or adaptation techniques might require more lead time, and so we have demonstrated our technique's performance on different forewarning times ranging from 5 to 900 seconds (Section 4).

2 INITIAL STUDY

We used three benchmark applications for our model tests and conducted preliminary experiments on one of them to identify resource parameters that can be used to predict memory exhaustion. We first describe the benchmarks used, and then the experiments.

2.1 Benchmarks

To evaluate our models, we used three benchmark applications that are written in high-level programming languages (JavaScript and Python). They are:

2.1.1 Automatic License Plate Detection (LPD) (Python-skimage). This is an image pre-processing application that identifies license plate positions on cars locally from the smart traffic posts (i.e., *network edge*) by using video cameras attached to them (Figure 1). The different processing steps in Figure 1 are both CPU and memory intensive. The pre-processed images generated as output can then be used to automatically recognize license plates of suspicious vehicles, by using different character segmentation and recognition techniques, which can also be run *at the network edge* – these are not considered in this paper.

2.1.2 Video Surveillance (node.js). This is a third-party application derived from Gascon-Samson et. al. [15] (hitherto called *Surveillance*), which can be used to analyze and process camera videos at the network edge. This application has two components, (1) the `video-streamer`, which streams video frames through a pub/sub (i.e., publish-subscribe) broker, and (2) the `motion-detector`, which receives the frames and runs a motion-detection algorithm. Both of these components consume memory to stream and process respective video frames.

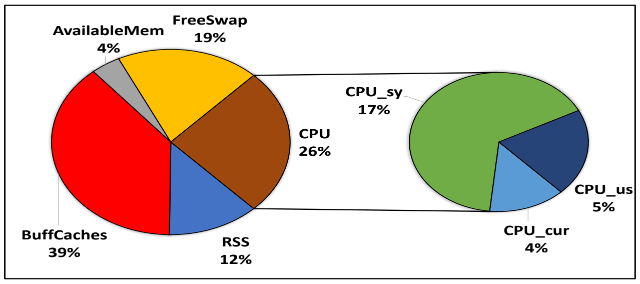

Figure 4: Spearman's Rank-Order Correlation between the benchmark application's response time and system resource parameters during memory exhaustion in IoT devices.

While both of these applications (i.e., LPD and Surveillance) are realistic, they do not offer us a fine-grained control over their memory and computational overheads, which is essential for studying the corner cases of our prediction technique. Therefore, we developed a custom application for sensor data processing in which we can tune the memory overhead at will.

2.1.3 Sensor Data Processing (node.js). Our sensor data processing application (hitherto called *SensorSim*) is a simulation of a sensor data processing server, which upon receiving data from its sensors, stores them into the device memory to do some fixed CPU and memory intensive computations. SensorSim was run as a foreground process concurrently with our memory stressor as a background process in the system. The job of the memory stressor is to gradually consume system memory so that we can observe the behaviour of our SensorSim application under varying and stringent memory conditions. We can tune the workload and monitor internal metrics (such as execution time and resource usage) of SensorSim under various workloads to discover corner cases of a system.

2.2 Experiments on memory exhaustion

To study memory exhaustion, we experimentally vary the memory consumption of the SensorSim application running on a Raspberry Pi 0W (Figure 3) – this is the more memory constrained of the two IoT platforms and hence we use it to explore memory exhaustion failures. However, similar results were obtained on the Raspberry Pi 3B platform as well. As can be seen, when memory is exhausted, the response time of the application shows an unusual super-linear increase. The principal reason for this increase is swap-based thrashing, and the `kswapd` system process [14] (which itself uses up to 85-90% of the CPU on both Pi 3B and 0W) that is in charge of finding free pages when memory appears scarce.

To discover the resource parameters causing this response time increase, we calculated the Spearman's Rank-Order Correlation [29] between relevant resource metrics and application execution time during memory exhaustion for both Pi 3B and 0W. We use the Spearman's coefficient rather than the Pearson correlation coefficient because the relationship between the metrics and the execution time may be non-linear[1].

We observe (Figure 4) that the `BuffCache` (i.e., reserved system memory), free swap space and the `CPU_sy` (i.e., system CPU usage) parameters have the strongest correlation to the application response time during memory exhaustion on our resource-constrained Raspberry Pi IoT devices. On the other hand, the parameters used in cloud systems, such as available system memory ([6, 9, 28]) or the RSS memory ([10]), have minimal correlation with the execution time, and, consequently, are not strong memory exhaustion failure indicators for IoT devices.

We use two of the three parameters which have the highest correlation with memory exhaustion failures, namely (1) `BuffCache`, which represents the reserved system memory, and (2) `CPU_sy`, which represents the system process CPU usage. Free swap space, although it has a strong correlation with the execution time increase, is not a good indicator of failures as it provides very little lead time before the failure. Therefore, we do not use this parameter.

3 PROPOSED APPROACH

We develop a technique to predict memory exhaustion failures that can be used on resource-constrained IoT devices. The information generated by our technique can be used by both (1) *system maintainers*, who want to maintain systems safely by taking proactive actions adaptively, with advance failure warnings, and (2) *system developers*, who want to keep track of which tasks are more likely to generate failures (so that they can be optimized later).

Our prediction models (4.2.4) take as input the current system state, corresponding to the parameters described in Section 2, as well as a given history window of the past system resource usage. The goal is to predict, given these parameters, the likelihood of future system states (i.e., safe or fail). As per our observations, due to the repetitive nature of typical IoT applications, the historical behavior of the system state can be used to predict future states. To that end, we built a technique MARK, that automatically extracts, analyzes, and processes relevant system resource parameters.

3.1 MARK overview

Building a model for predicting resource usage in complex resource-constrained heterogeneous IoT systems running high-level OSes (e.g., Linux) with dynamic resource allocation is challenging [9]. In addition, the fact that we run rich, high-level applications that may exhibit different behaviors on different device profiles (i.e., Raspberry Pi 3B vs Pi 0W), further complicates the analysis and construction of an accurate model. Moreover, due to the frequent and sometimes drastic changes in available resources on IoT devices, the model needs to be dynamic and reactive.

Given these considerations, we developed MARK (**M**onitor and **A**nalyze **R**esource **K**eys), a technique which extracts and analyses the resource usage of IoT systems running complex programs in resource-constrained environments (Figure 5). Overall, the main contribution of MARK is to extract, isolate and combine a wide range of system resource parameters, to provide a more fine-grained system resource usage data for prediction purposes.

[1]Unlike Pearson's coefficient, Spearman's rank correlation coefficient does not assume linearity of the metrics.

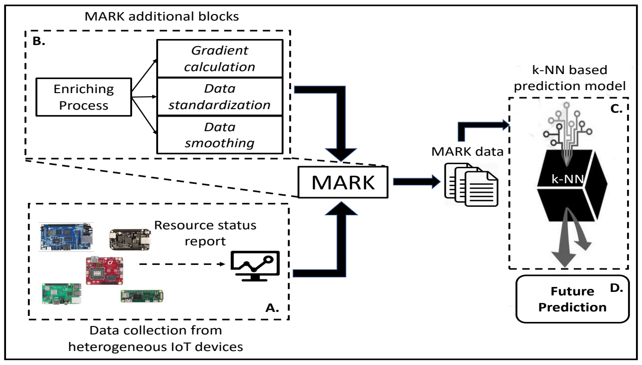

Figure 5: MARK workflow

3.2 Framework architecture

System statistics collected by MARK are used to train and test our prediction models (4.2.4). To do so, we take into consideration the following questions:

(1) Given that mitigation techniques might be used upon an imminent failure being detected, how much advance warning time, or Look **A**head **W**indow (LAW), is needed?
(2) How long does MARK need to observe the system for the prediction model to predict an accurate enough future state?
(3) What phase of the system will be suitable for the prediction model to predict possible future system states?
(4) What is the computational overhead incurred by our prediction model and is that overhead suitable for the system running the model?

The prediction model (Figure 5(C)) predicts possible future states of a system from a suitable location (adjusted according to the computational overhead), and based on the information received from MARK. In general, MARK feeds system statistics with respect to time (Figure 5(A)) to the prediction model since the start of the system, or the launch of a particular application. The base model then predicts future system states; i.e., safe or failed (Figure 5(D)), based on demand. Upon receiving the prediction results of the future system state, if a potential failure is forecasted, a decision is made on whether to initiate recovery or take mitigation actions, (recovery and mitigation techniques are outside this paper's scope).

3.3 Prediction model strategy

As input features to our memory failure prediction model, we consider a set of system resource parameters (from Section 2) at different timestamps that can be denoted as $R_{i,t}$, i.e., the value of $R_i = \{r_1, r_2, r_3, \dots\}$ measured at time t. For simplicity, we assume the set of time points to be $T = \{t_1, t_2, t_3, \dots\}$ with a 1 second time interval. However, this interval can be changed based on system needs as T is a tunable parameter to be adjusted for MARK. But before sending our data to our prediction model, we preprocess them with an *Enriching Process* (Figure 5(B)).

3.3.1 *Enriching process.*

The main purpose of the Enriching Process is to add additional and derived variables to the raw system observations. It performs the following three functions:

Gradient calculation. A challenge in IoT systems is that their values of $r \in R$ vary more over time compared to commodity computers as seen in Figure 2. The limited hardware resources of these IoT devices leads to an increase of this variation. Therefore, a simple approximation of R by averaging $r \in R$ over $t \in T$ is not sufficient to make an accurate prediction. Hence, it is important to measure the changes of R over T in both directions to capture its variation - we have used gradient $Grad(R)$ to do so.

Data standardization. Resource parameters such as `BuffCache` (which are calculated in megabytes), can overwhelm the influence of other parameters measured on a smaller or different scale, such as `CPU_sy`, which is calculated in percentage. We have used data standardization techniques so that we can import data produced in different units and from different devices into a common format for the memory-failure prediction model.

Data smoothing. Due to the limited available resources, noise in the resource usage on IoT devices is very common. To mask the effect of sudden variations, we used the Hamming window [27] method to smooth out noisy features. Depending on the degree of the noise, the size or length of the smoothing window was adjusted.

3.3.2 k-NN based prediction model. We use the enriched feature set to train our prediction model using the k-NN algorithm [21]. We opted for k-NN due to its simplicity, robustness to noisy training data and effectiveness for large training samples. The limited memory space in IoT devices generates significant noise in MARK data, which k-NN can handle robustly. Moreover, users can train k-NN based learning models for new applications quickly in comparison with other algorithms (e.g., Support Vector Machine [37]).

The first step includes hyper-parameter tuning using 10-fold cross-validation where we select the value for k (number of neighbors near a given point), which corresponds to the lowest model training error rate. Then, we train our model with training data collected from numerous execution traces of different workloads from different IoT devices. As this is a supervised learning problem, we have manually classified the training data with target values to represent binary system state using *domain knowledge*.

Our goal is to learn a function $f : R \to S$, so that given a set of resource measurements R, $f(R)$ can predict the corresponding system state $S = \{\text{safe}, \text{fail}\}$. However, this model would yield a prediction of the current system state, as can be expected from a typical k-NN based model. Given that our goal is to predict the future system state, we have trained our prediction model with future target values instead.

The formal representation of our model is given in equation 1. In this equation, A represents a set of k points in the training data that are closest to the current system resource measurement r. $I(arg.)$ is an indicator function which evaluates to 1 when its argument is true and 0 otherwise. Y is the target value and $s \in S$ represents possible system states. In this equation, for a system resource R measured at time t, its corresponding target label Y is set to be at a future timestamp $t + LAW$. This helps us to train and test our model to perform future system state classifications.

$$P(Y_{t+LAW} = s_{t+LAW}) = \frac{1}{k} \sum_{i \in A_t} I(Y_{i,t+LAW} = s_{t+LAW}) \qquad (1)$$

4 EXPERIMENTAL RESULTS

We first present the metrics for evaluating MARK, followed by the experimental setup. We then present results of the evaluation, and compare it with other approaches. Finally, we present an optimization of MARK for even more resource-constrained IoT devices.

4.1 Model evaluation metrics

By considering the *crash states* as "positive class", MARK classifies the future states of a system as either *safe* (i.e., no imminent memory exhaustion failure), or *crash* (i.e., potential upcoming failure in the system). To evaluate the prediction models of MARK and demonstrate its performance, we have considered three classification metrics: (1) *recall* (2) *precision* and (3) *F1-Score* (i.e., a combined metric that indicates a balance between the *recall* and *precision* metrics through computation of their harmonic mean [26]). High *recall* means our model can predict more failures (opposite for low recall); high *precision* means our model can predict failures with minimal false positives (opposite for low precision); and high *F1-Score* means in the prediction, both the *precision* and *recall* are high (whereas for the low *F1-Score*, either the *precision* or *recall* or both of them are low). All the results presented in this paper are for the precision, recall and F-1 Scores related to the *crash* state only, as our goal is to demonstrate the capability of MARK in identifying possible future *failures*. In case of a safe program execution, both the precision and recall values of *crash* states would be almost zero.

Prior work has used *accuracy* to evaluate their models [5], [6], [28]. However, for an imbalanced classification problem such as ours where one category (i.e., safe states) represent the overwhelming majority of the data points, *accuracy* is not an adequate metric for assessing model performance. In this work, we give higher priority to *recall* than *precision* so that our model can detect as many crash states (i.e., failures) as possible regardless of some false alarms. This is reflected in our initial experimental results – *recall* values were usually higher than *precision* values. However, our model can also predict possible future states with high *F1-Score* (i.e., high precision and recall) through an optimization discussed in Section 4.6.

4.2 Experimental setup

4.2.1 Test application 1 - Surveillance. To initially evaluate prediction models of MARK over a third-party IoT application, we used the *Surveillance* application [15] described in Section 2. More specifically, we used the `motion-detector` component of that application, and ran it on a Raspberry Pi 0W device. As explained previously, the `motion-detector` receives video frames (i.e., from the `video-streamer` component – not evaluated in this experiment), stores them in memory and performs motion-detection on them. In this example, we vary the video resolution, which has an effect on the memory usage of the `motion-detector`. We initially start the experiment with video frames having 420x220 resolution, which runs safely on the Pi 0. Then, as we increase the resolution (i.e., to 440x340), memory exhaustion eventually occurs and the application crashes.

4.2.2 Test application 2 - LPD. Besides the *Surveillance* application, we have tested models of MARK with the *LPD* application [32] on a Pi 3B. The application consumes memory to store its unprocessed

Table 2: List of Configurations for Training and Testing

Sets	S_1	S_2			S_3	
Models	***R_10***	***E_10***	***E_60***	***E_300***	***EE_10***	***EE_60***
Train LAWs	10 Seconds	10 Seconds	60 Seconds	300 Seconds	10 Seconds	60 Seconds
Train Applications	SensorSim	SensorSim + Surveillance			SensorSim + Surveillance + LPD	
Test Applications	Motion-Detector	Motion-Detector + SensorSim + LPD	Motion-Detector + LPD + Multitenancy	Motion-Detector	LPD	
Test Platform	Pi 0W	Pi 0W + Pi 3B + EC2	Pi 0W		Pi 3B	
Performance compared with	E_10	Threshold Tech., Compute Overhead	E_300	E_60	E_10	E_60

frames, which grows as the processing delay increases. For our experiment, we executed this application on a Pi 3B with its default (*unoptimized*) configuration, which is computationally more expensive than what the device could handle. Consequently, it eventually crashed on the Pi 3B – we attempted to predict the crash as shown in Section 4.3.3. As the *LPD* application is more resource intensive than the *Surveillance* application, it exhausted the memory faster.

4.2.3 Test application 3 – SensorSim. As described in section 2.1.3, our *SensorSim* application gathers and processes data from simulated sensors. As the number of sensors increases, *SensorSim* can end up consuming all the available limited memory of an IoT device, resulting in a memory exhaustion. We simulated this scenario on an Amazon EC2 `t2.micro` instance (Table 1), and we attempted to predict the memory exhaustion using one of the models of MARK as shown in Section 4.3.4.

4.2.4 Testbed. To demonstrate the applicability of MARK and to evaluate its performance on different use-cases, we trained three sets of its models (Table 2). The first set $S_1 = \{R\_10\}$ was trained with datasets containing highly randomized workloads. To highlight the overall performance improvement of our prediction technique, the second set $S_2 = \{E\_10, E\_60, E\_300\}$ was trained by enhancing the dataset of S_1 with some case-specific workloads. We further enhanced this set with some other case-specific workloads to train the last set $S_3 = \{EE\_10, EE\_60\}$. These models were trained with short (10 seconds), medium (60 seconds) and long (300 seconds) `Training LAWs` to showcase their use-cases in different scenarios (i.e., `Testing LAWs`, which represents various advance warning or lead times).

In MARK, different `Training LAWs` indicate the future time for which the model is trained to do the prediction. However, in the evaluation, models of MARK are tested for predicting beyond that specific point i.e., the `Testing LAW`. For instance, model E_60 is trained to predict the state after 60 seconds – however, we use it to predict the state over a variety of times ranging from 5 to 900 seconds (`Testing LAW`). These experiments show that our model can predict the state of the system at any point in time (`Testing LAW`); i.e., that we are not limited to the specific time used for training (`Training LAW`).

4.2.5 S_1-Training the model with random loads (`R_10`). In order to assess the general usability of our prediction technique, we initially trained one of its models using a large set of randomly generated successful and unsuccessful execution traces (i.e., by using our `SensorSim` application), under different workloads and across different models of Raspberry Pi (Table 1). The successful execution traces were labelled as *safe*, while the unsuccessful traces were labelled either *safe* or *crash* depending on their point of failure. We refer to this model as `R_10` (R for Random) and we trained it using a short *training LAW*. (i.e., 10 second)

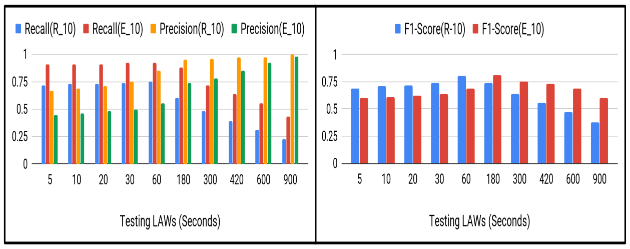

Figure 6: Comparing the recall, precision and F1-Scores of our `R_10` and `E_10` models predicting advance failures (in the form of Testing LAWs) of our `Surveillance` (motion-detector) benchmark on a Pi 0W.

4.2.6 S_2-Training the model with case specific loads (`E_10`, `E_60`, `E_300`). We later enhanced the training dataset of the `R_10` model with additional execution traces of both successful and unsuccessful executions of the *Surveillance* application [15] (which includes both of its `video-streamer` and `motion-detector` components), running on a Raspberry Pi 3B under different workloads. With this enhanced dataset, we trained another three models with three *Training LAWs* (i.e., 10, 60, 300 seconds). We call these models `E_10`, `E_60` and `E_300` respectively (Table 2), which include (some) similar but not identical test data to avoid the risk of overfitting [23].

4.2.7 S_3-Enhancing the case specific loads (`EE_10`, `EE_60`). The last expansion of our training dataset is done by including a few unsuccessful execution traces of our LPD benchmark on a Pi 3B (different than the *LPD* test data in 4.3.3). With short and medium (i.e. 10 and 60 seconds) *Training LAWs*, two models were generated with this training data, which we call `EE_10` and `EE_60` respectively.

4.3 Performance analysis

4.3.1 Model evaluation with Surveillance benchmark on a Pi 0W - Comparing `R_10` with `E_10`. We started evaluating our models by testing them with an unsuccessful (i.e., failing) execution trace

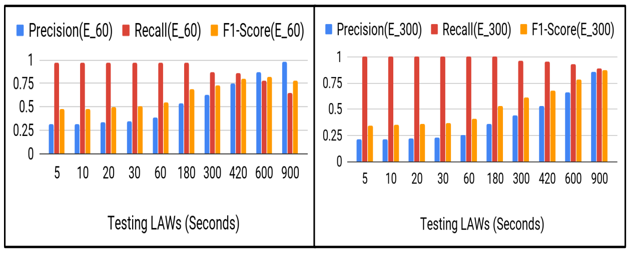

Figure 7: Comparing the recall, precision and F1-Scores of our `E_60` and `E_300` models predicting advance failures (in the form of Testing LAWs) of our `Surveillance` (motion-detector) benchmark on a Pi 0W.

of the *Surveillance* benchmark (`motion-detector` component executed on a Raspberry Pi 0W). Results are shown in Figure 6. Given a *Training LAW* of 10 seconds, we observe that both models predict a crash (*failure*) with high recall values for up to 180 seconds of *Testing LAWs*. We also observe that, in terms of recall, the `E_10` model performs better than the `R_10`, as it has gained some familiarity with the applications under concern. Nevertheless, the `R_10` model predicts failures with 71% recall on average for up to 180 seconds forewarning time which demonstrate general usability of our prediction model in cases where it is completely unfamiliar with its test applications. That being said, for our other evaluations, we focused on the models trained with case specific loads only as users are expected to train the model on their specific applications.

4.3.2 Model evaluation with Surveillance benchmark on a Pi 0W - Performance of `E_60` *and* `E_300` *models.* Figure 7 shows the performance of our models trained with medium and long *Training LAWs*. The results show that with long *Training LAW* (i.e., 300 seconds), our technique can better predict future states for higher *Testing LAW* (900 seconds and, 87% F1-Score) than the model with medium (i.e., 60 seconds) *Training LAW* (78% F1-Score). However, with high *Training LAWs*, our technique predicts future states that are *closer in time* with a lower precision, despite high recalls. This is expected, as in these cases, the model is trained to predict `fail` states *further in the future*. Given that such failures are more likely to occur in a more distant future, the model predicts that *the system is more likely to eventually crash* (which is an assumption that trivially holds true). Nevertheless, with a carefully fine-tuned `Training LAW` parameter (i.e., 60 seconds), our technique can predict failures well enough for both *farther* and *closer* Testing LAWs. We stress that developers and system maintainers can fine tune the *Training LAW* depending on the reliability needs of the target system, and the time needed to apply eventual failure mitigation actions (e.g., migration).

4.3.3 Model evaluation with LPD benchmark on a Pi 3B. For this evaluation, we initially used our `E_10`, `E_60` models to predict the onset of failure as shown in Figure 8. With about 95% recall, our E_60 model can detect failures in advance. However, the precision rates were much lower – given the nature of the system, we consider this to be acceptable, as we prefer not missing failures, at the expense of some false alarms being reported.

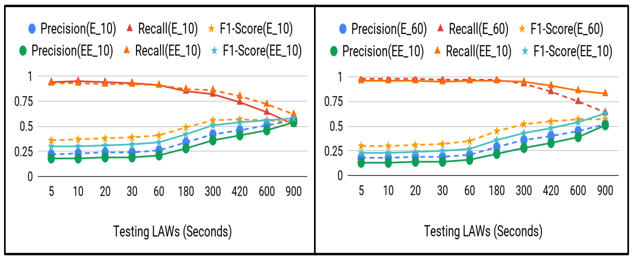

Figure 8: Performance of E_10, E_60 and EE_10, EE_60 models predicting failures of the LPD benchmark on a Pi 3B.

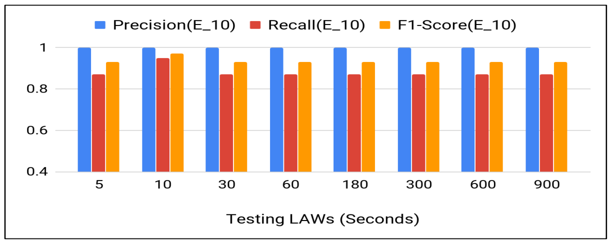

Figure 9: Performance of our `E_10` model predicting memory failures of the SensorSim benchmark on an Amazon EC2 t2.micro instance.

Nevertheless, we investigated the reason why our model obtained low precision scores in this case, and we found that limitations of the kNN algorithm itself caused the performance to degrade. As we have mentioned earlier, the `LPD` benchmark is much more resource intensive than the `Surveillance` benchmark. Hence, during memory exhaustion, the `LPD` demonstrates a sudden memory exhaustion, in contrast to the `Surveillance` which causes a more gradual memory exhaustion on our IoT device. Note that even though the memory exhaustion in *Surveillance* is more gradual than *LPD*, it is still more sudden than in cloud or commodity computers (Table 1). In the present context, the kNN algorithm can easily detect gradual memory exhaustion patterns with the help of its observation points near the centroids, but fails to do so in the case of more sudden memory exhaustions (i.e., `LPD`). This observation holds true even in the case of our EE_10 and EE_60 models (Figure 8) which were trained with similar case-specific datasets, but yet could not improve the precision scores (Sections 4.3.5 and 4.7).

4.3.4 Model evaluation with SensorSim benchmark on an Amazon EC2 `t2.micro` *instance.* In addition to testing against the *Surveillance* application on a Pi 0W and the *LPD* application on a Pi 3B, we tested our E_10 model against an unsuccessful (i.e., failed) execution trace of our `SensorSim` application on a Amazon EC2 `t2.micro` cloud instance (Figure 9). From Figure 9, we can observe that our E_10 model can achieve a 93% F1-Score even for higher Testing LAWs. The findings of this experiment are:

- Memory exhaustion failures are not only limited to OSes like Raspbian [3] used by the Raspberry Pi, but can occur even on an Amazon EC2.t2 micro instance.

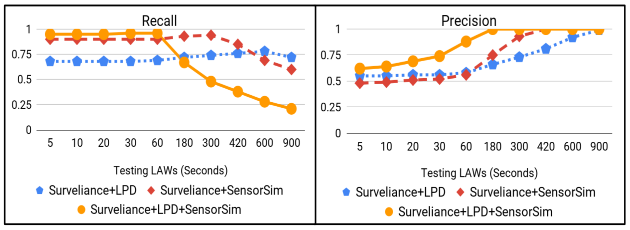

Figure 10: Performance of our `E_60` model predicting various `multitenancy` memory failures of our benchmark applications on a Pi 3B.

- Despite a flexible[2] CPU quota, `SensorSim` also crashed on EC2 `t2.micro` cloud instances, due to their limited memory, which again reinforces the impact of having limited memory.
- Our memory-failure prediction technique is not limited to different models of Raspberry Pi, but also applies to other memory-constrained system.

4.3.5 Model evaluation under multitenancy [2]. In prior experiments, we evaluated our memory-failure technique through different benchmark applications that were executed *independently* and caused memory exhaustion failures on different devices. Most of these memory exhaustions occurred either for non device-specific optimized applications (e.g., Sections 2.1.1, 2.1.2), or for scenarios in which we *forced* a memory exhaustion to explore different corner cases of the system (e.g., Section 2.1.3, Figure 2(F)). As we explained, an alternate mitigation strategy was to *degrade* the performance of the edge applications (e.g., reducing the processing frame-rate or resolution, at the cost of reduced processing accuracy as in Figure 1(I)), which allowed for a safe standalone execution of these programs (Fig. 2(C)). However, despite being individually optimized, attempting to run them together (e.g. *LPD* and *Surveillance* applications on a Pi 3B) can lead to memory failures (Figure 2(F)).

In this experiment, we executed various combinations of our benchmark applications in *multitenancy* [2], on a Raspberry Pi 3B. We observed memory exhaustion failures that were not reported when running these applications independently. With our `E_60` model, we attempted to predict the onset of such failures (Figure 10). From our results, we can see that different combination of applications have different prediction scores while running in multitenancy. Similar to single use cases, gradual memory exhaustion failures in multitenancy can be better detected, in comparison with sudden exhaustion failures (as seen from the sudden drop in recall when predicting failure with a lead time greater than 60 seconds when executing three benchmarks on a Pi 3B in Figure 10).

4.4 Usability of a threshold-based technique

We compared the performance of our memory-failure prediction technique with a threshold-based technique that detects failures by applying different thresholds on *available system memory* (used as memory exhaustion indicators in [9], [28], [10], [6]). We used this model on a Raspberry Pi 0W, and compared it against our `E_10` model (Figure 11), using the Surveillance benchmark. As can be

[2]https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/burstable-performance-instances.html

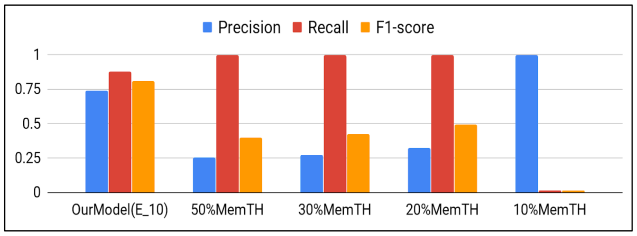

Figure 11: Comparing the `E_10` model performance with a threshold-based technique to detect memory-failures on a Pi 0W.

seen, the threshold-based technique performs much worse in comparison to our model (`E_10`). In particular, for higher thresholds, the recall is high but the precision of the threshold technique is very low implying that many false-positives are incurred; for lower thresholds, the precision is high, but the recall is almost zero, implying that virtually none of the failures are detected. Further, we stress that the threshold is different for different devices. Given the heterogeneity of the IoT landscape, choosing an appropriate threshold for all devices is a tedious and error-prone process, unlike our solution which requires minimal manual intervention.

4.5 Computational and Memory Overhead

Once trained, our memory-failure prediction technique is fast enough to be used on a Raspberry Pi 3B, which we believe is representative of a modern IoT device, at a reasonable price point (about 35$ USD). However, it is not necessary for it to be placed on the system itself. Rather, with MARK, prediction can take place on another device, as long as the system parameters are fed through the network in a continuous fashion. However, a memory driven algorithm such as k-NN [29] that we used consumes memory linearly with the increase of test data, as shown in Table 3. This memory-overhead is insignificant for a desktop computer (i.e., Table 1); however, it can be high for resource-constrained IoT devices. We defer reduction of the memory overhead to future work.

Table 3: Computational overhead of our `E_10` model predicting failures on a Pi 3B with different length of Test data

Test data length in Seconds	Time Overhead in Seconds	Memory Overhead in MB
100	4.1	92.4
1000	5.1	93
5000	8.7	94.5
10300	13.1	96.9

4.6 Optimization

The computational overheads of MARK can be mitigated with some device specific optimization techniques. In general MARK builds the prediction model by using test data collected since the start of the system or launch of any particular application. Instead, MARK can be initialized only when the system is assumed to be under stress, as failures are likely to occur mostly (but not necessarily)

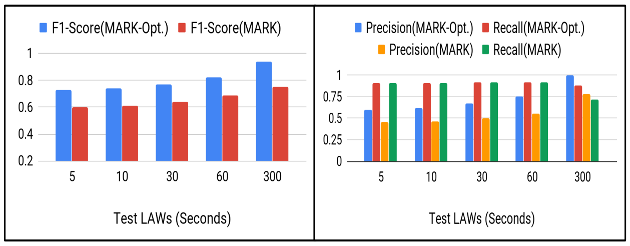

Figure 12: Comparing the `E_10` model performance with optimized and non-optimized MARK on a Pi 3B.

under high load [13, 16, 25]. Our observations revealed that for most of the time, the system remained unstressed and hence, we could omit the collection of testing data during these times. Hence, MARK launched during system stress periods only could reduce the computational overhead of the model as it needs to predict using less test data e.g., 100 observations collected during high system stress, compared to nearly 10000 observations collected from the beginning of the system (Table 3).

Moreover, as per our *domain knowledge*, the memory of IoT devices gets stressed when they start consuming pages from their swap space. This classification draws a clear line between the likelihood of memory exhaustion and normal operation. Therefore, due to the higher probability of failure, MARK initialized during system stress provides a better outcome in terms of prediction as shown in Figure 12. We observe that our `E_10` model predicts possible future failures more accurately (better F1-score) with MARK being optimized (i.e., launched during system stress). More precisely, for both cases, the recall values were found to be similar, but the precision varied – with non optimized MARK, the model tended to have higher false alarms (i.e., lower precision and F-1 scores).

However, doing device specific optimization for MARK is challenging as it has to be initiated only when the system is in stress. Coming up with suitable thresholds to distinguish between high-stress and normal system states must be done on a per-device basis. While the precision for optimized MARK is better (i.e., less false alarms are reported) than normal MARK, recall rates are similar (i.e., similar rates of failures are detected). Therefore, the tolerance to false positives should be weighed against the extra complexity in deploying device specific optimized MARK to determine which approach should be deployed for a given context. Further, due to the nature of memory-driven algorithms like k-NN, if we expand the model's training data, we would also increase the memory consumption of the MARK. In future work, we will examine other learning algorithms that could further reduce these overheads, so that MARK can be directly run on more lower-powered IoT devices.

4.7 Discussion

From the results presented so far, we can see that our technique can predict memory-related failures for different use cases of edge applications written in high-level languages (e.g., JavaScript, Python), with dynamic memory allocation and garbage collection, and executing on different IoT devices. We have identified two leading causes of memory exhaustion failures: (1) the lack of edge device-specific optimization, and (2) the limited system memory of edge devices. However, given the heterogeneous nature of the IoT landscape, we believe that deploying device-specific optimizations *for different applications* would be tedious and impractical. Also, most device-specific optimizations do not work in the presence of multi-tenancy (Fig. 2(F)). Moreover, increasing the memory of IoT devices is not possible, unlike cloud-based VMs. The technique that we are proposing can predict memory-based failures caused by both unoptimized (device-specific) single applications, and optimized applications running in multitenancy mode. Our technique provides enough forewarning time as to allow mitigation or self-adaptation actions to be performed *prior to* the failure occurring. In the future, we envision that our technique could be used to deploy cross-platform IoT applications written in high-level languages, while abstracting low-level device considerations.

Given the large and flexible memory space in the cloud or in normal computer systems, a sudden available memory outage is highly unlikely. On the contrary, this is not the case for IoT where quick and sudden memory exhaustions can be observed very often. However, a limitation of our technique is that it is unable to anticipate sudden memory exhaustions in highly intensive applications (in terms of precision) - this is due to our use of the kNN algorithm for prediction. Nevertheless, given that our prediction technique is decoupled from the black-box learning algorithms, the kNN algorithm can be replaced easily with alternate learning algorithms that perform better - this is a potential direction for future work.

5 RELATED WORK

The idea of predicting failures due to memory exhaustion is not new. Previous approaches have used different machine learning and analytic approaches with successful results in cloud and cluster contexts [6, 10, 24, 28, 31]. However, in contrast with such systems, IoT devices exhibit more stringent and variable resource constraints. This in turns increases the likelihood of rapid memory exhaustion failures, which are challenging to detect, distinguish and predict.

Software aging (i.e., the tendency of a program to fail after running for a while) due to aging-related bugs [17] has been identified as the main reason behind memory exhaustion and consequent failures [20]. Some other papers [5, 6] present solutions specific to the web for predicting failures caused by dormant faults (*Heisenbugs*). Specific to Google's cluster, Chen et al. [10] presents a Recurrent Neural Network (RNN)-based model to predict failures. Different from these studies, our observations revealed that due to their memory limitations, typical IoT devices were often subject to sudden memory exhaustion failures, and that these failures were not attributable to typical causes of failures found in larger-scale cloud and cluster systems (i.e., software aging and memory leaks). Further, the wide device heterogeneity of the IoT landscape renders the task of designing and applying device-specific solutions impractical as in the above-mentioned papers. This motivates the need for a generic solution that can adapt to a wide range of devices, environments and applications with minimal manual intervention.

Memory has been used as a parameter for anomaly detection in traditional computer systems [8]. However, they used some fixed thresholds to detect anomaly with the change of memory which is inapplicable for resource-limited IoT devices (Section 4.4). A Bayesian Network-based technique is proposed in Cohen et. al.[12]

for analyzing system memory. Nevertheless, it incurs a complexity of N^4 (N being the number of devices) [4]. Given that IoT edge-based systems can comprise many more devices compared to cloud environments, the technique quickly becomes unscalable. Linear regression techniques have also been proposed in prior work to model resource usage behaviour [11, 38] . However, Alonso et al. [5] found that linear regression fails to predict the behaviour of systems under anomalies (i.e., a sudden change in workloads), which renders such techniques less applicable in the IoT space where sudden workload changes are common. In comparison, we found that a simple k-NN based classification technique such as ours can predict failures caused by memory exhaustion in IoT devices, even under anomalies. We evaluated such a technique on a class of embedded devices to predict memory-exhaustion failures.

6 CONCLUSION AND FUTURE WORK

We proposed MARK, a memory-failure prediction technique in resource-constrained, heterogeneous IoT devices. MARK provides a device-independent solution for transparently monitoring change patterns in relevant system resources so that potential failures can be predicted with enough lead time as to apply eventual mitigation actions. We evaluated MARK on various real-world edge benchmark applications executed on two different IoT devices and on an Amazon EC2 `t2.micro` cloud instance. For both single and multitenancy use cases, MARK was able to predict failures with high recall, and acceptable precision values. As future work, we plan to test the applicability of MARK on a wider range of resource-constrained IoT systems and applications, and for other kinds of failures.

ACKNOWLEDGEMENTS

This work was supported in part by the Natural Sciences and Engineering Research Council of Canada (NSERC), a research gift from Intel, and the International Doctoral Fellowship from UBC.

REFERENCES

[1] 2018. Embedded Linux. http://elinux.org
[2] 2018. Gartner IT Glossary: Multitenancy. https://www.gartner.com/it-glossary/multitenancy
[3] 2018. Raspbian. https://www.raspberrypi.org/documentation/raspbian/
[4] Omid Alipourfard, Hongqiang Harry Liu, Jianshu Chen, Shivaram Venkataraman, Minlan Yu, and Ming Zhang. 2017. CherryPick: Adaptively Unearthing the Best Cloud Configurations for Big Data Analytics.. In *NSDI*, Vol. 2. 4–2.
[5] Javier Alonso, Jordi Torres, and Ricard Gavalda. 2009. Predicting web server crashes: A case study in comparing prediction algorithms. In *Autonomic and Autonomous Systems, 2009. ICAS'09. Fifth International Conference on*. IEEE.
[6] Javier Alonso Lopez, Josep Ll, Ricard GavaldÃă, and Jordi Torres. 2018. Predicting web application crashes using machine learning. (04 2018).
[7] Flavio Bonomi, Rodolfo Milito, Preethi Natarajan, and Jiang Zhu. 2014. Fog computing: A platform for internet of things and analytics. In *Big data and internet of things: A roadmap for smart environments*. Springer.
[8] Antonio Bovenzi, Francesco Brancati, Stefano Russo, and Andrea Bondavalli. 2015. An os-level framework for anomaly detection in complex software systems. *IEEE Transactions on Dependable and Secure Computing* 12, 3 (2015), 366–372.
[9] Gabriella Carrozza, Domenico Cotroneo, Roberto Natella, Antonio Pecchia, and Stefano Russo. 2010. Memory leak analysis of mission-critical middleware. *Journal of Systems and Software* 83, 9 (2010).
[10] Xin Chen, Charng-Da Lu, and Karthik Pattabiraman. 2014. Failure prediction of jobs in compute clouds: A google cluster case study. In *Software Reliability Engineering Workshops (ISSREW), 2014 IEEE International Symposium on*. IEEE.
[11] Ludmila Cherkasova, Kivanc Ozonat, Ningfang Mi, Julie Symons, and Evgenia Smirni. 2008. Anomaly? application change? or workload change? towards automated detection of application performance anomaly and change. In *Dependable Systems and Networks With FTCS and DCC, 2008. DSN 2008. IEEE International Conference on*. IEEE, 452–461.
[12] Ira Cohen, Jeffrey S Chase, Moises Goldszmidt, Terence Kelly, and Julie Symons. 2004. Correlating Instrumentation Data to System States: A Building Block for Automated Diagnosis and Control.. In *OSDI*, Vol. 4. 16–16.
[13] Domenico Cotroneo, Salvatore Orlando, and Stefano Russo. 2007. Characterizing aging phenomena of the java virtual machine. In *Reliable Distributed Systems, 2007. SRDS 2007. 26th IEEE International Symposium on*. IEEE.
[14] P Daniel, Cesati Marco, et al. 2007. Understanding the Linux kernel.
[15] Julien Gascon-Samson, Kumseok Jung, Shivanshu Goyal, Armin Rezaiean-Asel, and Karthik Pattabiraman. 2018. ThingsMigrate: Platform-Independent Migration of Stateful JavaScript IoT Applications. *European Conference on Object-Oriented Programming (ECOOP)* (2018).
[16] Michael Grottke, Lei Li, Kalyanaraman Vaidyanathan, and Kishor S Trivedi. 2006. Analysis of software aging in a web server. *IEEE Transactions on reliability* (2006).
[17] Michael Grottke and Kishor S Trivedi. 2005. A classification of software faults. *Journal of Reliability Engineering Association of Japan* 27, 7 (2005), 425–438.
[18] Michael Grottke and Kishor S Trivedi. 2007. Fighting bugs: Remove, retry, replicate, and rejuvenate. *Computer* 40, 2 (2007).
[19] Jayavardhana Gubbi, Rajkumar Buyya, Slaven Marusic, and Marimuthu Palaniswami. 2013. Internet of Things (IoT): A vision, architectural elements, and future directions. *Future generation computer systems* 29, 7 (2013), 1645–1660.
[20] Yennun Huang, Chandra Kintala, Nick Kolettis, and N Dudley Fulton. 1995. Software rejuvenation: Analysis, module and applications. In *ftcs*. IEEE.
[21] Gareth James, Daniela Witten, Trevor Hastie, and Robert Tibshirani. 2013. *An introduction to statistical learning*. Vol. 112. Springer.
[22] P Lamkin and S Charara. 2017. The best smartglasses 2017: Snap, Vuzix, ODG, Sony & more.
[23] David J Leinweber. 2007. Stupid data miner tricks: overfitting the S&P 500. *Journal of Investing* 16, 1 (2007), 15.
[24] Lei Li, Kalyanaraman Vaidyanathan, and Kishor S Trivedi. 2002. An approach for estimation of software aging in a web server. In *Empirical Software Engineering, 2002. Proceedings. 2002 International Symposium n*. IEEE.
[25] Rivalino Matias and JF Paulo Filho. 2006. An experimental study on software aging and rejuvenation in web servers. In *Computer Software and Applications Conference, 2006. COMPSAC'06. 30th Annual International*, Vol. 1. IEEE, 189–196.
[26] I Dan Melamed, Ryan Green, and Joseph P Turian. 2003. Precision and recall of machine translation. In *Proceedings of the 2003 Conference of the North American Chapter of the Association for Computational Linguistics on Human Language Technology: companion volume of the Proceedings of HLT-NAACL 2003–short papers-Volume 2*. Association for Computational Linguistics, 61–63.
[27] Lawrence Rabiner, Marvin Sambur, and Carol Schmidt. 1975. Applications of a nonlinear smoothing algorithm to speech processing. *IEEE Transactions on Acoustics, Speech, and Signal Processing* 23, 6 (1975).
[28] Xiaojuan Ren, Seyong Lee, Rudolf Eigenmann, and Saurabh Bagchi. 2006. Resource failure prediction in fine-grained cycle sharing system. In *International Conference on High Performance Distributed Computing*. 93–104.
[29] Laerd Statistics. 2013. Spearman's rank-order correlation. *Laerd Statistics* (2013).
[30] Xian-He Sun and Lionel M Ni. 1993. Scalable problems and memory-bounded speedup. *J. Parallel and Distrib. Comput.* 19, 1 (1993), 27–37.
[31] Kalyanaraman Vaidyanathan and Kishor S Trivedi. 1999. A measurement-based model for estimation of resource exhaustion in operational software systems. In *issre*. IEEE, 84.
[32] Stefan Van der Walt, Johannes L Schönberger, Juan Nunez-Iglesias, François Boulogne, Joshua D Warner, Neil Yager, Emmanuelle Gouillart, and Tony Yu. 2014. scikit-image: image processing in Python. *PeerJ* 2 (2014), e453.
[33] Chen Wang, Hoang Tam Vo, and Peng Ni. 2015. An IoT application for fault diagnosis and prediction. In *Data Science and Data Intensive Systems (DSDIS), 2015 IEEE International Conference on*. IEEE, 726–731.
[34] Pascal Weisenburger, Manisha Luthra, Boris Koldehofe, and Guido Salvaneschi. 2017. Quality-aware runtime adaptation in complex event processing. In *Software Engineering for Adaptive and Self-Managing Systems (SEAMS), 2017 IEEE/ACM 12th International Symposium on*. IEEE.
[35] Geoff Werner-Allen, Konrad Lorincz, Jeff Johnson, Jonathan Lees, and Matt Welsh. 2006. Fidelity and yield in a volcano monitoring sensor network. In *Proceedings of the 7th symposium on Operating systems design and implementation*. USENIX Association, 381–396.
[36] Ben Zhang, Nitesh Mor, John Kolb, Douglas S Chan, Ken Lutz, Eric Allman, John Wawrzynek, Edward A Lee, and John Kubiatowicz. 2015. The Cloud is Not Enough: Saving IoT from the Cloud.. In *HotStorage*.
[37] Hao Zhang, Alexander C Berg, Michael Maire, and Jitendra Malik. 2006. SVM-KNN: Discriminative nearest neighbor classification for visual category recognition. In *Computer Vision and Pattern Recognition, 2006 IEEE Computer Society Conference on*, Vol. 2. IEEE, 2126–2136.
[38] Qi Zhang, Ludmila Cherkasova, Ningfang Mi, and Evgenia Smirni. 2008. A regression-based analytic model for capacity planning of multi-tier applications. *Cluster Computing* 11, 3 (2008), 197–211.