Proceedings of the 7th Small Systems Simulation Symposium 2018, Niš, Serbia, 12th-14th February 2018

System-Level Anomaly Detection using Hardware Performance Counters

Mark Zwolinski, Lai Leng Woo, and Basel Halak

***Abstract*—In computer-based systems, anomalous behaviour can result from physical effects, such as variations in temperature and voltage, single event effects and component degradation, as well as from various security attacks such as control hijacking, malware, reverse engineering, eavesdropping and many others. In this paper, we will present a detection technique to detect a change in the system before the system encounters a failure, by using data from Hardware Performance Counters (HPCs). We show how HPC data can be used to create an execution profile of a system based on measured events and any deviation from this profile indicates an anomaly has occurred in the system. The first step in developing a detector is to analyse the HPC data and extract features from the collected data to build a forecasting model. Anomalies are assumed to happen if the observed values fall outside given confidence intervals, which are calculated based on the forecast values and prediction confidence. A detector should provide a warning to the user if anomalies occur consecutively for a certain number of times. We evaluate our detection algorithm on benchmarks that are affected by single bit flip faults. Our initial results show that the detection algorithm is suitable for use for this kind of univariate time series data and is able to correctly identify anomalous data from normal data.**

I. Introduction

Improvements in transistor size and integrated circuit performance have allowed an increase in the number of affordable embedded sensors. With the emergence of the Internet of Things (IoT), these sensors are now being connected together in networks where huge amounts of time series data are streamed, collected and shared. The sensors used in IoT are considered inexpensive and replaceable, however, there is increasing expectation that these sensors function safely, securely and reliably. The concerns have been studied for many years. Safety in embedded systems means reducing the frequency of failures whereas reliability means ensuring the system completes the task without experiencing any failure [1]. Security in the context of an IoT application is to ensure that malicious attackers do not gain control of any of the embedded devices or systems that could lead to disastrous consequences.

Although care has been taken to ensure these systems and sensors function in a safe, secure and reliable manner, they are still exposed to various environmental conditions which may cause problems for the systems and sensors. For example, the sensors may be imperfect, a bit error may appear, or the nature of the physical processes may have some variations. Security attacks on IoT applications, such eavesdropping, control hijacking, malware and others also cause problems to IoT applications.

The impact of these problems is anomalous behaviour in the system, which could lead to device failure. Very often, users are aware of the anomalous behaviour only after a failure has occurred. One practical approach is to detect anomalies from streaming real-time data. Here, we have used *Hardware Performance Counters* (HPCs) to monitor the behaviour of a system. HPCs are sets of special-purpose counters built into processors to record events precisely and accurately in real-time. A system that behaves normally (no error is detected in the system) exhibits a particular profile, and any deviations from this profile indicate an anomaly in the system. The research on anomaly detection in real-time streaming data is not something new, however, but we have yet to find research attempting to detect a change in the behaviour of the system using HPCs. This paper is the first attempt that focuses on early detection of anomalies (deviation from the normal patterns in the system) by utilising the real-time streaming HPCs that is available in the processor itself, and thus, no modification is required to the physical system. By creating a system that has some self-awareness capability and that is able to provide a warning to the user before a failure occurs, we aim to minimise or even avoid potential risk to the user. Overall, the main contributions of our work are as follows:

- We develop an algorithm for early detection of system-level anomalous behaviour using HPCs;
- We explore several anomaly detection methods in a case study;
- We develop a new attribute called the detection time that evaluates the effectiveness of the early detection algorithm; and
- Our results show that the algorithm can be used for early detection of system-level anomalous behaviour.

This paper is organised as follows. Section II looks at anomaly detection in the context of real-time time-series data. Our experiments with HPCs are presented in Section III. Our proposed detection algorithm and experiment based on hardware performance counter are presented in Section IV. In Section V, we discuss the data we obtained from our experiment. Finally, in Section VI, we conclude the paper and make suggestions for future research.

Mark Zwolinski, Lai Leng Woo and Basel are with Halak
Department Electronics and Computer Science, University of Southampton, Southampton, United Kingdom SO17 1BJ
Email: mz@ecs.soton.ac.uk

II. Anomaly Detection

Anomalous behaviour, or in short, anomalies, is behaviour that does not conform to a normal, expected pattern and can also be identified as outliers, exceptions, peculiarities, contaminants or other terms according to the domain [2] and anomaly detection refers to finding patterns in data that do not conform to expected behaviour [3]. The science of detecting anomalies is typically applied in applications like fraud detection in credit card applications, loan facilities applications, state benefits, fraudulent usage of credit cards and mobile telecommunication [2], network intrusion detection [4], network performance detection [5] as well as activity monitoring [6].

Anomaly detection is not an easy problem to solve due to various factors such as the nature of the data itself, the availability of labelled data, the types of anomalies to be detected, the application domain and many more. There are numerous existing anomaly detection techniques such as classification-based, clustering-based, nearest-neighbour-based, statistical, information theoretic and spectral [3], however, most of these techniques are often used for detecting anomalies in batches of data and are unsuitable for real-time streaming applications. Techniques that require data labelling such as supervised learning are also not suitable for real-time anomaly detection.

Most of the anomaly detection methods used in real-time streaming time series data are statistical techniques that are computationally lightweight, as one of the main requirements is the ability of the algorithm to learn continuously without storing the whole stream of data. These techniques include sliding windows [7], [8], ARIMA [9], Exponential Smoothing [10], Hierarchical Temporal Memory (HTM) [11] and change detection [12]. For real-time streaming time series data, early detection of anomalies is valuable as it allows actions to be taken, possibly even preventing system failure [11]. As evaluating all the methods is beyond the scope of this paper, we will focus on applying sliding window, Exponential Smoothing and ARIMA techniques to detect anomalies in the streaming data from HPCs.

III. Methodology and Experiment

Similar to what was done elsewhere, [13], [14], we use HPCs to create fault-free execution profiles for several different type of benchmarks.

There are several different fault models used in digital circuits. In our experiment, we focus on the single stuck-fault (SSF) model because this fault model is applicable to many different physical fault regardless of whatever technology is applied.

We have chosen to use the following counters to profile the executions: the number of committed instructions (instructions that were executed); number of function calls; number of integer instructions; and number of load instructions.

Once we have identified the fault model and specific hardware counters for monitoring, we created fault-free executables from various benchmarks and performed the simulation using Gem5 and GemFI, described below. We obtain initial execution profiles using the counters. The next step is to inject faults and observe the anomalies recorded using the counters.

This experiment was conducted on a Microsoft Azure virtual machine, [15]. We created a Linux virtual machine with 16 central processing units (CPUs), 32 GBs of memory and 1TBs of data storage. We used Ubuntu version 14.04 for compatibility with Gem5 and GemFI.

A. Architectural Simulator

Gem5 [16] simulator is an instruction set simulator, widely used in computer architecture research. It supports various Instruction Set Architectures (ISAs) such as X86, ARM, Alpha, Sparc, Mips and Power. Gem5 can operate in two modes: *System Call Emulation (SE)* and *Full System (FS)*. SE mode allows users to emulate most common system calls, thus avoiding the need to model devices or even an operating system (OS). In FS mode, Gem5 models complete system including the OS and devices, executing both user-level and kernel-level instructions.

GemFI [17], is a cycle accurate fault injection tool developed based on Gem5 with the primary objective of enabling fault injection. GemFI supports the Alpha and Intel X86 ISAs. There are two intrinsic functions provided by GemFI API:

- **void FI_init()** initialises the fault injection module.
- **void FI_activate (int id, int command)** is a pseudo-assembly instruction to toggle a fault on a specific thread. The thread is given a numerical identification number.

A set of faults is generated using the fault generator that comes together with GemFI. Each fault contains four attributes: *Location*; *Thread*; *Time*; and *Behaviour*.

B. Benchmarks

The benchmarks used in this experiment are from MiBench [18], which is a set of 35 embedded applications divided into six suites with each suite targeting a specific area of the embedded market. We have chosen the *basicmath*, *bitcount* and *qsort* benchmarks from the Automotive and Industrial Control suite, as well as *Dijkstra* from the Network suite. The basicmath program performs simple mathematical functions. The bitcount algorithm tests the bit manipulation ability of a processor by counting the number of bits in an array of integers and *qsort* uses the popular qsort algorithm to sort a large array of strings into ascending order. *Dijkstra* calculates the shortest path between every pair of nodes in a graph.

C. Experimental Setup

To extract the HPCs features that will be used to monitor system reliability, there are several steps:

1) Set up the benchmarks required for testing.
   Each benchmark was compiled dynamically in two versions – one in the original form and another with GemFI intrinsic functions added. Both versions were compiled for the X86 ISA. For *basicmath*, no input data was required, whereas for *bitcount*, the input data is an array of integers and for *qsort*, the input data contains a

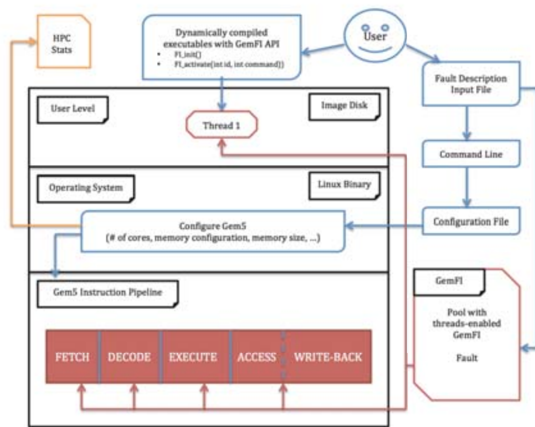


Fig. 1. Overview of the GemFI API, after [17]. The red components show possible fault injection locations; the red octagon is where the executables run.

list of words. The input data for *Dijkstra* is a large graph in the form of an adjacency matrix. The executable files are then placed in the disk image serving as the virtual disk for GemFI.

2) Perform the simulation.
Simulation of the benchmarks was performed in both the Gem5 and GemFI simulators in FS mode. FS mode simulates the execution of the benchmarks in an OS-based simulation environment. A script file is created to assist in the execution of the benchmarks. After fault injection has been initialised and enabled, a set of faults is then created using the fault generator in GemFI. A fault configuration file describing the fault to be injected is provided for GemFI. This file is parsed at startup and each fault is injected into one of the four internal queues, which correspond to a pipeline stage. The simulation continues as normal until it is time for the fault to be injected. Figure 1 provides a general overview of how the simulation works using GemFI API. The blue lines indicates that the tasks belong to the user, the red lines indicate the responsibility of GemFI, and the orange line denotes the HPC values as outputs from the OS.
Each experiment was executed six times: (i) initial run; (ii) with fault activation; (iii) fault injected in the Fetch pipeline; (iv) fault injected in the Decode pipeline; (v) fault injected in the Execute pipeline; and (vi) fault injected in the Load/Store pipeline. The fault model applied in this experiment is a stuck-at-1 fault model, and it is applied at every level in the pipeline.

3) Trace and record the required HPC values.
Two different tracing methods were tried to log the HPCs values obtained. The first method was to obtain the HPCs after the operating system (OS) has booted and another set of HPCs at the end of the execution of the benchmark. However, this method can only provide an indication that an error has occurred which causes the program to either hang, crash or provide incorrect output, but is unable to determine when the fault occurred. The second method was to log the HPCs values at certain intervals. Using this method, we can demonstrate that we are able to create an execution profile for each benchmark, and to detect the instance when an error has occurred. We found that tracing the HPC data at time intervals of 10ms (which is equivalent to 200,000 cycles) is sufficient to create a profile for each benchmark. The HPC data presented below are only for the benchmarks and do not include the OS.

TABLE I
TOTAL INSTRUCTIONS FOR EACH BENCHMARK

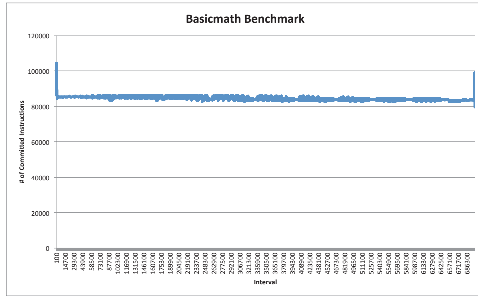
Total Instructions	Benchmarks				
	Basicmath	Bitcount	QSort	QSort (2)	Dijkstra
GemFi	590984224	40508678	83324366	82063568	52370245
GemFI w/ Fault Activated	590989240	40511222	83339395	82065794	52373445
Injected Fault - Fetch	13718386802	27945500	59913002	421330792	50701576
Injected Fault - Decode	590989240	40511222	83244369	82065794	52373445
Injected Fault - Execute	586196426	20397641	21025988	42442442	302299133
Injected Fault - Memory	1410304814	40511222	83339395	82065792	52373446

D. Results and Analysis

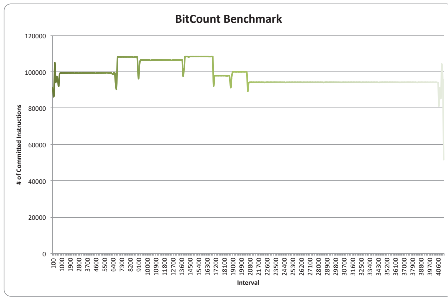
Figure 2 compares the execution profiles obtained from four different benchmarks using the number of committed instructions (axis Y) plotted against the time interval (axis X). By inspection, the profiles for each benchmark differ from one another, which suggests that HPCs can be used to identify the normal behaviour of the system. Profiles using the number of integer instructions, the number of function calls as well as the number of load instructions were plotted as well (but not displayed here due to space) and these profiles also show distinct differences, suggesting that it is sufficient to monitor the reliability of the system based on one or two counters.

In our experiments, as we injected a stuck-at-1 fault in every pipeline, we discovered that this stuck-at-1 (or 0) fault led to errors such as *segmentation faults*, *invalid opcodes*, *kernel panics* and *invalid instruction pointers*. These errors will then cause the program to either *crash* or *hang*. Table I shows the total instructions executed for each benchmark. In particular, "GemFI" and "GemFI w/ Fault Activated" represent the baseline data for the program being executed successfully. The value in "GemFI w/ Fault Activated" will always be slightly higher compared to "GemFI" due to the additional intrinsic functions that are added. Table I also displays the total instructions for *QSort* and *QSort2* with two different sets of input data.

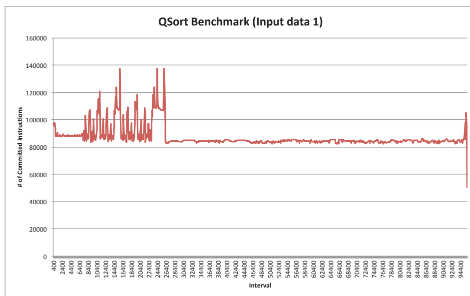
Table I lists the total instructions executed after a fault is injected in every pipeline. Values that differ from the baseline "GemFI w/ Fault Activated" number indicate that an anomaly has occurred in the program. A value that is below the baseline indicates that the program terminates early, whereas a value that exceeds the baseline indicates that the program hangs. As we discussed earlier, obtaining the data at the end of the execution is only able to tell us whether the program has terminated successfully but cannot determine when an error has occurred. For example, consider the total instructions



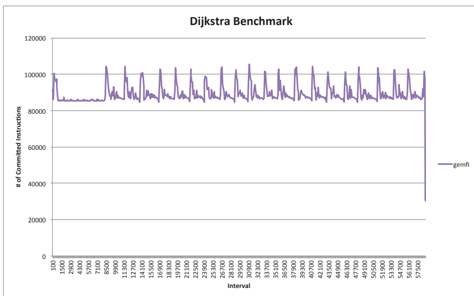
(a)



(b)

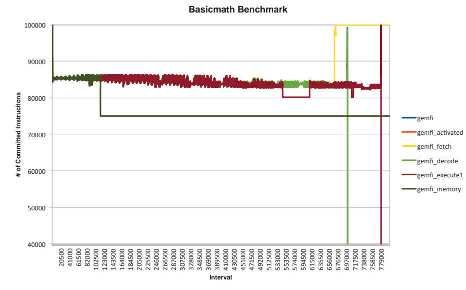


(c)

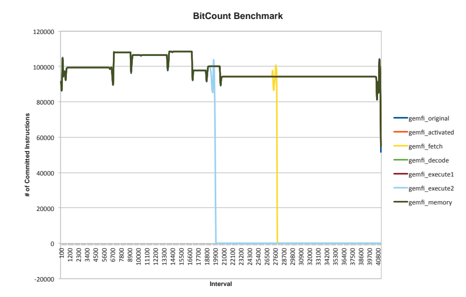


(d)

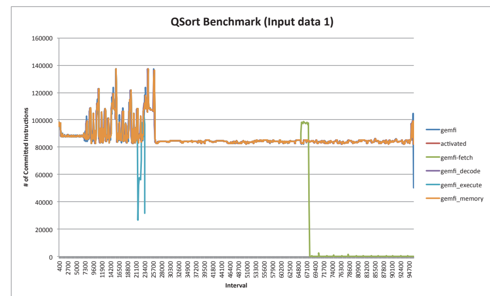
Fig. 2. Execution profiles based on the number of committed instructions for different benchmarks - (a) Basicmath, (b) Bitcount, (c) QSort and (d) Dijkstra



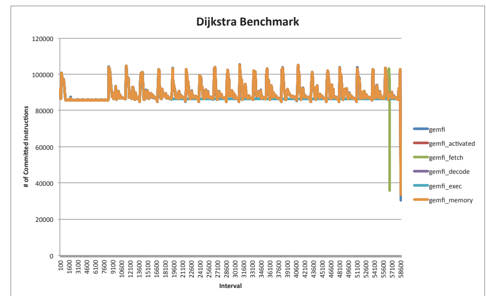
(a)



(b)



(c)



(d)

Fig. 3. Execution profiles that shows how a failure that occurred can be detected using only counters for different benchmarks - (a) Basicmath, (b) Bitcount, (c) QSort and (d) Dijkstra

recorded for the *basicmath* benchmark when a fault is injected in the Execute pipeline. If we compare the value of 586196426 against the baseline 590989240, the initial conclusion would be the program had not terminated successfully. However, if we perform instructions tracing in a time interval, we can see that an error has occurred but that the program did terminate successfully. This is illustrated better in Figure 3 (a) where the profile shown by the maroon line indicates a drop at interval 553500 to value 80000 compared to the expected value between 84000 and 85000. The value 80000 stays for a number of cycles until the program managed to recover from the fault and resumed the execution at interval 615000 until completion. We have shown that by tracing the data at intervals and creating a profile using the HPCs, it is possible to capture even a slight change in the program.

In Figure 3 (a) for the Basicmath benchmark, the profile was able to capture the error occurring in the *fetch* pipeline (yellow line) and the *memory* pipeline (dark green line). As noted, it is also possible to detect an error that occurs due to a fault, but is recovered, as shown by the maroon line in the same figure. This figure shows how an error can be visible if we performed data tracing. Other benchmarks, Figure 3 (b), (c) and (d), also show that errors are detectable using the counters. In 3 (b), the HPCs were able to detect a system hang in the *fetch* pipeline (yellow line) and *execute* pipeline (blue line), whereas in 3 (c), the *execute* pipeline (blue line) shows that the program has crashed (hence the line stopped in the midst of execution), and the *fetch* pipeline (green line) shows the program hangs.

These findings provide justification for our hypothesis that tracing HPCs in an interval can be used for anomaly detection in embedded systems.

IV. Early Detection Algorithm

Using GemFI, a single bit-flip fault model was injected into the Dijkstra benchmark. A single hardware counter, the *number of committed instructions*, was sampled at every 0.1μs and gathered at an equally spaced time interval of 100μs; this data is used to monitor the system's behaviour. We then developed an early detection method with a univariate type of time-series data.

The HPC data is a univariate type of time series data that can be represented as $X = \{x(t)|1 \leq t \leq m\}$ where $x\{t\}$ is a vector of continuous streaming data input at time t and can be represented as $x\{t\} = x_1, x_2, x_3, ..., x_t$. Briefly, in our algorithm, we propose early detection of system-level anomalous behaviour using a sliding window where q is the number of hardware counters used to predict the next sequential data x_{t+1}. Data will be classified as anomalous if it falls outside the upper and lower ranges of the predicted value calculated using D^t as the confidence interval, p. If five anomalies are detected consecutively, the system sends a warning to the user, else, the sliding window moves a step forward by removing x_{t-q+1} from the back of the window and adding the current data x_{t+1} to the front of the window to create D^{t+1}. The detail of the algorithm is explained further in following subsections.

A. One-Step-Ahead Prediction

The first step of this algorithm is to determine the size of the sliding window, q. Generally, there are two types of data used to detect anomalies – historical data and real-time data. The difference between using historical data compared to real-time data is that historical data uses previous and subsequent data in the window as input parameters to determine if the current data is anomalous, whereas real-time data uses only the previous data in the window to determine if the next data is anomalous. When the system experiences some anomalies, it is observed that the counter data starts to deviate from the point at which the fault was manifested. Based on this observation, the sliding window, q, will use previous data to predict the next data and this can be written as:

$$D^t = \{x_{t-q+1}, x_{t-q+2}, x_{t-q+3}, ..., x_{t-q+q}\} \quad (1)$$

Once the size of the sliding window has been determined, a univariate autoregressive model of the hardware counter data is used to predict the next counter value using the previous data as input. According to [7], [8], univariate autoregressive models are models that are used to predict future data in a sensor data stream by using specified set of previous measurements from the same sensor. This model is suitable for use in this experiment as it uses only one variable and data is being sampled at the same frequency. D^t is used as the input into the autoregressive model, M, to predict the next data which can be written as:

$$\overline{x}_{t+1} = M(D^t) \quad (2)$$

This work compares three methods for creating a one-step-ahead prediction model, namely Single Exponential Smoothing (SES), Single-Layer Linear Network Predictor (LN) and Autoregressive Moving Average (ARMA).

1) Single Exponential Smoothing (SES): SES is a type of exponential smoothing that weighs past observations with exponentially decreasing weights to forecast future values. The predicted value, $\overline{x}_{t+1}$ is calculated using:

$$\overline{x}_{t+1} = \alpha x_t + (1-\alpha)(D^t) \quad \text{where} \quad 0 < \alpha \leq 1 \quad (3)$$

The constant parameter α is a smoothing constant, used to smooth or damp older observations. The value of α was determined using a trial-and-error approach and the α value that provides the model with the best performance was selected. SES requires at least 3 previous data points for initialisation before a prediction of the next data can be made.

2) Single-Layer Linear Network Predictor (LN): The LN method predicts the next data $\overline{x}_{t+1}$ as a linear combination of the q previous data points using the following equation:

$$\overline{x}_{t+1} = \frac{\sum_{i=0}^{q-1} w_i x_{t-i}}{\sum_{i=0}^{q-1} w_i} \quad (4)$$

where b and $w_0, w_1, ..., w_{q-1}$ is a set of constant weights used to predict the next expected data based on the sliding window D^t. For simplicity, we have assigned the weight vectors to $1, 2, ..., q$ with the weight vector is inversely proportional to the distance of the points in the sliding window, that is, the further the point x_t from $\overline{x}_{t+1}$, the smaller the weight vector will be.

3) Autoregressive Moving Average (ARMA): The ARMA method (also known as the Box-Jenkins method) consists of an autoregressive (AR) part and a moving average (MA) part where the AR part regresses the variable on its past values, while the MA part models the error term. There is no *Integrated* term, as the time series data is found to be stationary based on the *Augmented Dickey-Fuller (ADF)* test. Therefore, the ARMA (p, q) model is used, defined as follows:

$$\overline{x}_{t+1} = c + \phi_1 Y_{t-1} + ... + \phi_p Y_{t-p} + \epsilon_t - \theta_1 \epsilon_{t-1} - ... - \theta_q \epsilon_{t-q} \quad (5)$$

In our work, we tested the value (p, q) between 0 and 5 in increments of 1, based on an autocorrelation plot (ACF) and a partial autocorrelation plot (PACF); the (p, q) values that provides the model with the best performance are selected.

B. Anomaly Classification

Once the next data has been predicted using either the SES, LN or ARMA prediction methods, the next step is to determine the upper and lower bounds of the predicted value using the confidence interval, p, where the upper and lower bounds determine the acceptable range of values the future hardware counter data can take. We assumed the models' residuals have zero-mean normal distributions and the standard deviation is calculated based on the window size. Therefore, the range of acceptable values is calculated using the following equation:

$$\overline{x}_{t+1} \pm (t_{\alpha/2,q-1} * \frac{\sqrt{\sum(x_t - \bar{x}_t)^2}}{q-1}) \tag{6}$$

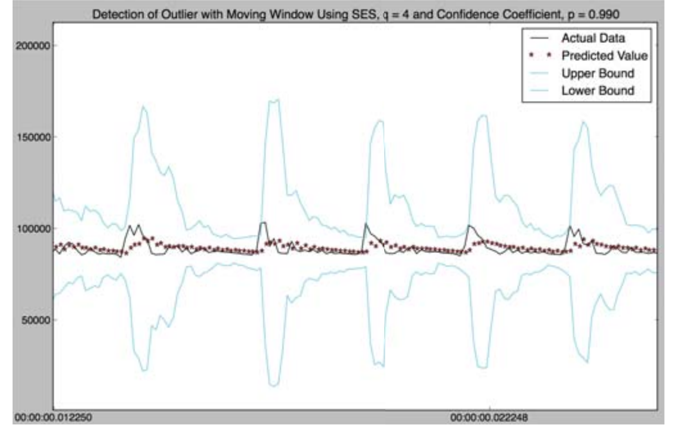
where $\overline{x}_{t+1}$ is the predicted data, $t_{\alpha/2}$ is pth percentile of the t-distribution with $q-1$ degrees of freedom, $\sqrt{\sum(x_t - \bar{x}_t)^2}$ is the standard deviation of the model residual and q is the size of the sliding window. If the next actual hardware counter data falls within the range, the data is considered non-anomalous, else if it falls outside the range, the actual data is marked anomalous. As we had observed, the hardware counter data does not have a fixed value, therefore, using a confidence interval is more relevant and beneficial compared to using some random threshold as confidence interval maintain the width of the upper and lower range according to the actual data.

C. Early Detection

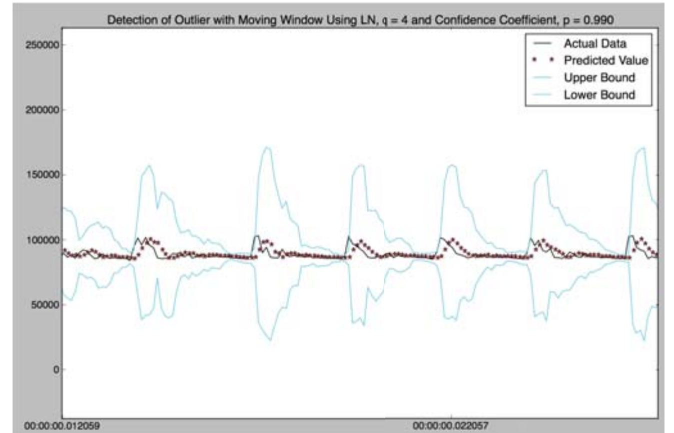
The main objective of the proposed detector is to be able to detect the anomalous behaviour as quickly as possible but at the same time, to avoid the detector being overly sensitive. From our observation, when the system experiences a failure, the hardware counter data begins to deviate from its normal pattern. The number of anomalies detected consecutively is determined using a trial-and-error approach, where the aim is to choose the value that gives the quickest detection time.

V. Discussion and Analysis

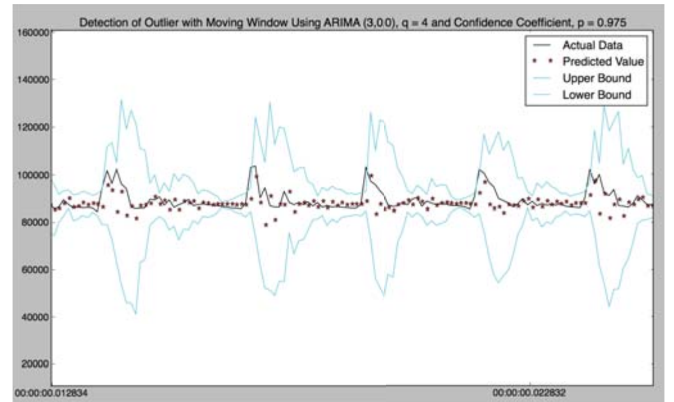
Fig 4 shows the result of one-step-ahead prediction using the three different methods outlined earlier. The upper and lower ranges calculated from Equation 6 are also shown in the same figures. It can be seen that the actual hardware counter data lies very near to the predicted data in all three methods. However, the SES method has provided a rather smoother predicted value, hence there is a bigger residual difference between actual data and predicted data compared to the LN and ARMA methods. Another observation is that both the SES and LN methods have also produced a bigger range of acceptable values compared to the ARMA method. However, as we will show in our analysis, a bigger range of acceptable values will make it harder to detect anomalies.



(a)



(b)



(c)

Fig. 4. One-step-ahead prediction methods - (a) SES Method, (b) LN Method, and (c) ARMA Method

In order to evaluate the effectiveness of the early detection algorithm, we look at how well the detector has performed in classifying the anomalies using the sensitivity, specificity and accuracy statistical attributes. Sensitivity (also known as true positive rate) evaluates how well the detection algorithm correctly identifies the anomalies and specificity (also known as true negative rate), measures how well the detection algorithm

correctly identifies the non-anomalies. Accuracy measures how well the detection algorithm detect both anomalies and non-anomalies. True Positive (TP) and True Negative (TN) are the ideal situation where data points are detected and identified correctly, while False Positive (FP) and False Negative (FN) are undesirable cases which are impossible to eliminate but need to be kept to a minimum. The formulae to calculate sensitivity, specificity and accuracy are defined as follows:

$$Sensitivity = \frac{TP}{(TP + FN)} \quad (7)$$

$$Specificity = \frac{TN}{(TN + FP)} \quad (8)$$

$$Accuracy = \frac{TP + TN}{(TP + FN + TN + FP)} \quad (9)$$

Another attribute that is important in evaluating our detection algorithm is detection time. This is an important attribute as it will determine which method is quickest in identifying the anomalous behaviour in the system. We have developed a formula to calculate detection time as shown below:

$$\text{Detection time} = (TP + FN) * \text{Logging Interval} \quad (10)$$

The goal of the detector is to detect the anomalous behaviour at the earliest time, therefore, the key attribute is the lowest detection time that can be attained from the proposed methods. A low (or early) detection time means low values of TP and FN are required. Lower values of TP and FN means a smaller value of sensitivity will be attained. The window size, k, and confidence interval, p, are the two parameters that are optimised in order to improve the detection algorithm. The parameter k determines how many previous points are used in the calculation to predict $\overline{x}_{t+1}$ and the value k takes in from 4 to 12 in increments of one. Parameter p calculates the acceptable range for a data point to be classify as non-anomalous and the value p varies from 85% to 99% with the increment of 5%. Larger value of p means bigger range of values are acceptable.

Table II shows the statistical analysis of using the ARMA method for predicting the next data with the table showing the top three results where detection time, TP, FN and FP are the lowest. The logging interval is set at 100 μs, and the earliest detection time was found to be 1700 μs. The result shows that the value of specificity where the detector was able to detect and identify non-anomalous points is at least 95% and above in most cases (except where (k, p) is $(4, 90\%)$, the specificity recorded was 92.6%). The best performance of the detector is achieved with $k = 4$ and $p = 99\%$ where actual data points that anomalous, TP is 2 and data points that are normal and identified as normal, TN is 264. The accuracy achieved with this setup was 93.0%.

Table III shows the statistical results for all three methods. For simplicity, only the top three results from each method are presented. From the results, it is clear that all three methods shows high accuracy of at least 89% achieved using LN method and highest being 93.0% which was achieved using the ARMA method. The ARMA method has also been shown to be superior compared to SES and LN, where the anomalous behaviour was detected earliest at 1700μs compared to 1900μs and 2700μs using SES and LN, respectively. The specificity attribute using ARMA method is 93%, which is the highest of all. This means that the one-step-ahead prediction model using ARMA was more accurate as it is able to predict values that lie very close to the actual values.

VI. Conclusion

In this study, we have developed an early detection algorithm that detects anomalies by detecting a change in the hardware performance counter data. This detection algorithm is performed on a univariate time series data using a one-step-ahead prediction applied to a sliding window, k, and the range of acceptable data is calculated from the predicted data using a confidence interval, p. The first step is to slice the data into small windows before applying a prediction model to predict the next counter data. Anomalies are assumed to have occurred if the value falls outside the acceptable range. The algorithm goes a step further to detect if there are 5 anomalies that have occurred consecutively.

The detection algorithm was tested on the Dijkstra benchmark that was affected with single bit flip fault. From the results, we conclude that the early detection algorithm developed in this study is useful in early detection of system-level anomalous behaviour. The detection algorithm is a simple algorithm which is suitable for application in IoT devices as it only requires the time series data of the hardware performance counter and does not require any initial classification of the data. In summary:

- We have developed an algorithm for early detection of system-level anomalous behaviour using hardware performance counter data;
- We explored several methods for one-step-ahead prediction which can be applied in our case study;
- We also developed a new attribute called the detection time that evaluates the effectiveness of the early detection algorithm; and
- Our results show that the algorithm can be used for early detection of system-level anomalous behaviour.

Our study will be further expanded by looking at other plausible techniques which can optimise our detection algorithm to reduce the detection time. One of the ways to reduce the detection time is by minimising the false negatives where the detector identified anomalous data points as normal. We also plan to develop and test the detector in a real IoT device and evaluate the detector in terms of its complexity, computational power, area and cost.

VII. Acknowledgement

This work has been partly supported by Microsoft Azure Research Award number CRM: 0518905.

TABLE II
STATISTICAL ANALYSIS OF ARMA METHOD WITH VARIOUS PARAMETER SETUP

Parameters (k, p)	**(4, 90%)**	**(4, 95%)**	**(4, 99%)**	**(5, 95%)**	**(5, 99%)**	**(6, 95%)**	**(6, 99%)**
TP	3	3	2	3	2	3	2
FN	14	14	15	14	15	14	15
FP	20	11	5	11	8	14	9
TN	249	258	264	258	261	255	260
Sensitivity	0.176	0.176	0.118	0.176	0.118	0.176	0.118
Specificity	0.926	0.959	0.981	0.959	0.970	0.948	0.967
Accuracy	88.11%	91.25%	93.00%	91.26%	91.96%	90.21%	91.61%
Detection Time (μs)	1700	1700	1700	1700	1700	1700	1700

TABLE III
STATISTICAL ANALYSIS USING ARMA, SES AND LN METHODS

	ARMA			**SES (α = 0.3)**			**LN**		
Parameters (k, p)	**(4, 99%)**	**(5, 99%)**	**(6, 99%)**	**(4, 99%)**	**(5, 99%)**	**(6, 99%)**	**(4, 99%)**	**(5, 99%)**	**(12, 99%)**
TP	2	2	2	4	4	4	9	9	9
FN	15	15	15	15	15	15	18	18	18
FP	5	8	9	6	8	9	12	14	13
TN	264	261	260	263	261	260	257	255	256
Sensitivity	0.118	0.118	0.118	0.211	0.211	0.211	0.333	0.333	0.333
Specificity	0.981	0.970	0.967	0.978	0.970	0.967	0.955	0.948	0.952
Accuracy	93.00%	91.96%	91.61%	92.71%	92.01%	91.67%	89.86%	89.19%	89.53%
Detection Time (μs)	1700	1700	1700	1900	1900	1900	2700	2700	2700

REFERENCES

[1] P. Koopman, "Reliability, safety and security in everyday embedded systems," *Dependable Computing, Lecture Notes in Computer Science*, vol. 4746/2007, 2007.

[2] V. J. Hodge and J. Austin, "A survey of outlier detection methodologies," *Artificial Intelligence Review*, vol. 22, no. 2, pp. 85 – 126, 2004.

[3] V. Chandola, A. Banerjee, and V. Kumar, "Anomaly detection: A survey," *ACM Computer Survey*, vol. 41, no. 3, pp. 15: 1 –15: 58, July 2009.

[4] J. Lin, Q. Zhang, H. Bannazadeh, and A. Leon-Garcia, "Automated anomaly detection and root cause analysis in virtualized cloud infrastructures," in *NOMS 2016 - 2016 IEEE/IFIP Network Operations and Management Symposium*, April 2016, pp. 550–556.

[5] Y. Zhang, S. Debroy, and P. Calyam, "Network-wide anomaly event detection and diagnosis with perfsonar," *IEEE Transactions on Network and Service Management*, vol. 13, no. 3, pp. 666–680, Sept 2016.

[6] P. Fiadino, A. D'Alconzo, M. Schiavone, and P. Casas, "Towards automatic detection and diagnosis of internet service anomalies via dns traffic analysis," in *2015 International Wireless Communications and Mobile Computing Conference (IWCMC)*, Aug 2015, pp. 373–378.

[7] D. J. Hill and B. S. Minsker, "Anomaly detection in streaming environmental sensor data: A data-driven modeling approach," *Environmental Modeling and Software*, vol. 25, no. 9, pp. 1014–1022, September 2010. [Online]. Available: http://dx.doi.org/10.1016/j.envsoft.2009.08.010

[8] Y. Yu, Y. Zhu, S. Li, and D. Wan, "Time series outlier detection based on sliding window prediction," *Mathematical Problems in Engineering*, no. 879736, 2014.

[9] Z. Hasani, "Robust anomaly detection algorithms for real-time big data: Comparison of algorithms," in *2017 6th Mediterranean Conference on Embedded Computing (MECO)*, June 2017, pp. 1–6.

[10] A. Kumar, A. Srivastava, N. Bansal, and A. Goel, "Real time data anomaly detection in operating engines by statistical smoothing technique," in *2012 25th IEEE Canadian Conference on Electrical and Computer Engineering (CCECE)*, April 2012, pp. 1–5.

[11] S. Ahmad, A. Lavin, S. Purdy, and Z. Agha, "Unsupervised real-time anomaly detection for streaming data," *Neurocomputing*, vol. 262, no. Supplement C, pp. 134 – 147, 2017, online Real-Time Learning Strategies for Data Streams. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0925231217309864

[12] N. Chen, Z. Yang, Y. Chen, and A. Polunchenko, "Online anomalous vehicle detection at the edge using multidimensional ssa," in *2017 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, May 2017, pp. 851–856.

[13] V. Jyothi, X. Wang, S. K. Addepalli, and R. Karri, "Brain: Behavior based adaptive intrusion detection in networks: Using hardware performance counters to detect ddos attacks," in *2016 29th International Conference on VLSI Design and 2016 15th International Conference on Embedded Systems (VLSID)*, Jan 2016, pp. 587–588.

[14] C. Malone, M. Zahran, and R. Karri, "Are hardware performance counters a cost effective way for integrity checking of programs," in *Proceedings of the sixth ACM workshop on Scalable trusted computing*. ACM, 2011, pp. 71–76.

[15] Microsoft, "Get started with azure," World Wide Web, retrieved 2017-02-15. [Online]. Available: https://azure.microsoft.com/en-gb/get-started/

[16] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A.Wood, "The gem5 simulator," *SIGARCH Comput. Archit. News*, vol. 39, no. 2, pp. 1 – 7, Aug 2011. [Online]. Available: http://doi.acm.org/10.1145/2024716.2024718

[17] K. Parasyris, G. Tziantzoulis, C. D. Antonopoulos, and N. Bellas, "Gemfi: A fault injection tool for studying the behavior of applications on unreliable substrates," in *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, June 2014, pp. 622–629.

[18] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "Mibench: A free, commercially representative embedded benchmark suite," in *2001 IEEE International Workshop, Proceedings of the Workload Characterization, 2001. WWC-4*, ser. WWC '01. Washington, DC, USA: IEEE Computer Society, 2001, pp. 3–14. [Online]. Available: http://dx.doi.org/10.1109/WWC.2001.15