University of Southampton

University of Southampton Research Repository

Copyright © and Moral Rights for this thesis and, where applicable, any accompanying data are retained by the author and/or other copyright owners. A copy can be downloaded for personal non-commercial research or study, without prior permission or charge. This thesis and the accompanying data cannot be reproduced or quoted extensively from without first obtaining permission in writing from the copyright holder/s. The content of the thesis and accompanying research data (where applicable) must not be changed in any way or sold commercially in any format or medium without the formal permission of the copyright holder/s.

When referring to this thesis and any accompanying data, full bibliographic details must be given, e.g.

Thesis: Author (Year of Submission) "Full thesis title", University of Southampton, name of the University Faculty or School or Department, PhD Thesis, pagination.

Data: Author (Year) Title. URI [dataset]

UNIVERSITY OF SOUTHAMPTON

Predicting Potential Failure in Real-Time through Monitoring and Detection of Anomalous Behaviour using Hardware Performance Counters

by

Woo Lai Leng

A thesis submitted in partial fulfillment for the
degree of Doctor of Philosophy

in the
Faculty of Engineering and Physical Sciences
School of Electronics and Computer Science

November 2019

UNIVERSITY OF SOUTHAMPTON

ABSTRACT

FACULTY OF ENGINEERING AND PHYSICAL SCIENCES
SCHOOL OF ELECTRONICS AND COMPUTER SCIENCE

Doctor of Philosophy

by Woo Lai Leng

Safety-critical embedded systems can be found in many application areas such as automotive control systems, medical devices, and nuclear systems. Failure in these systems can have catastrophic results and devastating effects on human lives and the surrounding environment. Variations in temperature and voltage, single event effects and component degradation are just some contributors that cause faults in these systems. Existing research into techniques that deal with errors due to the presence of faults has mostly focused on replication of hardware components, information redundancy or inclusion of additional components to perform self-testing. However, these techniques either have high overheads or are resource-intensive. This thesis presents a detection method that can predict potential failure in real-time by detecting a change in system behaviour using hardware performance counters that are readily available in a processor. The early detection and prediction algorithm consists of two main stages — one-step ahead prediction and anomaly classification. Evaluation on the early detection and prediction algorithm were performed on benchmarks that are perturbed by single bit flip faults. The analysis on the early detection algorithm shows that it achieves 99.7% accuracy and earliest detection time was recorded at 325μs, which is less than a typical time to failure about 4,000μs. The proof of concept results show that the detector manages to detect when the system had started to behave anomalously and is able to stop execution before the system encounters a critical failure. Analyses on the performance and size of the detector show that the detector can be realised with minimal computational time and resources.

Contents

List of Figures

List of Tables

Listings

Research Thesis: Declaration of Authorship

Print Name:	**Woo Lai Leng**
Title of thesis:	**Predicting Potential Failure in Real-Time through Monitoring and Detection of Anomalous Behaviour using Hardware Performance Counters**

I declare that this thesis and the work presented in it are my own and has been generated by me as the result of my own original research.

I confirm that:

1. This work was done wholly or mainly while in candidature for a research degree at this University;
2. Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this had been clearly stated;
3. Where I have consulted the published work of others, this is always clearly attributed;
4. Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work;
5. I have acknowledged all main sources of help;
6. Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself;
7. Parts of this work have been published as listed in Section 1.5 of this thesis.

Signature: Date:

Acknowledgements

My deepest gratitude to my main supervisor, Prof. Dr. Mark Zwolinski and my co-supervisor, Dr. Basel Halak for their continuous advice, support, guidance, supervision, motivation and useful insight during my entire Ph.D study. I could not have imagined having better mentor and advisor for without their support and guidance, I would not have come this far.

A very special gratitude to the School of Electronics and Computer Science (ECS), University of Southampton for granting me the PhD scholarship.

I would like to acknowledge the Microsoft Azure through the Microsoft Azure Research Award number CRM: 0518905 which had provided all the necessary resources to undertake the required research.

I also like to thank my fellow team members who have supported me through our formal and informal discussions, exchange of ideas as well as knowledge. A special shout-out to the parishioners of St. Edmund Catholic Church especially Adrienne Hendry, Simon Lewis, Kate Murelli, Monsignor Vincent Harvey and many others who supported us in one way or other.

My eternal gratitude to my parents (Woo Choe Keong and Chan Sau Ching), my siblings (Woo Tiki, Woo Lai Yenn, Woo Lai Yee and Chow Kok Kien) and my in-laws for their unconditional love, unending support and motivation, prayers and sacrifices throughout the whole period of my studies.

Lastly, my utmost appreciation and love to my husband, Egbert Adolf Naintin. You were my source of support and encouragement, who stood by me through thick and thin, motivates me when I am feeling lost, encourage me when I am feeling down, challenge me when I am feeling overly confident and pray for me every single day. I am truly thankful for having you in my life.

To my children, Daniel, Elisha and Joelle, who taught me love and patience (plus lots of hard work) the key to success.

Abbreviations

AAKR	Autoassociative Kernel Regression
ADF	Augmented Dickey-Fuller
AIC	Akaike Information Criterion
AMD	Advanced Micro Devices
ANN	Artificial Neural Network
API	Application Programming Interface
ASIC	Application Specific Integrated Circuits
AR	Autoregressive
ARMA	Autoregressive Moving Average
BIST	Built-In-Self-Test
CMOS	Complementary Metal-Oxide-Semiconductor
CPLD	Complex Programmable Logical Devices
CPU	Central Processing Unit
DDoS	Distributed Denial of Service
DMR	Double Modular Redundancy
DSP	Digital Signal Processors
DVFS	Dynamic Voltage and Frequency Scaling
FFT	Fast-Fourier Transform
FN	False Negative
FP	False Positive
FPGA	Field Programmable Gate Arrays
FS	Full-System
HPC	Hardware Performance Counter
ISA	Instruction Set Architecture
LLC	Last Level Cache
LN	Single-Layer Linear Network
LV-SVM	Least Squares Support Vector Machine
MA	Moving Average
MAE	Mean Absolute Error
MLP	Multi-Layer Perceptron
MMU	Memory Management Unit
MSE	Mean Squared Error

MSR	Model Specific Registers
MSE	Mean Squared Error
NEqO	Near-Equatorial Low Earth Orbit
PC	Program Counters
PME	Performance Monitoring Event
PMU	Performance Monitoring Unit
SAA	South Atlantic Anomaly
SE	System-call Emulation
SES	Single Exponential Smoothing
SLP	Single Layer Perceptron
SSFM	Single Stuck-at Fault Model
SVM	Support Vector Machine
TLNN	Time-Lagged Neural Network
TMR	Triple Modular Redundancy
TN	True Negative
TP	True Positive
UAV	Unmanned Aerial Vehicle

Nomenclature

C	Number of consecutive anomalies
c	Constant offset in an Autoregressive Moving Average model
df	Degree of freedom
e_t	Residual between observed value and forecast value at time t
$\bar{e}$	Residual average for n sample
h	Forecast horizon
k	Number of estimated parameters in an Autoregressive Moving Average model
L	Maximum value of the likelihood function for an Autoregressive Moving Average model
l_t	Level series at time t
MSE	Average of squared errors for $(n-1)$ sample
n	Sample size
PI	Students T-Distribution at $100(1-a)$ percentile with df
p	Order of the Autoregressive model
q	Order of the Moving Average model
v	Weight vector in Single Layer Linear Network model
W	Size of sliding window
Y_t	Observed value at time t
$\hat{Y}_{t+1}$	Forecast value at time t
$\bar{Y}$	Average of Y values for W number of sample
z	Number of standard deviations away from residual average, $\bar{e}$
z_{upper}	Upper bound threshold value in Prediction Interval
z_{lower}	Lower bound threshold value in Prediction Interval
z_{thresh}	Threshold value for z in Residual Distribution
α	Smoothing parameter in Single Exponential Smoothing model
ϵ_t	White noise at time t
σ^2	Residual variance for n sample

Chapter 1

Introduction

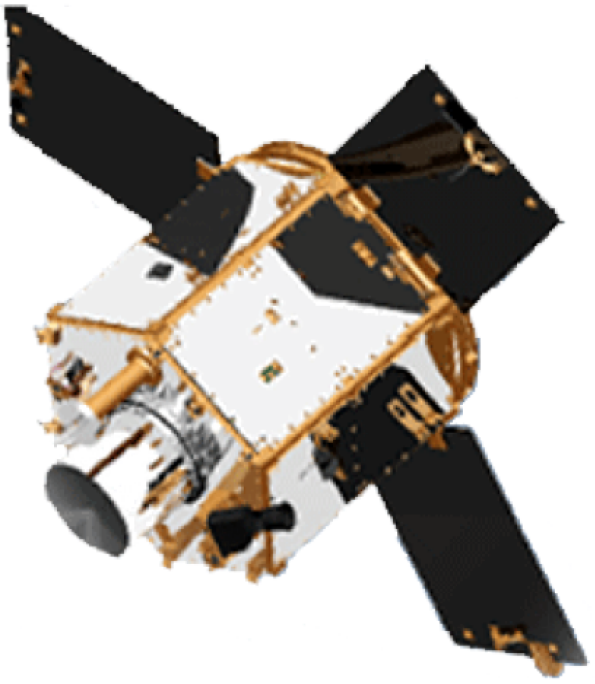


FIGURE 1.1: RazakSAT satellite, source from `https://www.angkasa.gov.my`

RazakSAT, as shown in Figure 1.1, is an earth observation satellite that was launched on July 14, 2009. It was the first satellite in the world placed into a Near-Equatorial Low Earth Orbit (NEqO), providing many imaging opportunities for countries around equatorial region, such as Malaysia. It was targeted to have an operational lifespan of three years, however, it ceased operation on August 30, 2010, just a year and sixteen days from the launch date. The NEqO orbit exposes the satellite to the South Atlantic Anomaly (SAA) phenomenon on every orbit it takes around the earth. SAA is a region of reduced magnetic intensity where the inner radiation belt makes its closest approach to the Earth's surface. Satellites in low-Earth orbit pass through the SAA periodically,

exposing them to several minutes of strong radiation each time, creating problems for scientific instruments, human safety, and single event upsets (SEU) [4]. The failure of RazakSAT resulted in a loss of RM10.89 million in 2009, of which RM7.7 million went towards insurance premiums for the faulty satellite [5].

The Therac-25 was a computer-controlled radiation therapy machine produced by Atomic Energy of Canada Limited (AECL) in 1982. It suffered a concurrent programming error which saw the system giving its patients radiation doses that were hundreds of times greater than normal, thus resulting in death or serious injury [6].

On June 4, 1996, the maiden flight of Ariane 5 launcher, known as Flight 501, veered off its flight path, broke up and exploded about 40 seconds after the initiation of the flight sequence. The end result was that the entire mission was a failure and the cost which includes the destroyed spacecrafts was approximately $370 Million. The report issued by the Inquiry Board in charge of inspecting the Ariane 5 Flight 501 failure concluded that the failure of the active and back-up Inertial Reference System caused the two solid boosters to steer or swivel into extreme positions, and slightly later, the Vulcain engine swivelled, causing the launcher to veer abruptly [7].

RazakSAT, Therac-25 and Ariane 5 Flight 501 are examples of failure in a critical embedded system. A critical system can be divided into three categories:

1. Mission-Critical Systems: A system whose failure may result in the failure of some goal-directed activity. Some examples of mission-critical systems are an on-board computer or a navigational system in a spacecraft.

2. Business-Critical Systems: A system whose failure may result in very high costs for the business using that system. Examples of business-critical systems are the customer accounting system in a bank or the online shopping cart.

3. Safety-Critical Systems: A system whose failure may result in loss of life, injuries, or significant damage to property or the environment.

The improvement on transistors size and integrated circuit performance, known as technology scaling, has allowed the growth of these computing systems across various missions [8]. As shown in Figure 1.2, the number of transistors on integrated circuits doubles every two years, driven by Moore's Law. Technology scaling has set the pace for semiconductor industries over the last decade whereby, with every technology generation, it had resulted in lower cost, lower power consumption, higher performance and higher transistor density per die but it also came with a cost: cheaper and better performance transistors are becoming less and less reliable [8, 9].

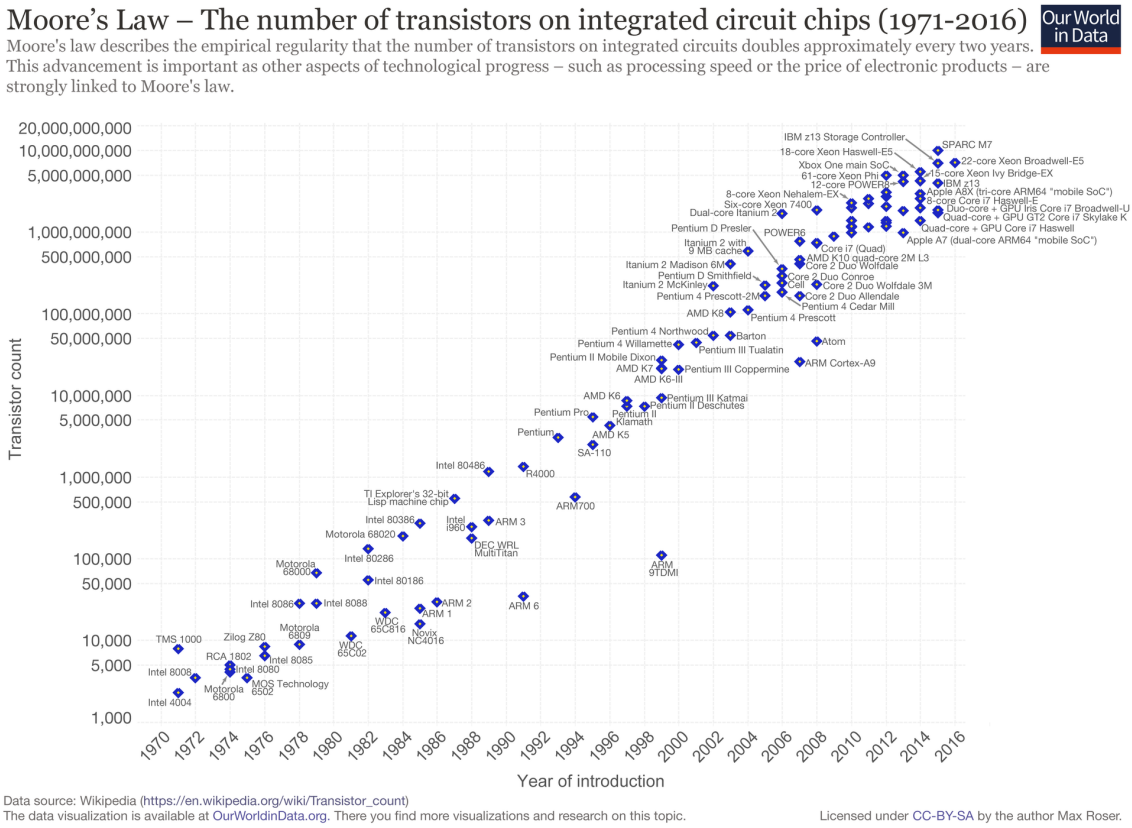


FIGURE 1.2: Transistors count against date of introduction, source from `https://ourworldindata.org/wp-content/uploads/2013/05/Transistor-Count-over-time.png`

1.1 Reliability in Safety-Critical Embedded Systems

1.1.1 Embedded Systems

Embedded systems are becoming more common and widely used in various applications and devices such as automotive industry, factory automation, medical and health, power plants, telecommunication, smart homes, robotics and many others [10, 11]. Driven by advances in microelectronics and software, embedded systems are becoming more affordable for daily usage, and had thus, enrich our lives and connect people together. According to the new market research report on embedded systems market [12], this market is expected to be valued at USD 110.46 Billion by 2023, driven mainly by the increasing adoption of embedded systems in the automotive industry, use of multicore processor technology in military applications, growing market for wearable devices, increase in usage of embedded systems in smart appliances of smart homes, and rising demand for embedded systems in healthcare equipment.

Unlike general-purpose computing system, embedded systems are computing systems that are embedded within larger mechanical or electrical systems and are dedicated to

perform specific functions. Embedded systems are widely associated with microprocessors or microcontrollers, although some embedded systems can contain other technologies like digital signal processors (DSPs), complex programmable logical devices (CPLDs), application-specific integrated circuits (ASICs), and field programmable gate arrays (FPGAs). An embedded system can be defined as [10]:

> Embedded systems are information processing systems embedded into enclosing products.

1.1.2 Characteristics of Embedded Systems

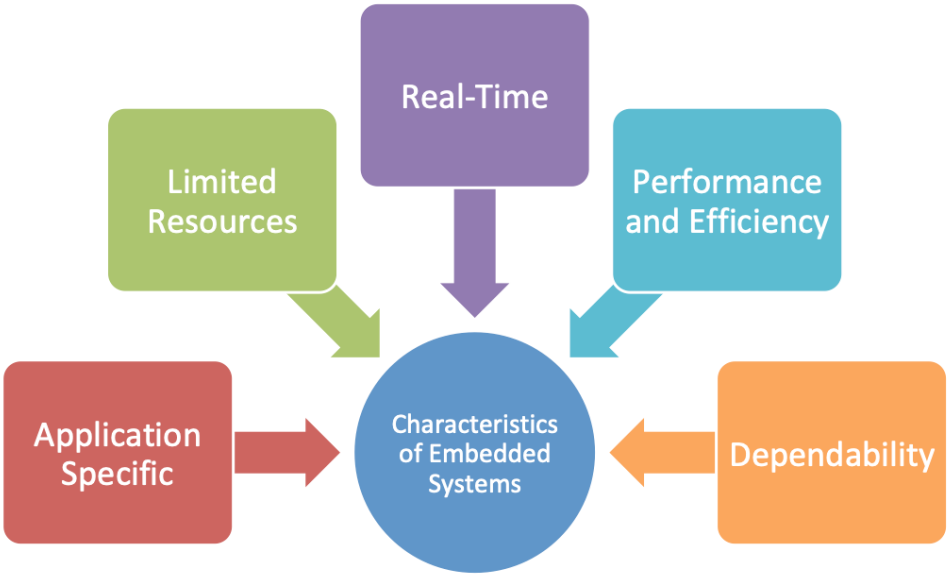


FIGURE 1.3: Characteristics of an embedded system

Despite various types of applications and implementation methods that are available, embedded systems are bound by some common characteristics as shown in Figure 1.3, which are briefly explained as follows:

- Application Specific: An embedded system is used to perform specific tasks related to its specific application, in contrast to a general-purpose computing system that executes a variety of applications [13].
- Limited Resources: Due to the many reasons such as the nature of application, production costs, and available hardware technology, embedded systems have tight constraints and limited resources concerning hardware resource, processor speed, power consumption and memory size [14]. For example, a microprocessor used in a general computing system operates at a clock speed above 2GHz, while the clock speed for a microcontroller varies between 20MHz and 300MHz, a fraction compared to the clock speed of a microprocessor.

- Real-Time: Embedded systems have to perform tasks or interact with the external environment within specific timing constraints, where the correctness of the system depends on the output results as well as the time the results are produced [11].
- Performance and Efficiency: Embedded systems are expected to achieve high performance, usually defined by the amount of tasks completed within certain execution time. Given the limited resources faced by these systems, embedded systems also have to be efficient in utilising the power consumption, memory utilisation and hardware resources [10].
- Dependability: Dependability is the ability to avoid service failures that are more frequent and more severe than what is acceptable [15, 16]. The common issues that arise in creating a dependable system are reliability, safety, security, availability, integrity and repairability [10, 13, 15, 16].
  - *Safety* means the system is able to function without catastrophic failure or reducing the frequency of failures.
  - *Reliability* means ensuring the system completes the task without experiencing any failure.
  - *Security* means the ability of the system to protect itself against deliberate or accidental intrusions.
  - *Availability* means the system is able to deliver the service when it is required.
  - *Integrity* means the system is protected against improper or unauthorised system alterations.
  - *Repairability* means the system can undergo modifications or repairs.

  As the number of embedded systems being deployed increases, the designers have a duty to ensure these systems are dependable.

In this thesis, the focus is given to addressing safety and reliability in a safety-critical embedded system. The reliability of an embedded system will be further discussed in Section 2.2. However, this does not diminish the importance of other attributes.

1.1.3 Safety-Critical Embedded Systems

According to [17], *safety* can be defined as *a property of a system that it will not endanger human life or the environment.* Therefore, a safety-critical embedded system can be defined as:

> A system where a failure or a malfunction might result in loss of life or severe injury to people, loss or severe damage to property or equipment, and severe damage to the environment.

Safety-critical embedded systems exist in the automotive industry and in various industries such as aviation, medical, nuclear engineering, power plants and many more [18]. The falling cost of hardware and the improvement in hardware quality will continue to be a catalyst driving the growth of safety-critical embedded systems.

With technology scaling, increased complexity in a system, introduction of new materials and devices, as well as increasing constraints in terms of time and money, experts have predicted that reliability will soon become a major concern [19]. Not only that, various conditions such as faulty devices, bit errors due to Single Event Upsets (SEUs), more pronounced ageing effects, process variations [9, 20, 21] or inadequate testing and verification processes coupled with increased time-to-market pressure [22, 23] may cause a system to experience faults.

These faults can manifest themselves as errors and cause the system to experience anomalous behaviour, which could lead to system failure, and thus contribute to a system behaving unreliably. This is a major concern and challenge not only for users, but also for technology vendors, system designers and system architects. One existing technique for preventing system failure in the presence of faults is by having a fault-tolerant system, which is usually achieved through the implementation of error detection and recovery [15, 16, 23, 24]. Existing fault tolerance techniques look at detecting errors through the failure that results, and very often, users are only aware of the presence of anomalies in the system after a failure has occurred.

1.2 Anomalous Behaviour

FIGURE 1.4: Illustration of anomaly behaviour, source from `http://www.dbta.com/Editorial/`

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 being studied [25, 26]. Figure 1.4 depicts an analogy of what an anomaly is all about and Grubbs [27] has defined an outlier as:

> An outlying observation, or outlier, is one that appears to deviate markedly from other members of the sample in which it occurs.

In data mining, anomaly detection refers to scientific techniques applied to identify behaviours, events or data points that do not belong to the rest of the data in a dataset. Chandola [26] defined anomaly detection as:

> The problem of finding patterns in data that do not conform to expected behaviour.

The reliability of a system can be compromised by various sources such as (a) design errors, (b) manufacturing problems, (c) external disturbances, (d) harsh environmental conditions, and (e) system misuse. The impact from these reliability problems causes anomalous behaviour in the system. Research on anomalous behaviour is usually associated with malicious activities, cyber intrusions or terrorist activities [26, 28], but anomalous behaviour in the system could also be due to the breakdown of a system caused by reliability issues [26, 29]. Anomaly detection can be applied in various domains and applications such as fraud detection in credit card applications, loan facilities applications, state benefits, fraudulent usage of credit cards and mobile telecommunication [25], network intrusion detection [30], network performance detection [31], activity monitoring [32, 33], system health management [34], sensor networks [35] and many more. Chapter 2 provides further understanding and discussion of anomaly detection.

1.3 Research Motivation

The failure observed in incidents like RazakSAT, Therac-25, Ariane 5 Flight 501 or in other similar examples, had a devastating impact not only to society in general, but also to the economy and environment of a country. The presence of anomalies in the systems had gone undetected, and users were only aware that something had gone wrong when a failure occurred.

Research in fault prevention looks into ways of strengthening the circuit, architecture or even system from these reliability issues. Fault prevention techniques are applied during the design and manufacturing phases with the focus on designing a better circuit, a better architecture, or a better system to prevent fault [23]. Techniques such as radiation hardening, shielding and others modify existing circuits or architectures, which more

often than not, struck a raw nerve with hardware designers because this means the circuits that have been designed are deemed not reliable enough. However, modification of existing circuits does not address the issue of reliability that still occurs after the post-silicon validation stage.

Another way to attain reliability for systems that are already in operation is through fault tolerance. Fault tolerance research is about preventing system failure in the presence of errors, and it is usually achieved through the implementation of redundancy, error detection and recovery [23,36]. Fault forecasting techniques aim to estimate the number of faults in the system, possible occurrence of faults in the future and the consequences of those faults and fault forecasting is done by evaluating the system's behaviour when a fault occur or is activated [16].

Current research on fault tolerance looks at detecting errors by examining the failure that occurs, but to date, there had been no research on predicting potential failures in real-time by detecting anomalous behaviour in the system before the user encounters the failure. Research in fault forecasting mostly revolves around mechanical systems or physical structures by using sensors to collect data [26] and there is no research that attempts to predict potential failure in an embedded system by monitoring anomalies in the system. Predicting the possible failure in a safety-critical embedded system by monitoring and detecting anomalous behaviour can help to minimise or even avoid failure-induced risk which could jeopardise the safety of the user or the surrounding environment.

The aim of this thesis is to complement current fault tolerance techniques and contribute to a better protection strategy by presenting the design of a detector that will be able to predict potential failure in real time through the detection of anomalous behaviour in a processor. In this thesis, we evaluate various strategies for achieving quick detection and high accuracy with minimal computational time and resources.

1.4 Research Objectives

As discussed in Section 1.3, it is crucial to detect the error before a failure occur as this could help to reduce or avert any risk which could threaten the safety of the user or the surrounding environment. Therefore, the fundamental research question for this thesis is:

Is it possible to predict a potential failure in an embedded system by monitoring and detecting the anomalous behaviour in the system?

In order to assist the fundamental research question, several other research questions had been formulated:

1. What are the available hardware performance counters in a processor which can represent a behaviour of a system and can be monitored online and in real-time?
2. What are the suitable techniques to model the behaviour of the system and perform early detection of anomalies to predict potential failure?
3. How do different prediction algorithms impact the implementation of the detector?

The following objectives are presented as follows to address the above questions:

- Investigate how a manifested fault affects the behaviour of the system and identify the various Performance Monitoring Events (PMEs) available that can be used across different types of processors. Identify the number of hardware performance counters available in processors used in embedded systems, in particular, the number of available counters in an Intel Atom processor used in this work. Select different PMEs and compare them to determine which is better for detection.
- Develop an algorithm for early detection suitable to be implemented in embedded systems taking into account the constraints and limitations of an embedded system. Explore several methods for one-step ahead prediction and anomaly classification rules and perform evaluation on methods used in the early detection algorithm.
- Implement the developed algorithm as a hardware-based detector. Validate the implementation through experimental simulations and analyse the performance and cost of the proposed detector.

1.5 Publications

Part of the research in this thesis have been published as:

1. E. W. L. Leng, M. Zwolinski and B. Halak, "Hardware performance counters for system reliability monitoring," 2nd International Verification and Security Workshop (IVSW), 2017
2. L. L. Woo, M. Zwolinski and B. Halak, "Early detection of system-level anomalous behaviour using hardware performance counters" in Design, Automation Test in Europe Conference Exhibition (DATE), 2018. IEEE
3. L. L. Woo, M. Zwolinski and B. Halak,"Predicting Potential Failure from Anomalous Behavior in Embedded Systems", in IEEE Transactions on Reliability (submitted in July 2019)

1.6 Thesis Organisation

This thesis is organised as follows:

- Chapter 2: This chapter provides an overview of reliability and discussed how defects, faults, errors and failures can affect the reliability of an embedded system. This chapter also covers existing online error detection techniques with particular emphasis on anomaly detection techniques. Discussion on hardware performance counters (HPCs) and how they can be used to detect anomalous behaviour are also presented in this chapter. The chapter concludes by identifying the gap in current online error detection and how utilising HPCs for anomaly detection and prediction of potential failure can address the gap.

- Chapter 3: This first objective of the thesis is addressed in this chapter. Embedded systems that performs a routine task exhibit a certain profile. When a fault is manifested as an error, it first causes deviations in the profile, indicating some anomalous behaviour has occurred in the system before the user encounters the failure. This chapter presents how HPCs can be used to observe the anomalous behaviour in the system by observing the deviations in the execution profile. Using GemFI, a fault injection tool developed based on Gem5 architectural simulator, single bit-flips are injected in various stages of a pipeline, and the behaviour of the system is observed. The first contribution of the thesis is also presented in this chapter where the HPC is utilised for monitoring anomalous behaviour that occur at a system level due to a reliability issue. A suitable Performance-Monitoring Event (PME) and sampling interval are proposed based on the experiment conducted.

- Chapter 4: This chapter addresses the second objective and provide the second and third contributions of the thesis. In this chapter, the novel early detection algorithm that detects anomalous behaviour in a processor core using HPCs and predict potential failure in real-time is presented. The algorithm consists of three stages: (i) a one-step ahead algorithm to predict the next value in the time-series, (ii) measurement of deviation algorithm between predicted value and observed value, and (iii) mechanism to classify if the observed value deviates too much from the expected behaviour and is deemed anomalous. Through the experiments, the optimal values of each parameter in the one-step ahead prediction methods and anomaly classification methods are identified. The novel detection time measurement attribute developed refers to the earliest time for the algorithm to predict potential failure. This attribute provides an indicator of how well the early detection and prediction algorithm has performed.

- Chapter 5: The last contribution of the thesis is addressed in this chapter, where a proof of concept for a detector that predicts potential failure in real-time by

detecting anomalous behaviour that occurs is developed using the early detection algorithm developed and optimised in Chapter 4. This chapter also presents the analyses on the performance and size of the detector, where it shows that the detector can be realised with minimal computational time and resources. It overcomes the drawback in existing error detection techniques where an error is only detected after a failure has occurred.

- Chapter 6: The findings and contributions of this thesis is concluded in this chapter. Suggestions for future research direction are provided in this chapter as well.

Chapter 2

Background and Related Work

2.1 Introduction

In Chapter 1, an overview of reliability, especially in safety-critical embedded systems, and how it leads to the failure of these systems was presented. In this chapter, a more in-depth study on reliability and how anomalous behaviour affects reliability is discussed, with particular focus on hardware performance counters, and how they can be used to detect anomalies in embedded systems. This chapter is organised as follows. Section 2.2 discusses what reliability is and the available means to attain a reliable system. Section 2.3 will look at the threats to reliability, which are usually defined in defects, faults, errors, and failures, and how a fault is propagated into an error which causes a failure in the system. From the user's perspective, the system fails when it is unable to deliver its intended function although the reason for failure may be unknown to the user. However, there could be some traces of anomalous behaviour in the system prior to a failure which can be detected. One of the main techniques for online error detection is via anomaly detection, which is presented in Section 2.4. This section also discusses damage detection, the application domain for anomaly detection which is of interest in this thesis, and the available techniques for anomaly detection in this domain. In Section 2.5, other techniques for online error detection, such as Built-In-Self-Test, Redundancy and Dynamic Verification are presented. Past research has shown that Hardware Performance Counters (HPCs) can be used to detect anomalous behaviour in a CPU [37,38]. In Section 2.6, an overview of HPCs is provided and some past research that uses HPCs are mentioned in brief. Section 2.7 concludes the whole chapter.

2.2 Reliability

Reliability can be described as the probability that a system will produce the correct or required outputs at time $t + 1$, given that the system was performing correctly at

time, t [16, 23, 24, 39]. Reliability is a crucial aspect in a computer system, even if it is at the expense of the performance of the system. Safety-critical embedded systems are systems that require high reliability whereby these systems are expected to be operational without interruptions or when maintenance is unavailable [16], and failure and data loss is almost unacceptable. Nanometre technology scaling has allowed systems to be built for a higher performance at a lower cost and power consumption, but, this is also accompanied by reliability problems that cause different failures over time.

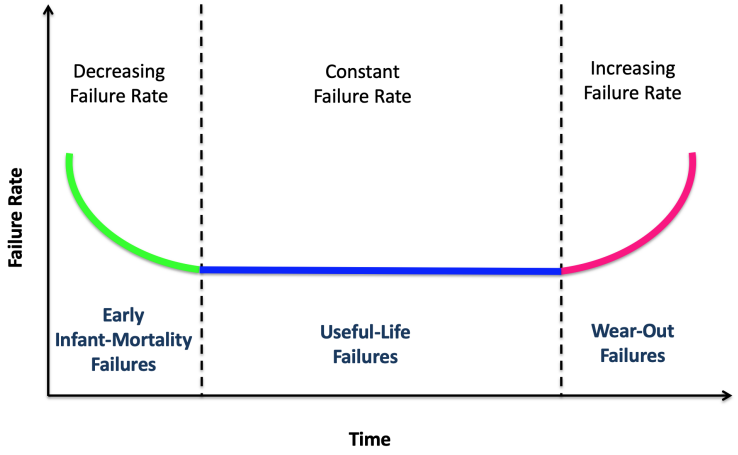


FIGURE 2.1: The 'bathtub curve', after [39, 40], is a combination of a decreasing hazard of early failure (green line) and an increasing hazard of wear-out failure (pink line), and a constant hazard of useful-life failure (blue line)

The bathtub-curve shown in Figure 2.1 is widely accepted and used to represent the failure rate of equipment and systems over time. It consists of three parts – early failures, constant failures, and wear-out failures. Early failures, which happen in the first zone, have a decreasing pattern where the rate of failures decreases during the early times of operation. In complementary metal-oxide-semiconductor (CMOS) technology, early failures are mainly caused by oxide defects, particulate masking defects or contamination-related defects. In the middle zone, the failures remain at a constant rate. The occurrence of these failures are mostly random, manifesting in the form of soft errors over the major part of the system operation life. The failures that occur in the middle zone will be the target of this thesis. In the third zone, wear-out occurs in the final stage of the system lifetime where the failure rate increases. For example, electromigration-related defects, oxide wear-out, or hot carrier injection which occur in integrated circuits are some of the conditions which causes the failure rate in these circuits to increase [39].

The issue of computer reliability has been a major concern with researchers looking for methods and solutions for improving resilience and reliability particularly in dealing with embedded systems as they have limited resources in terms of hardware, processor speed, memory size and power consumption. There is also a rising demand for embedded systems in life-critical or system-critical applications such as military, space, medical or

even automotive industries where an error or a sudden breakdown of a system may cause catastrophic results. Various means have been developed to attain reliability in a system which can be grouped into four major categories [15, 16, 23]:

- Fault Prevention: Techniques that aim to prevent the introduction or occurrence of faults in the system;
- Fault Tolerance: Techniques that aim to ensure the system continues to function correctly in the presence of faults;
- Fault Removal: Techniques that aim to reduce the number of faults which are present in the system; and
- Fault Forecasting: Techniques that aim to estimate how many faults are present in the system, possible future occurrences of faults and the consequences of faults.

The techniques in each category can be applied on their own or used in combination with techniques from other categories to attain reliability. The aim of this thesis is to predict potential failure through detection of anomalous behaviour in the system. This is to ensure that the reliability of a safety-critical embedded system is not compromised in the presence of faults. Fault tolerance techniques are used to achieve the aim of this thesis as fault tolerance is aimed at failure avoidance [15,39]. A fault tolerant system can be achieved via fault avoidance, fault masking, detection of erroneous or compromised system operation, containment of error propagation, and recovery to normal system operations [39]. However, in this thesis, the focus is specifically towards online error detection, which will be discussed further in Section 2.4 and Section 2.5.

2.3 Threats to Reliability

The relationship between the system's function, behaviour, structure and service is important in order to understand how a reliability of a system can be threatened. As described in [15], the *function* of such a system refers to what the system is tasked to do and is described by the functional specification in terms of functionality and performance. The *behaviour* of a system is what the system does to implement its function and is defined as a sequence of states (e.g., computation, communication, stored information, interconnection, and physical condition), and the *structure* of a system is what enables the system to generate the said behaviours. From a structural viewpoint, a system's structure is composed of a set of components (e.g., hardware modules, software modules or other systems) which are bound together to interact and to provide a service. The *service* delivered by the system is the behaviour of the system as perceived by the receiver.

The reliability of a system can be threatened at any stage – function, behaviour, structure or even at the service stage. The following terms are crucial as it relates to how anomalous behaviour in a system occurs, and thus, it is imperative that these terms be clearly defined. The four types of threats to reliability are *defects*, *faults*, *errors*, and *failures*. Although they give the message that something is not right, there exists a difference between each type. The difference between a defect, a fault, an error and a failure given below is extended from [41]:

The difference between a defect, a fault, an error, and a failure is that, in the case of a defect, the problem occurred on the physical level; in the case of a fault, the problem occurred on the functional level; in the case of an error, the problem occurred on the computational level; and in the case of a failure, the problem occurred on a system level.

Section 2.3.1 provides a detailed definition of defect, fault, error and failure, adapted from [15, 42].

2.3.1 Defect, Fault, Error and Failure

A *defect* is defined as the unintended difference between implemented hardware and its intended design [42]. Defects can be divided into process defects such as a bad etching or soldering, parasitic transistors, oxide breakdown, etc; material defects such as a broken pin, surface impurities, etc; and age defects such as dielectric breakdown, electro-migration, etc.

A *fault* is the logic level abstraction of a physical defect. A fault is used to describe the change in the logic function of a device caused by the defect [42]. Both defects and faults are the imperfections in the hardware and function respectively. Physical defects are modelled as logical faults to reduce the complexity of fault simulation. Logical faults may be in the form of static (e.g. shorts, breaks), dynamic (components out of specification, timing failures) or intermittent (environmental factors). Some examples of faults could be a frozen memory bit, a stuck-at fault, an uninitialised variable in software, or a bit flip due to an alpha particle hit or cosmic ray ionisation.

An *error* occurs at the behavioural stage of the system where a part of the total state of the system deviates from its correct state, and hence, it may subsequently lead to service failure [15]. An error happens when the result of a computation is inaccurate due to a fault that was present in the system. One example of an error could be when the system is trying to access a portion of the memory that may have been hit with a fault, thus resulting in incorrect output due to a computational flaw. However, not every fault causes an error. An error is created by a fault that is active.

A *failure* is defined as "an event that occurs when the delivered service deviates from correct service" [15]. In other words, a failure is said to have occurred when the system has failed to implement its intended function.

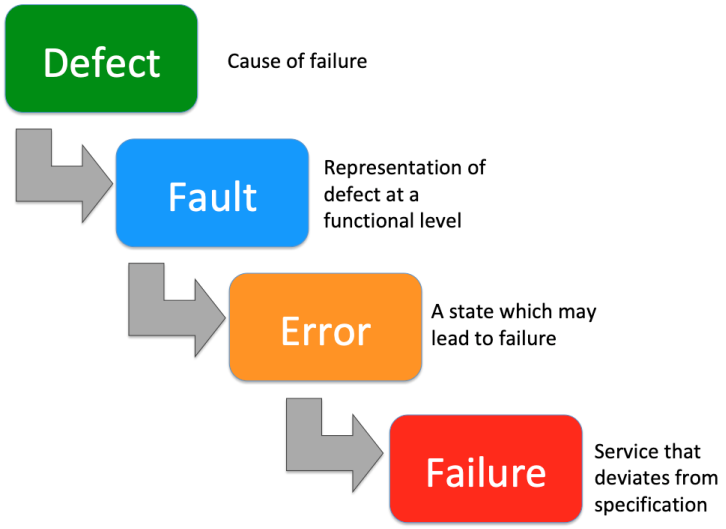


FIGURE 2.2: Relationship between fault, error, failure

Figure 2.2 above shows the relationship between defect, fault, error and failure. Faults are logical abstraction of physical defects. Faults will develop into errors, and multiple errors may cause a system to fail. In other words, faults are the reason for errors, and errors are the reason for failures. From the user point of view, the system has failed but the reason for failure may be unknown to the user. However, there could be some anomalous behaviour in the system prior to a failure which can be observed and detected.

2.3.2 Origin of Faults

As mentioned in Section 2.3.1, a failure is caused by the presence of one or more errors, and an error is caused by the presence of a fault. A fault is a representation of a physical defect and a defect could happen at any of these stages: during the design stage or during the operational stage. In the design stage, a defect could be caused by problems arising during specification, implementation or even the fabrication process [16]. During the operational stage, a defect could happen due to both internal and external factors, such as component degradation, external phenomena such as temperature and voltage variation, electromagnetic disturbances that result in single event effects [23] or malicious attacks by users [16].

2.3.3 Duration of Faults

The duration of a fault can be classified as permanent, transient or intermittent according to [16, 43].

- A *permanent fault* is a fault that is persistent or remains active; that is, it continues to exist until the faulty component is repaired or replaced. Typically, this type of fault is caused by defective components such as broken wire, an incorrect bonding, etc during the manufacturing process. It could also be caused by a component that is starting to degrade.
- A *transient fault* is a fault that occurs temporarily and does not leave any permanent damage on the chip. It is usually caused by environmental conditions such as radiation or noise. It happens randomly and therefore, these faults are hard to detect.
- *Intermittent faults* are faults that occur for few cycles, and apparently at random, then vanish, and then reoccur again, then vanish again. Unstable or ageing hardware that got activated by a change in environmental condition (e.g. temperature change) are usually the cause of these faults.

2.3.4 Fault-Error-Failure Chain

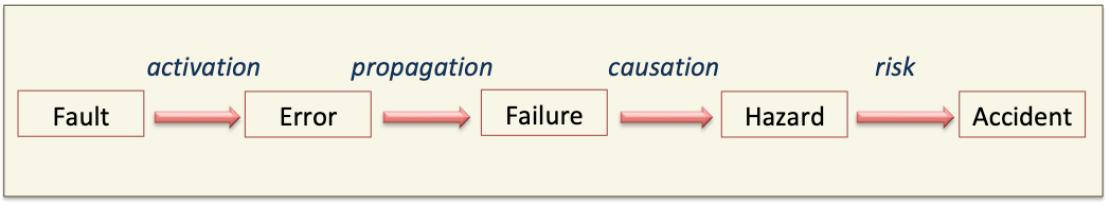


FIGURE 2.3: The fundamental chain of threat in a system, [44]

Figure 2.3 shows how the activated chain of threats of fault-error-failure may lead to an accidental situation which in turn risks the human life. Barton, Christian, Siewiorek, Koopman and Kaliorakis [45–49] have identified the various type of failures that can be observed by users due to the activation of faults in the system which can be classified in the following Table 2.1. These failures can be grouped into four categories, namely: (a) *System Failure*, (b) *Application Failure*, (c) Output Failure and (d) *Silent Failure*. A *System Failure* is where the Operating System (OS) has stop functioning properly or crashed. An *Application Failure* happens when a system ceases to respond to input, but the OS is still running and responding. *Output Failure* is where there is a mismatch between the actual output and the expected output while *Silent Failure* is where the failure is masked or was not noticeable by the user.

Table 2.1: Comparison of Failures Type

Categories of Failure	**Barton (1990)**	**Christian (1991)**	**Siewiorek (1993)**	**Koopman (2000)**	**Kaliorakis (2015)**
System Failure	Machine Crash Task Stop	Crash	Crash Abort (crash with abort message)	Catastrophic Abort	Crash Assert
Application Failure	Response too late	Timing (Early, Late)	Late Response (Timeout)	Restart Hindering	Timeout
Output Failure	Invalid Output	Incorrect Response (value, state)	Failure (Incorrect answer)	-	Detected Unrecoverable Error
SIlent Failure	No Error	Omission	Results returned OK (no error detected)	OK Silent	Masked Silent Data Corruption

Error detection is the vital aspect in fault tolerance because a processor cannot tolerate a problem that it is not aware of. Even if the processor cannot recover from a detected fault, it can still alert the user that an error has occurred and halt. Thus, error detection provides, at the minimum, a measure of safety [50]. Online error detection is the ability to detect any form of violation of system specifications during run-time. However, due to the increased complexity of processors, inadequate pre- and post-silicon testing and verification as well as pressure to reduce time-to-market, major design bugs or faults can still exist even after the chip is in operation [51–53]. Hardware defects due to variations in temperature and voltage, components degradation and Single Event Upsets (SEU) also contribute to the increased presence of faults in a chip. Thus, various online error detection schemes are applied to ensure these less-than-perfect chips can still operate effectively.

Online error detection is mainly applied to protect the CPU cores, memory hierarchy control logic and interconnection logic, whereas memories such as caches and register files are well protected using well-known error-correcting codes (ECC) [53]. Section 2.4 and Section 2.5 will describe the techniques that have been applied for online error detection, namely (a) *Anomaly Detection*, (b) *Built-In-Self-Test (BIST)*, (c) *Redundancy*, and (d) *Dynamic Verification*.

2.4 Anomaly Detection

2.4.1 Definition of an Anomaly

The failures listed in Table 2.1 occur when a fault has manifested itself an error and causes some high-level behaviour to be anomalous. Anomalous behaviour, or in short, anomalies, are behaviours that do not conform to a normal, expected pattern which can also be identified as outliers, exceptions, peculiarities, contaminants or other terms according to the domain being studied [26]. Chandola [26] has also categorised anomalies into three different structures, namely (a) point anomalies, (b) contextual anomalies, and (c) collective anomalies.

- *Point anomalies*: An individual data instance is deemed to be anomalous with respect to the rest of the data if it is "too far" from the rest of the data. Figure 2.4 shows an example of point anomalies where points O1 and O2 are deemed anomalous with respect to S1 and S2 as these points are located "too far" from the rest of the data.
- *Contextual anomalies*: An individual data instance is deemed to be anomalous in a specific context, but not otherwise. This type of anomaly is common in time-series data. Figure 2.5 shows an example of a contextual anomaly, T2. The graph shows

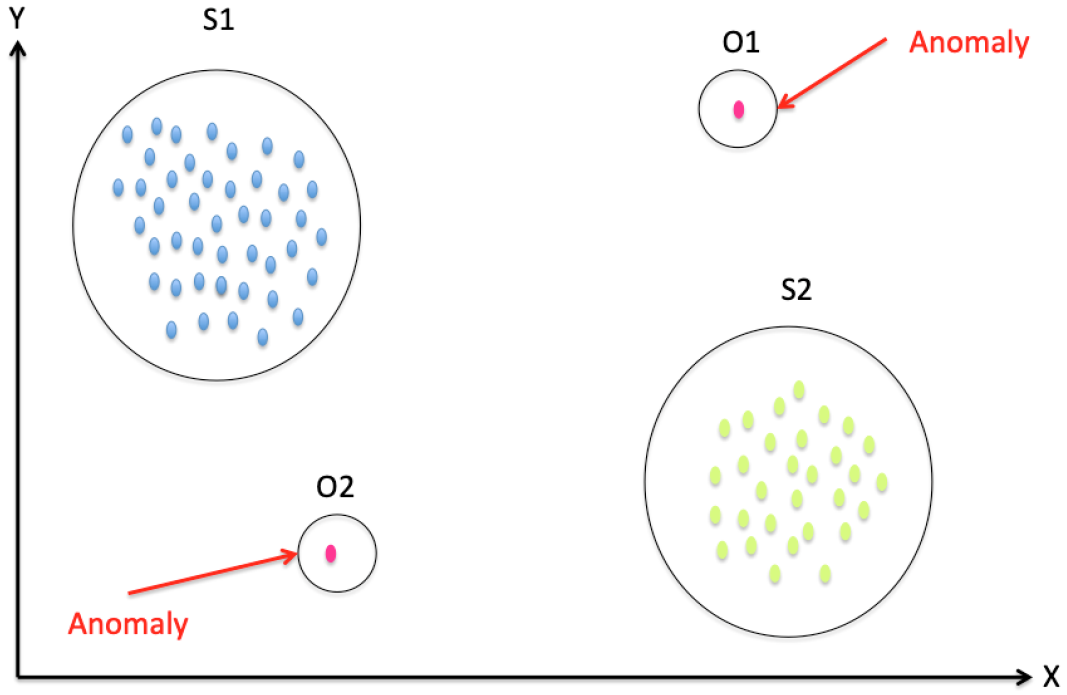


FIGURE 2.4: Point O1 and Point O2 are point anomalies as they deviates far from the rest of the data.

the average monthly temperature collected at Southampton from January 2017 until December 2018 obtained from Southampton Weather Station [1]. The value of T2 bears similarity to the value of T1, but the low temperature of T2 happened during the summer period instead of the winter period, hence it is considered anomalous.

- *Collective anomalies*: Individual data instances may not be anomalies themselves, but their occurrence together as a collection with respect to the rest of the data is deemed anomalous. Figure 2.6 shows an example of a collective anomaly corresponding to an Atrial Premature Contraction in a human electrocardiogram output. The value -5.5 as shown in 2.6 is not a value that deviates far from the rest of the data, but because it occurs consecutively for a period, hence it is considered anomalous.

As we will show in Chapter 3, the anomalies in this study are of the collective type, where individual data instances in a collective anomaly may not be anomalies themselves but their occurrence together as a collection is considered anomalous. The presence of collective anomalies in the system could potentially lead the system to failing to execute correctly and these failures can be classified as masked errors, silent data corruption, affected output, crash or hang as shown in Table 2.1.

[1] `http://www.southamptonweather.co.uk`

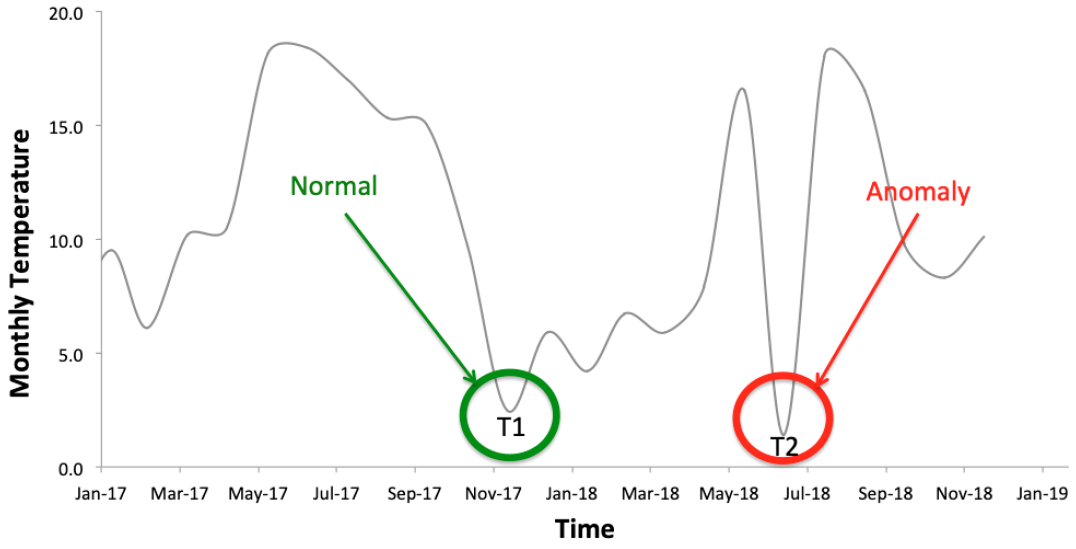


FIGURE 2.5: An example of contextual anomaly, $T2$ in a monthly temperature time series. Note that the temperature at $T2$ is similar to $T1$, but because it occurs in the month of July instead of November, hence it is considered anomalous.

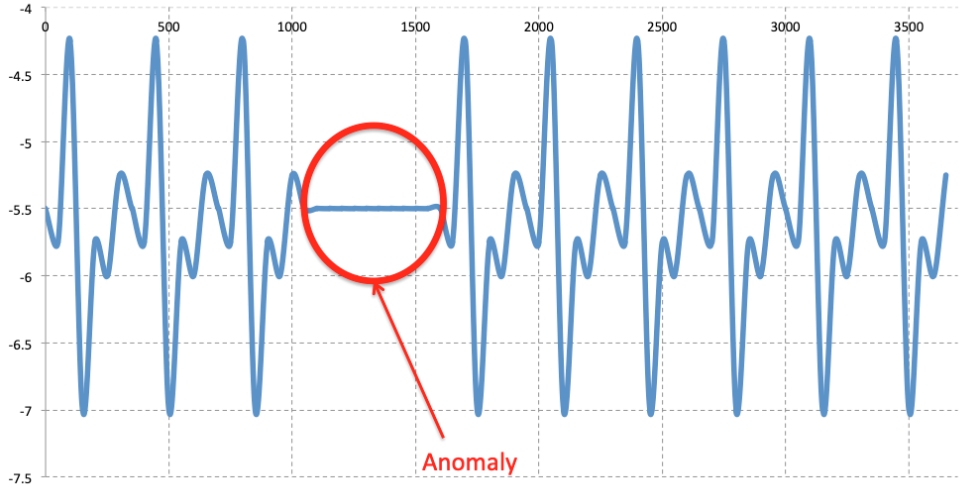


FIGURE 2.6: An example of collective anomaly corresponding to an Atrial Premature Contraction in a human electrocardiogram output

Anomaly detection is one of many approaches used for online error detection in a microcontroller core. It is about detecting "likely errors" by detecting anomalous behaviours. Some example of detectable anomalies in a microcontroller are unusual data values, branch mispredictions, exceptions, page faults, crashes, etc.

2.4.2 Anomaly Detection Techniques

Due to the nature of data and the type of anomalies that occur in different application domains, applying an anomaly detection technique developed for one domain into another domain is not a straightforward task [26]. Availability of labels in the dataset also plays an important role in deciding the type of techniques to be used. There are numerous techniques proposed for anomaly detection such as Replicator Neural Networks (also known as Autoencoders), One-Class Support Vector Machines, Bayesian Networks, Hidden Markov Models (HMMs), K-based Nearest Neighbours, Fuzzy Logic-based techniques and many more. Choosing an appropriate technique for anomaly detection depends very much on the available dataset as illustrated in Figure 2.7. In general, anomaly detection techniques can be divided into three broad categories as below [54]:

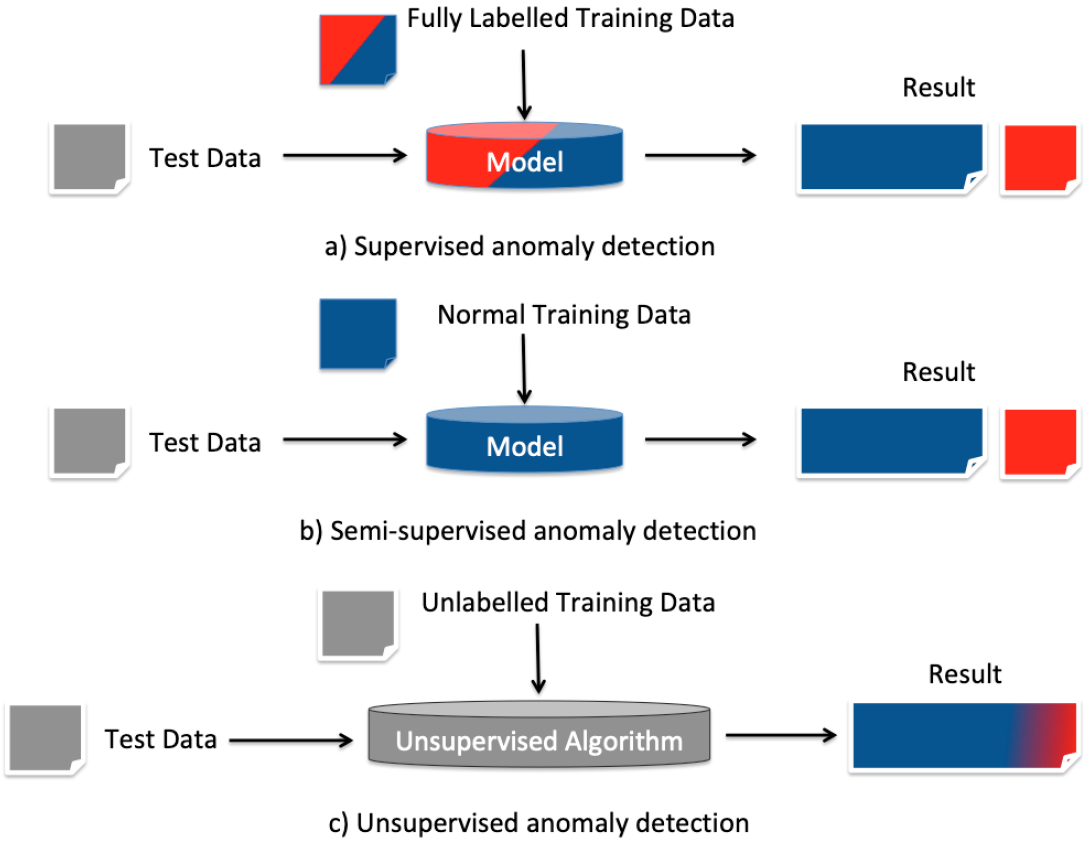


FIGURE 2.7: Different anomaly detection modes depending on the availability of labels in the dataset. (a) Supervised anomaly detection uses a dataset that contains both normal and anomalies for training. (b) Semi-supervised anomaly detection uses a "normal behaviour" training dataset. (c) Unsupervised anomaly detection algorithms use unlabelled dataset for training.

- *Supervised anomaly detection*:
  Techniques that fall under the category of supervised anomaly detection require the available samples in each data set to be labelled as either "normal" or "abnormal". A common approach involves training a classifier and building a predictive model

for both normal and abnormal class. New, unobserved data is then compared to the model to determine which class it belongs to. Some widely used techniques for supervised anomaly detection are Support Vector Machine (SVM) [37], decision trees, logistic regression, multi-layer perceptron networks [55] and linear regression. However, there are some issues using this approach as discussed in [26, 55]. Firstly, the available data are imbalanced as anomalous data is harder to obtain and less frequent compared to normal data. Imbalanced data causes over-fitting and the trained model lacks generalisation. Another issue is the challenges faced in obtaining labelled data, especially for anomalous data. Labelling the data requires a human expert, and it is a costly exercise, not to mention time-consuming, to obtain a huge amount of all possible samples of anomalous data.

- *Semi-supervised anomaly detection*:
  For techniques that fall under this category, it involves building a model representing normal behaviour from a given *normal* training dataset [56]. As it does not require labels from anomalous class, this approach is more widely used compared to supervised techniques. Unknown samples are classified as outliers when their behaviour is far from that of the known normal samples. Common techniques for semi-supervised anomaly detection include statistical-based techniques [33, 34, 57, 58], one-class classifiers [59], cluster-based techniques [60], probability density function and others.

- *Unsupervised anomaly detection*:
  Techniques for unsupervised anomaly detection do not require any training data. The data samples in the training set could contain both normal and anomalous data. However, it makes the assumption that there are more normal data instances compared to anomalous data in the training set. If the assumption is incorrect, then these techniques suffer from high false alarm rate [26]. Clustering-based techniques [61], Hierarchical Temporal Memory (HTM) networks [35], Principal Component Analysis (PCA) [62], one-class SVM and Self-Organising Map (SOM) are some techniques used for unsupervised anomaly detection.

Supervised techniques require both normal and anomalous data in building a model. Imbalanced data will give a low accuracy to the model. As the amount of anomalous data available from the dataset is less than 10% while the majority of the data consist of points that depict a normal behaviour, supervised techniques for anomaly detection are not suitable and thus, will not be considered in this thesis. Unsupervised techniques can be effective only if the assumption of having more "normal" data holds true. Else, a high false detection will occur. Semi-supervised techniques which use a normal training dataset to build a model is a more balanced approach compared to supervised and unsupervised techniques. According to [63], using this approach, it is possible to achieve high detection rate with good accuracy. Although there exist numerous techniques for

semi-supervised anomaly detection, not all techniques are suitable for detecting anomalies online and in real-time [35]. Most of the techniques used in real-time streaming time series data are statistical techniques that are computationally lightweight, as one of the main requirements are the ability of the algorithm to learn continuously without storing the whole stream of data [64].

2.4.3 Anomaly Detection in Damage Detection Domain

Anomaly detection is applied in various domains such as fraud detection, intrusion detection, medical and public health anomaly detection, industrial damage detection, sensor networks, image processing and many more [26]. As the interest of this thesis is the application of anomaly detection in the domain of industrial damage detection, the background review will be limited to the damage detection domain.

Industrial damage detection refers to detection of different faults and failures in complex industrial systems, structural damage, intrusions in electronic security systems, suspicious events in video surveillance, abnormal energy consumption, etc that may result from deterioration and breakage due to continuous usage and normal wear and tear. Such damage needs to be detected early to prevent further escalation and losses, as well as reduce risks to users. The fundamental question in a damage detection problem is whether a fault is present [65]. The problem is simply to identify from measured data if a machine or structure has deviated from normal condition, i.e., if the data is anomalous. Damage detection is mostly applied either in detecting faults in mechanical components or detecting defects in physical structures.

The observed data in this domain has a temporal aspect because it is observed and recorded with respect to the passage of time [26, 66]. Typically, an array of sensors is used to collect measurements either continuously or at a regular time interval [67, 68]. The type of anomalies that are found in this domain are anomalies that occur mostly because of an observation in a specific context (contextual anomalies) or as an anomalous sequence of observations (collective anomalies) [26]. Semi-supervised techniques are usually applied in damage detection as datasets describing the "normal behaviour" are readily available. Of the many approaches to the problem, some are drawn from condition monitoring, others from the field of pattern recognition and yet others from univariate and multivariate statistics. The latter field has a very substantial body of theory to support it and is proving to be a fruitful source of algorithms for damage detection [69].

In [70], Autoassociative Kernel Regression (AAKR) is used to model the normal behaviour using a multi-sensor monitoring observation to produce the healthy baseline of engine monitoring. On-line detection is then performed to detect any abnormal conditions based on deviation from the baseline model. The results show some anomalies

are observed in the cooling system and a pre-warning is generated. In [71], the Least Squares Support Vector Machine (LS-SVM) algorithm is used in its on-line and non-invasive anomaly detection system to continuously monitor the sensors and hardware components via flight data in an Unmanned Aerial Vehicle (UAV) while in [72], the LS-SVM is used to monitor for collective anomalies in satellite telemetry data. These are just some examples of research in damage detection where the same concept is applied – first a model of normal or expected behaviour is produced, then the model is used to predict the next data in the series. Once the actual data is available, it is used to compare with the predicted data. If the deviation is above a threshold, the data is deemed anomalous. The key difference in this type of research lies in the type of techniques used by the model to predict the next data in the series and the measurement of deviation between predicted data and observed data.

The application of anomaly detection in industrial damage detection domain was basically meant for mechanical components or physical structures, and there is no research so far on applying damage detection in electronic components, in particular, to a microcontroller. Existing anomaly detection approach for error detection in a microcontroller is developed based on the concept that every activated fault must manifest itself as some form of anomalies, and by monitoring these anomalies, the error, and subsequently the fault can be identified. A good example of this approach is SoftWare Anomaly Treatment (SWAT) [73] and subsequently, mSWAT [74] that were both developed using zero to low-cost hardware and software monitors to monitor and detect anomalous software behaviour such as *fatal traps*, *hangs*, *high-OS*, and *panic*. Once an anomaly is detected, the control is then transferred to the firmware to invoke a diagnosis process that will distinguish between transient faults, permanent faults and bugs. Although error detection on mSWAT has minimal overhead, the diagnosis component itself can incur a high overhead [74], which may not be very beneficial for an embedded system. Furthermore, this method is only able to detect the anomalous behaviour as it happens but gives no indication of forthcoming anomalies which may happen. In the case of detecting anomalies in an embedded system, the detection has to be performed online and in real-time as early detection of possible failure in the processor is important to minimise or reduce potential risk and damage.

2.5 Other Online Error Detection Techniques

Other techniques that have been explored for online error detection in a processor include *Built-In-Self-Test (BIST)*, *Redundancy*, and *Dynamic Verification*.

2.5.1 Built-In-Self-Test

Another category for error detection is based on the traditional Built-In-Self-Test (BIST) mechanisms. BIST is a design-for-testability technique that includes additional hardware or software features into integrated circuits to allow them to perform self-testing. It was originally used for manufacturing testing as there is a huge saving concerning the amount of time for testing as compared to testing using an external automated test equipment, and it is less costly. In a generic BIST scheme in testing mode, the test pattern generator applies a series of test patterns to the circuit under test and the test responses are evaluated by the response monitor. The test responses are then compacted to form a signature to be compared with the reference signature stored on-chip. If there is any discrepancy between the two signatures, an error signal is sent. The basic architecture of BIST is as shown in Figure 2.8.

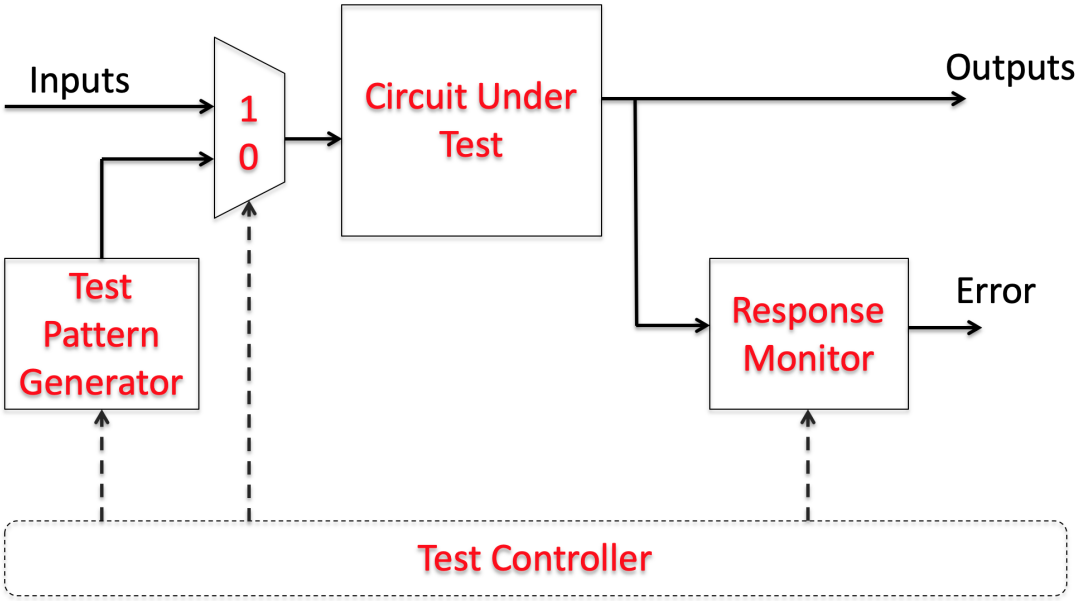


FIGURE 2.8: Basic architecture of BIST

BIST could come in the form of hardware or software and is performed either during idle time or executed periodically. Hardware-based BIST requires extensive and manual design changes as it uses dedicated hardware to generate the test patterns. However, this causes an increase in the circuit area and degrades its performance [75]. There are four primary concerns in BIST-based approaches – (a) fault coverage; (b) size of test set; (c) hardware overhead; and (d) performance penalty [76]. To address the hardware overhead and performance degradation in a hardware BIST, software BIST is proposed as it is non-intrusive and has a low test overhead [77]. Software BIST does not require any change in the design but utilises existing processor resources and instructions to perform self-testing [51,75]. However, software BIST has a longer testing time compared to hardware BIST [78].

Unfortunately, BIST has the limitation that it is only able to detect permanent faults [51, 53]. Newer approaches of software-based self-testing from [79] utilise the multiple cores in a multiprocessor to run parallel test programs in a scheduled manner. However, the experiment revealed that running the test program on multiple cores causes serious performance loss as it creates a congestion for shared hardware resources. [52] provided a solution to run the test program in parallel, however, it is noted that the existing workload has to be suspended in all the cores to allow the cores to be synchronised to execute the test program in parallel. This limitation is also agreed by [80] where test programs will not only have to share processor resources but are also faced with limited memory resources and this could limit the effectiveness of the usual methodologies used for writing test programs.

2.5.2 Redundancy

Another category of error detection uses redundancy. Redundancy is a common technique used to achieve fault tolerance, and can be achieved via hardware or software. According to [16], redundancy is the provision of additional functional capabilities that can either be a replicated hardware component, an additional check bit attached to a string of digital data, or a few lines of program code verifying the correctness of a program's result. There are four types of redundancy used in fault tolerant systems, which are (a) hardware redundancy; (b) time redundancy; (c) information redundancy; and (d) software redundancy.

2.5.2.1 Hardware Redundancy

Hardware redundancy can be divided into passive or active redundancy [16]. Active hardware redundancy such as dual modular redundancy (DMR) refers to two identical hardware modules performing the exact same task, but only the output from one module will be used, while the output from the other module is not used. The module which the output is usually used is the main module, while the other module is termed as the backup module. Both hardware modules are equally powerful to ensure that the system will not suffer any performance degradation. Figure 2.9 illustrates the error detection using active hardware redundancy called duplication with comparison. The voting logic in the comparison module will compare the results between the main module and the backup module. If the results are the same, no error has occurred. However, if the results are different, a mechanism to switch the output from the main module to the backup module is used. The most challenging aspect in DMR is to determined when to switch the output from the main module to the backup module. A more reliable form of voting logic involves an odd number of three devices or more, such as in triple modular redundancy (TMR) and N-modular redundancy. All perform identical functions and the outputs are compared by the voting logic. The voting logic establishes a majority when

there is a disagreement, and the majority will act to deactivate the output from other module(s) that disagree.

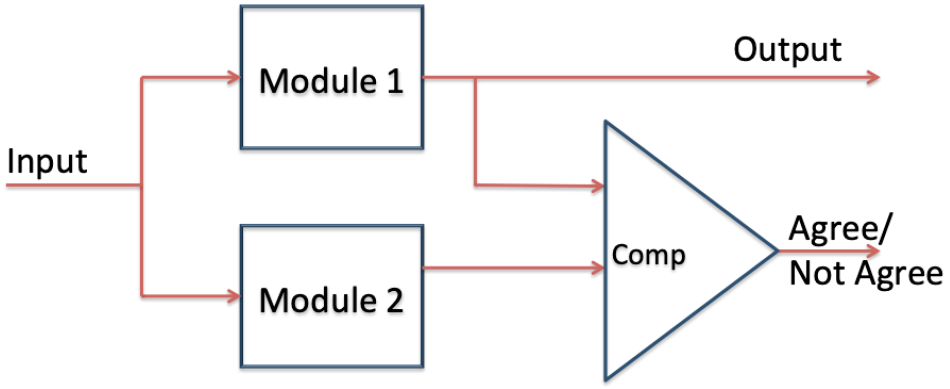


FIGURE 2.9: Active hardware redundancy

A passive hardware redundancy means the backup module is in idle mode until the main module fails or break down. When the main module has failed, the backup module takes over the task. In passive hardware redundancy, the switching mechanism is required to switch both the input and output from the main module to the backup module. Both active and passive hardware redundancy have high cost and introduce high overheads, both of which are unacceptable in a microcontroller. Not only it comes with high hardware overheads, but the additional voting step in active hardware redundancy may increase the runtime [81].

2.5.2.2 Time Redundancy

Hardware redundancy has a huge impact on physical entities such as cost, weight, size, power consumption, etc [16, 82]. An alternative to hardware redundancy is time redundancy, where certain operations, computations or data transmissions are repeated on the same module and the results are compared with a stored copy of the previous results. Time redundancy can be used to distinguish transient faults from permanent faults. If it is a transient fault, the fault disappear after re-computation, else it is a permanent fault. Therefore, time redundancy by means of task re-execution is a common technique to mitigate soft errors at system level [83]. Although time redundancy reduces the amount of hardware required, the total time and the active energy consumption are doubled because twice as much work is performed [50]. Figure 2.10 shows how a fault can be detected in the processor at the expense of time.

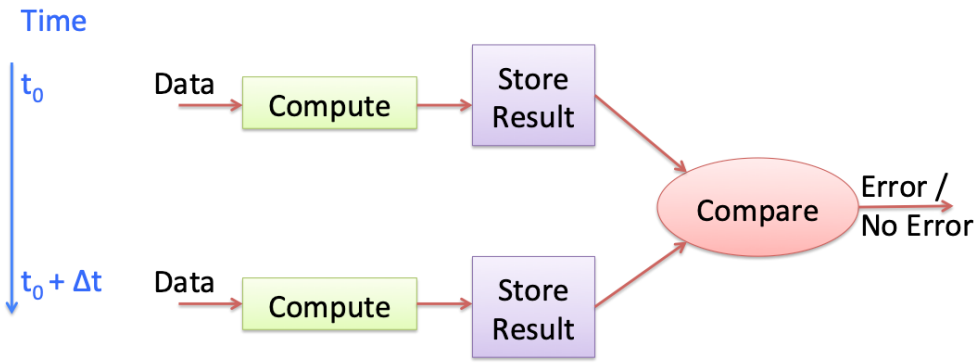


FIGURE 2.10: Time redundancy

2.5.2.3 Information Redundancy

Information redundancy, according to [82], is the addition of extra information to data to allow error detection and correction. The addition of extra information means adding redundant bits to a datum to detect when it has been affected by an error. This is also known as encoding, according to [16]. This includes error-detecting codes (EDC), error-correcting codes (ECC), and self-checking circuits. The data will be encoded into a code word during the encoding process. The data is recovered from the code word during the decoding process. If there is a mismatch between the decoded data and the expected data, an error has occurred. Error detection by means of information redundancy requires additional hardware resources because the encoded data and programs require a larger amount of memory for storage. Another downside of using encoding for error detection is the increase of runtime, as shown in [81].

2.5.2.4 Software Redundancy

Software redundancy techniques largely leverage on the experience of hardware redundancy. For example, in N-version programming, the program is written N times, and all N programs are executed in parallel with majority votes taken, much like having N-modular redundancy. However, just like N-modular redundancy being an expensive effort, N-version programming is equally costly and difficult to maintain. For example, Error Detection by Duplication (EDDI) [84] or SWIFT [85] duplicates all instructions and has 100% overhead in performance. A recent approach in [86] uses a data-flow graph where a redundant instance is inserted into the graph. The output from the execution of both the redundant instance and the original instance is compared. Although it is flexible and allows for parallelism, it is noted that benchmarks that stress the memory bus show performance degradation while other benchmarks have recorded up to 23% overheads.

A combination of full code duplication with selective comparison, as presented in [87], is aimed to improve the fault detection ratio and decrease the imposed overhead. While it did improve the code size overhead and execution time overhead of the combination methods by 43.5% and 22.2%, respectively, compared to a full instruction duplication method, it was less sensitive in detecting the errors caused by an injected fault. In [88], instructions-based TMR was implemented to mitigate against space-borne single event effects on processor architectures. However, the overhead of instructions-based TMR was 10.32%.

The emergence of simultaneous multi-threading (SMT) and multi-core processors (CMP) also saw techniques such as Redundant Multithreading gaining wide interest. For example, in [89, 90], two copies of the same program are run on separate threads and the outputs are compared. Although this is successful in detecting faults, it cannot be applied in embedded systems, as embedded systems have limited resources for redundant multi-threading.

The REPAIR architecture, introduced in [91], is targeted at detecting hard errors in a multicore processor. It first identifies instructions that will use or have used faulty processor structures. These potentially incorrectly executed instructions are routed to an Instruction Re-Execution unit (IRU) to be re-executed and the results are compared. The results from the IRU will be applied if the results from the core and IRU do not match. However, the comparison is done post execution, where the pipeline operations are halted pending the re-execution. On the availability of the results, the necessary registers are updated and the regular operation of the processor pipeline continues. There is also an issue of loss of performance from stalling the instructions while awaiting the duplicated results, which may not be favourable in safety-critical systems. To overcome this problem, PreFix was introduced in [92]. It is similar to REPAIR, but re-execution of faulty instructions are reduced substantially because in PreFix, these faulty instructions are predicted and verified beforehand. With this technique, a faulty core is allowed to continue its operation on the assumption that the second core running duplicated instructions is healthy and free of error. The issue arises when the secondary core is faulty and the results from running the duplicated instructions cannot be trusted.

2.5.3 Dynamic Verification

Dynamic verification is another form of online error detection that operates during runtime execution of the software. It differs from software BIST and software redundancy as the behaviour of the software is observed dynamically using dedicated hardware checkers to verify selected high-level invariants. Invariants are program properties that must be preserved when the code is modified and can be classified into preconditions, post conditions and loop invariants. Thus, the key to dynamic verification is identifying the invariants to check. The selection of high-level invariants that define the correct

behaviour such as control-flow checking, dynamic [93] or static data-flow checking, computation results, software invariants [94] or any mixture of these [95] are pre-identified. However, this method of using invariants for error detection incurs additional area mainly due to history fields and signature computation logic used for data-flow and control-flow checking. For example, in [95], there is a 16.6% increase in area of the core component alone and the decrease in performance was measured to be around 3.2% average.

2.5.4 Summary of Existing Online Error Detection Techniques

Existing online error detection techniques such as redundancy-based techniques, BIST or dynamic verification have high overheads or require more resources than an embedded processor can offer. Anomaly detection techniques have proved to be a promising approach for online error detection as they incur low overheads [73, 74], and this approach will be used to detect anomalous behaviour in a system prior to a failure. Hardware performance counters (HPCs) are a valuable tool for detecting anomalous behaviour in a processor [96]. Section 2.6 provides an overview of HPCs as well as existing applications that use HPCs.

2.6 Hardware Performance Counters (HPCs)

2.6.1 Overview of HPCs

Most modern processors have special, on-chip hardware that can monitor performance known as *Hardware Performance Counters* or HPCs. HPCs are sets of special-purpose counters built into processors such as Intel Pentium, ARM, Cray, PowerPC, Ultrasparc and MIPS architectures [97] to track low-level Performance Monitoring Events (PMEs) within the processors, such as the number of cache misses, the number of instructions retired, the number of branch instructions retired in real-time.

HPCs are part of the wider Performance Monitoring Unit (PMU) built into most modern processors. A PMU consists of two components: performance event select registers and event counters. A counter is paired with an event select register to monitor a particular PME. In an Intel x86 processor, the performance event select registers are known as model specific registers (MSRs) [1], and for an ARM Cortex-A series processors, the registers are controlled through the event bus, PMUEVENT [2]. The PMU is interrupt-based, such that an interrupt is generated after a certain interval of time or the number of occurrences of the desired event exceeds a predefined threshold. In other words, it is possible for PMU to do either time-based sampling or event-based sampling. The counters are incremented on an instruction-by-instruction basis, thus ensuring accurate

results [98,99]. As these counters are built-in, there are no additional overheads to access the enormous information available in the CPU.

The number of available counters in each processor is limited and the available PMEs differ from one processor to another due to architectural differences. The number of available counters limits the number of PMEs that can be monitored in real-time. For example, an Intel Atom has only two programmable performance-monitoring counter registers per processor core. This means that only two PMEs can be monitored simultaneously. Therefore, it is not practical to utilise more microarchitectural events than the number of available counters to achieve high accuracy as it requires executing the application multiple times, since the hardware can only count a small subset of events concurrently [100]. It has been shown, however, that a single counter is sufficient to describe the behaviour of a program [101, 102]. Table 2.2 shows the number of available counters and the number of available PMEs for some common processors such as Intel, ARM, POWER4, and UltraSparc II. The total available counters in the processor and number of available PMEs are taken from the technical reference manual of each processor.

TABLE 2.2: Number of available counters and events for some processors

Processor	**Number of Available HPCs**	**Number of Available PMEs**
Intel Atom [1]	2 + 3 (fixed functions)	129
Intel Core i7 Nehalem [1]	4 + 3 (fixed functions)	129
ARM Cortex-A9 [2]	6	57
POWER4 [103]	8	>100
UltraSparc II [104]	2	>4 bil

Table 2.3 and Table 2.4 list some PMEs that can be observed from an Intel® and ARM architecture.

TABLE 2.3: Pre-defined architectural performance monitoring events for Intel® architecture [1]

Bit Position CPUID.AH.EBX	**Performance Monitoring Event (PME) Name**
0	UnHalted Core Cycles
1	Instruction Retired
2	UnHalted Reference Cycles
3	LLC Reference
4	LLC Misses
5	Branch Instruction Retired
6	Branch Misses Retired

TABLE 2.4: Examples of performance monitoring events for ARM architecture [2]

Name	Event Number	Description
PMUEVENT[0]	0x00	Software increment
PMUEVENT[1]	0x01	Instruction cache miss
PMUEVENT[2]	0x02	Instruction micro TLB miss
PMUEVENT[3]	0x03	Data cache miss
PMUEVENT[4]	0x04	Data cache access
PMUEVENT[5]	0x05	Data read
PMUEVENT[6]	0x06	Data writes
⋮	⋮	⋮
PMUEVENT[56]	0xA4	PLE FIFO Overflow
PMUEVENT[57]	0xA5	PLE request programmed

2.6.2 Application of HPCs

HPCs were originally designed to be used as hardware verification or debugging tools for performance analysis or tuning purposes [105], but have since been used for performance evaluation [98,106], workload estimation [107], detection of malicious activities [96,100–102,108–111], integrity checking [99] and anomaly detection [37,74].

For example, in [109], the authors proposed BRAIN, which stands for BehaviouR based Adaptive Intrusion detection in Networks and that uses statistics gathered from hardware, network and application to detect and mitigate Distributed Denial of Service (DDoS) attacks on an application. The HPCs that form the hardware in BRAIN are used to characterise the host behaviour during load and attack. The result shows that by correlating the HPCs with network statistics and application statistics can successfully detect DDoS with high accuracy, low cost and performance overheads. In [112], the authors present NumChecker, a Virtual Machine Monitor (VMM) based framework that securely and efficiently monitors the execution of system calls to detect kernel rootkits by leveraging on existing HPCs. In [110], ConFirm is a low-cost technique that uses HPCs as a signature to verify the execution of the computational paths in order to detect malicious modification of firmware in embedded control systems.

Wang et al. [106] use HPCs to monitor and quantify the interference between virtual machines located in the same host and competing for shared physical resources. Using Last Level Cache (LLC) miss-rates, one of the counters available, the data is fed into the interference prediction model to predict performance degradation between virtual machines and through the information gathered, it can determine which virtual machine is utilising most of the resources. Another example of how HPCs are used for performance monitoring is shown in [107] where the authors proposed to monitor L1 cache activity counters in order to estimate the workload and set the Dynamic Voltage and Frequency Scaling (DVFS) based on the estimated workload. This method resulted

in energy saving of 23% compared to the on-demand frequency setting policy used in Linux.

A common trait in all these works is that HPCs have been used to identify or detect some form of deviation from normal or expected behaviour. As explained in detail in Chapter 3, a single HPC is used to monitor the execution profile of a processor core running a workload. A system that behaves normally exhibits a certain profile, and any deviation from the profile indicates anomalous behaviour has occurred in the system.

2.7 Summary

This chapter presents the fundamental concepts and definitions in the scope of this thesis. The existing online error detection techniques require either some external monitors to be built-in or some means of redundancy. A dedicated hardware-based detector can be intrusive as it meddles with the rest of the hardware and a pure software detector, though unobtrusive, may be too slow to react. As well as the additional overhead, these techniques are only able to detect an error in the processor after a failure had occurred. This thesis addresses the gap where it is possible to use an HPC to detect anomalous behaviour and predict potential failure before the failure happens. Predicting potential failure in an embedded system is important to minimise or reduce potential risk. Since these systems usually operate continuously, the detection of anomalous behaviour has to be performed online and in real-time. By utilising the built-in HPC, the overhead incurred will be lower than for software profilers [100,112]. However, prior to performing online error detection, it is important to understand what constitutes normal behaviour of a system, and what causes the system to behave anomalously. A system that behaves normally exhibits a certain pattern, thus any behaviour that deviates from that normal pattern should be identifiable. Chapter 3 discusses how anomalous behaviour can be identified using HPCs and the results gathered from the experiment conducted also show a correlation between errors that occur and failures caused by these errors.

Chapter 3

Identification of Anomalous Behaviour using Hardware Performance Counter (HPC)

3.1 Introduction

Embedded systems typically perform routine or repetitive tasks, and anything that is repetitive has a pattern. Based on this understanding, a system that is behaving normally (i.e. functioning without any error) has a pattern, and thus, any abnormal behaviour that deviates from this normal pattern should be identifiable. The problem is to determine what kind of features in a system or which system components provide meaningful information about the system's behaviour. Appiah et. al in [113] and Zhai et. al in [114] have shown that it is possible to use Program Counters (PCs) as a feature to characterise a Central Processing Unit (CPU).

Several experiments were performed where the PCs were extracted to characterise the behaviour of the system running simple programs such as Queens, SQR Root and Angle Conversion. The amount of data generated using PCs as the monitoring feature is huge, amounting to 10 GB and over for each program. Not only that, the time involved to generate data was long, around one to two hours for a simple program. This suggest that using PCs to observe the behaviour of a system is not a feasible solution. An alternative to PCs is using Hardware Performance Counters (HPCs) to characterise normal behaviour. As presented in the previous chapter, these counters are built into a processor and record specific events that occur in the processor precisely and accurately.

In this chapter, identification of anomalous behaviour using HPC is presented. The main objectives of this chapter are as follows:

1. Investigate the type of events available to be monitored using an HPC. In particular, two events which are (a) the number of instructions retired, and (b) the number of cache misses will be investigated.

2. Investigate and determine a suitable sampling rate for HPCs. Several sampling rates are proposed and selection of an appropriate sampling rate will be presented.

3. Investigate the behaviour of HPC data when a fault has manifested itself as an error and subsequently, led the system into a failure. The behaviour of the HPC data with a manifested fault that leads to a failure, in particular, (a) a hang; or (b) a crash will be presented.

This chapter is organised as follows. The process of gathering anomalous and non-anomalous data is presented in Section 3.2. In Section 3.3 and Section 3.4, the selection of fault model and HPC event used in the experiment are presented. The selection of benchmarks for the experiment is presented in Section 3.5, while Section 3.6 explains the simulator used for the experiment. Analyses of the results are presented in Section 3.8, while correlations between a manifested fault (also known as an error) and a failure are presented in Section 3.9. Section 3.10 concludes the chapter.

3.2 Methodology

Similar to what has been done in [99, 109], the hardware counters are used to create execution profiles for several benchmarks based on the methodology shown in Figure 3.1. Briefly, the first two steps involve identifying the fault model to be used and the type of event to be monitored using an HPC. After identifying the benchmarks to be used in the experiment, the next step involves creating, executing and obtaining initial pattern from fault-free executable, which will form as the baseline patterns for the benchmarks. Fault injection is performed and anomalous behaviour from the system is observed through the counter data. The following Sections 3.3, 3.4, 3.5, 3.6 and 3.7 explained in detail the steps that have been taken.

3.3 Selection of Fault Model

The first step is to identify which fault model will be used in the experiment. There are several fault models that are typically used to investigate the effects of physical faults on higher levels of abstraction. The fault models outlined by Wang and Chattopadhyay [115] are as follows:

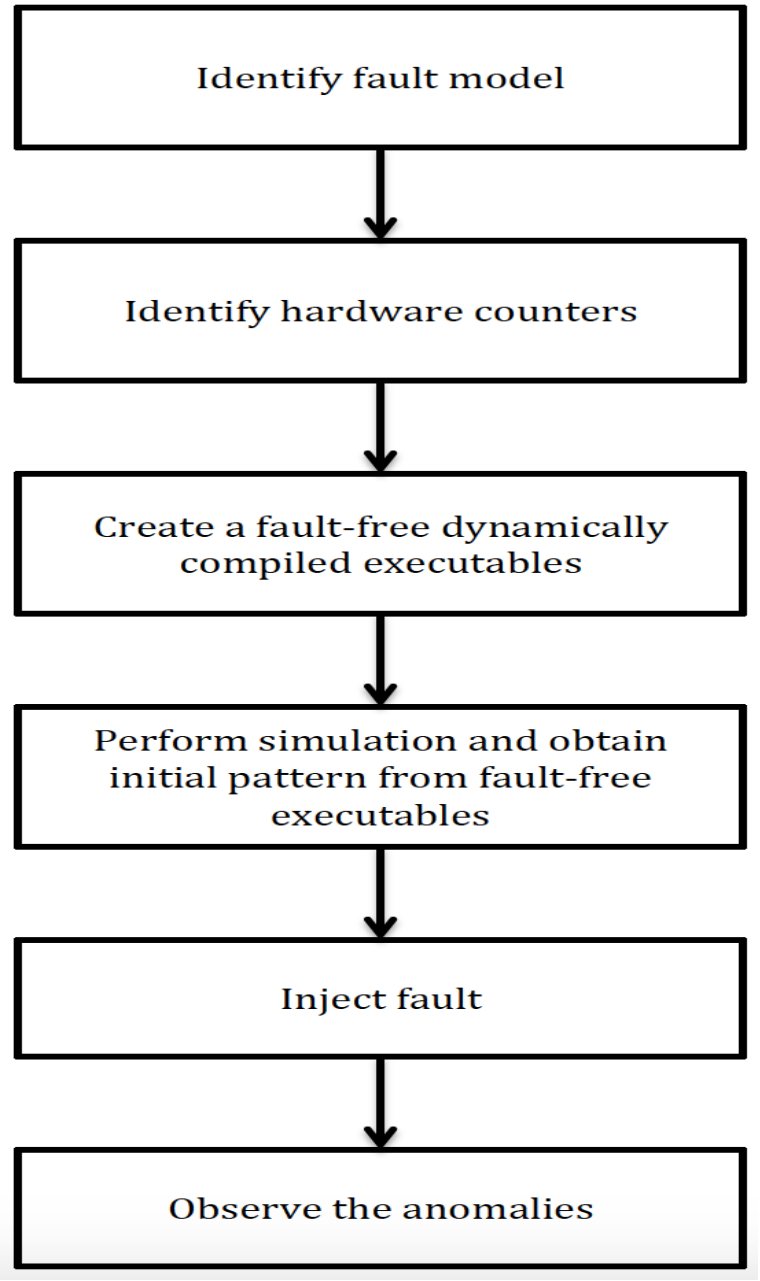


FIGURE 3.1: The methodology set-out for this experiment

- *Single Stuck-at Fault model* (SSFM) is used to model the condition where a value in a memory cell or a logic gate is permanently stuck at either logic value zero or one. This fault model is the most common model used in digital test pattern generator due to its simplicity. SSFM can be used to model many physical defects.
- *Single Bit-flip Fault* model is used to model transient faults due to soft errors that can occur either in the register file, arithmetic logic units, or in different pipeline registers of a processor. This fault model causes the flipping of a logic value from one value to another when a defect occurs. The bit-flipping takes place within

the duration of the defect, and the value flips back to its original value after the duration of the defect.

- *Multiple Bit-flip Fault* model is used to represent simultaneous change of logic values for multiple bits. Like single bit-flip fault model, multiple bit-flip fault model involves flipping various bits located either in the register file, arithmetic logic units, or in different pipeline registers of a processor at the same time, and for the duration of the fault.

As discussed in Chapter 2, a transient fault is a fault that happens at a random time and is hard to detect. However, for a safety-critical embedded system such as a control system in a spacecraft, detection of errors caused by transient faults is very important to ensure the errors are mitigated and thus, prevent the system from entering into a failure that is caused by radiation. Therefore, in this work, the single bit-flip fault model is chosen as a study case as this fault model closely represents the transient faults experienced by the system.

3.4 Selection of Event

TABLE 3.1: Architectural events that can be monitored in an Intel Atom processor

Bit Position	Event Name	Explanation
0	UnHalted Core Cycles	Counts core clock cycles when the clock signal on a specific core is running (not halted).
1	Instruction Retired	Counts the number of instructions that were completely executed, and it only counts for instructions that are on the correct execution path.
2	UnHalted Reference Cycles	Counts reference clock cycles at a fixed frequency while the clock signal on the core is running.
3	LLC Reference	Counts requests originating from the core that reference a cache line in the last level on-die cache.
4	LLC Misses	Counts each cache miss condition for references to the last level on-die cache.
5	Branch Instructions Retired	Counts branch instructions at retirement.
6	Branch Misses Retired	Counts mis-predicted branch instructions at retirement.

The second step is to identify which Performance-Monitoring Events (PMEs) the hardware counter will monitor to create a profile of the system. As the number of available

counters in a processor is limited, this imposes a limit on the number of PMEs that can be monitored concurrently in real-time. For example, the Intel Atom used in this work only has two programmable performance monitoring counter registers per processor core, which means only two PMEs can be captured simultaneously. However, researchers in [101] and [102] have shown that using a single counter to monitor a single PME is sufficient to describe the behaviour of a program.

Due to the limitation of available counters for monitoring, selection of the PME is important to ensure it is applicable across benchmarks that have different instruction distributions and that run on different processors. Architectural PMEs are the common events that can be monitored across different processors and architectures. Table 3.1 shows some examples of architectural PMEs available in an Intel Atom processor. In this work, two different PMEs, namely the number of instructions retired and the number of cache misses are chosen. These two PMEs were chosen as it was found that the same PMEs are also available on other processors, namely the ARM Cortex processor [2] and the POWER4 processor [103]. The data from the two PMEs are collected and monitored separately before being compared to determine which PME is better suited for detection of anomalous behaviour.

3.5 Benchmarks

The benchmarks used in this experiment are from MiBench [116], which consists of a set of 35 embedded applications divided into six suites, with each suite targeting a specific area of the embedded market. These 35 embedded applications can also be grouped according to different classes of instruction. There are three main classes of instructions, namely (a) logical and program control instructions (such as unconditional and conditional branch instructions), (b) arithmetic instructions, which includes both integer and floating-point instructions, and (c) data transfer or memory operation instructions (load and store). It was impossible to conduct experiments utilising all 35 benchmarks as each benchmark is executed more than 100 times. Therefore, at least one benchmark from each suite was chosen. The chosen benchmarks also have different computational characteristics to ensure the results obtained will not be dependent on one type of benchmark. The following benchmarks have been chosen for this experiment:

- *Bitcount* taken from Automotive and Industrial Control Suite, is an algorithm that tests bit manipulation ability of a processor by counting the number of bits in an array of integers. This is known as a computationally intensive benchmark as it has a large percentage of arithmetic instructions.
- *QSort*, which is also from Automotive and Industrial Control Suite, uses the popular quick-sort algorithm implemented in the GNU C standard library to sort a

large array of strings into ascending order. QSort is also another computationally intensive benchmark as it has more than 79% arithmetic instructions.

- *Dijkstra* is a benchmark taken from the Network suite that calculates the shortest path between between two nodes using an adjacency matrix of size 100x100. 100 paths are calculated during each execution. This benchmark is categorised as a memory intensive benchmark as it contains 40% memory operation instructions.
- *FFT* is a benchmark taken from Telecommunication suite that performs a *Fast Fourier Transform* on an array of 32,768 floating point data. Fourier transforms are used in digital signal processing to find the frequencies contained in a given input signal. The input data is a polynomial function with pseudorandom amplitude and frequency sinusoidal components. This benchmark can be categorised as both computational and control intensive benchmark due to the high arithmetic as well as logical and program control instructions.
- *StringSearch* is a benchmark taken from Office suite that searches for given words in phrases using a case-insensitive comparison algorithm. This benchmark has a high number of arithmetic and memory operation instructions.

3.6 Architectural Simulator

In the work described in this thesis, we studied the presence of transient faults in a processor's registers and their effect on the overall behaviour of the processor. Building a real system to study the effects of transient faults in the processor's registers involves a huge cost [117]. A more cost-effective way is to rely on computer architecture simulators. A simulator also provide the ease of evaluation, debugging and understanding of the behaviour of the existing system, allowing one to see what is happening "in the system" [118].

There are several simulators available such as Gem5 [119], Multi2Sim [120], PTLSim [121], MARSSx86 [122] and others. However, Gem5 was selected as the simulation tool in this work because of its ease of configurability, support of various Instruction Set Architecture (ISAs), support of a complete operating system (OS) and support from its active community of developers. Another reason for selecting Gem5 was the availability of GemFI [123], a fault injection tool that was developed based on the Gem5 simulator.

All the experiments were performed using the Gem5 architectural simulator and GemFI fault injection tool. The Gem5 simulator is an instruction set simulator, widely used in computer architecture research. Gem5 provides a flexible, modular simulation system that is capable of evaluating a broad range of systems, encompassing system-level architecture down to processor micro-architecture. It supports various ISAs such as Intel x86, ARM, MIPS, Alpha, Sparc and Power, and can be used either in *System-call Emulation*

(SE) or *Full-System* (FS). SE mode allows users to emulate most common system calls, thus avoiding the need to model devices or even an OS. In FS mode, Gem5 models the complete system including the OS and devices, executing both user-level and kernel-level instructions.

In principle, GemFI is able to support any processor model and ISA available in Gem5, but at the time of our experiments, GemFI only supported Alpha and Intel x86 ISAs. For this work, Intel x86 ISA is used. Running in FS mode, GemFI injects fault onto registers in the processor core while simulating both user-level and kernel-level instructions and models a complete system including the central processing unit (CPU), memory and peripheral devices. It evaluates the impact of faults from the architectural level up to the application level. There are two intrinsic functions provided by the GemFI Application Programming Interface (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.

These two API functions are required to be added into the benchmark to be tested. GemFI also generates a list of possible faults which can be injected into various locations such as: (a) the *Fetch* instruction, (b) the selection of read/write registers during *Decoding* stage, (c) the result of an instruction in an *Execution* stage or (d) memory transactions during *Load and Store*. Each fault in the list is characterised by four attributes: *Location*, *Thread ID*, *Time* and *Type*.

- **Location** defines the targeted location for fault injection. Supported locations include the fetched instruction, the selection of read/write registers during the decoding stage, the result of an instruction at the execution stage, and finally memory transactions (load/stores).
- **Thread ID** allows a fault to be injected into a specific thread by using the numerical identification number given to the thread upon the execution of the API function, void FI_activate (int id, int command).
- **Time** defines when the fault will be injected, which means, the fault is injected after a certain number of instructions have been executed from the point the fault injection is initiated.
- **Type** defines how the value of a specified location can be corrupted: (a) by flipping the running value at specific bit location (which mimics a single bit-flip), and (b) by setting all bits of the location to either 0 or 1 (which mimics a single stuck-at-fault).

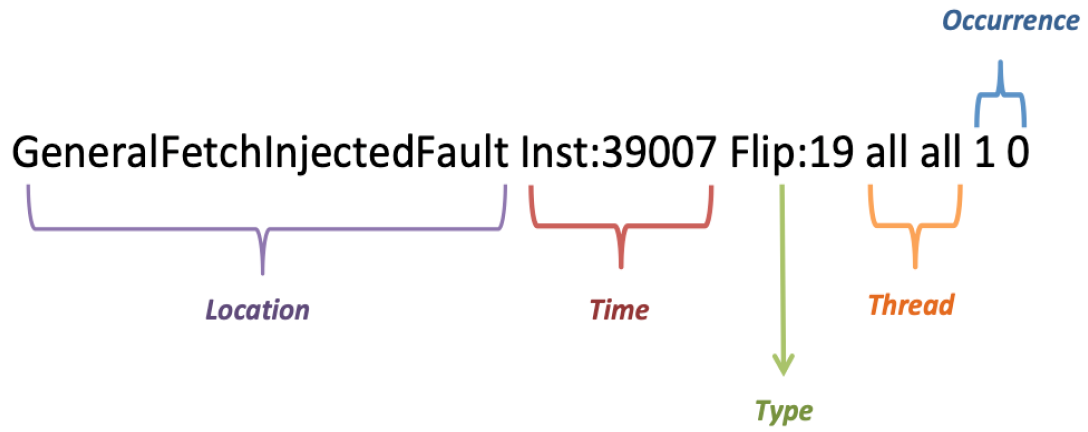


FIGURE 3.2: A sample of fault being injected into the Fetch instruction

Figure 3.2 shows an example of how a fault is injected. The fault is injected in bit 19 of the fetched instruction in the CPU, when the application fetches the 39007th instruction after the initiation of fault injection for this thread (FI_activate). The fault is activated for all threads at the 39007th instruction, and stays active for a period of one instruction and the type of fault being injected is a single bit-flip fault, identified with the keyword *Flip* as shown in Figure 3.2.

The experiments have been conducted on a Linux virtual machine as both Gem5 and GemFI require the Linux operating system. The virtual machine was created using Microsoft Azure virtual machine platform with 16 central processing units (CPUs), 32 gigabytes (GBs) of memory and 1 terabyte (TB) of data storage and running Ubuntu version 16.04 LTS as the operating system. Microsoft Azure is a public cloud computing platform providing a range of cloud services, including services needed for storage, networking, databases, artificial intelligence and machine learning, analytics and compute. A major part of the work in this thesis has been supported by Microsoft Azure that provided the computing resources required to perform the experiments. These resources allowed the experiments to be executed at a much faster pace compared to using a normal desktop computing system. It provided significant storage capacity to store all simulation results.

3.7 Experimental Setup

To extract the HPC data that will be used to monitor system reliability, there are several steps involved:

1. Set up the benchmarks required for testing.
   Each benchmark is compiled dynamically in two different versions - one in the original version and another with GemFI intrinsic functions added. Both versions

are compiled for Intel x86 ISA. For *Bitcount*, the input to the benchmark is an array of integers while for *QSort*, the input to the benchmark contains a list of words. The input for *Dijkstra* benchmark is a large graph in the form of an adjacency matrix, whereas for *FFT*, the input to the benchmark is an array of 32,768 floating point data. Lastly, the input for the *StringSearch* benchmark is a list of phrases.

2. Perform the simulation.
   Simulations of the benchmarks were performed in GemFI under FS mode. FS mode simulates the execution of the benchmark 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 created using the fault generator in GemFI. A fault configuration file describing the faults 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, each of which corresponds to a pipeline stage. The simulation continues as normal until it is time for the fault to be injected. Once the fault has been injected (i.e. a bit has been flipped), the simulation continues. If an injected fault is activated or manifest as an error, it leads to the system experiencing some form of failure, else, the experiment terminates successfully. Figure 3.3 provides a general overview of how the simulation works using the GemFI API. The blue lines indicate 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 is executed in six conditions:

   (a) *Initial Run* refers to running the binary executables without any GemFI API functions added to it. The *Initial Run* condition was executed to obtain the original behaviour of the benchmarks without any GemFI API functions added to it.

   (b) *With Fault Activated* refers to running the binary executables that have been added with GemFI API functions, but fault is not being injected. This condition provides the baseline behaviour for all benchmarks

   (c) *Fault injection in the Fetch Instruction* refers to running the binary executables with GemFI API functions added, and a fault is injected in the Fetched instruction.

   (d) *Fault injection in the read or write register during Decoding stage* refers to running the binary executables with GemFI API functions added, and a fault is injected in either a read or a write register during Decoding stage.

   (e) *Fault injection in the result of an instruction during Execution stage* refers to running the binary executables with GemFI API functions added, and a fault is injected in a register that contains the result of an instruction at the Execution stage.

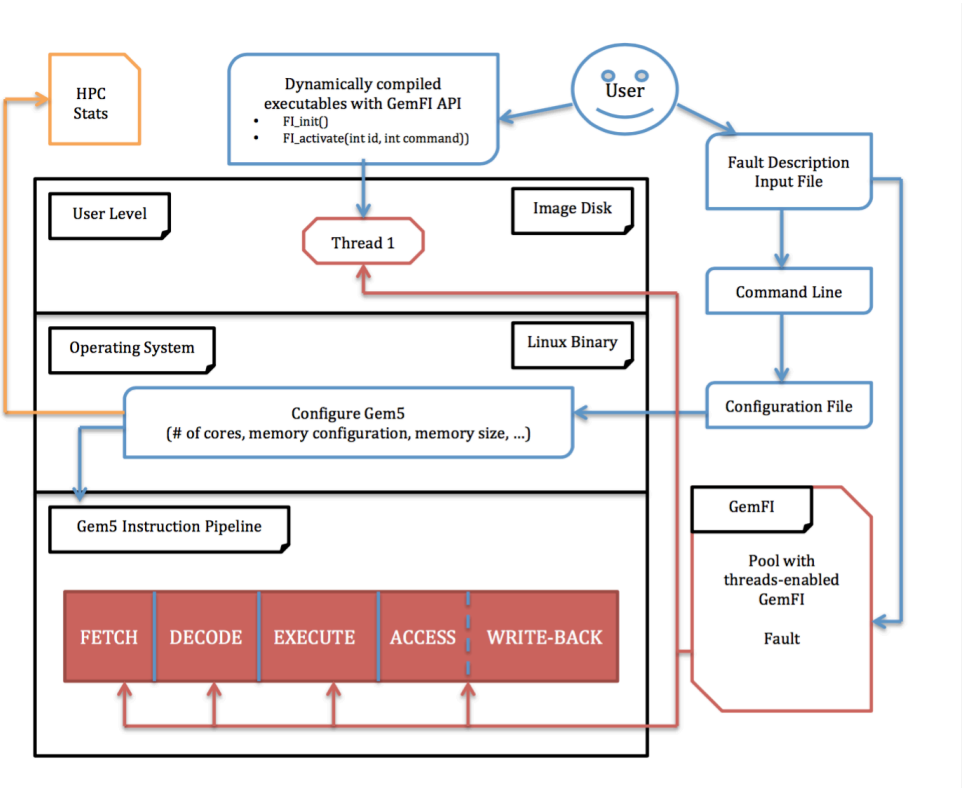


FIGURE 3.3: Overview of the GemFI API, after [123]. The red components show possible fault injection locations; Thread 1 is where the executables run.

(f) *Fault injection during memory transactions in Load/Store stage* refers to running the binary executables with GemFI API functions added, and a fault is injected in the memory instruction that is either loading a value from a register, or storing a value into a register.

For each experiment, the fault model applied is a single bit-flip fault model where a single bit-flip fault is injected randomly at a stage in the pipeline. The benchmarks were simulated at two clock speeds: (a) at clock speed of 2GHz, a speed that is typically found in microprocessors, and (b) a clock speed of 250MHz, a typical speed found in microcontrollers. Two different speeds were applied to identify the number of clock cycles it takes for the system to suffer a failure, particularly, a failure that resulted in a crash. This information is used to evaluate the effectiveness of the early detection and prediction algorithm presented in Chapter 4.

The runtime for each experiment ranges from a minimum of five minutes to two hours depending on the benchmark, the clock speed and the sampling interval. Lower clock speeds and smaller sampling intervals result in longer runtime and generation of a huge amount of data. For example, for the FFT benchmark running at clock speed 250MHz and sampling at 5000ns, the total runtime required was two hours and the total data generated was 8GB. Compare with the same benchmark but running at a clock speed of 2GHz and sampling at 100,000ns, the total runtime

required was just around thirty minutes and the total data generated was 100MB. For each experiment conducted, the HPCs are traced using the method outlined below.

3. Trace and record the required HPC values.
   Two different tracing methods were used to log the counter values obtained. The first method was to obtain the counter value after the execution of the benchmark has been completed. The total count for the benchmark with fault injection will be compared against the total count for baseline benchmark (benchmark with fault activated but without fault injection). However, this method was only able to provide an indication that an error has occurred which causes the application to either hang, crash or provide incorrect output, but was unable to determine when the error occurred. The second method was to log the counter value at certain intervals. Using this method, the execution profile for each benchmark is created, and the execution profile is able to detect the instance an error has occurred which causes a failure to the system.

   Sampling interval plays an important factor in determining the accuracy of time sampling methods [124]. It is important to ensure that the execution profiles created contain sufficient amount of data that can be used to identify the anomalous behaviour in the system. Several sampling intervals were chosen to determine which interval duration is most suitable for this work. The sampling intervals listed below were chosen from the order of magnitude 2 to the order of magnitude 5, increasing one order of magnitude each time.

   - 100000ns;
   - 50000ns;
   - 10000ns;
   - 5000ns;
   - 1000ns; and
   - 800ns

4. Obtain, compare and analyse the results.
   The counter values obtained from *Initial Run* condition and *With Fault Activated* condition are first plotted and compared. Besides trying to establish the base line behaviour for each benchmark, the comparison is also done to ensure that the insertion of GemFI API will not alter the behaviour of the benchmarks. The base line behaviour is used to study and compare between two different PMEs, the various sampling interval as well as using different input data, which are presented in Section 3.8.1, Section 3.8.2 and Section 3.8.3.

   Next, the counter values obtained from the remaining four conditions, (a) *Fault injection in the Fetch Instruction*, (b) *Fault injection in the read or write register*

during Decoding stage, (c) *Fault injection in the result of an instruction during Execution stage*, and (d) *Fault injection during memory transactions in Load/Store stage* are obtained. The counter values from the four conditions are compared against counter value obtained from *With Fault Activated* condition that forms the baseline of the behaviour for each benchmark. The characteristics of the counter values observed are further discussed in Section 3.8.4.

The outcome from each experiment are categorised either as (a) *Crash*, (b) *Hang*, *Fail Silence Violation* or *Not Manifested*. A *Crash* is said to have occurred when the experiment terminated unexpectedly while a *Hang* occurred when the experiment had stalled or do not response within specific time. *Fail Silence Violation* occurred when the counter values has some slight deviation from the baseline counter values but no apparent failure can be observed by the user as the experiment terminated successfully. *Not Manifested* is where the experiment terminated successfully, and the counter values do not deviate from the baseline counter values. The errors that causes these anomalous behaviours are further discussed in detail in Section 3.9.

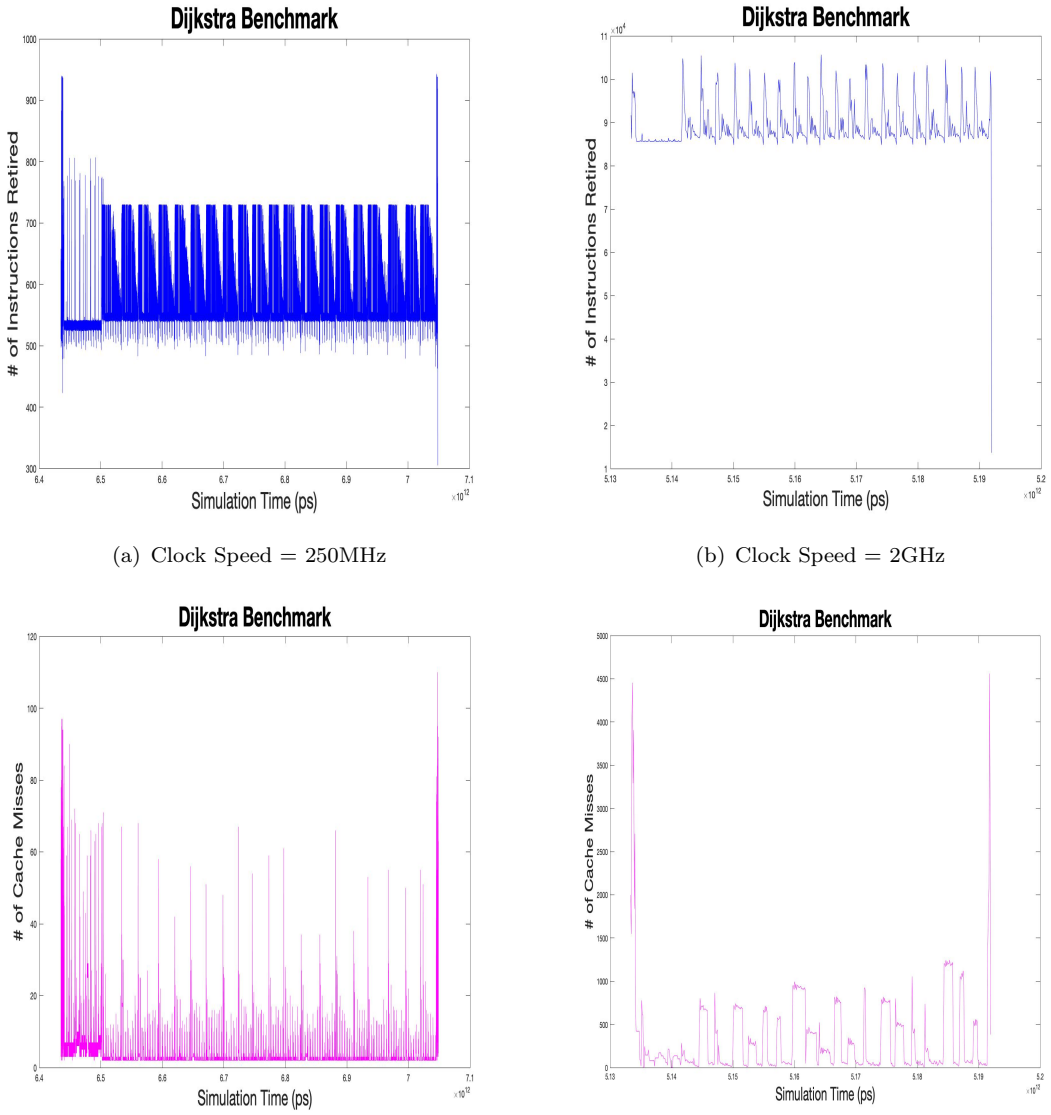
3.8 Results and Discussion

In this section, the results obtained from the experiment is discussed. Section 3.8.1 compares the results obtained using two different PMEs while Section 3.8.2 compares the results obtained using various sampling intervals. In Section 3.8.3, the execution profiles of two benchmarks using different input data are presented and discussed. All the results in Section 3.8.1, Section 3.8.2 and Section 3.8.3 are obtained from experiments running *With Fault Activated* condition, and it shows the normal behaviour of the benchmarks.

The characteristics of anomalous behaviour observed in the processor are presented in Section 3.8.4, where in this section, the counter values from the remaining four conditions, (a) *Fault injection in the Fetch Instruction*, (b) *Fault injection in the read or write register during Decoding stage*, (c) *Fault injection in the result of an instruction during Execution stage*, and (d) *Fault injection during memory transactions in Load-/Store stage* conditions are compared against counter value obtained from *With Fault Activated* condition. These anomalous behaviour are categorised as either (a) *Crash*, (b) *Hang*, *Fail Silence Violation* or *Not Manifested*.

3.8.1 Comparisons between two PMEs

Figure 3.4 compares the execution profiles obtained from Dijkstra benchmark running at two clock speeds (a) 250MHz, and (b) 2GHz. The profiles were generated from *With Fault Activated* condition (i.e. fault was not injected), which forms as the baseline



(a) Clock Speed = 250MHz

(b) Clock Speed = 2GHz

(c) Clock Speed = 250MHz

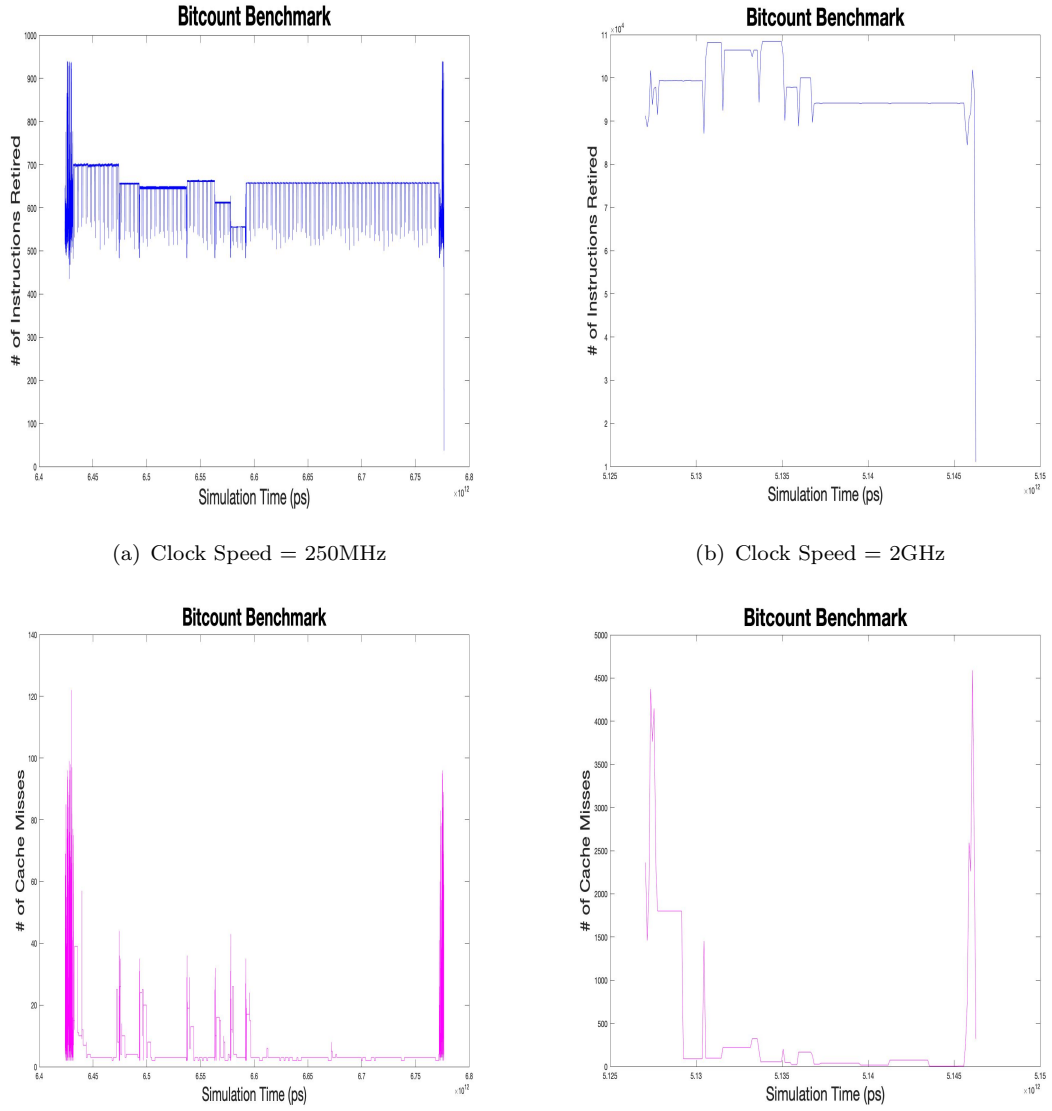
(d) Clock Speed = 2GHz

FIGURE 3.4: Execution profiles using Number of Instructions Retired and Number of Cache Misses for Dijkstra benchmark running at 250MHz and 2GHz clock speed

behaviour for this benchmark. Figure 3.4(a) and Figure 3.4(b) show the behaviour of Dijkstra benchmark monitored using the number of retired instructions against the simulation time in picoseconds while Figure 3.4(c) and Figure 3.4(d) show another set of results where the behaviour of the benchmarks were monitored using the number of cache misses plotted against simulation time in picoseconds. The counter values for each experiment begin after the OS has boot-up. For example, in Figure 3.4(a) and Figure 3.4(c), it takes about 6.44s for the OS to boot-up, while in Figure 3.4(b) and Figure 3.4(d), it takes about 5.134s for the OS to boot-up. The difference in the boot-up time is caused by running Ubuntu on different virtual machines.

As can be seen in Figure 3.4, the profile for each benchmark is similar even though it is

running at a different clock speed. From the results, it shows that a program exhibits the same behaviour regardless of any clock speed it runs on. This suggests that the PME monitored using HPC can be used to create the execution profile of an application, and thus, identifying the normal behaviour of the system.



(a) Clock Speed = 250MHz

(b) Clock Speed = 2GHz

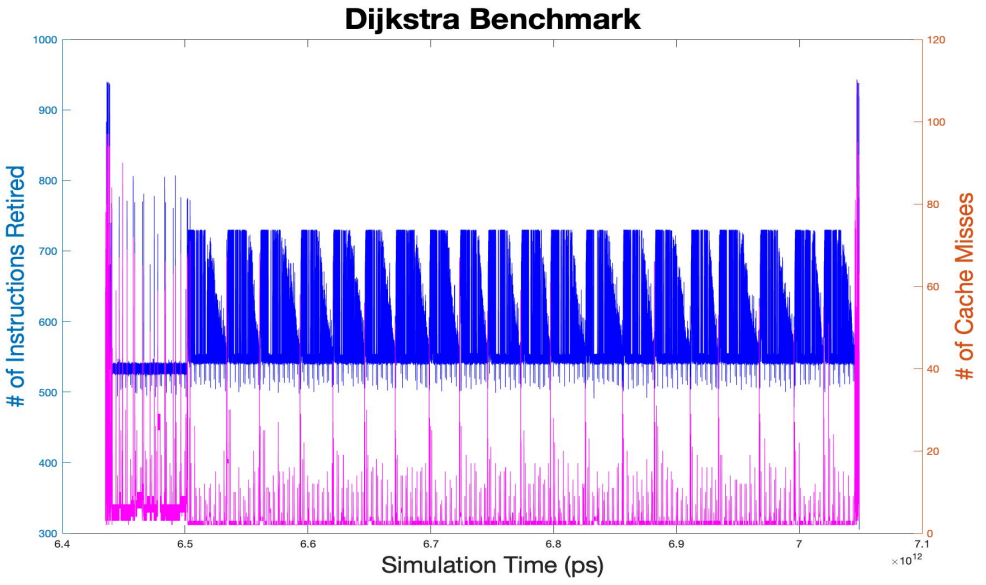
(c) Clock Speed = 250MHz

(d) Clock Speed = 2GHz

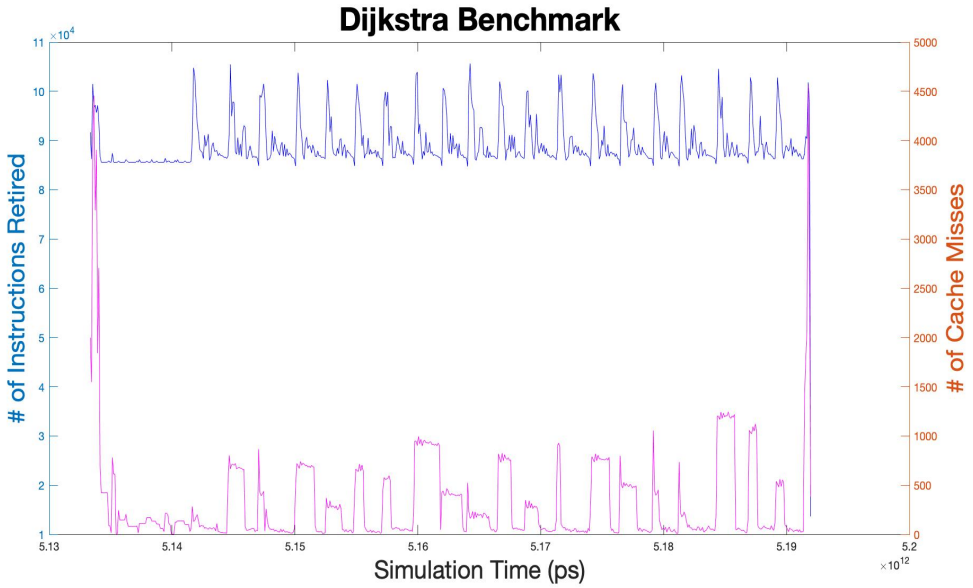
FIGURE 3.5: Execution profiles using Number of Instructions Retired and Number of Cache Misses for Bitcount benchmark running at 250MHz and 2GHz clock speed

Figure 3.5 shows the execution profiles obtained from Bitcount benchmark where Figure 3.5(a) and Figure 3.5(b) show the behaviour of Bitcount benchmark monitored using the number of retired instructions (axis Y) against the simulation time in picoseconds (axis X) while Figure 3.5(c) and Figure 3.5(d) show another set of results where the behaviour of the benchmarks were monitored using the number of cache misses plotted on axis Y. It is clear that the execution profiles for Dijkstra benchmark shown in Figure 3.4 completely differs from the execution profiles for Bitcount benchmark shown

in Figure 3.5 whether it is using instructions retired PME or cache misses PME. Like Dijkstra benchmark, it takes about 6.43s for the OS to boot-up for the Bitcount benchmark running at 250MHz as shown in Figure 3.5(a) and Figure 3.5(c), and 5.128s for the Bitcount benchmark running at 2GHz as shown in Figure 3.5(b) and Figure 3.5(d). Additional results on the execution profiles for FFT, StringSearch and QSort benchmarks can be referred to in Appendix A. From the results obtained, it is clear that each application has its own signature profile, which can be monitored using HPC.



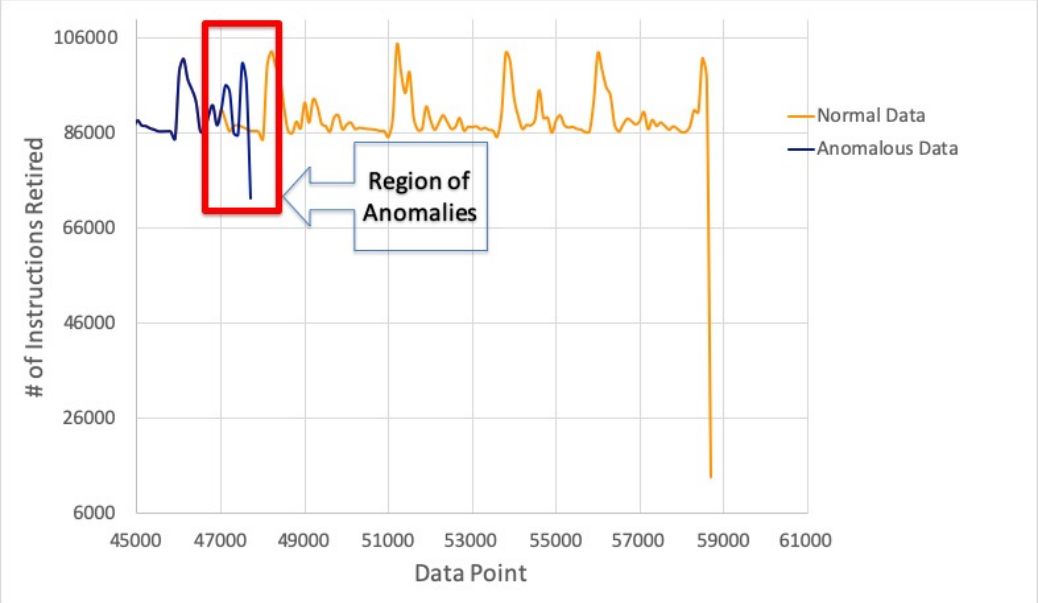
(a) Clock Speed = 250MHz



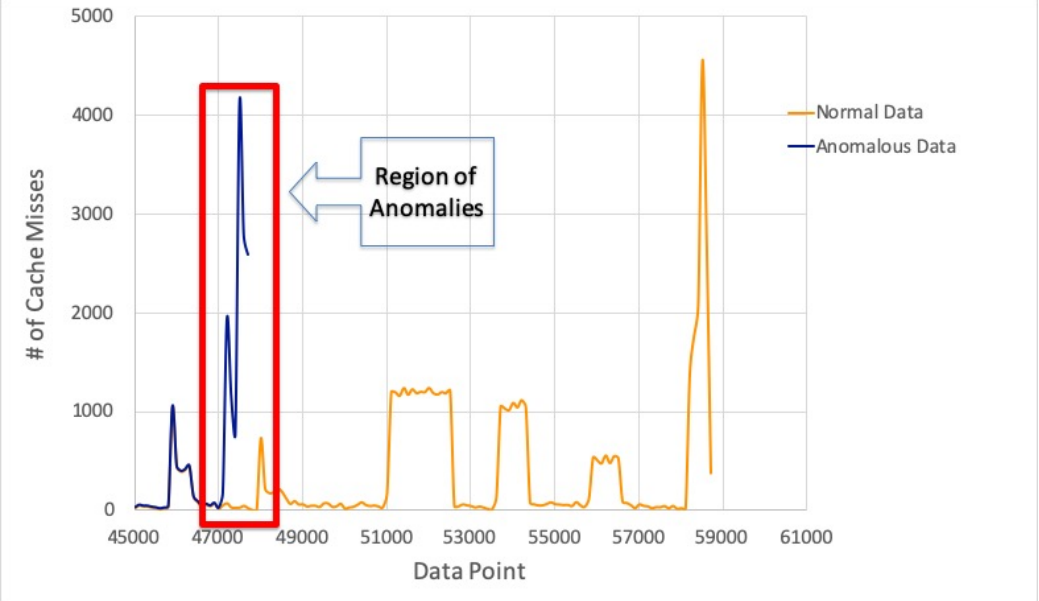
(b) Clock Speed = 2GHz

FIGURE 3.6: Correlation between Number of Instructions Retired and Number of Cache Misses for Dijkstra benchmark running at 250MHz and 2GHz clock speed

Figure 3.6 shows the correlation between instructions retired PME and cache misses PME for Dijkstra benchmark. As can be observed, there is a positive correlation between the two PMEs, which is more prominent in the execution profile obtained when running at lower clock speed (Figure 3.6(a)), where both the counter values increase and decreases in parallel. This is expected as higher cache misses means higher latency, where more clock-ticks are required for an instruction to retire.



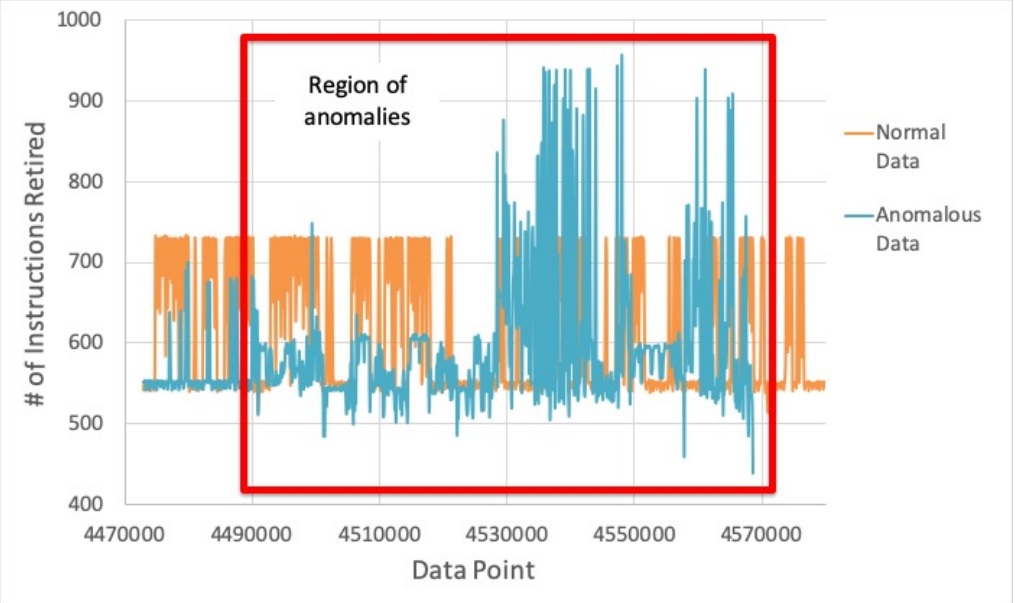
(a)



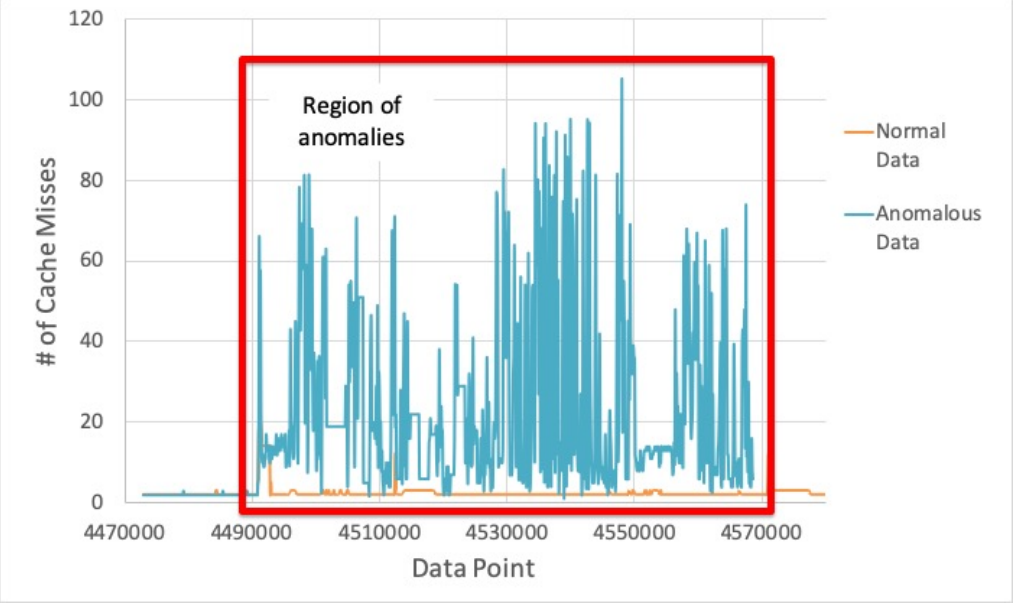
(b)

FIGURE 3.7: Comparison between Number of Instructions Retired and Number of Cache Misses for Dijkstra benchmark running at 2GHz clock speed

Figure 3.7 and Figure 3.8 show the comparison between cache misses PME and instructions retired PME for Dijkstra benchmark running at 2GHz and 250MHz. As can be



(a)



(b)

FIGURE 3.8: Comparison between Number of Instructions Retired and Number of Cache Misses for Dijkstra benchmark running at 250MHz clock speed

observed from Figure 3.7 and Figure 3.8, the value of both the counters record a sudden and huge increase at Data Point 47,000 in Figure 3.7 and at Data Point 4,491,100 in Figure 3.8. This indicates that an error is present in the system, thus causing the system to behave anomalously.

By comparing the values recorded using cache misses PME and instructions retired PME, it is found that values recorded using cache misses are lower compared to instructions retired. For example, in Figure 3.7, the values recorded for instructions retired PME

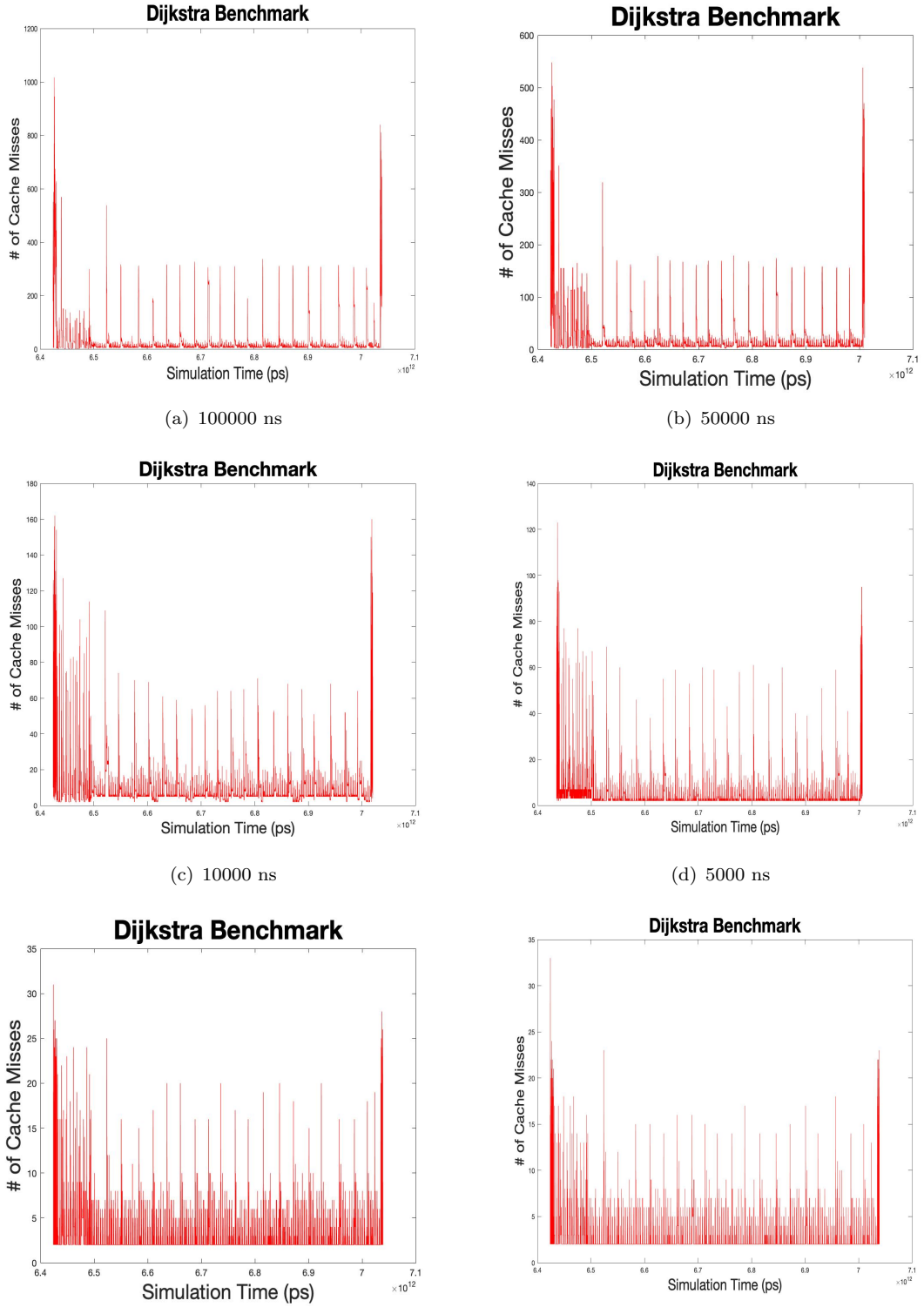
are in the range of 67,000 and 100,000, while for cache misses PME, the values recorded are between 0 and 4,500. Similarly, for Figure 3.8, the values recorded for cache misses are between 0 and 110, while values recorded for instructions retired PME are between 450 and 950. In general, the values recorded for cache misses are between three and seven bits while value recorded for instructions retired PME are around seventeen bits, more than twice the values recorded for cache miss PME. Therefore, the computational size and speed can be greatly reduced by monitoring the behaviour using cache misses compared to using instructions retired.

Another observation is that the cache misses PME is also more susceptible to detection where the counter data recorded bigger deviation (more than 10% as seen in Figure 3.7(b) and Figure 3.8(b)) when the pattern begins to deviate from the normal behaviour compared to number of instructions retired PME, where the deviation recorded is around 5%. A bigger deviation in the counter values is easier for detection, and thus, provides better detection accuracy. Based on all these findings, the cache misses PME is found to be more suitable for monitoring anomalous behaviour in the system.

3.8.2 Comparison on various Sampling Interval

Figure 3.9 shows the results for Dijkstra benchmark running at 250MHz clock speed with various sampling interval. The results display various execution profiles which were plotted with number of cache miss plotted against simulation time. The difference between each figure is the amount of data collected from the counter. For example, in Figure 3.9(d) where experiment with the sampling interval set at 5000ns, there were more data collected compared to the experiment with 100000ns sampling interval as shown Figure 3.9(a). A total of 120,000 data points are collected from the experiments running at 5000ns while only 6000 data points are collected from the experiments running at 100000ns.

The shorter the duration of the interval, the higher the amount of data is generated. Shorter intervals allows the anomalous behaviour to be detected earlier as the amount of data from the point the fault manifested as an error to the point where a failure occurs increases. Another observation is the value of the counter gets smaller as the sampling interval gets smaller. This finding is consistent with the behaviour of HPC itself, where in time-based sampling, the counter is incremented on an instruction-by-instruction basis until an interrupt is generated. Smaller sampling interval means the interrupt is generated quicker. However, it is found that sampling interval below 5000ns is not suitable as it is difficult to distinguish between an anomalous point from a normal point. Another problem of having an interval that is too short is that the same activity can be recorded several times, thus inflating the sample size. Therefore, the most suitable sampling interval chosen is 5000ns.



(a) 100000 ns

(b) 50000 ns

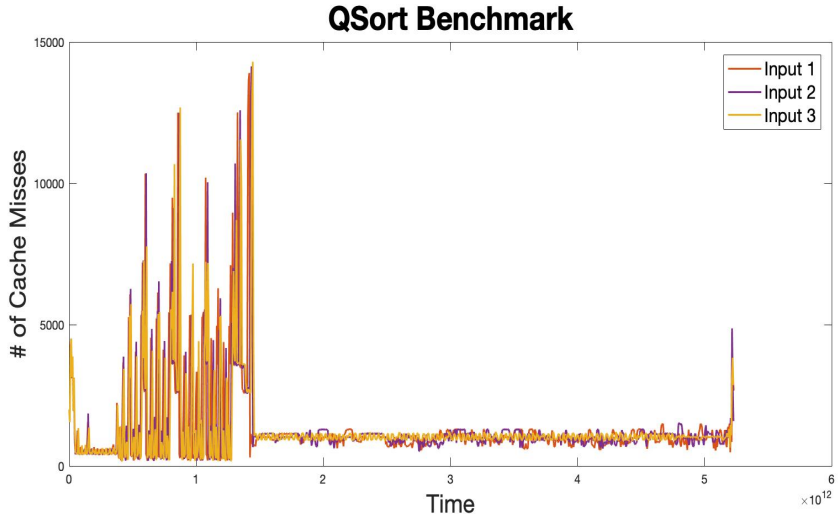
(c) 10000 ns

(d) 5000 ns

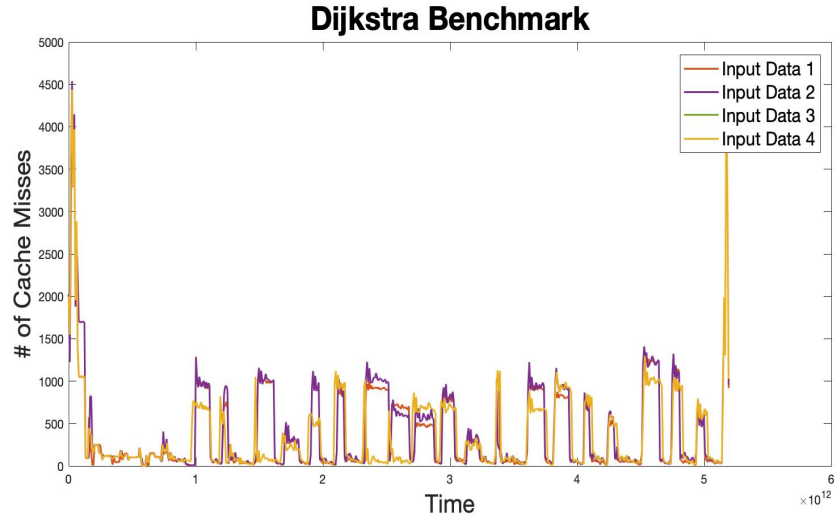
(e) 1000 ns

(f) 800 ns

FIGURE 3.9: Execution profiles using Number of Cache Misses for Dijkstra benchmark running at different sampling interval



(a)



(b)

FIGURE 3.10: Execution profiles using Number of Cache Misses for (a) QSort and (b) Dijkstra benchmarks with multiple inputs.

3.8.3 Comparison on using Different Input Data

The experiment was also conducted for QSort and Dijkstra benchmarks with different sets of input data and the execution profile for each benchmark was compared. An input data is defined as a file that contains data that serve as an input to a program. Executing the benchmarks with different sets of input data simulate the condition where applications do not always run on the same input data. There are three different sets of input data used for the QSort benchmark and four different sets of input data used for Dijkstra benchmark. For QSort benchmark, the first set of input data consisting of words and integers, the second set of input data consists of a mixture of words, integers and floating points and the third set of input data contains only integers and floating

points. Since the Dijkstra benchmark is a benchmark that calculates the shortest path between every pair of nodes in a graph, different sets of input data means having different nodes in each input data.

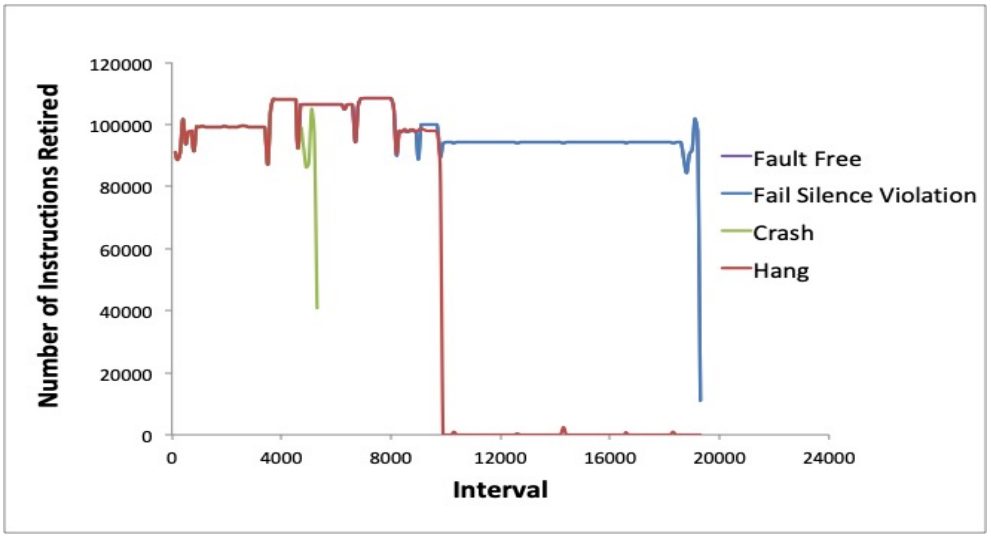
Figure 3.10 shows the execution profile generated for QSort and Dijkstra benchmarks with different sets of input data and from the results shown, the execution profile for each benchmark still bears a strong similarity. This finding suggests that regardless of any input used, the execution profile remains similar, and thus, it is possible to observe anomalies that may occur based on the profiles generated by the counter.

3.8.4 Characteristics of Anomalous Behaviour in a Processor

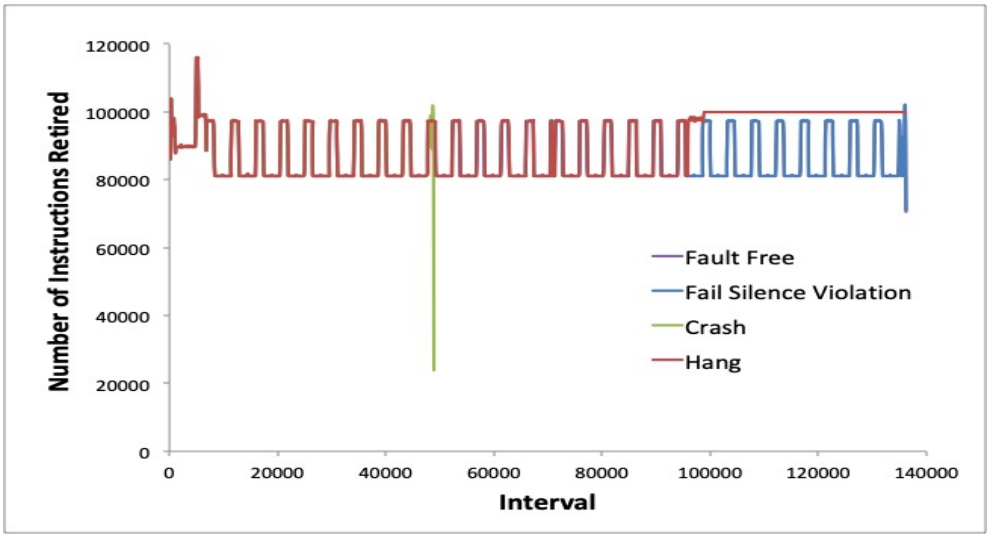
In the experiment conducted, as a single bit-flip fault is injected randomly in each stage of the pipeline, the manifested fault leads to different kind of errors such as *segmentation fault*, *invalid opcodes*, *kernel panic*, *bad paging* and others. These errors causes the program to experience anomalous behaviour, which can be divided into either (a) *masked error* (or fault free), (b) *fail silence violation*, (c) *hang*, or (d) *crash*. The correlation between these errors and failure observed in the system will be further discussed in detail in Section 3.9. However, every injected fault that manifests itself as an error can be shown using a single counter. The anomalous behaviour is captured in the execution profile for each benchmark as shown in Figure 3.11.

In the work described in this thesis, the characteristics observed from *Hang* and *Crash* are used to develop the early detection and prediction algorithm. For a *Hang*, there is a huge deviation in the beginning of the error before the counter value becomes constant at a point. The amount of time for the system to stay unresponsive or hang varies as it depends on when the user sends an interrupt to the system. For *Crash*, the counter will also spot a huge deviation before it stops. As for *Masked Error* and *Fail Silence Violation*, there is either no deviation from the counter values observed (for the case of *Masked Error*) or an extremely small deviation in the counter values (for *Fail Silence Violation*). In other words, the pattern profile remains the same as a fault-free system for both situations, and hence, it is not considered in this thesis.

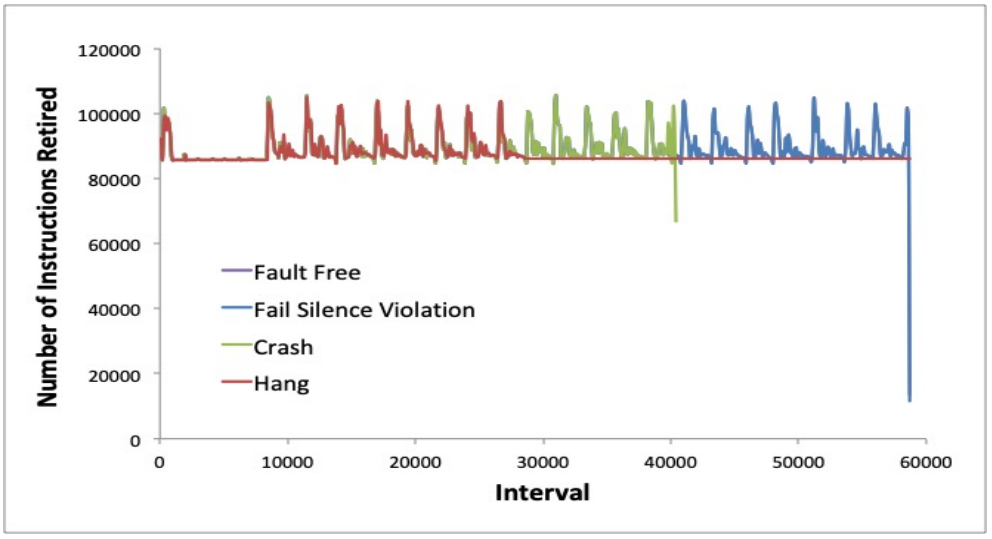
The different failures caused by manifested faults can be illustrated using Figure 3.12 which shows the temporal relationship between a fault, error and failure. Assume that a single bit-flip fault occurs at time t_f. When the fault propagates or manifests into an error, the counter begins to deviate, and a string of anomalies occur from time t_e onwards until the system encounters failure at time t_{fail}. In other words, a bit-flip fault may occur at anytime, but it does not causes an error in the system immediately. The counter values begin to deviate only when the fault is activated. Hence, it is crucial to detect when the fault has been activated or manifest as an error. The important time interval is δt_d, which is the time interval between error to failure. For a prediction



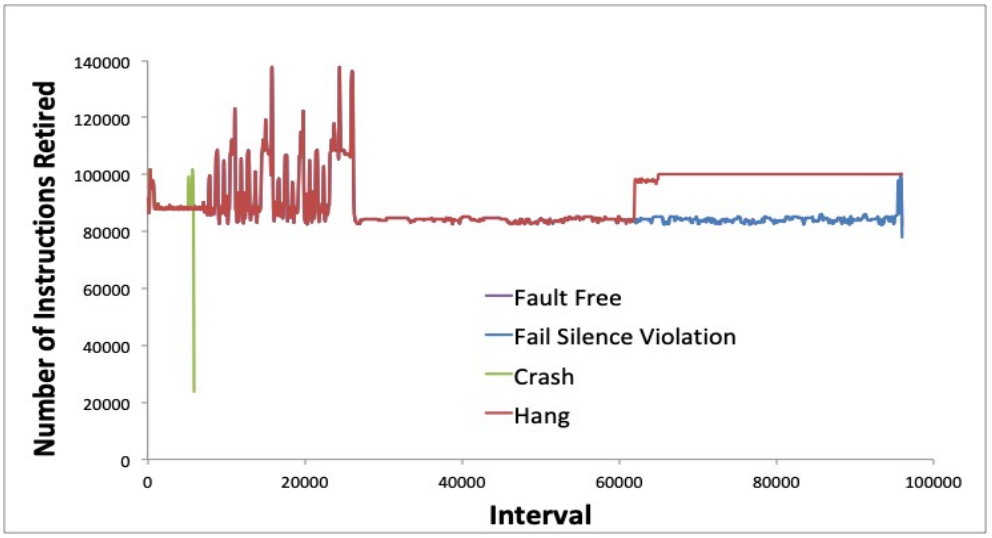
(a) Bitcount



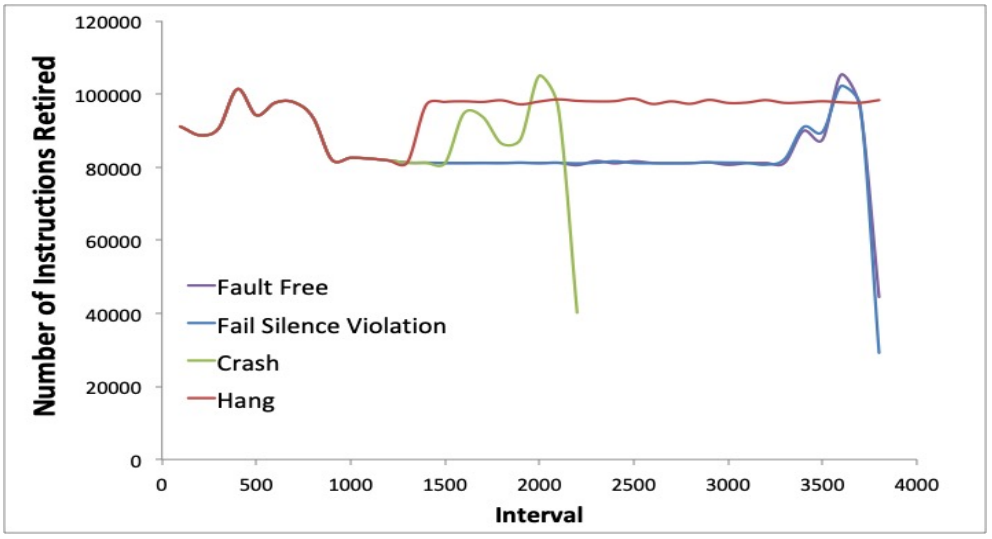
(b) FFT



(c) Dijkstra



(d) QSort



(e) StringSearch

FIGURE 3.11: Execution profiles that shows how different failures can be detected for the following benchmarks - (a) BitCount, (b) FFT, (c) Dijkstra (d) QSort and (e) StringSearch

method to become useful, the detection of anomalous behaviour has to be as early as possible within the time interval.

Figure 3.13 and a close-up of it in Figure 3.14 show how the fault manifested as an error after being injected into the system until the system fails. It shows how from the time the fault is manifested as an error until the time when the failure is observed on the system, there is a delay of approximately 5,000μs, equivalent to 1,250,000 (or 1.25M) clock cycles. Experiments were further conducted to determine the minimum number of clock cycles it takes for a system to crash after a fault has manifested as an error. A comparison was made between two benchmarks running at two different clock speed that suffered a crash failure after a fault has been manifested as an error. There were

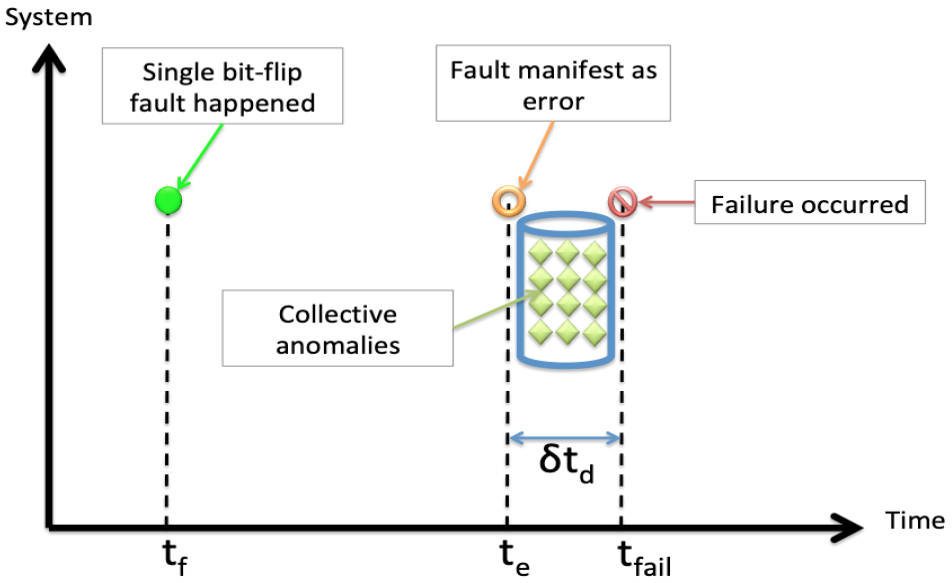


FIGURE 3.12: Temporal relationship between fault, error and failure

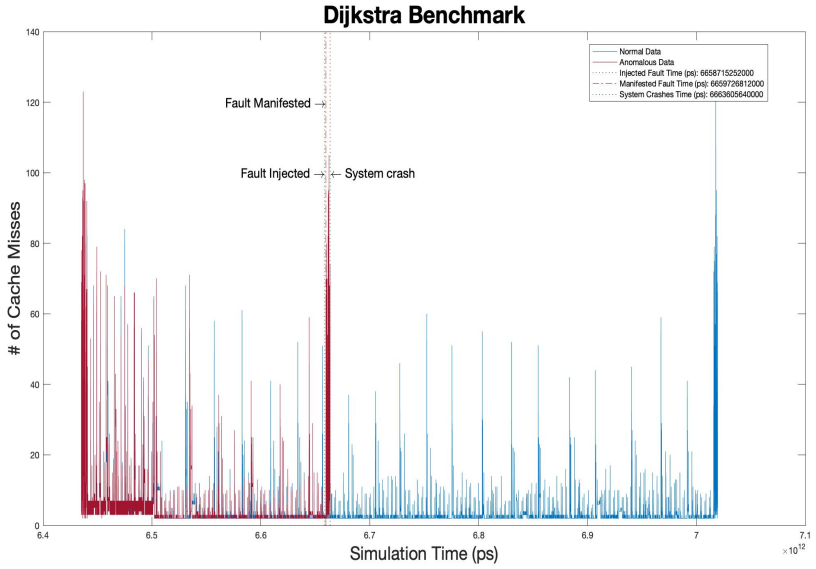


FIGURE 3.13: From injected fault to manifested fault and finally system failure

14 data sets and 11 data sets from Dijkstra and Bitcount benchmarks running at clock speed of 2GHz, and another 20 data sets and 5 data sets from Dijkstra and Bitcount benchmarks running at 250MHz. From the experiment, the minimum amount of clock cycles it takes for a system to crash is found to be approximately 1M clock cycles as shown in Figure 3.15, which is equivalent to 4,000μs for a system running at 250MHz clock speed. In other words, δt_d is 1M clock cycles or 4,000μs for a system running at 250MHz clock speed.

1M clock cycles seems to be a "big" number, but it only takes 4ms for a system to

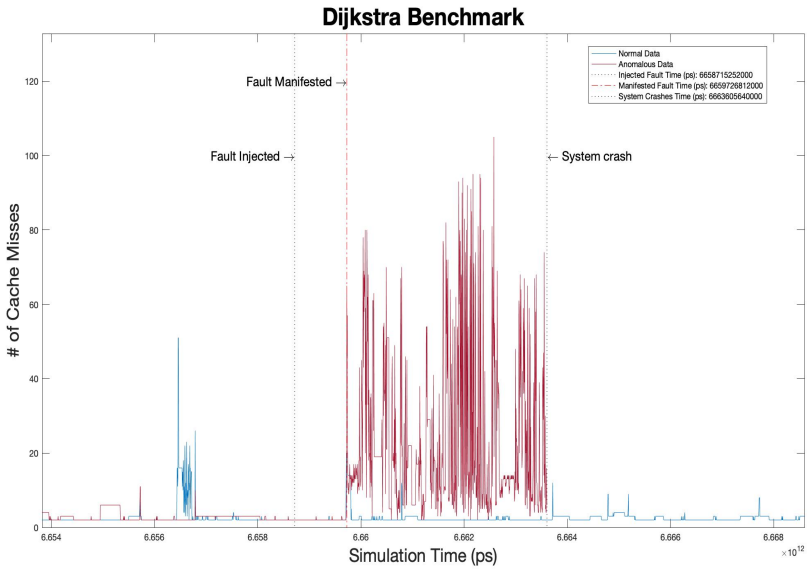


FIGURE 3.14: Close-up of Fault Injection to Fault Manifestation and System Failure

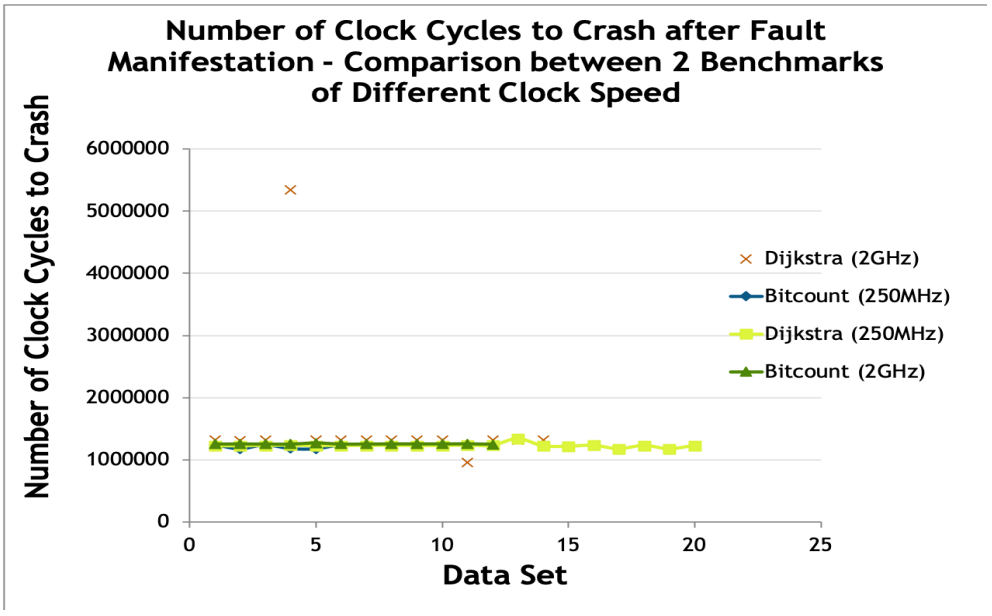


FIGURE 3.15: Number of clock cycles to crash

crash from the time the fault manifests itself as an error. In the most basic system, each instruction may occur in exactly one cycle. This involves fetching an instruction to execute from a program memory, decoding the instruction in the hardware, executing the instruction and finally storing the result of the instruction. However, systems nowadays are far more complex, even for a safety-critical embedded system, and not all CPU operations have equal operation of cost in terms of CPU cycles. [125] has provided a list of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD and VIA CPUs, where it shows that the number of clock cycles for each operation differs from one another.

To illustrate how it is possible to take 1M clock cycles to cause a crash, assume a single bit-flip fault is present in one of the registers of a certain fetch instruction. The fault gets activated when the faulty instruction is fetched from the program memory into instruction register. Fetching the instruction from a program memory may take a few clock cycles. The instruction register is split into two parts – *opcode* and *address*. This opcode is then decoded at the control unit, which takes another clock cycle. The number of clock cycles involved in the execute stage depends on the type of instruction decoded by the control unit. Arithmetic operations usually takes a few clock cycles while integer multiplication, integer division and floating point division take between 10 and 40 clock cycles [125]. If the bit-flip happened at the opcode part of the instruction, it causes a wrong instruction to be executed. If the bit-flip happened at the address part of the instruction, it causes the program to access a wrong location in the memory. The benchmarks were also executed under an operating system, where there are system APIs. These system APIs causes kernel calls, and switching between kernel and user mode, between address spaces and between threads is inherently expensive [126,127]. A single kernel call require at least 1000-1500 CPU clock cycles. All of this contributes to the 1M clock cycles.

Therefore, it is imperative to detect deviation as early as possible and below 1M clock cycles (or below 4,000μs). However, there could be instances where the system crashes in less than 1M clock cycles after a fault had occurred. This happens when the fault causes the system to perform an illegal operation such as erroneously try to access a non available address, memory or instruction. When an illegal operation is performed, the system terminates the process.

3.9 Correlation Between Errors and Failures

3.9.1 Analyses of the Distribution of Failure

As mentioned in Section 2.3, a failure is said to have occurred when the system transitioned from correct service to incorrect service. A failure is caused by the presence of an error in the system where an error is the terminology used for an active or manifested fault. A fault in the system is caused by a defect in the hardware, which in this case happens to be a single bit-flip at the instruction in the pipeline. To observe the correlation between errors and failures, the failures and errors are first classified and defined according to [128]. Table 3.2 described four categories of failures and Table 3.3 defines the various type of errors which may occur.

The experiment was performed on the *QSort* benchmark, where a total of 1200 single bit-flip faults were injected (300 faults at each stage of the pipeline). This amount of fault injections is sufficient to provide 95% confidence in the test results according

TABLE 3.2: Failure categories

Failure Category	**Description**
Crash	The system stops working.
Hang	System resources are exhausted, resulted in a nonoperational system.
Fail silence violation	Either the system or the application erroneously detects the violation presence of an error or allows an incorrect data/response to propagate out.
Not manifested	The corrupted instruction is used, but it does not have a visible abnormal impact to the system.

TABLE 3.3: Error categories

Error Category	**Description**
Segmentation fault	Access violation, raised by hardware with memory protection, notifying an OS the software has attempted to access a restricted area of memory.
Invalid opcode	An illegal instruction that is not defined in the instruction set is executed.
Kernel panic	The operating system detects an error.
NULL pointer	Unable to handle kernel NULL pointer de-reference.
Bad paging	A page fault. The kernel tries to access some other bad page except NULL pointer.
Assertion error	Assertion evaluates to false at run-time.
Bad trap	Unknown exception.
General protection fault	Exceeding segment limit, writing to a read-only code or data segment, loading a selector with a system descriptor, reading an execution-only code segment.

TABLE 3.4: Statistics on failure distribution on QSort benchmark

Pipeline Stage	**Num of Injected Fault**	**Not Manifested**	**Fail Silence Violation**	**Hang**	**Crash**
Fetch	300	232	13	35	20
Decode	300	300	0	0	0
Execute	300	199	18	42	41
Load/Store	300	207	22	44	27
Total	**1200**	**939**	**52**	**121**	**88**

to [129]. Faults are injected into fetch instructions, selection of read/write registers during decode stage, result of an instruction during execution stage and finally during memory transactions in load/store stage. Table 3.4 and Figure 3.16 show the total failures observed and the number of failures observed in each pipeline. The findings from this experiment can be summarised as follows:

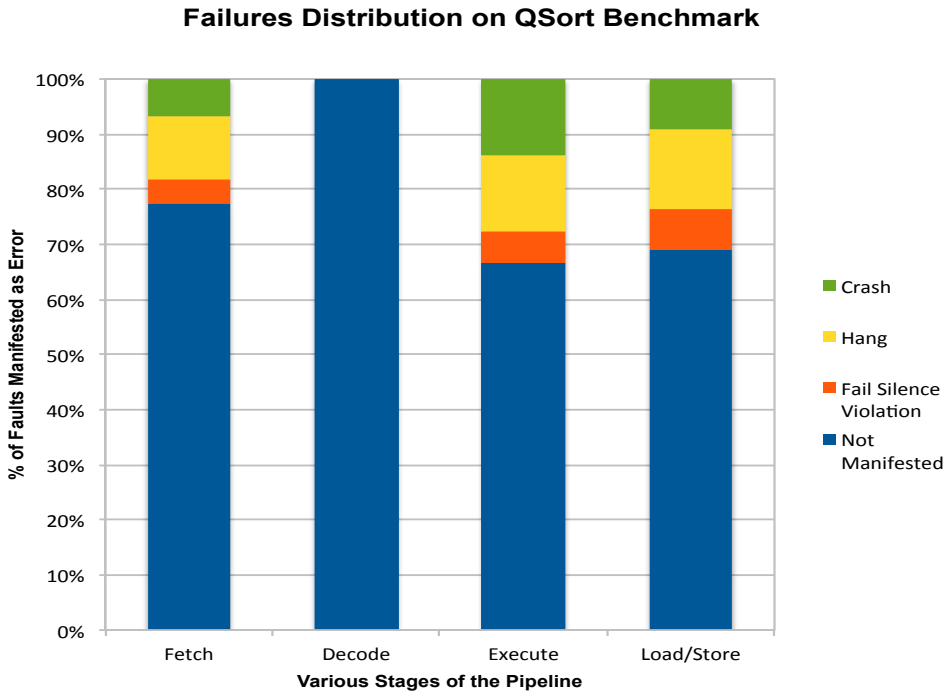


FIGURE 3.16: Percentage of failures distribution observed in the experiment conducted for QSort benchmark

- Out of 1200 faults injected in various stages of the pipeline, only a total of 261 faults manifested as errors and caused the system to behave anomalously.
- From the 261 failures observed, 121 of the failures are of type hang, 88 failures are of type crash and 52 failures are of type fail silence violation.
- The Execute stage in the pipeline is more susceptible to the presence of faults where out of 300 faults injected, 33.67% of faults were manifested as failures. This is followed by the LoadStore pipeline where there are 31.00% of faults manifested as failures and finally, in the Fetch pipeline, a total of 22.67% of faults were manifested as failures.
- An interesting observation from this experiment was none of the faults injected in the Decode stage of the pipeline were manifested as an error or caused some anomalous behaviour to the system. In fact, all 300 of the faults injected in this stage returns as Not Manifested. This could mean the corrupted register where a fault is injected was either not used during the execution or it was overwritten before the erroneous value was used, thus it did not affect the system.

As for *Dijkstra*, *FFT*, *Bitcount* and *StringSearch* benchmarks, a total of 120 single bit-flip faults were injected randomly at different bit location of each stage in the pipeline and Table 3.5, Table 3.6, Table 3.7 and Table 3.8 show the number of failures observed in

TABLE 3.5: Statistics on failure distribution on Dijkstra benchmark

Pipeline Stage	**Num of Injected Fault**	**Not Manifested**	**Fail Silence Violation**	**Hang**	**Crash**
Fetch	40	24	3	2	11
Decode	30	30	0	0	0
Execute	30	24	1	0	5
Load/Store	30	18	9	0	3
Total	**130**	**96**	**13**	**2**	**19**

TABLE 3.6: Statistics on failure distribution on FFT benchmark

Pipeline Stage	**Num of Injected Fault**	**Not Manifested**	**Fail Silence Violation**	**Hang**	**Crash**
Fetch	30	24	1	3	2
Decode	30	30	0	0	0
Execute	30	17	4	6	3
Load/Store	30	21	2	3	4
Total	**120**	**92**	**7**	**12**	**9**

TABLE 3.7: Statistics on failure distribution on Bitcount benchmark

Pipeline Stage	**Num of Injected Fault**	**Not Manifested**	**Fail Silence Violation**	**Hang**	**Crash**
Fetch	30	23	2	0	5
Decode	30	30	0	0	0
Execute	30	26	2	0	2
Load/Store	30	20	2	3	5
Total	**120**	**99**	**6**	**3**	**12**

the pipeline stages for each benchmark. From these four benchmarks, it is also observed that both Execute stage and LoadStore stage are more susceptible to the presence of faults, with the exception of Bitcount benchmark. However, this could be due to the faults being injected randomly, as well as the reduced number of experiments performed.

3.9.2 Analyses of Error Distribution and Its Effect to the System Behaviour

It is also interesting to find out what are the typical errors that cause crash and hang in the system. Table 3.9 provides the statistics on error distribution for QSort benchmark and Figure 3.17 and Figure 3.18 both show the distribution of errors that cause the system to either crash or hang. Additional results on the error distribution for Dijkstra, FFT, Bitcount and StringSearch benchmarks can be referred to in Appendix B. Based on this analysis, it can be concluded that:

TABLE 3.8: Statistics on failure distribution on StringSearch benchmark

Pipeline Stage	Num of Injected Fault	Not Manifested	Fail Silence Violation	Hang	Crash
Fetch	30	23	3	4	0
Decode	30	30	0	0	0
Execute	30	21	1	5	3
Load/Store	30	21	1	6	2
Total	**120**	**95**	**5**	**15**	**5**

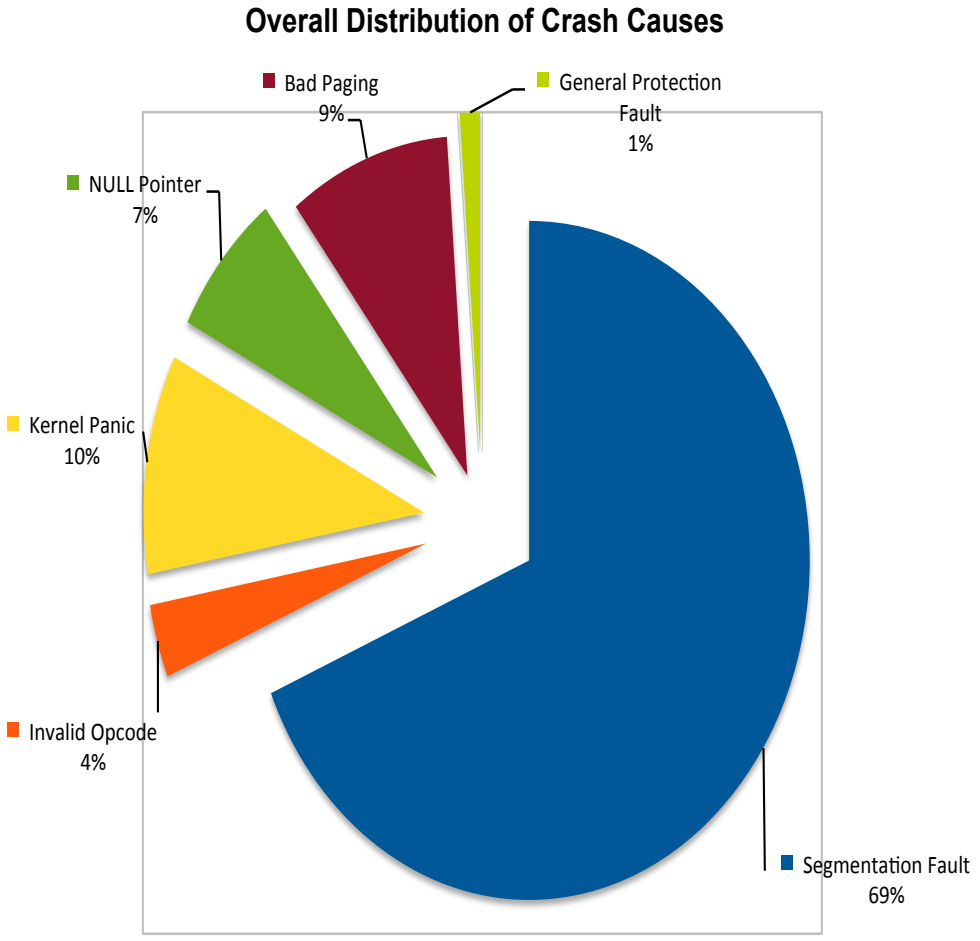


FIGURE 3.17: Analysis of different types of errors that causes crash in the system.

- The main cause of systems crashing is due to a segmentation fault, where 69% of the crashes observed were caused by this error. This is followed by kernel panic errors which account for 10% of the total. A segmentation fault is a common error that causes a system to crash. This happens when the system is trying to read from or write to an illegal memory location such as trying to access a variable which has been freed, writing to a read-only part of the memory, attempting to access memory that it does not have any right to. When a fault is injected in the

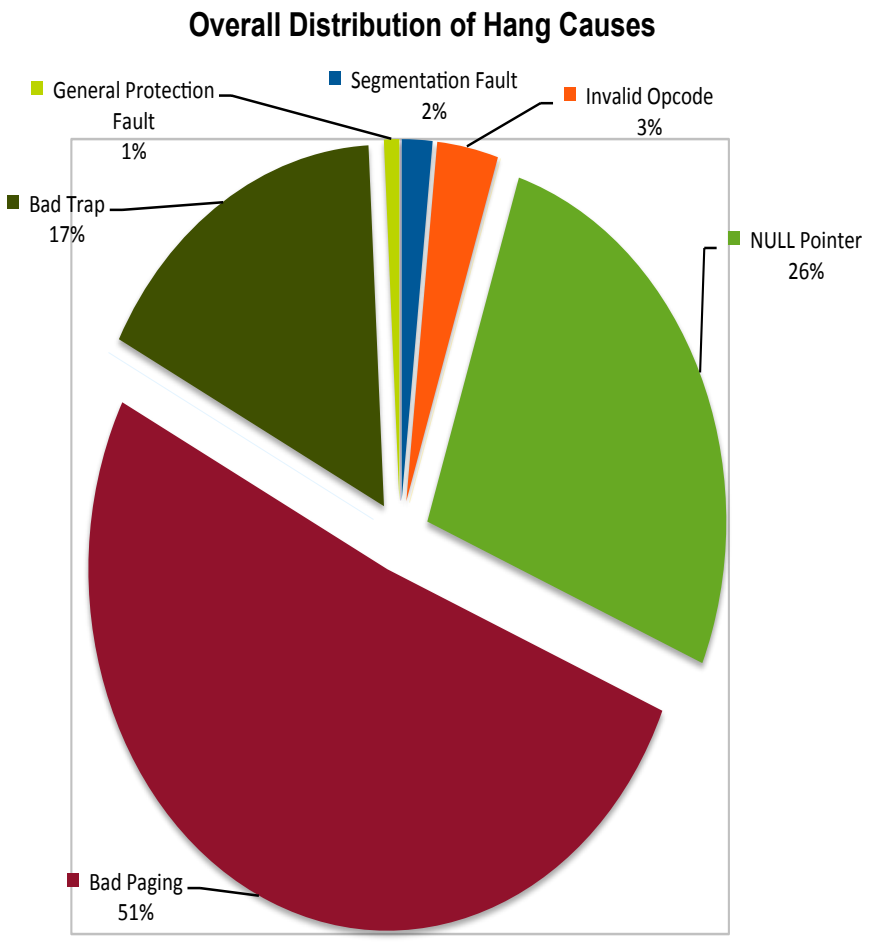


FIGURE 3.18: Analysis of different types of errors that causes hang in the system.

Execute pipeline, it alters the memory access instruction which usually leads to a segmentation fault due to the fault altering the resulting address.

- The main cause of a system resulting in a non-operational mode or hang is due to bad paging (51%), followed by a NULL pointer (26%). Bad paging is an exception raised by the system when the running program is trying to access a memory page that is not currently mapped by the memory management unit (MMU) into the virtual address space of the program. This is observed in the LoadStore pipeline where the injected fault alters the value of the address to an address located in an unmapped page.

TABLE 3.9: Statistics on error distribution for QSort benchmark

Pipeline Stage	Type of Failure	Segmentation Fault	Invalid Opcode	Kernel Panic	NULL Pointer	Bad Paging	Assertion Error	Bad Trap	General Protection Fault	No Error	Total
Fetch	Not Manifested	0	0	0	0	0	0	0	0	232	**232**
	Fail Silence Violation	0	0	0	0	0	0	0	0	13	**13**
	Hang	0	2	0	14	16	0	3	0	0	**35**
	Crash	13	2	1	0	0	3	0	1	0	**20**
Decode	Not Manifested	0	0	0	0	0	0	0	0	300	**300**
	Fail Silence Violation	0	0	0	0	0	0	0	0	0	**0**
	Hang	0	0	0	0	0	0	0	0	0	**0**
	Crash	0	0	0	0	0	0	0	0	0	**0**
Execute	Not Manifested	0	0	0	0	0	0	0	0	199	**199**
	Fail Silence Violation	0	0	0	0	0	0	0	0	18	**18**
	Hang	1	1	0	12	22	0	6	0	0	**42**
	Crash	31	0	6	2	1	1	0	0	0	**41**
Load/Store	Not Manifested	0	0	0	0	0	0	0	0	207	**207**
	Fail Silence Violation	0	0	0	0	0	0	0	0	22	**22**
	Hang	1	1	0	6	24	0	11	1	0	**44**
	Crash	15	1	2	0	5	4	0	0	0	**27**

3.10 Summary

Every system that behaves normally (i.e without fault) exhibits a pattern, and any deviation from that pattern indicates an anomalous behaviour has occurred. In this chapter, monitoring of these anomalous behaviours using HPCs is presented. Firstly, the profile of a system behaving normally is captured and plotted against time. The execution profile was captured using two different PMEs – (i) the number of instructions retired PME, and (ii) the number of cache misses PME. The results show that the execution profile for system running with different benchmarks differ from one another, but for system that runs the same benchmark but at different clock speed, the execution profiles bear striking similarities. This shows that by tracing the HPC data in a time interval and plotting the execution profile based on the data gathered, it can assist in monitoring the system for any anomalous behaviour.

Between the two PMEs, it was found that the cache miss PME is more suitable for detection where the counter data records bigger deviation (more than 10%) when the pattern begins to deviate from the normal behaviour compared to the instructions retired PME, where the deviation recorded is around 5%. Values recorded using the cache miss PME are also much lower compared to the instructions retired PME, where the values are between three and seven bits, while values recorded for instructions retired PME are around seventeen bits. Therefore, the early detection and prediction algorithm presented in Chapter 4 will use cache misses PME as the univariate time-series data. The suitable sampling interval for sampling HPC data in an embedded system was found to be at 5000ns where the amount of data generated was large enough to ensure quick detection can be performed. Sampling interval below 5000ns is not suitable as it is difficult to distinguish an anomalous point from a normal point.

Next, a single bit-flip fault is injected into a location on the processor pipeline and the behaviour of the system is captured. The failure triggered by manifested fault in the processor can be divided into a) masked error, b) fail silence violation, c) crash and d) hang. These four behaviours can be observed by using a single HPC. The work in this thesis focuses on two failures namely crash and hang. In the case of hang, a huge deviation is observed in the HPC when the fault is manifested into an error before the counter value becomes constant at a point. When the system experiences a crash, the counter also spots a huge deviation before the counter value stops. The number of clock cycles it takes for the system to fail from the time the fault is manifested as an error is about 1M clock cycles (equivalent to 4000μs).

The results in this chapter clearly show that the HPC can be used to identify two main types of failure: crash and hang. As for fail silence violation, it is a little harder as the execution profile does not deviate much from the fault free model. Each of these failures clearly exhibits different characteristics, which will be useful for developing a detector. The correlation between the type of errors that occur and the various failures

observed was also presented in this chapter. The following chapter will present the early detection algorithm which uses HPC data to detect anomalous behaviour in the system and predict potential failure in real time.

Chapter 4

Early Detection and Prediction Algorithm

4.1 Introduction

A dedicated hardware-based detector can be expensive and intrusive while a pure software-based detector, though unobtrusive, may be too slow to react. Existing online error detection techniques look at detecting errors through the failures they encounter, and very often, users are only aware of the anomalous behaviour after a failure has occurred. Chapter 3 showed how Hardware Performance Counters (HPCs) can be used to profile the behaviour of a system, and any deviation from the normal behaviour profile indicates anomalous behaviour. This chapter presents the novel algorithm that predicts potential failure in real-time by detecting anomalous behaviour in a processor using a single HPC. The main objectives of the work in this chapter are as follows:

1. Identify available forecasting techniques suitable for time-series forecasting. Based on design considerations, pattern and characteristics of the cache misses PME, three different techniques, (a) Single Exponential Smoothing (SES), (b) Autoregressive Moving Average (ARMA), and (c) Single Layer Linear Network (LN), will be investigated.

2. Devise a measurement test which distinguishes anomalous data instances from normal data instances. In particular, two methods that measure the deviation between predicted data and observed data, namely Residual Distribution and Prediction Interval will be presented. The rule to classify each observed data as normal or anomalous according to the deviation will be discussed.

3. Determine the rule in predicting potential failure. Failure in the system is not caused by point anomalies, but collective anomalies that begin when the fault has

been manifested into an error. The rule in deciding the number of consecutive anomalies to be detected before predicting potential failure will be presented.

This chapter is organised as follows. Section 4.2 describes the data sets used to develop the early detection and prediction algorithm while in Section 4.3, the characteristics and features of the HPC data sets collected are investigated. Section 4.4 presents the algorithm for prediction of potential failure through the detection of anomalous behaviour. Section 4.5 discuss about the number of consecutive anomalies to be detected in the algorithm in order to be able to predict potential failure. In Section 4.6, three different forecasting methods are presented and discussed in detail. Section 4.7 will explain the two different methods used to measure the deviation between predicted data and observed data and the rule to classify each observed data whether it is normal or anomalous. Discussion on results and analysis is presented in Section 4.8 while Section 4.9 concludes the chapter.

4.2 Generating Data Set

The Dijkstra benchmark is used as a case study in this chapter. Based on the findings presented in Chapter 3, it is found that the cache miss PME is more suitable compared to instructions retired PME for monitoring deviation in the profile. The values recorded using the cache miss PME are much lower, between three and seven bits. Besides that, the deviations recorded using the cache miss PME are also higher, making it easier to detect anomalous behaviour in the system. Hence, the cache miss PME is used to build the data sets, which are then used in the development of the early detection and prediction algorithm.

The sampling interval of 5000ns is selected to generate the required data sets as it is found that this interval duration generates sufficient amount of data for the algorithm to distinguish an anomalous point from a normal point. A total of nine data sets were obtained with the input data for each data set differ from one data set to another. All data sets contain approximately 120,000 data points and all the data sets exhibit fault-free behaviour, which means there is no anomalous behaviour detected in the execution profile. As each data set is different, all nine data sets will be used as training data sets. As for testing data set, a different data set is used. This data set contains 118,860 data points, compared to a normal data set that contains approximately 120,000 data points. This data set also contains collective anomalies. All experiments were performed using Matlab R2017b.

4.3 Understanding the Data Set

The hardware counter data obtained from the Dijkstra benchmark is a univariate type of time-series data. Time-series data is defined as a sequence of observations continuously streaming at time t and are gathered at an equally spaced time intervals [130–133]. It can be represented as $Y\{t\} = Y_1, Y_2, Y_3, ..., Y_t$. A univariate time-series is a sequence of measurements of the same variable collected over time, where in this case, the variable being the *number of cache misses* collected at every 5000ns.

To determine a suitable algorithm that can be utilised for monitoring and detecting a change in the data, the first step is to determine the data pattern in the data set. There are four different types of time-series data patterns: horizontal, seasonal, cyclical and trend. Spyros et al [134] provided a brief summary of these four types of data series:

- A *horizontal* (H) pattern exists when the data values fluctuates around a constant mean. Such data is called *stationary* in its mean.
- A *seasonal* (S) pattern exists when a series is influenced by seasonal factors. In other words, there is a clear pattern that repeats itself over fixed interval of time (e.g., the quarter of the year, monthly, or day of the week).
- A *cyclical* (C) pattern exists when the data exhibit rises and falls that are not of a fixed period. The main difference between a seasonal pattern and a cyclical pattern is that the former is of a constant length and recurs on a regular periodic basis, while the latter varies in length.
- A *trend* (T) pattern exists when there is a long-term increase or decrease in the data.

For time-series data, a *time plot* where data are plotted over time can reveal any seasonal behaviour, trend over time or any other features of the data. Figure 4.1 shows the time plot of Dijkstra benchmark with three different sets of input data. From Figure 4.1, there is no obvious trend, cyclical or seasonal patterns that is present in all three time plots. However, all three time plots show there exists a horizontal pattern (data is roughly horizontal along the time axis), which means this data could be a stationary time-series data. The time plot is also compared against Pegels' (1969) classification of trend and seasonality patterns as shown in Figure 4.2. Based on Pegels' classification, the time plot exhibits no trend and no seasonal effect.

A stationary time series is where its statistical properties like the mean, variance and autocorrelation structure do not change over time. Other than comparing time plots with Pegels' (1969) classification, one can also perform a *Unit Root Test* to determine if the time-series is stationary. Unit root tests are tests for stationarity in a time series. The

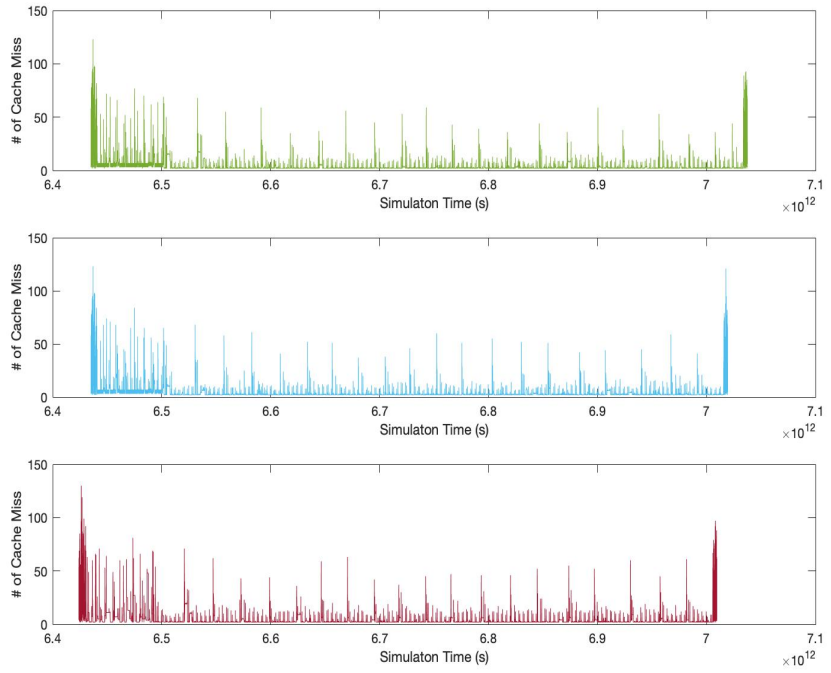


FIGURE 4.1: Time plot of Dijkstra benchmark with 3 different sets of input data

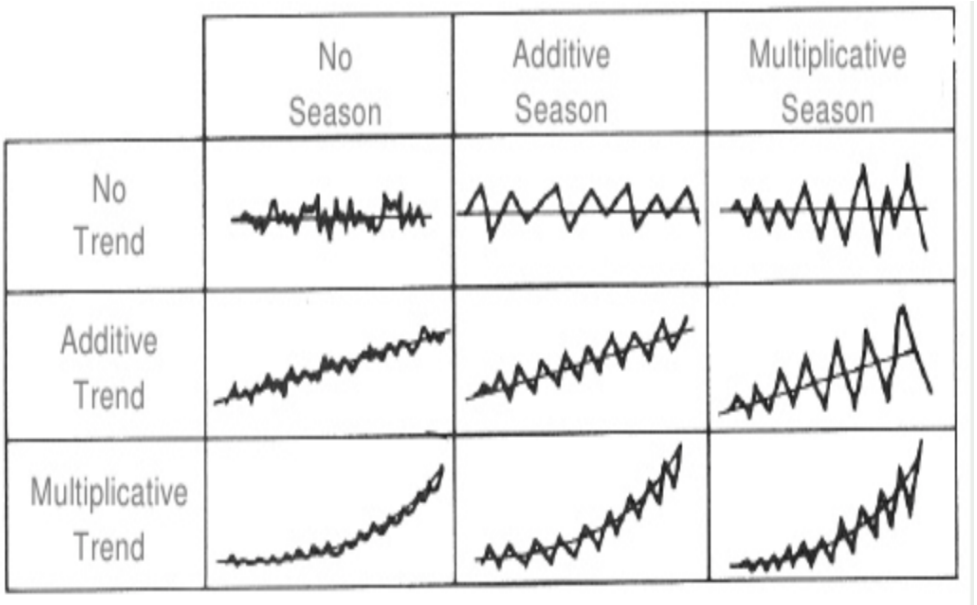


FIGURE 4.2: Patterns based on Pegels' (1969) classification

presence of unit roots are one cause for non-stationarity where it can cause unpredictable results in the analysis. The most widely used test for unit root testing is the *Augmented Dickey-Fuller* (ADF) test. The hypotheses used for the ADF test in this work are:

- The null hypothesis is that a unit root is present in the data; and

- The alternate hypothesis is that the time-series data is stationary.

The general equation used to carry out ADF test on the time-series is Equation 4.1. Since the mean of the series is non-zero, the value of α will not be restricted to 0. However, as there is no trend in the series, the value of β will be restricted to zero, thus the revised equation is Equation 4.2 using the basic regression model that has a constant and no trend:

$$\Delta Y_t = \alpha + \beta t + \gamma Y_{t-1} + \delta_1 \Delta Y_{t-1} + ... + \delta_p \Delta Y_{t-p+1} + \epsilon_t \tag{4.1}$$

$$\Delta Y_t = \alpha + \gamma Y_{t-1} + \delta_1 \Delta Y_{t-1} + ... + \delta_p \Delta Y_{t-p+1} + \epsilon_t \tag{4.2}$$

TABLE 4.1: Critical values for Dickey-Fuller t-distribution, source from [3]

Critical Values for Dickey-Fuller t-Distribution				
Sample size, T	No Trend		With Trend	
	1%	5%	1%	5%
T = 25	-3.75	-3.00	-4.38	-3.60
T = 50	-3.58	-2.93	-4.15	-3.50
T = 100	-3.51	-2.89	-4.04	-3.45
T = 250	-3.46	-2.88	-3.99	-3.43
T = 500	-3.44	-2.87	-3.98	-3.42
T = ∞	-3.43	-2.86	-3.96	-3.41

The test is then carried out under the null hypothesis $\gamma = 0$ against the alternative hypothesis of $\gamma < 0$. Once the value is computed, it is compared to the relevant critical value for ADF test. If the test statistic is less than the critical value, then the null hypothesis $\gamma = 0$ is rejected and the series is a stationary series. In general, if the *p-value* is less than 5%, the null hypothesis can be rejected. Comparison can also be made between the calculated DF_t statistic and the tabulated critical value in Table 4.1. If the calculated DF_τ is more negative than the table value, the null hypothesis is rejected. The equation for DF_τ is:

$$DF_\tau = \frac{\widehat{\gamma}}{SE(\widehat{\gamma})} \tag{4.3}$$

As the number of samples in each data set is more than 500, the critical value for $T = \infty$ is chosen. As observed in Figure 4.1 also, the benchmark data set also exhibits no trend. Hence, the 5% critical value from Table 4.1, which is -2.86, is chosen. The ADF test is performed on all nine data sets of the Dijkstra benchmark and the value of DF_τ for each sample of data is shown in Table 4.2.

From the test results shown in Table 4.2, the value of DF_τ for each data set in Dijkstra benchmark is found to be more negative compared to -2.86, therefore the null hypothesis $\gamma = 0$ is rejected and the time-series is found to be stationary.

TABLE 4.2: ADF Test Results for Dijkstra benchmark

ADF Test Results		
Sample	**Sample Size**	DF_τ
Data Set 1	122934	-91.09
Data Set 2	117091	-90.60
Data Set 3	117091	-90.60
Data Set 4	122697	-99.56
Data Set 5	116855	-96.52
Data Set 6	119122	-101.99
Data Set 7	114191	-99.54
Data Set 8	120289	-99.21
Data Set 9	120471	-100.08

4.4 Algorithm Overview

In designing the algorithm to predict potential failure in real-time through detection of anomalous behaviour, the following design considerations, which were adapted from [26, 58], were applied:

- Timeliness:
  The purpose of this algorithm is to be able to predict potential failure in the processor *before* the actual failure occurs. Therefore, it is imperative that the detection of anomalous behaviour has to be performed in real-time.
- Nature of Data:
  The data recorded using HPC is a continuous, univariate time-series data as it is recorded from one monitored PME, which is the number of cache misses.
- Data Label:
  The available data set consists of normal points, which is used to develop the model for a normal behaviour. Subsequently, the model is then tested on anomalous data.
- Rate of Change:
  The values of the PME are relatively static, whereby the changes between each data instances are rather small. Values that have sudden, huge changes indicate some anomalous behaviour.

To meet all the design requirements as outlined above, the algorithm to detect anomalous behaviour and predict potential failure using HPC data consists of three stages, with each stage building from its predecessor:

1. An algorithm to predict the next value in the time-series using a one-step ahead prediction method;

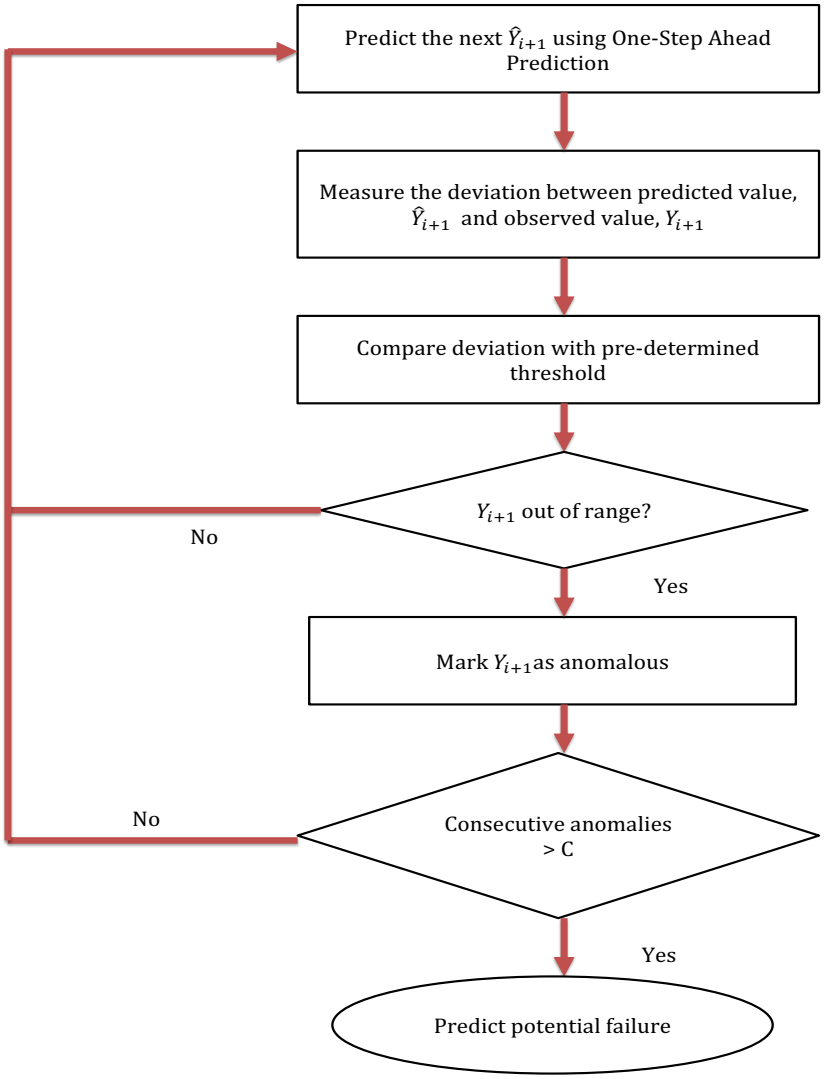


FIGURE 4.3: Early detection algorithm using hardware performance counter

2. Measurement of deviation between the predicted value and the observed value at current time;

3. A mechanism to classify if the observed value deviates "too much" and is deemed anomalous.

Figure 4.3 shows the algorithm for early detection of anomalous behaviour using a univariate type of time-series data gathered from HPC. Briefly, one-step ahead prediction is used to predict the next sequential data $\hat{Y}_{t+1}$. The predicted value will be measured against the observed value and the observed value will be classified as anomalous if

it falls outside the defined threshold. An alarm for a potential failure is raised if the number of anomalous points detected consecutively exceeds a predefined value, else, the actual observed value Y_{t+1} is added to the front of the series and the next sequential data is predicted.

4.5 Predicting Potential Failure

The main objective of this algorithm is to predict a potential failure in the system before the system experiences a failure. As mentioned in Chapter 2, the type of anomalies in this study is collective anomalies, which means, the failure of the system is not caused by one anomalous point, but rather, it is caused by a group of points. These points that have been identified as anomalous indicate the beginning of the system experiencing a failure. In Chapter 3, one of the behaviours exhibited by the counter when a fault has manifested into an error is that the counter begins to deviate from the normal behaviour. This leads to the crux of the algorithm, that is, to determine how many consecutive anomalies are required to be detected before raising an alarm on potential failure.

The optimal value of consecutive anomalies, denoted by C, is one that is able to predict potential failure in the shortest time possible but at the same time, avoid being overly sensitive. As shown in Chapter 3, the minimum amount of clock cycles it takes for a system to crash is about 1M or 4,000μs. Essentially, this means the algorithm needs to detect and predict potential failure below 1M clock cycles or 4,000μs. In this thesis, the following assumption is made:

The minimum value for C is 4. This means the algorithm has to detect at least four consecutive anomalies before raising an alarm. If $C \leq 4$, this creates an overly sensitive algorithm where an inappropriate alarm is raised on normal points wrongly identified as anomalies.

The optimal value for parameter C is presented in Section 4.8.2.

4.6 One-Step Ahead Prediction

As presented in Chapter 2, anomaly detection has been applied in various domains, and one of the domain is Damage Detection where anomaly detection is used for detecting damage in advance to minimise losses, prevents further escalation and reduces risk. However, instead of performing damage detection on mechanical components or structural components, the damage detection in this work is performed on an electrical component, in this case, being a processor. To detect manifested error as early as possible and predict possible failure, it requires time-series forecasting, i.e. making prediction

of the next data instance based on a model fitted to present and past data instances. Applying a threshold rule on the HPC data will not work because the anomalies that occur in HPC data are not point anomalies, but collective anomalies. Using an overall thresholding rule on the data set will not only result in high false alarms, but it will not be able to predict a potential failure in a timely manner. Schmueli et al. [135] gave a very clear distinction between time-series forecasting and time-series analysis as quoted below:

> In descriptive modelling, or time-series analysis, a time-series is modelled to determine its components concerning seasonal patterns, trends, relation to external factors, and the like. In contrast, time-series forecasting uses the information in a time-series (perhaps with additional information) to forecast future values of that series.

There is a variety of forecasting methods, and the choice depends on many factors, such as, type of data, background knowledge, the objectives to be achieved and others. In order to detect anomalies on-chip and in real time, the selection of one-step ahead prediction methods must satisfy the following requirements:

1. Minimal computational complexity; and
2. Does not require any pre-processing on the data;

As shown in 4.3, the nature of the data set is a univariate time-series data that is found to be stationary with no trend or seasonality. Other design considerations such as the unavailability of data label, the timeliness and the rate of change in the data also play a role in determining the suitable forecasting methods. The selection of a proper model is extremely important as it reflects the underlying structure of the series and this fitted model will in turn, be used for future forecasting [132]. Simple Moving Average and Exponential Weighted Moving Average were used to model the structure of the series and forecast the next data, but these methods not only gives a low accuracy and high detection time, but also produces high false alarms. Other methods like Replicator Neural Network (also known as Autoencoder) and Principal Component Analysis were not suitable as well because these methods require the entire fitted model to be stored in memory, which means, it requires huge amount of storage. Replicator Neural Network is also not suitable to be used for forecasting the next data point. It is a good method to replicate the whole data set from time $t = 1$ until time $t = n$, but it is unable to forecast the next data point, $t = n + 1$ and beyond.

Three other forecasting methods are considered namely, (a) Single Exponential Smoothing (SES), (b) Autoregressive Moving Average (ARMA), and (c) Single Layer Linear Network (LN). Initial results have been promising, where the detection time is lower and

accuracy rate is higher compared to Simple Moving Average and Exponential Weighted Moving Average. The amount of data to be stored in the memory for processing is also much lower compared to Replicator Neural Network and Principal Component Analysis. Based on these requirements, three forecasting methods have been selected namely, (a) Single Exponential Smoothing (SES), (b) Autoregressive Moving Average (ARMA), and (c) Single Layer Linear Network (LN).

A report by Makridakis et al. [136] provides an interesting finding on how statistical forecasting methods such as SES and ARMA outperform other machine learning algorithms such as K-Nearest Neighbour Regression (KNN), Bayesian Neural Network (BNN), Support Vector Regression (SVR) and many more in terms of accuracy and forecasting performance while at the same time ensuring low computation time and complexity. The findings from this report further strengthen the choice in choosing SES, ARMA and LN methods for one-step ahead prediction.

Another factor that plays an important role in time-series forecasting is the forecast horizon. Forecast horizon, denoted by h, is the length of time into the future for which the forecast data is prepared and could take the value of $h = 1, 2, 3, ...$ However, researchers in [137] had shown that for regular or fast-moving data, forecasting in a short term horizon is more accurate compared to longer forecasting horizon. In this thesis, the forecast horizon, h is set at 1 (hence, the name one-step ahead). Forecasting one-step ahead allows the deviation between predicted value and observed value to be observed as soon as possible. If there is a huge deviation between predicted value and observed value, an anomaly may have occurred. If the forecast horizon is set at several time steps away, there will be a delay in detecting the anomalous behaviour as the comparison between predicted value and observed value has to be made at the same time stamp.

4.6.1 Single Exponential Smoothing

Single Exponential Smoothing (SES) [33, 133, 138] method is a type of exponential smoothing prediction method that uses historical data and assigns weights to forecast future values. The one-step-ahead forecast for time $t+1$ is a weighted average of all the observations in the series $Y_1, \cdots, Y_t$. The rate at which the weights decrease is controlled by the parameter α, also known as the smoothing parameter. Weights are decreased exponentially as data is further in the past. In other words, the older the data, the smaller the weight is associated with as shown in Equation 4.4 with α between 0 and 1.

$$\hat{Y}_{t+1} = \alpha Y_t + \alpha (1-\alpha) Y_{t-1} + \alpha (1-\alpha)^2 Y_{t-2} + \cdots , \tag{4.4}$$

However, according to [138], SES can also be represented in a component form. As the time-series does not exhibit any trend or seasonal pattern, the only component included is the level, l_t. Component form representations of SES comprises a forecast equation and a smoothing equation for each of the components included in the method and is given by Equation 4.5.

$$\begin{aligned} &\text{Smoothing Equation:} \quad l_t = \alpha Y_t + (1-\alpha)\, l_{t-1} \\ &\text{Forecast Equation:} \quad \hat{Y}_{t+1} = l_t \end{aligned} \tag{4.5}$$

$\hat{Y}_{t+1}$ consists of the weighted average of the most recent observation Y_t and the smoothed value of the series, l_{t-1}. The smoothing parameter, α, is used to smooth or dampen older observations and it takes a value between 0 and 1. If α is small, i.e. close to 0, past observations are given more weight. Vice-versa, if the value of α is big, i.e. close to 1, more weight is given to the more recent observations.

In order to start the SES prediction process, the initial smoothed value, denoted as l_0 needs to be estimated. l_0 is needed in the recursive calculations that start with $l_1 = \alpha Y_1 + (1-\alpha)\, l_0$. There are two commonly used methods to estimate the initial l_0 [33, 131, 138, 139]:

1. Set $l_0 = Y_0$
2. Take the average of the available data or a subset of the available data, $\bar{Y}$, and set $l_0 = \bar{Y}_1$

For large data sets, the estimation of l_0 has little relevance [131]. However, it is important to note that smaller the value of α, the more sensitive the forecast will be on the initial forecast value, l_1. In this work, the initial forecast value, l_0 is set to the initial value of the time series as given in Equation 4.6:

$$l_0 = Y_0 \tag{4.6}$$

Once the initial smoothed value, l_0 has been set, it is substituted into Equation 4.5, where $l_1 = \alpha Y_1 + (1-\alpha)\, l_0$. The first predicted value at time 2, $\hat{Y}_2$ takes the value of l_1. Thus, the current smoothed value, l_t is an interpolation between the previous smoothed value, l_{t-1} and the current observation, Y_t. The forecast value for the next period, $\hat{Y}_{t+1}$ is simply the current smoothed value.

A typical practice in determining the optimal value for parameter α is by optimising the selected forecast-error metric. The forecast-error metric chosen for SES is *Mean Absolute Error* (MAE). The difference, or the residual between the actual observation

Y_t for the time period t and the forecast value $\hat{Y}_t$ of the same period is given by Equation 4.7:

$$e_t = \hat{Y}_t - Y_t \tag{4.7}$$

MAE is defined as a measure of the average absolute deviation between forecast values and observed (or original) values $|e_t| = \left|Y_t - \hat{Y}_t\right|$ and it shows the magnitude of the overall error occurred due to forecasting [3,131,132,134]. In MAE, the effects of positive and negative errors do not cancel out. To get a good forecast, the value of MAE should be as small as possible. MAE can be defined using Equation 4.8:

$$MAE = \frac{1}{n}\sum_{t=1}^{n}|e_t| \tag{4.8}$$

The SES model with α value that produces the smallest MAE is selected. In developing the model, the principle of parsimony is followed where the simplest model with the smallest possible number of parameters is to be selected to provide an adequate representation of the underlying time-series data [132].

The first step of this experiment involves obtaining the optimal value of α. The training data used involves all nine data sets. The α value is varied between 0 and 1 in increments of 0.1. The MAE is obtained for each α value for each data set during training, and the MAE values from all nine data sets are then averaged with the results as shown in Table 4.3. As can be seen from Table 4.3, the lowest mean MAE achieved is 0.49 when $\alpha = 0.7$.

The next step is to validate how well the model performs on the data that were not used when fitting the model. For this experiment, Data Set 8 was used as the training data, and Data Set 1, Data Set 3, Data Set 5 and Data Set 9 were used as the validation data. As had been mentioned in Section 4.1, each data set uses different input data for the benchmark. By using different sets of data, this helps to provide a reliable indication of how well the model is likely to forecast on new data. The validation of the SES model generated using $\alpha = 0.7$ identified in the previous step is presented in Section 4.6.4.

TABLE 4.3: Average Mean Absolute Error (MAE) for different α values in SES

Alpha (a)	**MAE**									**Average**
	Dataset 1	**Dataset 2**	**Dataset 3**	**Dataset 4**	**Dataset 5**	**Dataset 6**	**Dataset 7**	**Dataset 8**	**Dataset 9**	
0.0	9.3505	9.3135	9.3135	8.2542	8.2847	8.2410	8.2384	8.2525	8.2379	8.61
0.1	0.6670	0.6155	0.6155	0.7811	0.7287	0.7082	0.7214	0.7001	0.7020	0.69
0.2	0.5538	0.5064	0.5064	0.6667	0.6345	0.6265	0.6397	0.6134	0.6212	0.60
0.3	0.5002	0.4593	0.4593	0.6114	0.5936	0.5900	0.6020	0.5742	0.5829	0.55
0.4	0.4668	0.4326	0.4326	0.5778	0.5687	0.5673	0.5790	0.5505	0.5585	0.53
0.5	0.4447	0.4148	0.4148	0.5564	0.5515	0.5518	0.5639	0.5352	0.5419	0.51
0.6	0.4299	0.4030	0.4030	0.5427	0.5414	0.5424	0.5545	0.5264	0.5311	0.50
0.7	0.4203	0.3962	0.3962	**0.5356**	**0.5364**	**0.5386**	**0.5499**	**0.5227**	**0.5257**	**0.49**
0.8	0.4153	**0.3940**	**0.3940**	0.5364	0.5398	0.5418	0.5523	0.5258	0.5275	0.49
0.9	**0.4149**	0.3962	0.3962	0.5425	0.5484	0.5499	0.5597	0.5333	0.5345	0.50
1.0	0.4196	0.4027	0.4027	0.5547	0.5627	0.5637	0.5731	0.5466	0.5227	0.51

4.6.2 Autoregressive Moving Average

The second forecasting method is the Autoregressive Moving Average (ARMA) method, also known as the Box-Jenkins method [140]. ARMA has been widely used for forecasting as it is suitable for univariate time-series modelling [130–132, 134]. An ARMA(p, q) model is a combination of an autoregressive (AR) part and a moving average (MA) part. An AR(p) involves coefficients φ_t with $t = 1, ..., p$ that reflects the relationship between $\hat{Y}_{t+1}$ and the past values of the time-series. As mentioned in [132], AR(p) can be expressed mathematically as shown in Equation 4.9:

$$\begin{aligned} Y_t &= c + \varphi_1 Y_{t-1} + \varphi_2 Y_{t-2} + \cdots + \varphi_p Y_{t-p} + \epsilon_t \\ &= c + \sum_{i=1}^{p} \varphi_i Y_{t-i} + \epsilon_t \end{aligned} \tag{4.9}$$

Here, Y_t and ϵ_t refers to the actual value and white noise at time t, c is the constant and $\varphi_i (i = 1, 2, \ldots, p)$ are the AR model parameters, and p is the order of the model. The MA part involves coefficients θ_t with $t = 1, \ldots, q$ which reflects the relationship between $\hat{Y}_{t+1}$ and the residues. This can be expressed mathematically as shown in Equation 4.10:

$$\begin{aligned} Y_t &= \mu + \theta_1 \epsilon_{t-1} + \theta_2 \epsilon_{t-2} + \cdots + \theta_q \epsilon_{t-q} + \epsilon_t \\ &= \mu + \sum_{j=1}^{q} \theta_j \epsilon_{t-j} + \epsilon_t \end{aligned} \tag{4.10}$$

μ is the mean of the series and $\theta_i (i = 1, 2, \ldots, q)$ are the MA model parameters, and q is the order of the model. The data does not require any differencing as it is found to be stationary, as shown in Section 4.3. Autoregressive (AR) and Moving Average (MA) models can be effectively combined to form a general and useful class of time-series models, known as the ARMA models. Following [131, 132], an ARMA (p, q) model can be defined mathematically as shown in Equation 4.11:

$$\begin{aligned} \hat{Y}_{t+1} &= c + \varphi_1 Y_{t-1} + \cdots + \varphi_p Y_{t-p} + \epsilon_t - \theta_1 \epsilon_{t-1} - \cdots - \theta_q \epsilon_{t-q} \\ &= c + \epsilon_t + \sum_{i=1}^{p} \varphi_i Y_{t-i} + \sum_{j=1}^{q} \theta_j \epsilon_{t-j} \end{aligned} \tag{4.11}$$

where $\hat{Y}_{t+1}$ is the variable to be predicted using previous samples of the time-series, ϵ_t denotes white noise, and c is a constant offset.

Once the model has been described, the next concern is to select the appropriate model that can produce accurate forecast based on the historical data and determine the optimal model orders. Statisticians George Box and Gwilym Jenkins developed a practical approach called the Box-Jenkins methodology to build ARMA model, which best fit to a given time-series and also satisfy the parsimony principle. The Box-Jenkins methodology uses a three-steps iterative approach namely *model identification*, *parameter estimation*, and *model verification* to determine the best model from a general class of ARMA models [131, 132, 140]. This three-step process is repeated several times until a satisfactory model is finally selected and can be used for forecasting future values of the time-series.

A crucial step in deciding an appropriate model is to determine the model's optimal parameters including the coefficients denoted by p and q. To assist in identifying the suitable ARMA model, one of the popular method is by using the Autocorrelation Function (ACF) and Partial Autocorrelation Function (PACF) plots. However, ACF and PACF only give an estimation of what parameters p and q can be and cannot be used to measure the suitability of an ARMA model. Another widely used measure for model identification and parameter estimation is the *Akaike Information Criterion* (AIC) developed by Akaike in 1974 [141]. AIC is used to estimate the quality of each model, where the less amount of information the model loses, the better the quality of that model. The higher the quality of the model, the lower the value of AIC.

AIC avoids both the risk of overfitting and the risk of underfitting, and finding a balance between the goodness of fit of the model and the simplicity of the model. One way of selecting the optimal ARMA model order is by choosing the number of model parameters has the lowest AIC. AIC value for each corresponding model was obtained using Equation 4.12:

$$AIC = 2(k) - 2log(L) \tag{4.12}$$

where L is the maximum value of the likelihood function for the model, and k is the number of estimated parameters in the model. The model's parameters are estimated according [131]. By varying the coefficients (p, q) between 0 and 6 in increments of 1, a total of forty-eight ARMA models with different orders of model parameters were built, with the exception of $(0, 0)$. An ARMA $(0, 0)$ model is used on a time-series that contains basically a constant and white noise. Since the time-series obtained from the Dijkstra benchmark does not consists of a constant and white noise, ARMA $(0, 0)$ is not considered.

TABLE 4.4: Average Akaike Information Criterion (AIC) for different orders of ARMA model

Model Parameter (p , q)	**Average AIC**						
	q = 0	**q = 1**	**q = 2**	**q = 3**	**q = 4**	**q = 5**	**q = 6**
p = 0	NIL	6.541E+05	6.386E+05	6.233E+05	6.140E+05	6.104E+05	6.063E+05
p = 1	6.059E+05	5.876E+05	5.820E+05	5.819E+05	5.806E+05	5.802E+05	5.802E+05
p = 2	5.988E+05	5.824E+05	5.819E+05	5.817E+05	5.804E+05	5.801E+05	5.801E+05
p = 3	5.875E+05	5.824E+05	5.815E+05	5.807E+05	5.803E+05	5.801E+05	5.797E+05
p = 4	5.858E+05	5.800E+05	5.799E+05	5.799E+05	**5.798E+05**	5.798E+05	5.794E+05
p = 5	5.854E+05	5.799E+05	5.799E+05	5.799E+05	5.798E+05	5.794E+05	5.793E+05
p = 6	5.839E+05	5.799E+05	5.799E+05	5.796E+05	5.794E+05	5.794E+05	5.793E+05

Up to six autoregressive parameters and six moving average parameters are used to determine the best ARMA model for this problem. Increasing the number of parameters increases the complexity of the model by nearly 5% in terms of execution time [142]. Each data set is trained with all forty-eight ARMA models and the AIC value for each model on all nine data sets are calculated. Table 4.4 shows the average AIC values for all forty-eight ARMA models with different orders of model parameters. As can be observed from Table 4.4, the lowest AIC value obtained is 5.793 x 10^5 when $p = 6$ and $q = 6$. However, based on the initial experiments conducted, it is found that the detection accuracy reduces significantly while the detection time increases significantly for models having coefficients greater than 4. Due to that, a trade-off is made where the coefficients for p and q which is 4 is chosen.

Based on this result, the most suitable ARMA model to be used for one-step ahead prediction was found to be ARMA (4, 4). After deciding the model parameters, the final step is to perform model verification where the developed model is verified against validation data set. Data Set 8 was used as training data to develop ARMA (4,4) model while Data Set 1, Data Set 3, Data Set 5 and Data Set 9 were used as validation data sets. The MAE scores calculated between the model and the validation data sets are presented in Section 4.6.4.

4.6.3 Single Layer Linear Network

The third forecasting method chosen is the Single Layer Linear Network (LN) (also known as a single layer perceptron network) [57, 143]. LN is derived from Artificial Neural Networks (ANNs) approach, an alternative and popular technique to time-series forecasting. Inspired by the way biological neural networks in a human brain process information, ANNs' objective is to try to recognise regularities and patterns in the input data, learn from experience and then provide generalised results based on their known previous knowledge [132]. ANNs have been applied on a variety of tasks, such as speech recognition, machine translation, computer vision, social network filtering, playing board and video games and medical diagnosis.

There are several types of ANN architectures used in forecasting problems such as Single Layer Perceptrons (SLPs), Multi-Layer Perceptrons (MLPs) and Time-Lagged Neural Networks (TLNNs). SLPs, MLPs and TLNNs are derived from feed-forward neural network type and Sanger [144] has given a clear definition of the feed-forward neural network quoted below:

> It is a network which has a distinct set of input units onto which values are clamped. These values are then passed through a set of weights to produce the inputs to the next layer of "hidden" or internal units. These units modify the input using a non-linear function (usually in the shape of a sigmoid) to

produce the outputs. As many layers as desired of this form can be stacked, and the units of the final layer become the outputs of the network.

TLNN is not applicable for this study as TLNN is used for seasonal time-series data where the input nodes are the time-series values at some particular lags. In this work, SLPs is favoured over MLPs as the computational complexity for MLPs is much higher due to the presence of additional hidden layers. The hidden neurons and output neurons in the additional layers causes the computational time to increase. Overall, simulation time (or the time cost/complexity) increases linearly with the increase in pattern count (or neuron count in both hidden and output layers) [145]. Figure 4.4 illustrates how the input layers are connected to an output layer in a SLP and the information flows only in one direction.

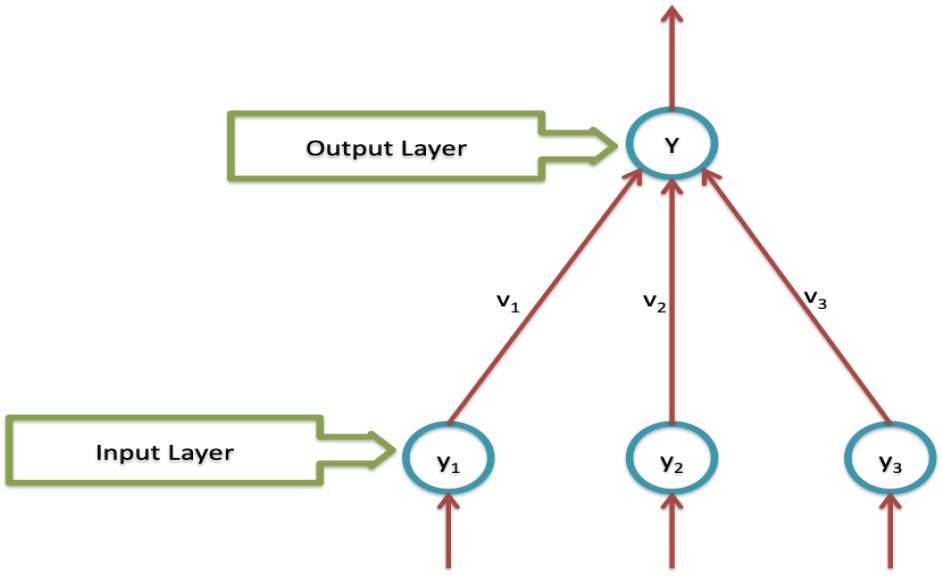


FIGURE 4.4: Single Layer Perceptron

LN is a type of SLP without any hidden layer, and the connections between the nodes do not form a cycle. Without the hidden layer, LN is a function of a linear combination of the input variable [146] and is the simplest form of neural network. It consists of a single layer of output nodes where the inputs are fed directly to the output which takes a weighted sum of all its inputs. It can be represented mathematically as shown in Equation 4.13:

$$Y = f\left(\sum_i v_i Y_i\right) \tag{4.13}$$

In the LN method, the next data $\hat{Y}_{t+1}$ is predicted as a linear combination using previous data multiplied by a set of weights represented by $v_0, v_1, ..., v_{W-1}$. The amount of

previous data used in the one-step ahead prediction is determined by the size of the sliding window, denoted by W. This can be mathematically expressed using Equation 4.14:

$$\hat{Y}_{t+1} = \frac{\left(\sum_{i=0}^{W-1} v_i Y_{t-i}\right)}{\sum_{i=0}^{W-1} v_i} \tag{4.14}$$

It defines the relationship between the sliding window $Y_{t-W}, ..., Y_t$ and the predicted value of $\hat{Y}_{t+1}$. The LN model has the same mathematical form as the traditional autoregressive (AR) model of Box and Jenkins, and thus has similar capabilities [57]. Following [143], the weight vectors are assigned as $1, 2, ..., v$ with the weight vector to be inversely proportional to the distance between each point in the sliding window, that is, the further point Y_t from $\hat{Y}_{t+1}$, the smaller the weight vector will be. Thus, the size of the sliding window, W, plays an important role in determining the optimal LN model for predicting one-step ahead. Unlike other popular methods such as the Delta Learning Rule or Yule-Walker equation to determine the parameters of this model (i.e. $v_0, v_1, ..., v_{W-1}$), the weight vector is assigned to be inversely proportional to the distance between each point in the sliding window. This is done to ensure that the forecasting model created is robust and applicable to any type of process. Selection of the optimal value for parameter W in LN model is based on the *Mean Absolute Error* (MAE) as the chosen forecast-error metric given in Equation 4.8.

Following the steps taken in determining the optimal values for parameters in SES and ARMA, finding the optimal window size, W involves using all nine data sets. For each data set, the value W is varied between 1 and 10 in increments of 1. The MAE value obtained for each W size from all nine data sets are then averaged and the results is as shown in Table 4.5. From the result in Table 4.5, the lowest average MAE obtained was 0.50 when $W = 3$.

Once the optimal window size has been determined, the developed LN model is verified against validation data set. For this purpose, Data Set 8 was used as training data to develop LN model and Data Set 1, Data Set 3, Data Set 5 and Data Set 9 were used as validation data sets. The MAE scores calculated between the LN model and validation data sets are presented in Section 4.6.4.

TABLE 4.5: Mean Absolute Error (MAE) for different size of sliding window, W in a LN model

Window Size, (W)	**MAE**									**Average MAE**
	Dataset 1	**Dataset 2**	**Dataset 3**	**Dataset 4**	**Dataset 5**	**Dataset 6**	**Dataset 7**	**Dataset 8**	**Dataset 9**	
1	**0.4197**	0.4028	0.4028	0.5548	0.5628	0.5637	0.5732	0.5467	0.5472	0.51
2	0.4208	**0.4003**	**0.4003**	0.5527	0.5573	0.5621	0.5732	0.5425	0.5447	0.51
3	0.4264	0.4042	0.4042	**0.5456**	**0.5491**	**0.5537**	**0.5667**	**0.5342**	**0.5386**	**0.50**
4	0.4367	0.4134	0.4134	0.5494	0.5518	0.5554	0.5695	0.5365	0.5415	0.51
5	0.4491	0.4233	0.4233	0.5571	0.5581	0.5610	0.5747	0.5417	0.5476	0.52
6	0.4608	0.4322	0.4322	0.5631	0.5620	0.5647	0.5784	0.5456	0.5519	0.52
7	0.4710	0.44	0.44	0.5715	0.5685	0.5706	0.5846	0.5519	0.5587	0.53
8	0.4812	0.4473	0.4473	0.5806	0.5757	0.5771	0.5915	0.5584	0.5658	0.54
9	0.4909	0.4545	0.4545	0.5897	0.5830	0.5837	0.5982	0.5652	0.5729	0.54
10	0.500	0.46	0.46	0.5985	0.5901	0.5899	0.6047	0.5717	0.5797	0.55

4.6.4 Comparison between Forecasting Methods

This section presents the comparison between three different forecasting methods, namely SES, ARMA and LN. The optimal model for each method was obtained using Data Set 8 as training data. Besides using Data Set 1 as a validation data set, the optimal models were also verified against Data Set 3, Data Set 5 and Data Set 9, which were used as validation data sets. Table 4.6 shows the performance of each data set forecasting method against four different validation data sets.

TABLE 4.6: Comparison between different forecasting methods in One-Step Ahead Prediction

Training Data (Dataset 8)	**MAE (Dataset 1)**	**MAE (Dataset 3)**	**MAE (Dataset 5)**	**MAE (Dataset 9)**
SES (α= 0.6) MAE = 0.526	1.91 (+3.63%)	2.12 (+4.03%)	1.36 (+2.59%)	1.42 (+2.70%)
ARMA(4,4) MAE = 0.592	1.86 (+3.14%)	2.06 (+3.48%)	1.32 (+2.23%)	1.38 (+2.33%)
LN (W = 3) MAE = 0.534	1.87 (+3.50%)	2.08 (+3.90%)	1.34 (+2.51%)	1.39 (+2.60%)

The MAE obtained from Data Set 8 for all three forecasting methods are used as the baseline. For example, in Table 4.6, the baseline MAE for SES method is 0.526. When the forecast model is used to validate Data Set 1, the MAE obtained is 1.91, an increase of 3.63%. This indicates that the forecast model is not overfitting or underfitting. A model that is underfit will have high training and high validation error while an overfit model will have extremely low training error but a high validation error. The results in Table 4.6 shows a low validation error, less than 5% in all data sets.

From the results shown in Table 4.6, all three forecasting methods are comparable with one another. However, the forecast model developed using the ARMA provides the lowest MAE score on all validation data sets compared to the SES and LN methods. This indicates that ARMA (4,4) creates a better forecast model for one-step ahead prediction.

4.7 Measurement of Deviation and Anomaly Classification

Once the next data has been predicted using either the SES, ARMA or LN forecasting methods, the next step is to define a measure to determine how much the observed behaviour of the time-series deviates from the expected pattern. If the observed value falls outside the defined threshold, it is classified as anomalous. Two different methods have been selected to measure the deviation between the observed data from the expected data, namely: (a) Residual Distribution; and (b) Prediction Interval, which will be explained further in Section 4.7.1 and Section 4.7.2.

To test how well the predicted model can be used to classify anomalies, a different Dijkstra data set is used — a testing data set that has not been trained or validated, and which contains anomalies. The anomalous dataset contains 118,860 data points, compared to a normal dataset that contains approximately 120,000 data points. The starting point for the occurrence of anomalies were detected at point 118,072, and ends at point 118,860 since these are collective anomalies type. Figure 4.5 shows the region of collective anomalies that occurred in the anomalous dataset as compared to a fault-free dataset.

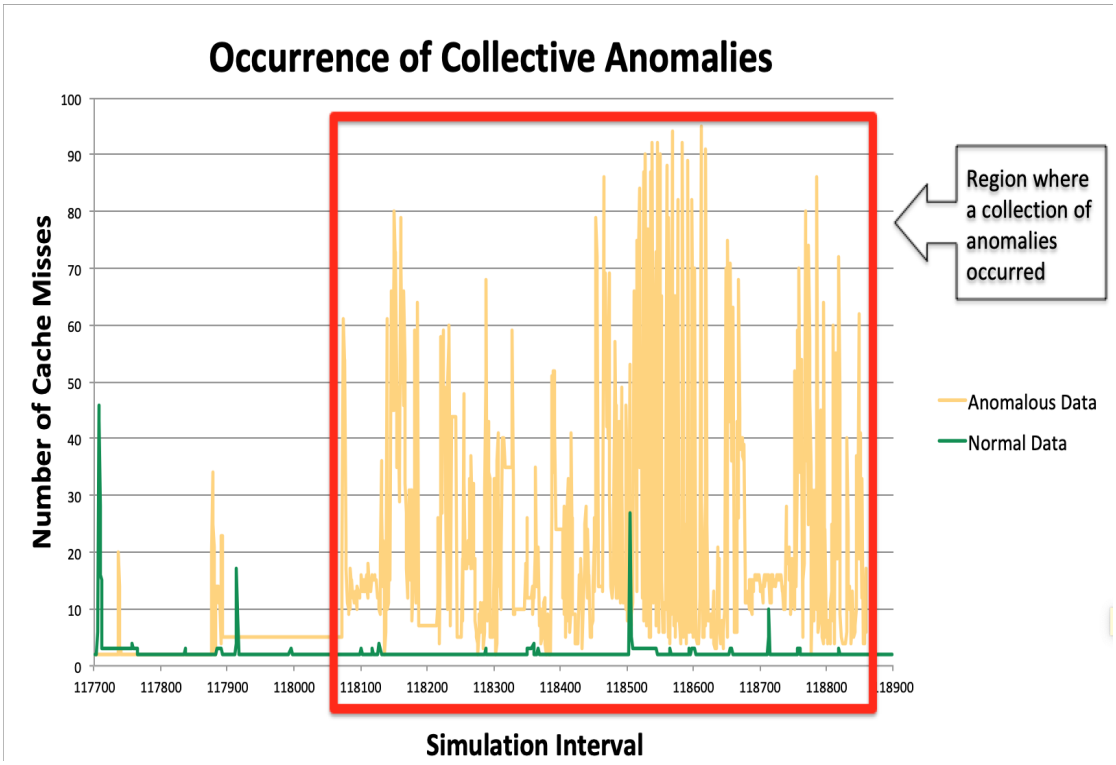


FIGURE 4.5: Collective anomalies which occurred in the Dijkstra anomalous dataset

4.7.1 Residual Distribution

This method is adapted from [33] where the residual at specific time t is used to define the deviation between predicted value and observed (or actual) value. The deviation between observed data and predicted data at time t is known as the forecast residual and is given in Equation 4.7. Analysis of residuals is done by looking at the distribution of the residuals for each method. Graphical methods are used to examine residuals and one of the common methods is to use a histogram to display the distribution of a group of residuals.

The residuals were obtained by calculating the absolute difference between actual value observed in the validation Data Set 1 against the predicted value from the models

developed using SES, ARMA or LN method at time t. The residuals series for each method is then plotted using a histogram and the result is shown in Figure 4.6.

As can be seen from Figure 4.6, the histograms show an approximately normal distribution curve. However, while they resemble the normal distribution family of curves, all three histograms have a much taller peak and the tails decay much slower compared to a normal distribution. This type of distribution is known as Cauchy Distribution [139,147], where the residuals are not normally distributed due to too many extreme positive and negative residuals. In other words, the distribution is *heavy tailed*. This can be observed in Figure 4.7 where the probability plot shows the residuals are not distributed normally. In contrast, a normal probability plot of the residuals is approximately linear supporting the condition that the error terms are normally distributed.

Unlike Normal distribution that is centred around zero-mean, μ, with a standard deviation, σ, a Cauchy Distribution has its mean undefined and the variance is infinite. However, it is possible to calculate the residual average of a sample, denoted by n, in a Cauchy distribution [148], where n is the total number of observations up until time t, and determine the current residual is lying how many standard deviations away from the average of the forecast residual. The equation for Residual Distribution as shown in Equation 4.17 consists of two main components — residual average, $\bar{e}$ and residual variance, σ^2 shown in Equation 4.15 and 4.16:

$$\text{Residual average,} \quad \bar{e} = \frac{\sum_{i=1}^{n} |e_i|}{n} \tag{4.15}$$

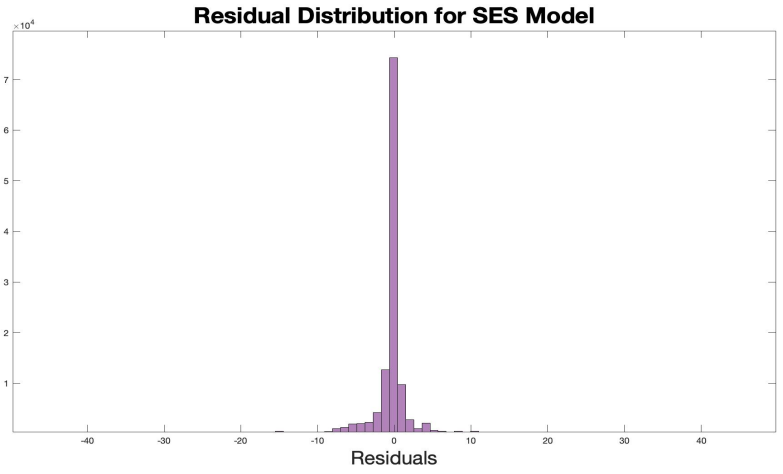
$$\text{Residual variance,} \quad \sigma^2 = \frac{\sum_{i=1}^{n} |e_i|^2}{n} - \bar{e}^2 \tag{4.16}$$

$$\text{Residual Distribution,} \quad z = \frac{e_t - \bar{e}}{\sqrt{\sigma^2}} \tag{4.17}$$

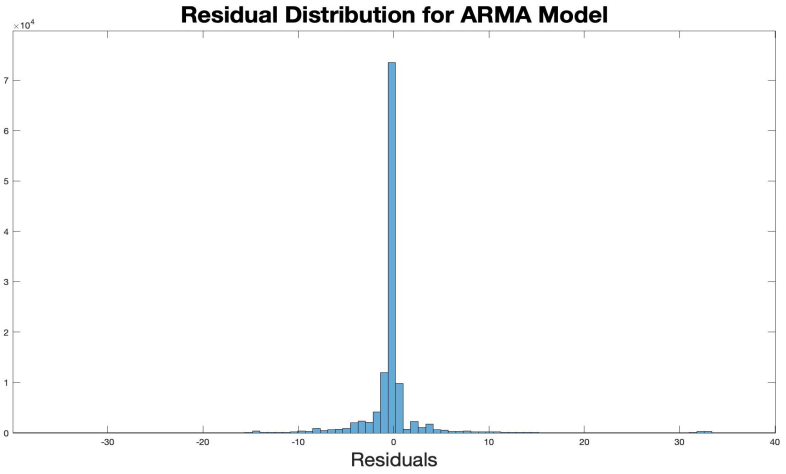
In order to determine if the observed value should be marked normal or anomalous, the threshold rule, z_{thresh}, is defined as the distance of the forecast error from the residual average in terms of standard deviations. The value z_{thresh} is varied between 1 and 10 in increments of 1. After measuring the deviation between predicted and observed values, if $z > z_{thresh}$, the observed value is marked as anomalous.

4.7.2 Prediction Interval

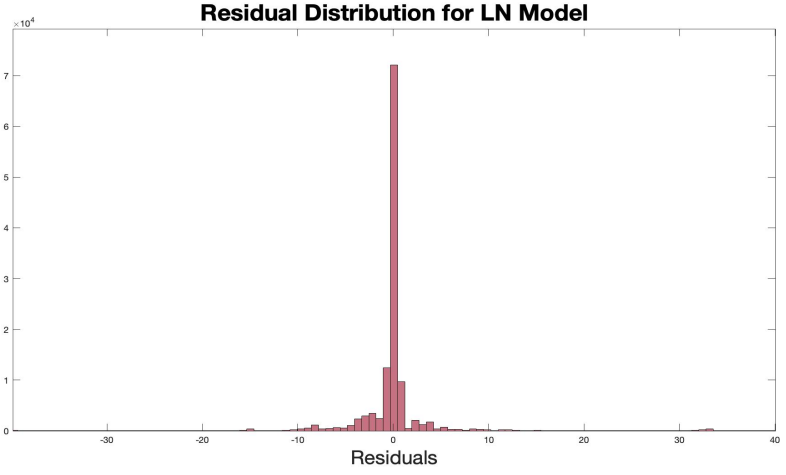
The second method is the prediction interval [149] which is commonly used in regression analysis. It is an estimate of a range where the observed values will fall with a certain probability. Prediction interval describes the uncertainty for a single specific value where



(a)



(b)



(c)

FIGURE 4.6: Distribution of forecast residuals from validation data set 1 and forecast models developed using SES, ARMA and LN method

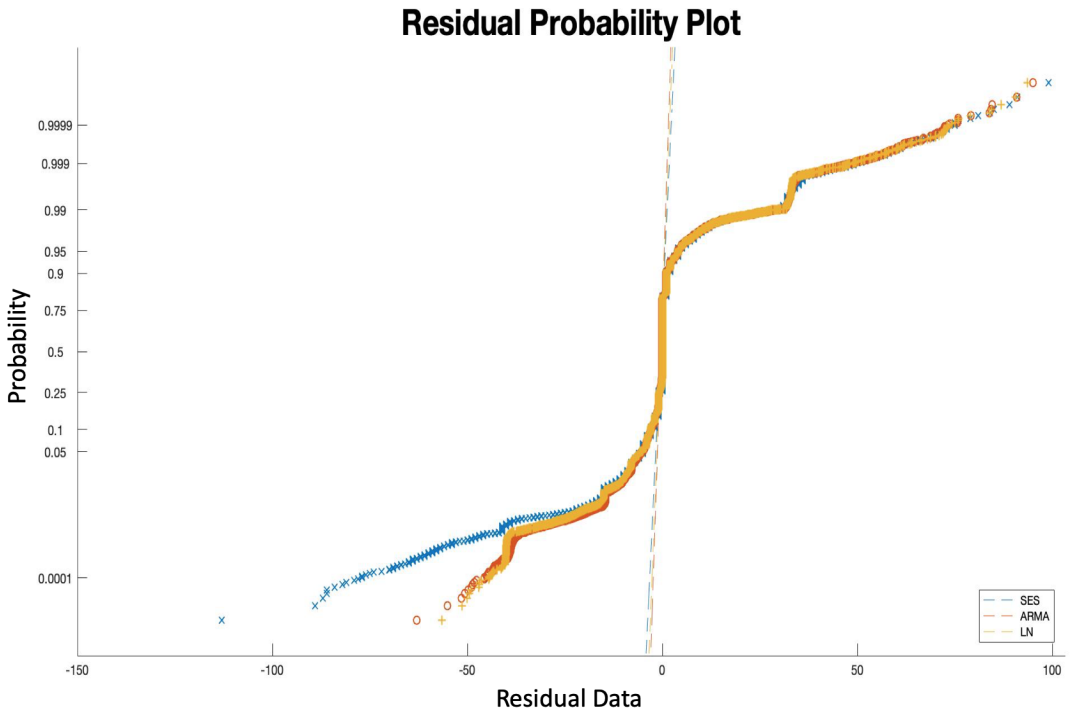


FIGURE 4.7: Probability plot of residuals for all three one-step ahead prediction methods

uncertainty comes from the errors in the model itself and noise in the input data and provides probabilistic upper and lower bounds based on the estimate of a predicted variable. If the observed data falls within the upper and lower bounds, it is considered to be normal and if it falls outside the upper and lower bounds, it will be marked anomalous. The formula to calculate Prediction Interval is given in Equation 4.18. z_{upper} and z_{lower} represent the upper and lower thresholds for acceptance or rejection of the observed data. The current observed data point is considered normal if it satisfies the condition $z_{lower} < Y_t < z_{upper}$, else it is marked as anomalous.

$$z_{upper} = \hat{Y}_{t+1} + PI * \sqrt{MSE * \left(1 + \frac{1}{n} + \frac{\left(\hat{Y}_{t+1} - \bar{y}\right)^2}{\sum_{i=1}^{n} |e_i|^2}\right)}$$
$$z_{lower} = \hat{Y}_{t+1} - PI * \sqrt{MSE * \left(1 + \frac{1}{n} + \frac{\left(\hat{Y}_{t+1} - \bar{y}\right)^2}{\sum_{i=1}^{n} |e_i|^2}\right)} \tag{4.18}$$

Three important parameters are required to calculate the prediction interval for new predicted data. The first parameter is $\hat{Y}_{t+1}$, which is the predicted data point at time

$t+1$. The second parameter given by $\sqrt{MSE * \left(1 + \frac{1}{n} + \frac{\left(\hat{Y}_{t+1} - \bar{y}\right)^2}{\sum_{i=1}^{n} |e_i|^2}\right)}$ is known as the *standard error of the predicted model* [150]. It depends on the mean squared error (MSE), the sample size n which is the total number of observations until time t, the distance in squared units the predicted value $\hat{Y}_{t+1}$ is from the average of Y values in the window W, and the sum of the squared absolute deviation, $|e_i|^2$. The equations to calculate MSE and the average of Y values are given in Equation 4.19 and Equation 4.20 respectively, while calculation of e_i follows Equation 4.7.

$$\text{MSE} = \sqrt{\frac{\sum_{i=1}^{n} |e_i|^2}{n}} \tag{4.19}$$

$$\bar{y} = \frac{1}{W} \bullet \sum_{i=t-W}^{t-1} Y_i \tag{4.20}$$

Finally, the third parameter is PI, which represents the $100\%(1-a;df)$ of the *Students T-distribution* with df degrees of freedom. It reflects the confidence associated with the calculation of the upper and lower bounds of $\hat{Y}_{t+1}$. Table 4.7 lists a few selected values for T-distributions with df degrees of freedom for a range of one-sided critical regions. The first column is df, the percentages along the top are confidence levels, and the numbers in the body of the table represent the $100\%(1-a;df)$. The values in the body of the main table refers to the critical values for the one-sided critical regions ranges from 75% to 99.5% with df degrees of freedom, and is substituted in parameter PI. In this work, the confidence level is varied from 80% to 97.5% while the value of df is varied between 1 and 3 and the size of the sliding window, W is varied between 3 and 10. The value df is chosen between 1 and 3 because the higher the value of df, the narrower the width between the upper bounds and lower bounds. A higher value of df will affect the anomaly classification process as normal data points will be marked erroneously as anomalous because the observed data points do not satisfy the threshold rule.

4.8 Analysis and Evaluation

4.8.1 Evaluation Metric

In order to evaluate the effectiveness of early detection algorithm, we look at how well the anomaly classification methods have performed in classifying the anomalies. The early detection algorithm consists of three stages. The first stage is to predict the next data point using either SES, ARMA or LN one-step ahead prediction as discussed in Section 4.6. The second stage is to measure how much the observed data has deviated

TABLE 4.7: Critical values of Student's T-distribution with *df* degrees of freedom, [139, 151]

Prob., 100%(1-a) / **DOF, *df***	**75%**	**80%**	**85%**	**90%**	**95%**	**97.5%**	**99%**	**99.5%**
1	1.000	1.376	1.963	3.078	6.314	12.71	31.82	63.66
2	0.816	1.080	1.386	1.886	2.920	4.303	6.965	9.925
3	0.765	0.978	1.250	1.638	2.353	3.182	4.541	5.841
4	0.741	0.941	1.190	1.533	2.132	2.776	3.747	4.604
5	0.727	0.920	1.156	1.476	2.015	2.571	3.365	4.032
6	0.718	0.906	1.134	1.440	1.943	2.447	3.143	3.707
7	0.711	0.896	1.119	1.415	1.895	2.365	2.998	3.499
8	0.706	0.889	1.108	1.397	1.860	2.306	2.896	3.355
9	0.703	0.883	1.100	1.383	1.833	2.262	2.821	3.250
10	0.700	0.879	1.093	1.372	1.812	2.228	2.764	3.169
11	0.697	0.876	1.088	1.363	1.796	2.201	2.718	3.106
12	0.695	0.873	1.083	1.356	1.782	2.179	2.681	3.055
13	0.694	0.870	1.079	1.350	1.771	2.160	2.650	3.012
14	0.692	0.868	1.079	1.345	1.761	2.145	2.624	2.977
15	0.691	0.866	1.074	1.341	1.753	2.131	2.602	2.947

from the defined threshold, which has been discussed in Section 4.7. And finally, in the third stage, if the measurement of deviation does not satisfy the threshold rule, the observed data point is marked as anomalous.

Two evaluation metrics are used to measure how well the detection algorithm has performed. The first metric is the accuracy statistical attribute. As the name implies, the accuracy metric defines how accurate the detection algorithm is in detecting both anomalies and non-anomalies. Another attribute that is important in evaluating the early detection and prediction algorithm is the detection time. This is a novel performance measurement attribute developed with the objective of determining which method is quickest in identifying the anomalous behaviour in the system. Existing performance attributes such as AUC (Area Under The Curve) and ROC (Receiver Operating Characteristics) curves are not suitable to be applied in the damage detection domain as the usage of AUC and ROC attributes require the algorithm to detect almost all anomalous points. As discussed in Section 2.4.3, in a damage detection domain, the type of anomalies can be either contextual anomalies or collective anomalies. As discussed in Section 4.7, the type of anomalies observed in this work are collective anomalies. Therefore, the main goal is to detect several anomalies consecutively and from there, predict potential failure as early as possible.

Calculation of both accuracy and detection time is done using True Positives (TP), True Negatives (TN), False Positives (FP) and False Negatives (FN) as described in Table 4.8.

TP refers to an anomalous observation while TN refers to a normal observation. TP and TN are the ideal situations where data points are detected and identified correctly,

TABLE 4.8: Confusion matrix for early detection of anomalous behaviour

Outcome	**Detection**	
	Anomalous	**Non-Anomalous**
Anomalous	True Positives (TP) (data points that are anomalous and identify as anomalous)	False Negatives (FN) (data points that are anomalous but identify as normal)
Non-Anomalous	False Positives (FP) (data points that are normal but identify as anomalous)	True Negatives (TN) (data points that are normal and identify as normal)

while FP and FN are undesirable cases which are impossible to eliminate but need to be kept to a minimum. The formulae to calculate detection time and accuracy are shown in Equation 4.21 and Equation 4.22.

$$\text{Detection time} = (TP + FN) * \text{Logging Interval} \tag{4.21}$$

$$Accuracy = \frac{TP + TN}{(TP + FN + TN + FP)} \tag{4.22}$$

While it is true that a good anomaly detection model should maximise the number of correct detections and keep the false detection as low as possible [31, 33, 57], the main objective of the detection model is to be able to predict failure at the earliest time. Therefore, the key attribute is the lowest detection time that can be attained from the proposed methods. The detection time has to be below $4,000\mu$s, which is equivalent to 1M clock cycles.

4.8.2 Minimum Consecutive Anomalies to be detected, C

As discussed in Section 4.5, the minimum number of consecutive anomalies to be detected, C, is 4. Table 4.9 shows the results of the lowest detection time achieved when $C = 4$, $C = 5$ and $C = 6$. As can be seen, the lowest detection time of 325μs is achieved with $C = 5$. This means, the anomalous behaviour was detected at 325μs after the fault manifested itself as an error. Increasing the value of C increases the detection time as more consecutive anomalies are required to be detected. When $C = 4$, no detection time was recorded as the algorithm was overly sensitive and not able to detect anomalies correctly. The algorithm managed to detect the anomalies only when the parameters for anomaly classification methods are enlarged. Another observation was all three forecasting methods gave the same detection time whether they use Residual Distribution or Prediction Interval for anomaly classification. All three forecasting methods have comparable performance in predicting the data point one-step ahead.

TABLE 4.9: Analysis on the optimum value for C

Number of Consecutive Anomalies (c)	**Lowest Detection Time (μs)**		
	Residual Distribution (z_{thresh} = 6)		
	SES	**ARMA**	**LN**
C = 4	NaN (1715μs with z_{thresh} = 9)	NaN (1715μs with z_{thresh} = 9)	NaN (1715μs with z_{thresh} = 9)
C = 5	325	325	325
C = 6	585	585	585

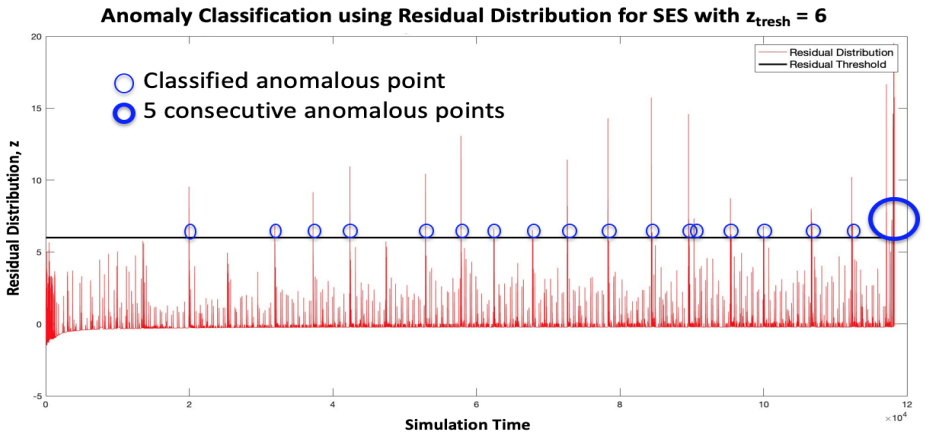
Number of Consecutive Anomalies (c)	**Lowest Detection Time (μs)**		
	Prediction Interval (PI = 3.078 where Probability = 90%, df = 1 and W = 3)		
	SES	**ARMA**	**LN**
C = 4	NaN (575μs with W = 5)	NaN (575μs with W = 5)	NaN (575μs with W = 5)
C = 5	325	325	325
C = 6	585	585	585

4.8.3 Detection Accuracy using Residual Distribution

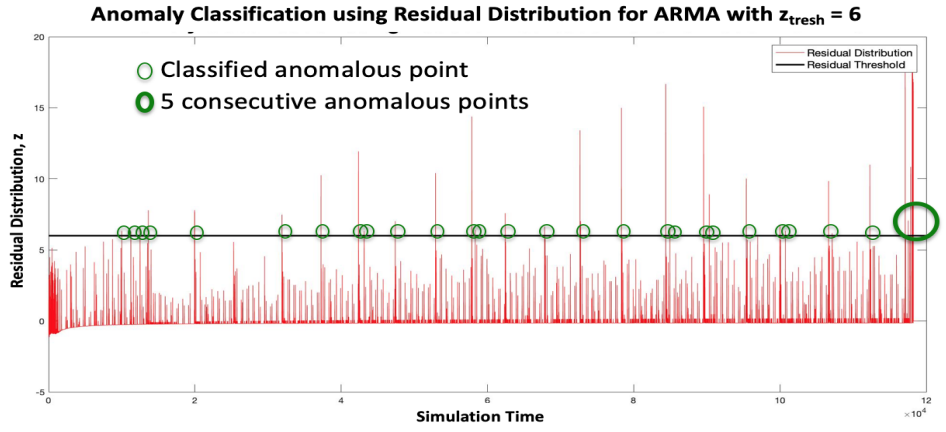
Anomaly classification using the Residual Distribution method analyses how many standard deviations the current residual is lying away from the average of the forecast residual. The value z is calculated using Equation 4.17 and is compared against the threshold, z_{thresh}. Figure 4.8 shows the results of how anomalies are classified using the Residual Distribution method. The value z_{thresh} in Figure 4.8 has been set to 6, and values that exceed the threshold are marked as anomalous. The complete detection result using this method where the value z_{thresh} is varied between 1 and 10 is shown in Table 4.10.

As can be observed in Table 4.10, when parameter z_{thresh} is set between 1 and 4, the detection was too sensitive with a lot of normal points being wrongly classified as anomalous points. However, for $z_{thresh} = 8$, $z_{thresh} = 9$ and $z_{thresh} = 10$, the anomaly classification method was found to be unresponsive as it was unable to detect both normal and anomalous points. The optimum value for z_{thresh} was found to be 6, where the number of TP is the lowest across all three prediction methods. The top three results from Table 4.10 were further analysed to obtain the detection time and accuracy using Equation 4.21 and Equation 4.22.

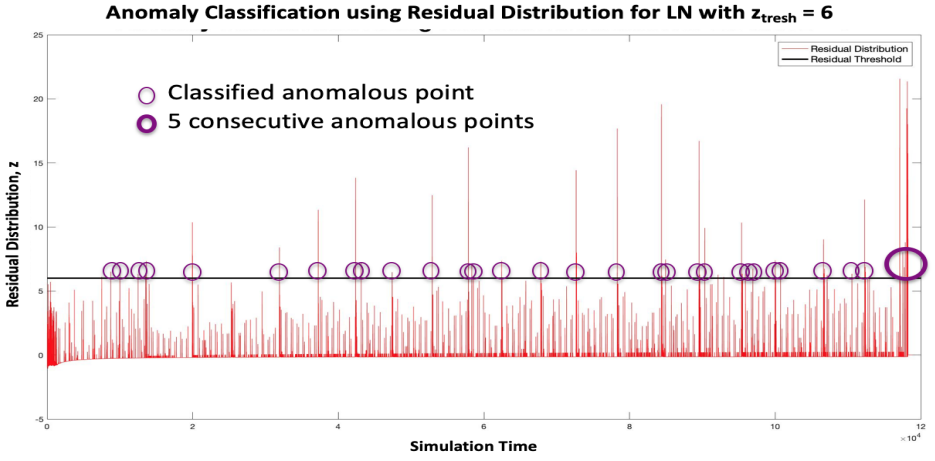
Table 4.11 shows the top three results obtained for this method where the lowest detection time was 325μs (or $\approx$ 82000 clock cycles) after a fault is injected. The number of anomalous points, TP, that are correctly detected is between 15 and 19, while the non-anomalous points, TN, that are correctly identified lie between 118050 and 118067. The number of missed anomalies and false alarms (FN and FP) is between 135 and 182, giving an accuracy between 99.85% to 99.89%. While all three prediction methods are able to achieve the lowest detection time of 325μs, the ARMA method has managed to detect fewer false alarms (FP) compared to SES and LN.



(a)



(b)



(c)

FIGURE 4.8: Anomaly classification using Residual Distribution

TABLE 4.10: Detection results for z_{thresh} between 1 and 10

z_{thresh}	**SES**				**ARMA**				**LN**			
	TP	**FN**	**FP**	**TN**	**TP**	**FN**	**FP**	**TN**	**TP**	**FN**	**FP**	**TN**
1	0	0	229	4506	0	0	209	4526	0	0	237	4498
2	0	0	161	19779	0	0	137	19801	0	0	178	19762
3	0	0	299	78007	0	0	226	78080	0	0	312	77994
4	0	0	146	78167	0	0	160	78213	0	0	138	78175
5	**17**	**48**	**216**	**117940**	**25**	**40**	**163**	**117993**	**20**	**45**	**205**	**117951**
6	**15**	**50**	**132**	**118064**	**19**	**46**	**89**	**118067**	**18**	**47**	**106**	**118050**
7	**21**	**95**	**68**	**118088**	**23**	**93**	**47**	**118109**	**21**	**95**	**60**	**118096**
8	0	0	0	0	0	0	0	0	0	0	0	0
9	0	0	0	0	0	0	0	0	0	0	0	0
10	0	0	0	0	0	0	0	0	0	0	0	0

TABLE 4.11: Top result for Residual Distribution using SES, ARMA and LN method with C = 5

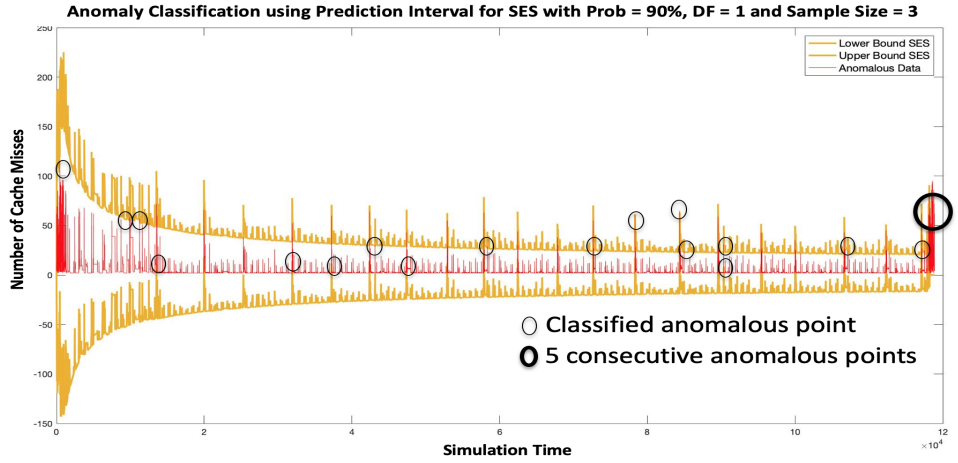
Prediction Methods	**SES (a = 0.7)**			**ARMA (4,4)**			**LN (W=3)**		
Parameter (z_{thresh})	**5**	**6**	**7**	**5**	**6**	**7**	**5**	**6**	**7**
TP	17	15	21	25	19	23	20	18	21
FN	48	50	95	40	46	93	45	47	95
FP	216	132	68	163	89	47	205	106	60
TN	117940	118024	118088	117993	118067	118109	117951	118050	118096
Accuracy	99.78%	99.85%	99.86%	99.83%	99.89%	99.88%	99.79%	99.87%	99.87%
Detection Time μ**s**	325	325	580	325	325	580	325	320	580

4.8.4 Detection Accuracy using Prediction Interval

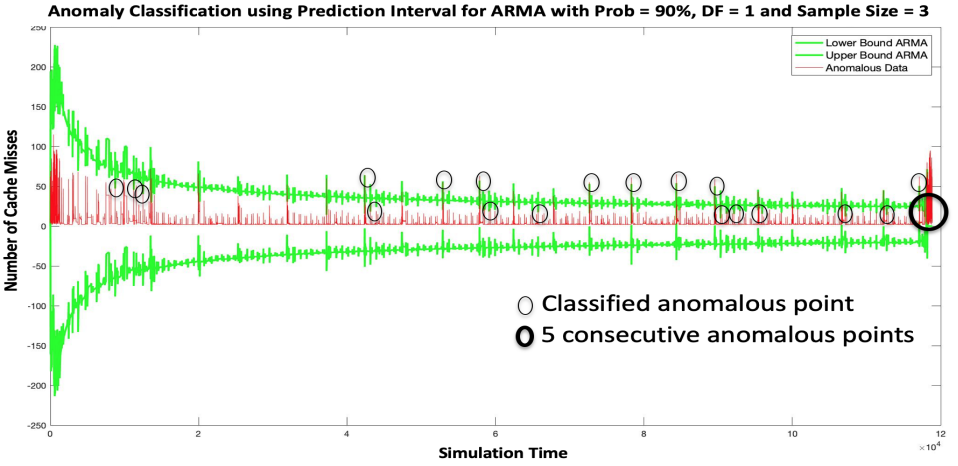
The second anomaly classification method is the Prediction Interval where the predicted data point is used to estimate, with a certain probability, the range where the observed values will fall. Prediction Interval provides upper and lower bounds where if the observed data point falls outside the upper and lower bounds, it is marked as anomalous. The upper bound z_{upper}, and lower bound z_{lower} values are calculated using Equation 4.18. Figure 4.9 shows the results of how anomalies are classified using this method. The upper and lower bounds in Figure 4.9 were calculated using 90% one-tailed probability with 1 Degree of Freedom. From these figures, it is shown that the upper and lower bounds calculated generate good envelopes for the actual data where the majority of the data points lies between the upper and lower bounds. However, it requires at least 20μs for the calculation to stabilise. This means, if anomalies happen at the start of the program, this method would not be able to detect those anomalies and subsequently predict a failure.

For anomaly classification using the Prediction Interval method as discussed earlier in Section 4.7.2, the parameter PI plays an important role in setting the threshold rule. Table 4.12, Table 4.13 and Table 4.14 show the complete detection results using SES, ARMA and LN prediction method respectively where the probability is varied between 80% and 97.5% with the degree of freedom, df between 1 and 3 and the size of W between 3 and 10. As can be observed from all three tables, when the parameter PI uses the probability of 80% with any corresponding value for degree of freedom, the detection was too sensitive and resulted in a higher number of normal data points being wrongly classified as anomalous points. The critical values from the probability of 80% with $df = 1$, $df = 2$ and $df = 3$ were 1.376, 1.080 and 0.978 respectively. These values created a narrow width between the upper bound and lower bound, thus more false alarms occurred where normal data points were wrongly classified as anomalous points.

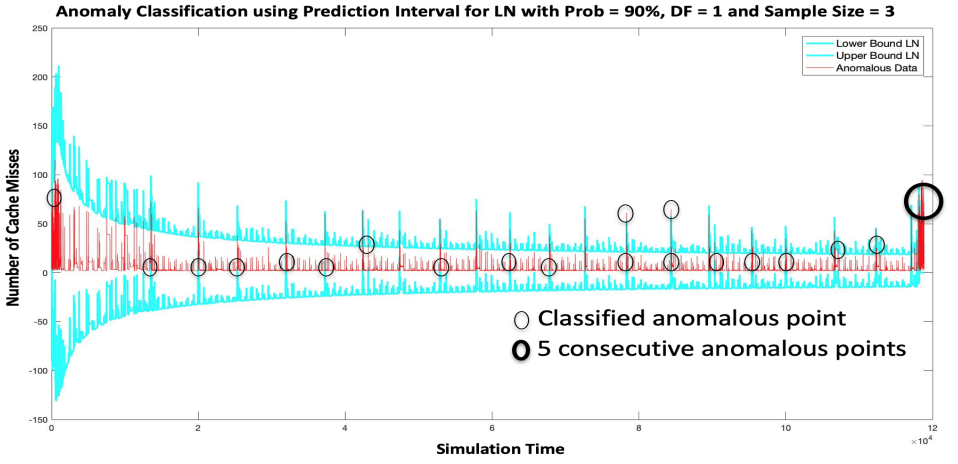
In Table 4.15, the top three results for each prediction method using Prediction Interval is presented. The detection time achieved for all top three results was 325μs, but the best results were achieved with $Probability = 90\%$, $df = 1$ and $W = 3$. This is due to the number of false alarms (FP) which are slightly lower, thus resulting in a higher accuracy compared to the results obtained using $PI = 2.920$ and $PI = 2.353$. The number of anomalous points, TP, that are correctly detected is between 15 and 19, while the non-anomalous points, TN, that are correctly identified lie between 118058 and 118072. The number of missed anomalies (FN) is between 46 and 50 while the number of false alarms (FP) is between 84 and 98, giving the accuracy of 99.88%, 99.89% and 99.88% for the SES, ARMA and LN method respectively.



(a)



(b)



(c)

FIGURE 4.9: Anomaly classification using Prediction Interval

TABLE 4.12: Detection results with probability between 80% and 97.5% using SES prediction method

DOF = 1

Probability	**80%**				**85%**				**90%**				**95%**				**97.5%**			
Window Size, W	**TP**	**FN**	**FP**	**TN**	**TP**	**FN**	**FP**	**TN**	**TP**	**FN**	**FP**	**TN**	**TP**	**FN**	**FP**	**TN**	**TP**	**FN**	**FP**	**TN**
3	0	0	123	19824	0	0	145	78168	**15**	**50**	**88**	**118068**	0	0	0	0	0	0	0	0
4	0	0	200	57643	0	0	98	78215	20	96	62	118094	0	0	0	0	0	0	0	0
5	0	0	236	78070	0	0	91	89454	77	292	49	118107	0	0	0	0	0	0	0	0
6	0	0	192	78114	0	0	68	89477	71	298	41	118115	0	0	0	0	0	0	0	0
7	0	0	147	78159	14	45	111	118045	64	305	38	118118	0	0	0	0	0	0	0	0
8	0	0	122	78191	15	50	98	118058	57	312	33	118123	0	0	0	0	0	0	0	0
9	0	0	103	78210	24	92	86	118070	0	0	0	0	0	0	0	0	0	0	0	0
10	0	0	111	89434	21	95	71	118085	0	0	0	0	0	0	0	0	0	0	0	0

DOF = 2

Probability	**80%**				**85%**				**90%**				**95%**				**97.5%**			
Window Size, W	**TP**	**FN**	**FP**	**TN**	**TP**	**FN**	**FP**	**TN**	**TP**	**FN**	**FP**	**TN**	**TP**	**FN**	**FP**	**TN**	**TP**	**FN**	**FP**	**TN**
3	0	0	149	9726	0	0	118	19829	0	0	171	78135	**15**	**50**	**100**	**118056**	66	303	38	118118
4	0	0	162	19780	0	0	195	57648	0	0	113	78200	21	95	71	118085	50	319	28	118128
5	0	0	138	19804	0	0	229	78077	0	0	106	89439	81	288	56	118100	0	0	0	0
6	0	0	118	19829	0	0	188	78118	0	0	77	89468	74	295	45	118111	0	0	0	0
7	0	0	102	19845	0	0	145	78168	0	0	62	89483	69	300	40	118116	0	0	0	0
8	0	0	186	57657	0	0	119	78194	14	45	104	118052	64	305	36	118120	0	0	0	0
9	0	0	245	78061	0	0	100	78213	15	50	93	118063	57	312	33	118123	0	0	0	0
10	0	0	211	78095	0	0	107	89438	24	92	81	118075	50	319	27	118129	0	0	0	0

DOF = 3

Probability	**80%**				**85%**				**90%**				**95%**				**97.5%**			
Window Size, W	**TP**	**FN**	**FP**	**TN**	**TP**	**FN**	**FP**	**TN**	**TP**	**FN**	**FP**	**TN**	**TP**	**FN**	**FP**	**TN**	**TP**	**FN**	**FP**	**TN**
3	0	0	169	9706	0	0	149	19793	0	0	166	57677	0	0	99	89446	22	94	79	118077
4	0	0	184	19754	0	0	119	19828	0	0	188	78118	0	0	66	89479	81	288	58	118098
5	0	0	159	19783	0	0	204	57639	0	0	133	78180	**15**	**50**	**102**	**118054**	74	295	44	118112
6	0	0	140	19802	0	0	167	57676	0	0	105	78208	24	92	83	118073	68	301	40	118116
7	0	0	124	19824	0	0	207	78099	0	0	105	89440	20	96	67	118089	58	311	35	118121
8	0	0	111	19836	0	0	174	78132	0	0	85	89460	20	96	59	118097	53	316	29	118127
9	0	0	200	57643	0	0	143	78170	0	0	68	89477	77	292	49	118107	0	0	0	0
10	0	0	174	57669	0	0	121	78192	0	0	60	89485	74	295	44	118112	0	0	0	0

TABLE 4.13: Detection results with probability between 80% and 97.5% using ARMA prediction method

DOF = 1

Probability	80%				85%				90%				95%				97.5%			
Window Size, W	**TP**	**FN**	**FP**	**TN**	**TP**	**FN**	**FP**	**TN**	**TP**	**FN**	**FP**	**TN**	**TP**	**FN**	**FP**	**TN**	**TP**	**FN**	**FP**	**TN**
3	0	0	74	19873	0	0	112	78201	**19**	**46**	**84**	**118072**	0	0	0	0	0	0	0	0
4	0	0	153	57690	0	0	84	78229	24	92	52	118104	0	0	0	0	0	0	0	0
5	0	0	170	78136	0	0	84	89461	70	299	36	118120	0	0	0	0	0	0	0	0
6	0	0	138	78168	0	0	66	89479	67	302	29	118127	0	0	0	0	0	0	0	0
7	0	0	110	78196	17	42	129	118027	58	311	25	118131	0	0	0	0	0	0	0	0
8	0	0	94	78219	21	44	98	118053	54	315	21	118135	0	0	0	0	0	0	0	0
9	0	0	86	78227	29	87	81	118075	0	0	0	0	0	0	0	0	0	0	0	0
10	0	0	98	89447	28	88	66	118090	0	0	0	0	0	0	0	0	0	0	0	0

DOF = 2

Probability	80%				85%				90%				95%				97.5%			
Window Size, W	**TP**	**FN**	**FP**	**TN**	**TP**	**FN**	**FP**	**TN**	**TP**	**FN**	**FP**	**TN**	**TP**	**FN**	**FP**	**TN**	**TP**	**FN**	**FP**	**TN**
3	0	0	116	9759	0	0	71	19876	0	0	121	78185	**22**	**43**	**108**	**118048**	58	311	27	118129
4	0	0	112	19830	0	0	150	57693	0	0	88	78225	27	89	65	118091	51	318	20	118136
5	0	0	80	19862	0	0	166	78140	0	0	93	89452	75	294	44	118112	0	0	0	0
6	0	0	71	19876	0	0	136	78170	0	0	77	89468	70	299	33	118123	0	0	0	0
7	0	0	57	19890	0	0	114	78199	0	0	57	89488	65	304	27	118129	0	0	0	0
8	0	0	143	57700	0	0	92	78221	17	42	122	118034	58	311	23	118133	0	0	0	0
9	0	0	174	78132	0	0	85	78228	20	45	101	118055	54	315	21	118135	0	0	0	0
10	0	0	156	78150	0	0	94	89451	29	87	78	118078	50	319	20	118136	0	0	0	0

DOF = 3

Probability	80%				85%				90%				95%				97.5%			
Window Size, W	**TP**	**FN**	**FP**	**TN**	**TP**	**FN**	**FP**	**TN**	**TP**	**FN**	**FP**	**TN**	**TP**	**FN**	**FP**	**TN**	**TP**	**FN**	**FP**	**TN**
3	0	0	139	9736	0	0	92	19850	0	0	133	57710	0	0	90	89455	29	87	70	118086
4	0	0	135	19803	0	0	71	19876	0	0	136	78170	0	0	61	89484	78	291	45	118111
5	0	0	107	19835	0	0	157	57686	0	0	107	78206	**22**	**43**	**114**	**118042**	68	301	33	118123
6	0	0	82	19860	0	0	129	57714	0	0	87	78226	29	87	79	118077	62	307	27	118129
7	0	0	72	19875	0	0	155	78151	0	0	93	89452	26	90	58	118098	56	313	23	118133
8	0	0	62	19885	0	0	126	78180	0	0	80	89465	23	93	47	118109	51	318	20	118136
9	0	0	152	57691	0	0	110	78203	0	0	69	89476	71	298	37	118119	0	0	0	0
10	0	0	137	57706	0	0	92	78221	0	0	55	89490	68	301	33	118123	0	0	0	0

TABLE 4.14: Detection results with probability between 80% and 97.5% using LN prediction method

DOF = 1

Probability	80%				85%				90%				95%				97.5%			
Window Size, W	**TP**	**FN**	**FP**	**TN**	**TP**	**FN**	**FP**	**TN**	**TP**	**FN**	**FP**	**TN**	**TP**	**FN**	**FP**	**TN**	**TP**	**FN**	**FP**	**TN**
3	0	0	112	19835	0	0	152	78161	**17**	**48**	**98**	**118058**	0	0	0	0	0	0	0	0
4	0	0	227	57616	0	0	107	78206	22	94	64	118092	0	0	0	0	0	0	0	0
5	0	0	260	78046	0	0	100	89445	69	300	49	118107	0	0	0	0	0	0	0	0
6	0	0	205	78101	0	0	82	89463	62	307	40	118116	0	0	0	0	0	0	0	0
7	0	0	159	78147	16	43	158	117998	59	310	34	118122	0	0	0	0	0	0	0	0
8	0	0	136	78177	18	47	123	118033	53	316	28	118128	0	0	0	0	0	0	0	0
9	0	0	112	78201	24	92	95	118061	0	0	0	0	0	0	0	0	0	0	0	0
10	0	0	119	89426	23	93	76	118080	0	0	0	0	0	0	0	0	0	0	0	0

DOF = 2

Probability	80%				85%				90%				95%				97.5%			
Window Size, W	**TP**	**FN**	**FP**	**TN**	**TP**	**FN**	**FP**	**TN**	**TP**	**FN**	**FP**	**TN**	**TP**	**FN**	**FP**	**TN**	**TP**	**FN**	**FP**	**TN**
3	0	0	141	9734	0	0	111	19836	0	0	175	78131	**18**	**47**	**127**	**118029**	59	310	36	118120
4	0	0	151	19791	0	0	219	57624	0	0	128	78185	23	93	76	118080	50	319	23	118133
5	0	0	122	19820	0	0	253	78053	0	0	110	89435	76	293	60	118096	0	0	0	0
6	0	0	110	19837	0	0	195	78111	0	0	94	89451	67	302	49	118107	0	0	0	0
7	0	0	91	19856	0	0	155	78158	0	0	76	89469	60	309	39	118117	0	0	0	0
8	0	0	207	57636	0	0	132	78181	15	44	144	118012	59	310	33	118123	0	0	0	0
9	0	0	264	78042	0	0	109	78204	18	47	118	118038	53	316	28	118128	0	0	0	0
10	0	0	228	78078	0	0	116	89429	24	92	93	118063	50	319	23	118133	0	0	0	0

DOF = 3

Probability	80%				85%				90%				95%				97.5%			
Window Size, W	**TP**	**FN**	**FP**	**TN**	**TP**	**FN**	**FP**	**TN**	**TP**	**FN**	**FP**	**TN**	**TP**	**FN**	**FP**	**TN**	**TP**	**FN**	**FP**	**TN**
3	0	0	176	9699	0	0	134	19808	0	0	186	57657	0	0	105	89440	24	92	86	118070
4	0	0	178	19760	0	0	110	19837	0	0	196	78110	0	0	79	89466	76	293	60	118096
5	0	0	148	19794	0	0	232	57611	0	0	144	78169	**18**	**47**	**135**	**118021**	67	302	46	118110
6	0	0	126	19816	0	0	184	57659	0	0	116	78197	24	92	84	118062	60	309	38	118118
7	0	0	112	19835	0	0	221	78085	0	0	111	89434	23	93	72	118084	55	314	31	118125
8	0	0	99	19848	0	0	182	78124	0	0	96	89449	21	95	61	118095	50	319	24	118132
9	0	0	225	57618	0	0	149	78164	0	0	85	89460	69	300	52	118104	0	0	0	0
10	0	0	195	57648	0	0	133	78180	0	0	73	89472	67	302	46	118110	0	0	0	0

TABLE 4.15: Analysis using SES, ARMA and LN method for Number of Successive Anomalies, c = 5 using Prediction Interval

Methods	**SES** (α = **0.7**)			**ARMA** (**4,4**)			**LN** (**W=3**)		
Parameter PI (**prob., df**)	**PI = 3.078** (**prob = 90%, df = 1, W = 3**)	**PI = 2.920** (**prob = 95%, df = 2, W = 3**)	**PI = 2.353** (**prob = 95%, df = 3, W = 5**)	**PI = 3.078** (**prob = 90%, df = 1, W = 3**)	**PI = 2.920** (**prob = 95%, df = 2, W = 3**)	**PI = 2.353** (**prob = 95%, df = 3, W = 5**)	**PI = 3.078** (**prob = 90%, df = 1, W = 3**)	**PI = 2.920** (**prob = 95%, df = 2, W = 3**)	**PI = 2.353** (**prob = 95%, df = 3, W = 5**)
TP	15	15	15	19	22	22	17	18	18
FN	50	50	50	46	43	43	48	47	47
FP	88	100	102	84	108	114	98	127	135
TN	118068	118056	118054	118072	118048	118042	118058	118029	118021
Accuracy	99.88%	99.87 %	99.87 %	99.89 %	99.87 %	99.87 %	99.88 %	99.85%	99.85 %
Detection Time (μs)	325	325	325	325	325	325	325	325	325

The results from Table 4.12, Table 4.13 and Table 4.14 also suggest that the value of PI can be selected among 2.920, 3,078, 3.182 and 4.303. From the results shown in Table 4.11 and Table 4.15, one-step ahead prediction using ARMA method provides higher accuracy compared to SES and LN. This can be attributed to lower number of false alarms (FP) identified when using ARMA for prediction. The optimised parameters obtained from this analysis will be used in the following chapter where the proof of concept for a lightweight detector that predicts potential failure through the detection of anomalous behaviour is presented.

4.9 Summary

In this chapter, a novel algorithm that detects anomalous behaviour in a processor using HPC data and predict potential failure in real-time is presented. The algorithm consists of three main stages, (a) one-step ahead prediction to predict the next data in the time-series, (b) measurement of deviation between predicted value and observed value, and (c) mechanism to classify if the observed value deviates "too much" and is deemed anomalous. Three forecasting methods have been identified for one-step ahead prediction namely, (a) Single Exponential Smoothing, (b) Autoregressive Moving Average, and (c) Single Layer Linear Network. Two methods are used for measurement of deviation and anomaly classification, namely: (a) Residual Distribution, and (b) Prediction Interval. The algorithm predicts a potential failure if the number of consecutive anomalies, C equals 5.

Two attributes are used to measure the performance of the algorithm. The first attribute is the accuracy, which measures how accurately the detector has predicted both anomalies and normal data points correctly. The second attribute is the detection time attribute, a novel performance measurement attribute developed specifically for this problem. It refers to the earliest time for the algorithm to raise an alarm at a potential failure.

Based on analysis conducted, the lowest detection time achieved was 325μs for Residual Distribution with threshold value, z_{thresh} equal to 6. Meanwhile, using Prediction Interval, the best detection time achieved was also 325μs with optimum parameters $PI = 90\%$, $df = 1$ and $W = 3$. One-step ahead prediction using ARMA(4,4) proves to be a better forecasting method as the number of detected false alarms (FPs) are lower compared to SES and LN. Although the best detection time using Residual Distribution and Prediction Interval is same, using Prediction Interval has a slight drawback – it requires time for the calculation to stabilise, thus it may be at a little disadvantage compared to Residual Distribution method.

The following chapter will present the proof of concept for a lightweight detector that predicts potential failure through the detection of anomalous behaviour based on the early detection and prediction algorithm developed in this chapter.

Chapter 5

Detector for Predicting Potential Failure from Anomalous Behaviour

5.1 Introduction

In previous research that uses HPC to detect anomalous behaviour, the data collected is usually sent for offline processing to be analysed and anomalies are only detected after a failure has happened [37, 74, 152]. In contrast to that work, the detector proposed and designed here aims to predict potential failure in real-time through the detection of anomalous behaviour. The key difference lies in the ability of the detector to detect and predict within a certain number of clock cycles and prevent the system from entering into a failure state. In Chapter 3, the minimum number of clock cycles before the system crashes was found to be approximately 1,000,000.

In Chapter 4, the algorithm for early detection and prediction of potential failure is presented where the best detection time was found to be 325μs or approximately 82,000 clock cycles. This chapter presents the proof of concept for a lightweight detector that predicts potential failure through detection of anomalous behaviour based on the algorithm presented in Chapter 4. The design of this detector is realised using the optimal parameters that have been determined earlier in the previous chapter. The main objectives of the work in this chapter are as follows:

1. To design and simulate the detector through experimental validation by implementing one-step ahead prediction and anomaly classification presented in the previous chapter.

2. To test the detector on several embedded benchmarks and evaluate the effectiveness of the detection and prediction in terms of "Time to Detect" against "Time to Failure".

3. To measure the performance of the detector based on its *total instructions*, *Cycles Per Instruction* (CPI), *total execution time* and *size*.

This chapter is organised as follows. Section 5.2 will present the proposed design of detector and Section 5.3 outlined the experimental setup to validate the design. Section 5.4 will present the results of testing the detector on several benchmarks while Section 5.5 and Section 5.6 present the performance analysis and source-byte analysis of the detector based on *total instructions*, *Cycles Per Instruction* (CPI), *total execution time* and *size* while Section 5.7 provides the summary of the performance analysis. The chapter is concluded in Section 5.8.

5.2 Proposed Design of the Detector

The computer industry, be it from a high-end, customised, special-purpose computing in avionics, telecommunications and networking to low-power embedded computing in video games and portable computing, the expectations from consumers remain the same – faster, more efficient and more powerful. However, single core products are showing a decline in the ability to boost performance to keep pace with consumer desire. Multicore-processors have now been recognised as one of the key components in improving computing performance.

Multicore technologies aim to either exploit concurrency, increase compute density, handle partitioned workloads, or achieve some combination of these objectives. Integrating multicore technologies for embedded systems requires developing multicore processors which can be integrated into a small area such as a classic microcontroller. Fortunately, this has been made possible with the recent developments. For example, most Intel Atom processors have between two and four cores while the NXP LPC4300 contains an ARM Cortex-M4 and a Cortex-M0 processor, and the Freescale Vybrid contains a Cortex-M4 and a Cortex-A5 processor. This illustrates that the need for multi-processor designs in certain microcontroller application areas is expanding.

The idea of using a dedicated hardware processor to detect anomalous behaviour in the main processor is aimed at achieving a quick response for detection and prediction with minimal performance overhead. Rather than placing the detector on the main core in a multicore microcontroller, it is designed to be placed on secondary core to ensure no overhead is imposed on the main core running the application. However, there are some challenges for using a multicore processor in a microcontroller, and the most notable

challenge is deciding which part of the system will be shared, and which part of the system will not be shared.

Following the design guidelines as proposed in [153], the main core and secondary core have been designed to have private caches and private memories. This is to ensure that the HPC data from the main core that uses the number of cache misses as its PME will not be compromised due to the presence of a secondary core. Figure 5.1 illustrates how the proposed detector is designed in such a way that the secondary processor can receive the PME counts from the main processor with minimal overhead to the main processor via the communication pipeline.

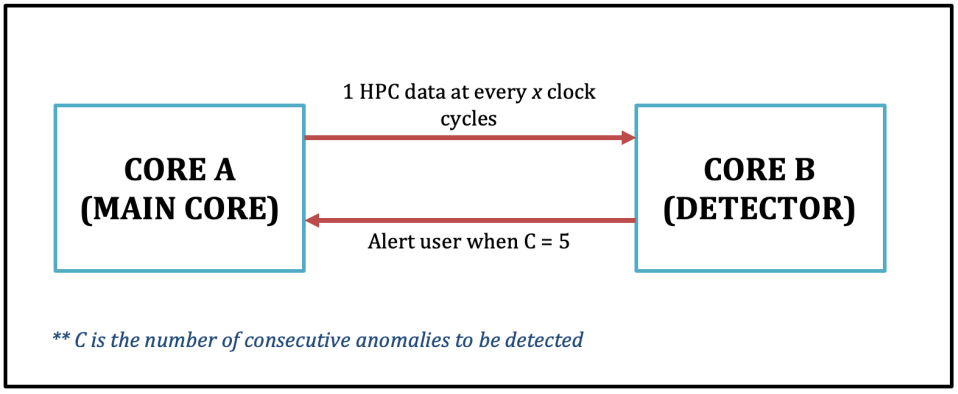


FIGURE 5.1: Proposed hardware-based detector utilising multi-cores architecture

The important consideration in this design is the inter-core communication pipeline where HPC data from the main core can be sent via a dedicated pipeline to the detector core as shown in Figure 5.1. The inter-core communication pipeline between the main core and the detector core is established when the main core sends a command to turn on the detector core. It is important to have a dedicated pipeline to ensure that the HPC data sent from the main core to the detector core will not be compromised that can affect the detection and prediction. Figure 5.2 shows the overall execution between the main core, called Core A, and the secondary core, called Core B.

Core A which functions as the main core, starts up the whole microcontroller, initialises the memory, peripherals and stack pointers. Core A will then turn on Core B and initialised the inter-core communication pipeline to Core B. After Core B has been turned on and the communication pipeline to Core A has been established, Core B loads the detector program. Core A will run the application and send one HPC data at every 1250 clock cycles to Core B. Core B, prior to receiving the HPC data, will perform one-step ahead prediction and predict the next data. Once the actual data (or observed data) is available, Core B will measure the deviation between predicted data and actual data. If it exceeds the threshold, the actual data will be marked as anomalous. If there are five anomalous data points detected consecutively, Core B sends an interrupt to

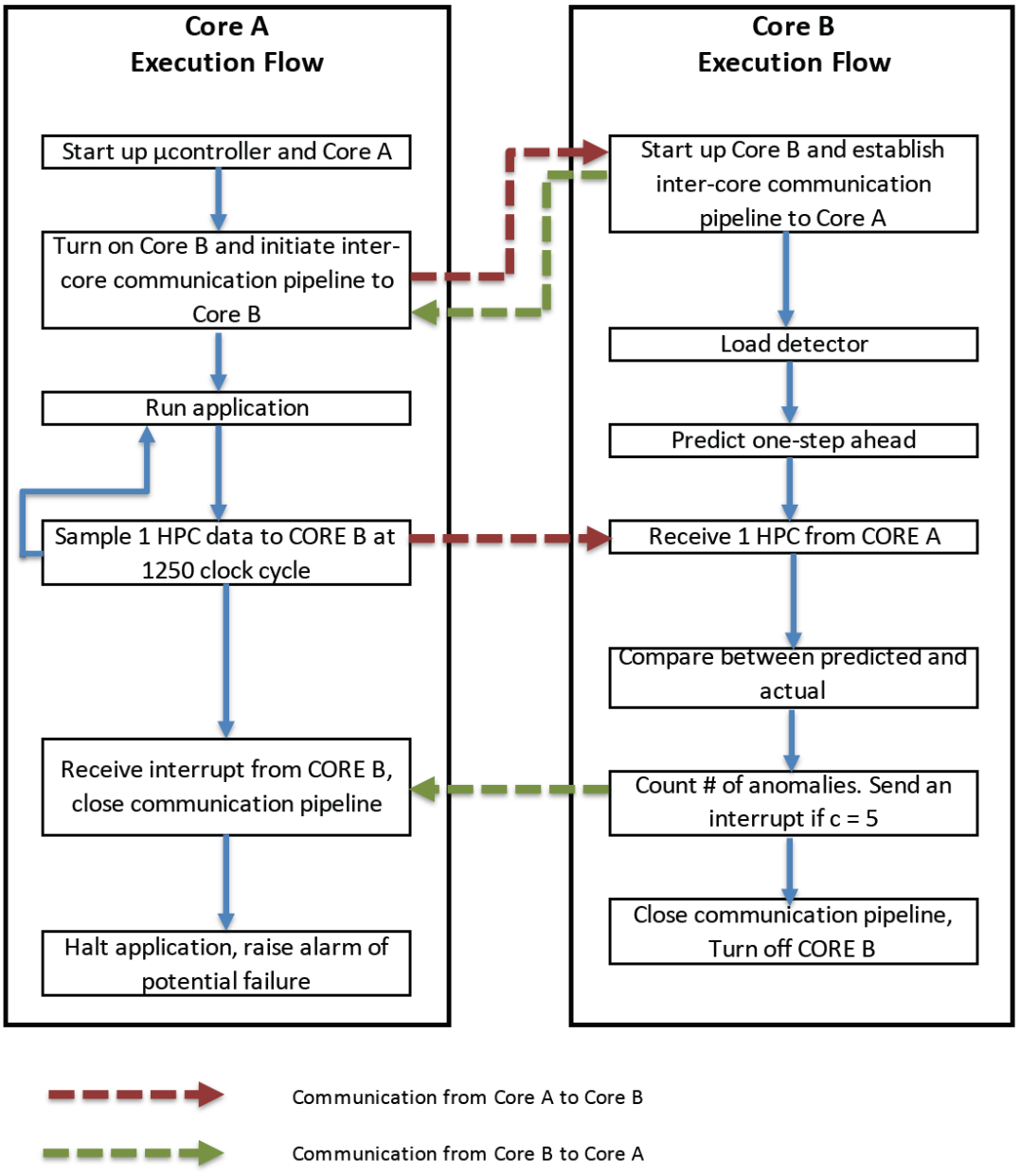


FIGURE 5.2: Overall execution flow between main core, Core A and secondary core, Core B

Core A. Core A, upon receiving the interrupt, will close the inter-core communication pipeline with Core B, halt the application, and raise an alarm for potential failure.

In this work, it is assumed that Core B, being the detector core, is protected against transient faults.

5.3 Experimental Validation of the Detector

The experiments were performed on a workstation running Ubuntu 16.04 LTS as the operating system with an Intel Core i5-5257U operating at 2.70GHz and 11.1 GB of memory. An environment is simulated in such a way that a main processor core (or Core A) will be running the benchmark, and a secondary core (also known as Core B) runs the early detection algorithm to predict and classify the stream of HPC data coming from Core A through a protected inter-core communication pipeline. In the experimental setup, to simulate Core A running a benchmark, a stream of HPC data from a benchmark is used as an input to Core A, which will then read and send a data to Core B. The time granularity has been fixed where the HPC data is being transferred from Core A to Core B at every 5μs.

The implementation of inter-core communication pipeline is realised using a *named pipe* (also known as a FIFO), as it is one of the methods for inter-process communication. Unnamed pipes allow any process to use the pipes to send commands while named pipes only allow processes that have establish connections with the pipes to send or receive commands. Using *named pipe*, Core A established an inter-core connection with Core B, and vice-versa. When five anomalies have been detected consecutively, Core B sends an interrupt to Core A and closes the connection to Core A. Core A, upon receiving the interrupt, stops the execution and prompt an error message on the screen.

Function 5.1 and 5.2 shows how a FIFO is created and used.

```
1 /* int mkfifo(const char *pathname, mode_t mode); */
2 int file1;
3 file1 = mkfifo("fifo_counter",0666);
```

Listing 5.1: Creating a FIFO

```
1 /* Open named fifo */
2 int fifo_in;
3 fifo_in = open("fifo_counter", O_WRONLY);
```

Listing 5.2: Using a FIFO

Three different benchmarks have been chosen to validate the detector. The benchmarks chosen are Dijkstra, FFT and Bitcount benchmarks, each from a different suite as explained in Section 3.5. These benchmark applications have been injected with a single bit-flip randomly at any of the location in the various stages of the pipeline. As the main processor core is running a benchmark application, it sends one piece of HPC data at every 1250 clock cycles through the communication pipeline to the detector core. Six different detectors were implemented with each detector running a combination of one-step ahead prediction with anomaly classification techniques.

The first stage is to predict the value for next data point using one-step ahead prediction. Three different methods were used for one-step ahead prediction, namely SES, ARMA

and LN. For one-step ahead prediction using the SES method, the optimum value for parameter α, which has been determined in Section 4.6.1, is 0.7. As for one-step ahead prediction using the ARMA method, the optimal value for parameter (p, q) is $(4, 4)$ as shown in Section 4.6.2. And for one-step ahead prediction using the LN method, the optimum window size, W, is 3 as shown in Section 4.6.3.

For the second stage, two different methods were applied. The first method is Residual Distribution where, the threshold rule, z_{thresh}, is defined as the number of standard deviations away from the average of the residual. If the residual distribution, z exceeds the threshold rule, z_{thresh}, the observed data is marked as anomalous. The optimum value for z_{thresh} is 6, as shown in Section 4.8.3. The second method is Prediction Interval, used to estimate the range where the observed data will fall. z_{upper} and z_{lower} are the upper and lower boundary thresholds defined in Equation 4.18. The ideal confidence level, determined in Section 4.8.4, was 90% with window size, $W = 3$ and $df = 1$. The algorithms for the Residual Distribution method and the Prediction Interval method are shown in Figure 5.3 and Figure 5.4, respectively.

In the secondary core, the detector will classify if the current point is anomalous and raise the alarm of the impending failure of the main core if five anomalous points are detected consecutively. The interrupt is transmitted to the main core via another dedicated communication pipeline. This is to ensure that the pipeline that is being used to send an HPC data from the main core to the detector core do not need to wait and check for any alarm from the detector core, which could impede the detection process.

Figure 5.5 shows the simulation of the detector core performing one-step ahead prediction, measurement of deviation between predicted values and observed values and classification of the observed values. The detector core in Figure 5.5 has used the ARMA method for one-step ahead prediction and the Residual Distribution method for measurement of deviation and anomaly classification. Upon five anomalies detected consecutively, an interrupt is sent to the main core. Figure 5.6 shows the main core upon receiving an interrupt for a potential failure, terminates the current process.

5.4 Experimental Results

The six different detectors were tested on three benchmarks namely Dijkstra, FFT and Bitcount benchmarks. These benchmarks are tasked to run at the same clock speed and provide HPC data at the same sampling interval of 5μs. As mentioned in Chapter 3 and Chapter 4, it takes about 1M clock cycles for the system to fail and crash, therefore any potential failure has to be predicted before 1M clock cycles, which is equivalent to 4,000μs.

Algorithm 1 Detection using Residual Distribution

for $i \leftarrow 1$ to 3 **do**
  $array[i] \leftarrow y_i$
end for
$\hat{y}_{t+1} \leftarrow Predict(array[2], i)$
while [Main program runs] **do**
  $y_{t+1} \leftarrow$ new HPC {From pipeline}
  {Shift array by 1}
  $array[2] \leftarrow y_{t+1}$
  $error \leftarrow |\hat{y}_{t+1} - y_{t+1}|$
  sum error $\leftarrow$ sum error $+ error$
  sum squared error $\leftarrow$ sum squared error $+ error^2$
  num observations $\leftarrow$ num observations $+ 1$
  $\mu \leftarrow \frac{\text{sum error}}{\text{num observations}}$
  $\sigma^2 \leftarrow \frac{\text{sum squared error}}{\text{num observations}} - (\mu^2)$
  $z \leftarrow \frac{(error - \mu)}{\sqrt{\sigma^2}}$
  if $(z > 6)$ **then**
    $count \leftarrow count + 1$
  else
    $count \leftarrow 0$
  end if
  if $(count > 4)$ **then**
    Pipeline $\leftarrow 0$ {Anomalous behaviour detected}
  else
    Pipeline $\leftarrow 1$ {No anomaly}
  end if
  $\hat{y}_{t+1} \leftarrow Predict(array[2], i)$
end while

FIGURE 5.3: Algorithm for early detection and prediction using Residual Distribution

5.4.1 Experimental Results for the Dijkstra Benchmark

For the Dijkstra benchmark, twenty experiments were performed where in each experiment, a single bit-flip fault is injected. Out of the twenty experiments conducted, three experiments were found to experience crash failure due to the injected fault manifested itself as an error. Table 5.1, Table 5.2 and Table 5.3 show the detection results for three different anomalous data sets. As can be seen from these results, potential failure in the main core can be predicted by at least three detectors out of a total of six detectors with the detection time are all under 4,000μs.

In Table 5.1, the detectors running SES with Residual Distribution, LN with Residual Distribution, SES with Prediction Interval and LN with Prediction Interval have managed to predict a potential failure in 1820μs, 285μs, 245μs and 1790μs, respectively.

Algorithm 2 Detection using Prediction Interval

for $i \leftarrow 1$ to W **do**
  $list[i] \leftarrow y_i$
end for
$\hat{y}_{t+1} \leftarrow Predict(list[W], i)$
while [Main program runs] **do**
  $y_{t+1} \leftarrow$ new HPC {From pipeline}
  {Shift error array by 1}
  $error[5] \leftarrow |\hat{y}_{t+1} - y_{t+1}|$
  sum squared error $\leftarrow$ sum squared error $+ error^2$
  for $i \leftarrow 1$ to W **do**
    sum samples $\leftarrow$ sum samples $+ array[i]$
  end for
  $\mu \leftarrow \frac{\text{sum samples}}{i+1}$
  for $i \leftarrow 1$ to W **do**
    sum diff $\leftarrow$ sum diff $+ (array[i] - \mu)^2$
  end for
  {Shift list array by 1}
  $list[5] \leftarrow num$
  num observations $\leftarrow$ num observations $+ 1$
  diff $\leftarrow (num - \mu)^2$
  $MSE \leftarrow \frac{\text{sum squared error}}{\text{num observations}}$
  $temp \leftarrow 1 + \frac{1}{\text{num observations}} + \frac{\text{diff}}{\text{sum diff}}$
  $lower \leftarrow model - (PI * \sqrt{MSE} * \sqrt{temp})$
  $upper \leftarrow model + (PI * \sqrt{MSE} * \sqrt{temp})$
  if $(num < lower \parallel num > upper)$ **then**
    $count \leftarrow count + 1$
  else
    $count \leftarrow 0$
  end if
  if $(count > 4)$ **then**
    Pipeline $\leftarrow 0$ {Anomalous behaviour detected}
  else
    Pipeline $\leftarrow 1$ {No anomaly}
  end if
  $\hat{y}_{t+1} \leftarrow Predict(array[W], i)$
end while

FIGURE 5.4: Algorithm for early detection and prediction using Prediction Interval

TABLE 5.1: Detection time for Dijkstra benchmark Anomalous Dataset 1

Anomaly Classification	**Residual Distribution**			**Prediction Interval**		
One-Step Ahead Prediction	**SES**	**ARMA**	**LN**	**SES**	**ARMA**	**LN**
Fault injection (s):	7.0134			7.0134		
System crash (s):	7.0183			7.0183		
Start of anomalous behaviour (s):	7.0149			7.0149		
Anomalies detected at (s):	7.0168	7.0148	7.0152	7.0151	7.015	7.017
Detection time (μs)	1820	-120	285	245	-145	1790

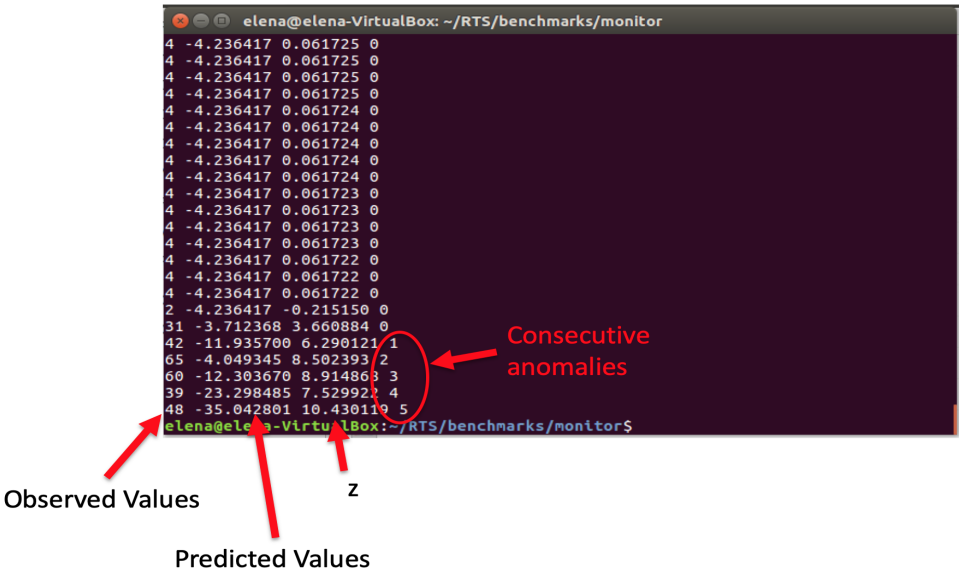


FIGURE 5.5: Simulation of the Detector Core

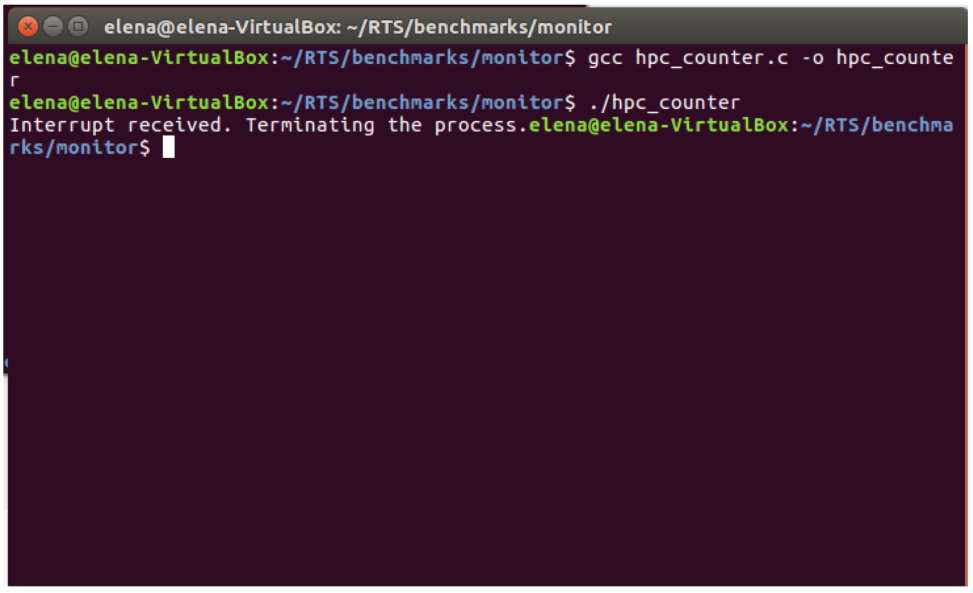


FIGURE 5.6: Simulation of the Main Core

TABLE 5.2: Detection time for Dijkstra benchmark Anomalous Dataset 2

Anomaly Classification	**Residual Distribution**			**Prediction Interval**		
One-Step Ahead Prediction	**SES**	**ARMA**	**LN**	**SES**	**ARMA**	**LN**
Fault injection (s):	6.9702			6.9702		
System crash (s):	6.9756			6.9756		
Start of anomalous behaviour (s):	6.9717			6.9717		
Anomalies detected at (s):	6.9741	6.9721	6.8700	6.9725	6.7887	6.9717
Detection time (μs)	2420	350	-101685	770	-183009	25

TABLE 5.3: Detection time for Dijkstra benchmark Anomalous Dataset 3

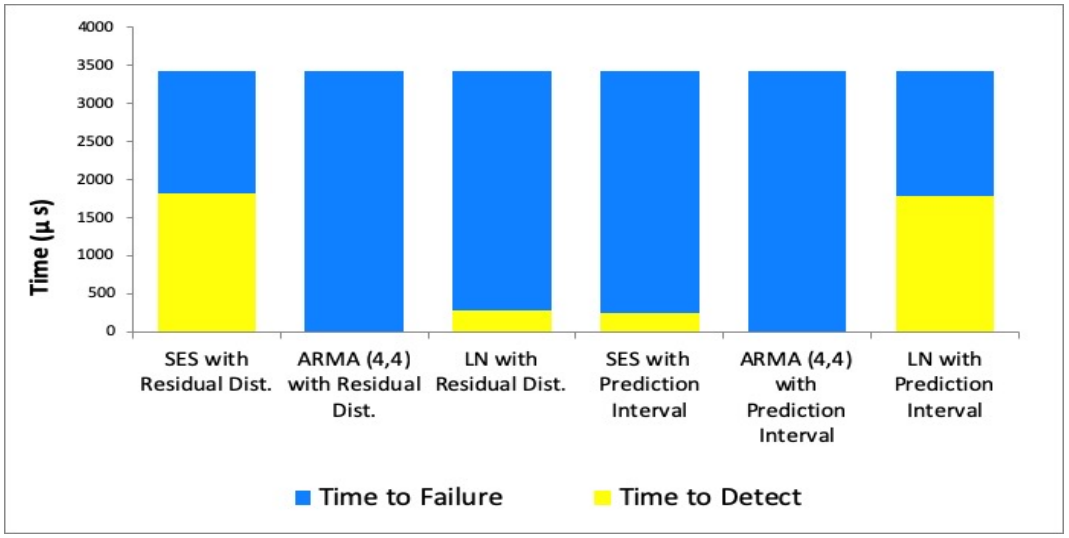
Anomaly Classification	**Residual Distribution**			**Prediction Interval**		
One-Step Ahead Prediction	**SES**	**ARMA**	**LN**	**SES**	**ARMA**	**LN**
Fault injection (s):	6.6587			6.6587		
System crash (s):	6.6636			6.6636		
Start of anomalous behaviour (s):	6.6597			6.6597		
Anomalies detected at (s):	6.6623	6.6600	6.6620	6.6622	6.6601	6.6608
Detection time (μs)	2595	350	2595	2425	405	1075

However, for detectors running with ARMA method, the detection time were -120μs and -145μs. This indicates that the detector was too sensitive and had detected false alarms instead of actual anomalies. In Table 5.2, the detectors running SES with Residual Distribution, ARMA with Residual Distribution, SES with Prediction Interval and LN with Prediction Interval have managed to predict a potential failure in 2420μs, 350μs, 770μs and 25μs, respectively, while in Table 5.3, all six detectors have managed to predict a potential failure in the system in 2595μs, 350μs, 2595μs, 2425μs, 405μs and 1075μs respectively. This means, the detector core had successfully predicted a potential failure of the main core before actual failure occurs.

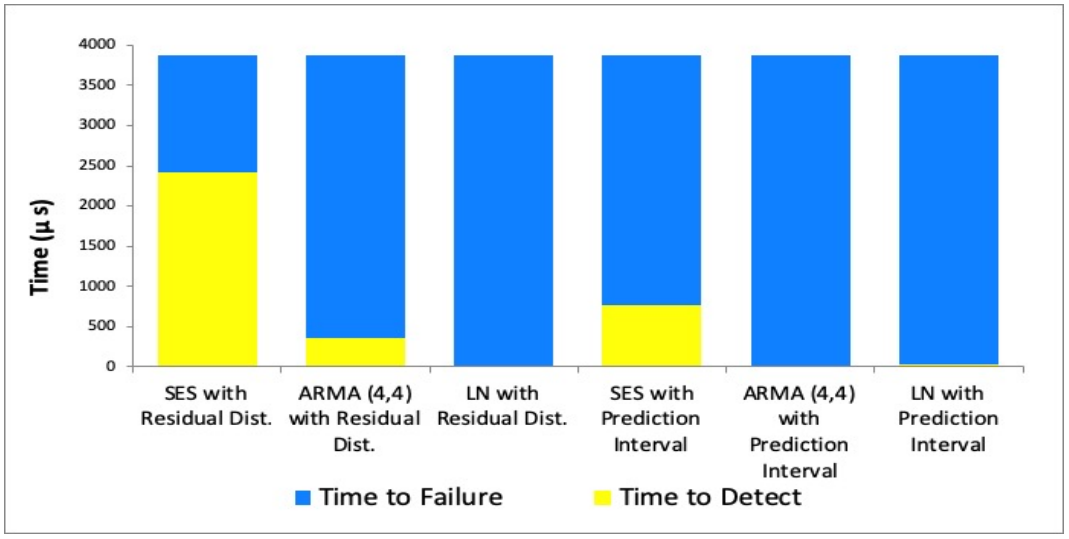
Figure 5.7 shows the result of "Time to Detect" versus "Time to Failure" for each detector. "Time to Failure" is calculated from the time the fault is injected into the system until the time when the system crashes. "Time to Detect" is calculated from the start of the anomalous behaviour in the system until the time the detector had detected 5 anomalous data points consecutively and send an interrupt to the main core notifying a potential failure. The minimum value on the Y-axis in Figure 5.7 is 0 as it shows the time to predict potential failure by detecting and identifying actual anomalies. Negative values indicate that the detector has detected false alarms and wrongly identified the normal data points as anomalies. Based on these figures, the time to detect is well-below 4,000μs. This means, there is a possibility for some preventive or corrective actions to be taken to avert the impending failure.

5.4.2 Experimental Results for the Bitcount Benchmark

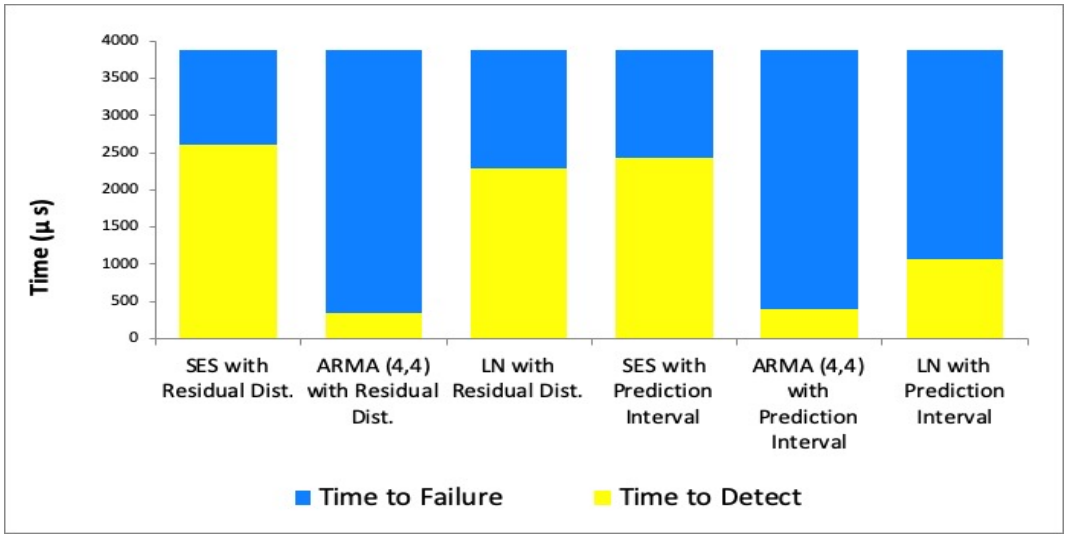
The detectors were built based on trained and optimised models, where training and optimisation were performed using the Dijkstra benchmark. These same detectors were also tested against the Bitcount benchmark. Out of twenty experiments conducted, one experiment was found to experience failure after a fault had been injected. Table 5.4 shows the result of the detector detecting anomalies and predicting a potential failure. All detectors managed to detect the anomalies and predict potential failure with the detection time ranging from 525μs to 2450μs.



(a) DataSet 1



(b) DataSet 2



(c) Dataset 3

FIGURE 5.7: Time to Detect vs Time to Failure for Dijkstra benchmark

TABLE 5.4: Detection time for Bitcount benchmark Anomalous Dataset 1

Anomaly Classification	**Residual Distribution**			**Prediction Interval**		
One-Step Ahead Prediction	**SES**	**ARMA**	**LN**	**SES**	**ARMA**	**LN**
Fault injection (s):	6.6756			6.6756		
System crash (s):	6.6806			6.6806		
Start of anomalous behaviour (s):	6.6766			6.6766		
Anomalies detected at (s):	6.6791	6.6771	6.6790	6.6791	6.6772	6.6790
Detection time (us)	2465	525	2420	2450	545	2375

In Figure 5.8, the detector that utilised the ARMA method for one-step ahead prediction has the quickest detection time compared to using SES or LN method. The time to detect using the ARMA method with either Residual Distribution or Prediction Interval is around 500μs, with at least 3000μs to spare for any preventive or corrective actions to be taken.

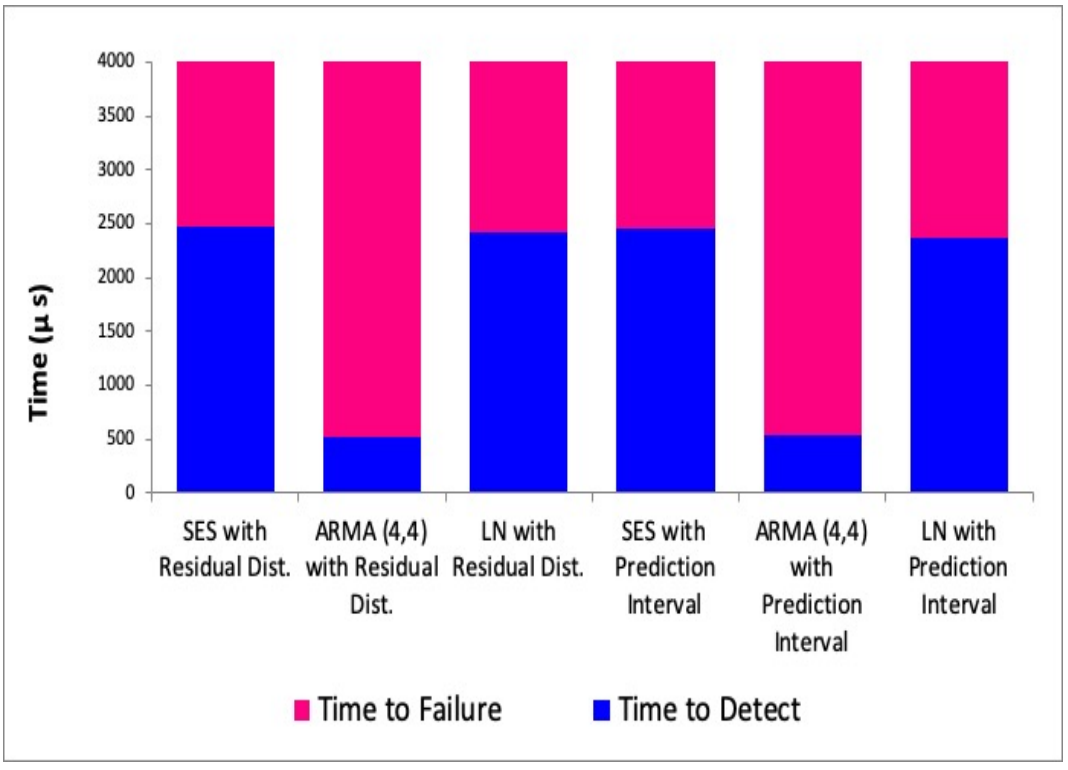


FIGURE 5.8: Time to Detect vs Time to Failure for Bitcount benchmark

5.4.3 Experimental Results for the FFT Benchmark

All six detectors were also used in the FFT benchmark. In the FFT benchmark, a total of 20 experiments were performed, and one experiment was found to experience failure after the injected fault had manifested itself as an error. Table 5.5 shows the results for each detector in detecting the anomalous behaviour and predicting a potential failure.

From the results, all detectors were able to predict a failure in the main core, with the detection time ranges from 930μs to 4025μs.

TABLE 5.5: Detection time for FFT benchmark Anomalous Dataset 1

Anomaly Classification	**Residual Distribution**			**Prediction Interval**		
One-Step Ahead Prediction	**SES**	**ARMA**	**LN**	**SES**	**ARMA**	**LN**
Fault injection (s):	7.1101			7.1101		
System crash (s):	7.1161			7.1161		
Start of anomalous behaviour (s):	7.1121			7.1121		
Anomalies detected at (s):	7.1157	7.1130	7.1141	7.1144	7.1137	7.1137
Detection time (us)	4025	930	1960	2270	1610	1635

Figure 5.9 shows the time to detect versus time to failure for all six detectors detecting the anomalies in the FFT benchmark. From the figure, it is seen that the detector that utilises ARMA for prediction has the quickest detection time compared to SES and LN. The detector took 930μs to predict a potential failure with 3070μs to spare.

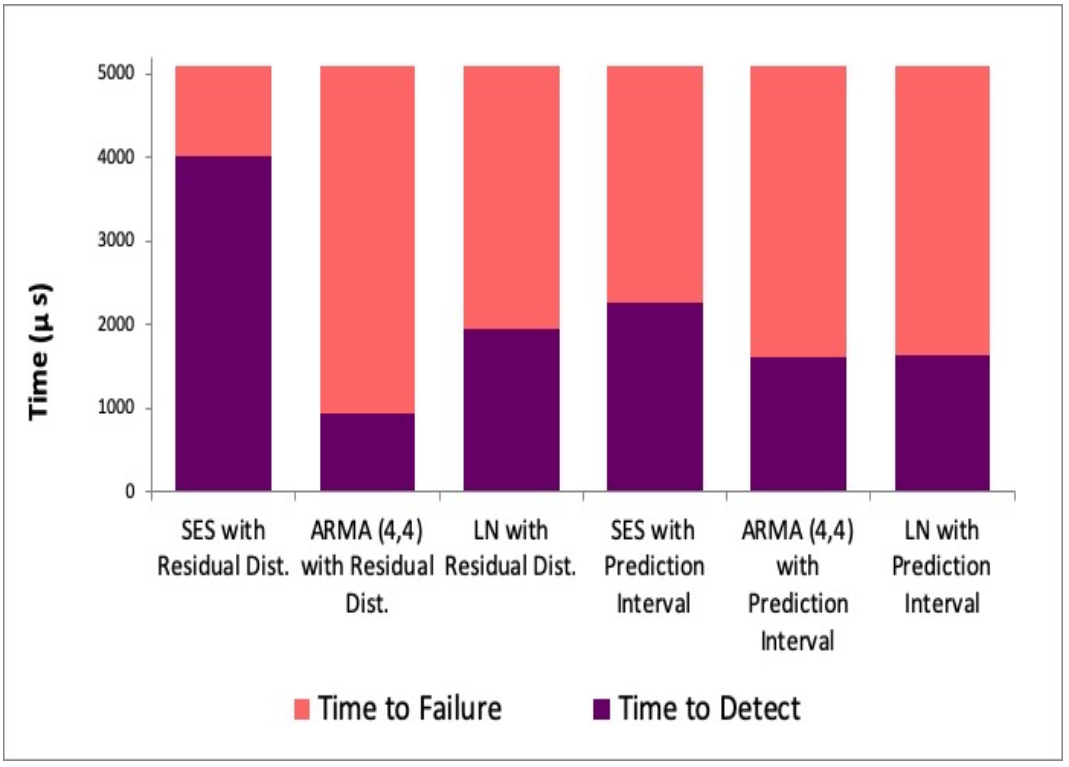


FIGURE 5.9: Time to Detect vs Time to Failure for FFT benchmark

All three benchmarks, as illustrated in Figure 5.7, Figure 5.8 and Figure 5.9, show that the ARMA method is the most suitable method for predicting the next data point. Both FFT and Bitcount benchmarks had not been used in the training, validation and testing of the algorithm in Chapter 4, but still the detector managed to detect the anomalous behaviour and predict potential failure well before the time of failure. For measuring the deviation between the predicted point and the actual point, both Residual Distribution

and Prediction Interval are comparable. The only downside of using Prediction Interval for anomaly classification is that it requires at least 20μs for the calculation to stabilise, which means, if the fault is manifested into an error during that initial period, it will not be detectable. This is due to the parameter in Prediction Interval algorithm which is the *standard error of the predicted model*. One of the component in the parameter is the Mean Squared Error (MSE), shown in Equation 4.19. In the beginning of the time series data, the calculated value of MSE is large when the total sum of squared error is divided by a small number of sample, n. As n increases, the MSE decreases until it stabilises at a point of time, which in this experiment, is after 20μs.

5.5 Performance Analyses of the Detector

Analysis on the performance of the detector is done by calculating the total execution time, T using the given formula in Equation 5.1.

$$T = I * CPI * \text{CPU Clock Cycle} \tag{5.1}$$

The detector is designed to run on secondary processor core. Both the main core and the secondary core were simulated using the same Intel processor, and the main processor is tasked to run the benchmarks as a form of single-tasking system. The assembly codes for all six different detectors which were developed are obtained, and the reciprocal throughput and latencies for each instruction are calculated based on Intel processor metrics obtained from [125]. The CPU clock cycle is calculated as 1/Clock Rate where the clock rate is set at 250MHz. Table 5.6 displays the results of total instructions, I, Cycles per Instruction, CPI, and the total execution time, T, for each detector.

TABLE 5.6: Performance in execution time of each method measured on an Intel architecture

Method	**Total Instructions (I)**	**Cycles per Instruction (CPI)**	**Total Execution Time (T) - μs**
SES with Residual Distribution	250	1.8780	1.878
ARMA with Residual Distribution	311	1.7186	2.138
LN with Residual Distribution	243	1.8992	1.846
SES with Prediction Interval	334	1.7006	2.272
ARMA with Prediction Interval	384	1.6810	2.582
LN with Prediction Interval	330	1.8273	2.412

As shown in Table 5.6, the total execution time for evaluating a single data point ranges from 1.8μs to 2.6μs, well below the sampling time of 5μs (or 5000ns), which is equivalent to sampling at every 1250 clock cycles for a processor with a clock rate of 250MHz. In

other words, as the current data point being sampled, the detector is able to determine if the current data point is normal or anomalous. The total number of instructions using Prediction Interval is higher compared to using Residual Distribution as the computation for lower and upper bounds require more values as can be seen in Algorithm 5.4. The CPI for methods using Residual Distribution are just slightly higher as there are more dependencies in the detection and prediction algorithm, hence there are more latencies.

5.6 Source Byte Analysis of the Detector

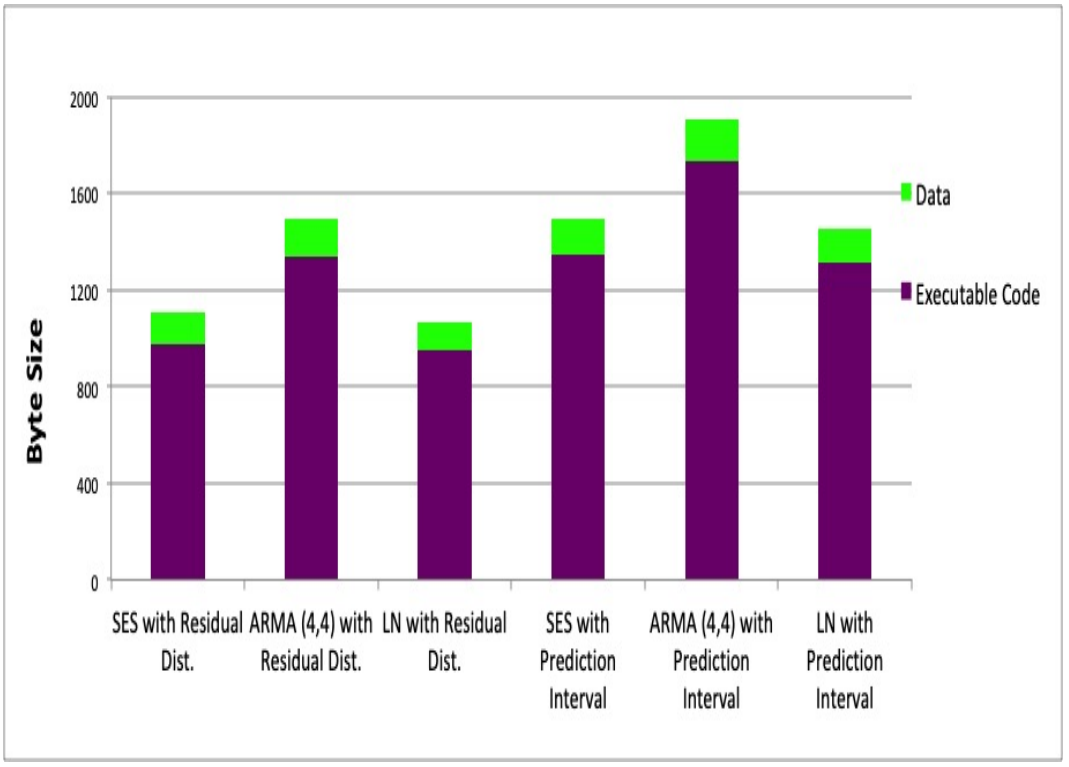


FIGURE 5.10: Size of detector in bytes

The size of the detector is measured by the size of the executable code and data. As can be seen from Figure 5.10, the combined size of executable code and data did not exceed 2000 bytes or 2 kB. As ARMA has the most instructions compared to SES and LN, naturally the size of the detector using ARMA will be bigger. The size of the detector that applies Prediction Interval method is also bigger by almost 500 bytes compared to using Residual Distribution. Figure 5.11 compares the size of the detector that uses ARMA with Residual Distribution with the size of the embedded benchmarks used for testing. As can be observed, the size of the detector is as light as the embedded benchmarks used. However, the size and complexity of the detector developed in this work are independent of the benchmarks.

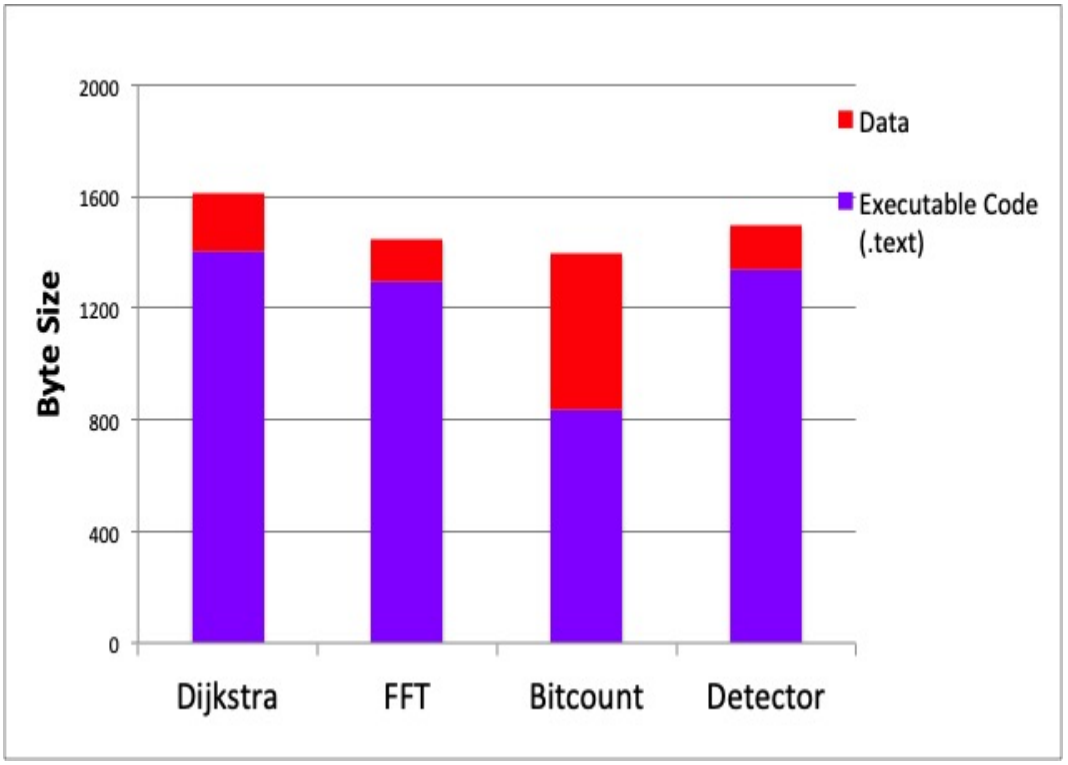


FIGURE 5.11: Size of detector in comparison with size of benchmarks

5.7 Summary on the Results Analyses

The one-step ahead prediction using the ARMA method has provided the quickest detection time in most of the datasets for Dijkstra, Bitcount and FFT benchmarks compared to using SES or LN prediction methods. From these analyses, it is observed that all the techniques detected anomalous behaviour well before the system failed, but the combination of using ARMA method with Residual Distribution method is the fastest, as shown in Figure 5.7, Figure 5.8 and Figure 5.9. The Residual Distribution methods are faster compared to Prediction Interval method for evaluating a single data point as shown in Table 5.6 and have slightly smaller code sizes as shown in Figure 5.10. Thus, by a small margin, the combination of ARMA with Residual Distribution is the best choice for prediction and detection of anomalies.

Placing the detector on a secondary core means there will be no additional hardware imposed on the main core. The main core utilises the existing hardware performance counter in its own core and send the data to the secondary core for detection of anomalous behaviour. Based on source byte analysis, the proposed detector can be deemed as a lightweight detector since the size is below 2 kB. It is also worth noting that the size of the detector is independent of the size of the benchmarks, which means, the size of the detector remains at 2 kB even if the size of the benchmark exceeds 2 kB. Although the experiments were performed in the form of simulations, other work has demonstrated that using a secondary processor core to monitor main processor core's

HPC is possible [37]. However, compared to this work which uses real-time streaming HPC data to detect for anomalies and predict potential failure, they capture the total count of HPC data after the application has completed and performed the analysis offline to determine if the application running is benign or anomalous.

5.8 Summary

In this chapter, a proof of concept for a lightweight detector to predict potential failure through the detection of anomalous behaviour in a microcontroller is presented. The detector has been designed to run from a secondary processor, where the main core sends one HPC data sampled at every 1250 clock cycles via an inter-core communication pipeline. The secondary core, dubbed as Core B, will first load the detector program after start-up, and begin predicting the next data. Once it receives the actual data from the main core, it will measure how much the actual data deviate from the threshold and if the deviation exceeds the threshold, the actual data will be classified as anomalous. Five anomalies detected consecutively will result in the secondary core sending an interrupt to the main core via another inter-core communication pipeline to alert for potential failure. The main core, after receiving the interrupt, will close the communication pipeline, halt the application and raise an alarm for potential failure.

The lightweight detector was designed based on the early detection and prediction algorithm developed in Chapter 4. It is worth noting that while the early detection and prediction algorithm was developed solely based on Dijkstra benchmark, it has been proven that it is a general model where it can be applied to other benchmarks, such as FFT and Bitcount. Based on the results presented, it is possible to predict potential failure in the real-time by detecting the anomalous behaviour that appeared in the processor when a fault has manifested into an error. The detector has managed to detect anomalous behaviour and predict potential failure in all three benchmarks below the "Time to Failure" of 4000μs, with ARMA prediction method and Residual Distribution for anomaly classification providing the best performance compared to other methods. There was no cost involved in the main core, as the HPC is built-into the processor, while for the detector, the total size of the detector does not exceed 2 kB. As mentioned earlier, each HPC data is sampled at 1250 clock cycles, which is equivalent to every 5μs for a 250MHz clock speed. The total execution time of a single data point using ARMA(4,4) with Residual Distribution was 2.138μs, well within the sampling time.

While both the main core and the detector core were simulated using the same Intel processor, it is possible for the detector core to operate on an Intel processor which is smaller and with reduced power compared to the main core as it is shown that the total size of the detector core is less than 2 kB.

Chapter 6

Conclusions and Future Work

Reliability in safety-critical embedded systems is a major concern because a failure in a safety-critical embedded system can result in death, injury, severe damage to equipment, property or environment. While existing research in fault tolerant systems focused on providing a complete error tolerant system, the techniques often come with high overhead, are resource-intensive and some techniques have only managed to detect the fault after a failure has occurred. The contributions from the work described in this thesis provide a mechanism to predict potential failure in real-time by monitoring and detecting anomalous behaviour in the system using a Hardware Performance Counter (HPC). The work in this thesis addresses the gap found in current research on fault tolerant systems and is targeted to complement current fault tolerance techniques and contribute to a better protection strategy for embedded systems. The proposed detector that can predict potential failure in the processor in real-time is the first of its kind, and not only it does not require any additional hardware resources, it is also very light (less than 2 kB of code).

6.1 Summary and Contributions

In this thesis, the work presented can be evaluated against the research questions and research objectives detailed in Chapter 1, Section 1.4, which are reiterated here for clarity.

Research questions:

1. *Is it possible to predict a potential failure in an embedded system by monitoring and detecting the anomalous behaviour in the system?*
2. *What are the available hardware performance counters in a processor which can represent a behaviour of a system and can be monitored online and in real-time?*

3. *What are the suitable techniques to model the behaviour of the system and perform early detection of anomalies to predict potential failure?*

4. *How do different prediction algorithms impact the implementation of the detector?*

Research objectives:

1. *Investigate how a manifested fault affects the behaviour of the system and identify the various Performance Monitoring Events (PMEs) available that can be used across different types of processors. Identify the number of hardware performance counters available in processors used in embedded systems, in particular, the number of available counters in an Intel Atom processor used in this work. Select different PMEs and compare them to determine which is better for detection.*

2. *Develop an algorithm for early detection suitable to be implemented in embedded systems taking into account the constraints and limitations of an embedded system. Explore several methods for one-step ahead prediction and anomaly classification rules and perform evaluation on methods used in the early detection algorithm.*

3. *Implement the developed algorithm as a hardware-based detector. Validate the implementation through experimental simulations and analyse the performance and cost of the proposed detector.*

The first objective is achieved in Chapter 3 where a HPC is used to monitor the anomalous behaviour caused by manifested faults. As presented in Chapter 2, the use of HPCs has been applied in various research such as performance evaluation, workload estimation and detection for malicious activities. A common trait in all these works is how HPCs have been used to identify or detect some form of deviation from normal or expected behaviour. But none have actually used HPC to predict potential failure in the processor, caused by a single bit flip, by monitoring and detecting anomalous behaviour at a system-level. A system that behaves normally exhibits a pattern, and any deviation from that pattern indicates an anomaly had occurred. The experiments which were performed using GemFI, show how the behaviour pattern of the system leading either to a crash or a hang can be clearly observed by using a single HPC. From the experiments conducted, it takes about 1,000,000 (or 1M) clock cycles for a system to crash from the time the fault manifested as an error. This information is then used to guide the detection algorithm where error detection and prediction of potential failure has to be done below 1M clock cycles (or 4,000μs) for a system running at a clock speed of 250MHz.

Two different architectural-level PMEs that were common across different types of processor were selected for comparison, namely (i) number of instructions retired PME, and (ii) number of cache misses PME. While both PMEs can be used for anomalous

behaviour monitoring, the latter is better suited for detection where the counter data records a bigger deviation (more than 10%) when the pattern begins to deviate from the normal behaviour compared to the former where the deviation recorded is around 5%. The cache miss values are much lower than the instructions retired values, and hence, where cache miss values have between three and seven bits, the instructions retired values have around seventeen bits. This means the computational effort and speed to perform early detection using cache misses will be smaller. Therefore, the first objective has been fulfilled.

The second objective is achieved in Chapter 4 where a novel algorithm to predict potential failure in real-time by monitoring and detecting anomalous behaviour in a processor is presented. Embedded systems have limitations and constraints concerning hardware resources, speed, power and memory size. Therefore, the algorithm for error detection and prediction of potential failure must be lightweight with minimal computational complexity and do not require any pre-processing on the data. Statistical methods are preferred over machine learning algorithms as these methods not only satisfy the above criteria, but statistical methods have also been found to outperformed machine learning algorithms in terms of forecasting accuracy.

The novel algorithm consists of three stages, with each stage building from its predecessor: (i) predicting the next data using one-step ahead prediction method, (ii) measuring the deviation between predicted data and actual data, and (iii) classifying if the actual data deviates "too much" and is deemed anomalous. Based on the analyses conducted in Chapter 3, when a fault manifested as an error, the counter value begins to deviate and the system starts behaving anomalously before it finally experiences failure. Based on this characteristic, prediction of potential failure relies on how many consecutive anomalies are required to be detected. From the experiments conducted, the algorithm is able to predict potential failure with low detection time and high accuracy if there are five anomalies detected consecutively. The detection time attribute in this thesis is a novel performance measurement attribute that specifically measures how well the early detection and prediction algorithm perform. It refers to the earliest time for the algorithm to predict potential failure.

Three different methods have been applied for one-step ahead prediction, namely, (i) Single Exponential Smoothing (SES), (ii) Autoregressive Moving Average (ARMA) and (iii) Single-Layer Linear Network (LN) while two anomaly classification methods namely, (i) Residual Distribution and (ii) Prediction Interval for measuring the deviation and classification are explored. Based on the results of the experiment, the earliest detection time achieved was 325μs from the time the fault has manifested itself into an error, and this was achieved with the optimum parameters $z_{thresh} = 6$ for Residual Distribution and $PI = 90\%, df = 1, W = 3$ for Prediction Interval. ARMA (4,4) has proven to be a better prediction method as the number of false alarms and missed anomalies (FPs and FNs) are lower compared to SES and LN methods. Between Residual Distribution and

Prediction Interval, the latter is at a certain disadvantage for it requires a time of at least 20μs for the calculation to stabilise. Hence, if an error occur at the initial start-up of the program, the Prediction Interval method for anomaly classification will not be able to detect those anomalous behaviours, and would not be able to predict a potential failure. Therefore, based on these results, the second objective has been fulfilled.

In Chapter 5, the third objective of the thesis is achieved where a proof of concept for a lightweight detector that predicts potential failure from the detection of anomalous behaviour is presented. A total of six different detectors were designed and simulated, with each detector built using either SES, ARMA or LN methods for one-step ahead prediction and Residual Distribution or Prediction Interval methods for anomaly classification. The detector is designed to run on a secondary core, where it receives one HPC data from the main core at every 1250 clock cycles via a dedicated, inter-core communication pipeline. It measures the deviation between the HPC data received from the main core and the data it predicted earlier and marked the observed data as anomalous if it exceeds the threshold set either in Residual Distribution or Prediction Interval method. If the detector detected five anomalies consecutively, it sends an interrupt to the main core to alert for potential failure. Upon receiving the interrupt from the detector, the main core will halt the application and raise an alarm for potential failure.

In Chapter 4, the detectors were developed based on Dijkstra benchmark. However, the detectors were also tested on two additional benchmarks that have not been trained or tested. Based on the results obtained, it is proven that the detector has managed to detect the anomalous behaviour that had occurred and was able to predict potential failure in real-time before the system experience failure. For Dijkstra benchmark, the detector raised the alarm for potential failure at 350μs, where else, for FFT and Bitcount benchmarks, the alarm is raised at 930μs and 525μs respectively, well below the 4,000μs limit. These results were recorded using the detector which implements ARMA(4,4) for one-step ahead prediction and Residual Distribution for measurement of deviation. The total execution time required for a single data point was 2.138μs, which is well-within the sampling time of 5μs. This means that within the time of 2.138μs, the detector is able to predict the next data, measure the deviation between predicted data and actual data, and classify the actual data as normal or anomalous before sending an interrupt to the main core if the number of consecutive anomalies exceeds 5. The size of the detector is also very small, less than 2 kB. No modification was required on the main core as all the prediction and classification of the HPC data was performed on the secondary core, and the main core is only tasked to send the existing, built-in HPC data from the processor core itself to the secondary core via the inter-core communication pipeline. Therefore, the third objective has been achieved.

All the four contributions in this thesis are aimed at providing a new strategy for online error detection in a processor. Based on the findings in this work, it can be concluded that it is possible to predict a potential failure in the embedded system by monitoring

the system for any anomalous behaviour. The lightweight detector proposed is suitable to be used in multicore microcontrollers and there is no additional cost imposed on the main core running the application. This novel algorithm for early detection and prediction of potential failure in a processor has proven to work even on benchmarks that were not used for training and testing. This algorithm can complement existing fault forecasting and fault tolerance techniques and will contribute to a better protection strategy for microprocessors, especially those that are used in embedded computing.

6.2 Future Work

The combination of fault forecasting, fault tolerance, fault removal and fault prevention techniques helps in developing a dependable system for embedded computing. The research in this thesis is focused on predicting potential failure through the detection of anomalous behaviour in a processor. Some possible areas for future research which have been identified are presented as follows.

6.2.1 Diagnostics

This research could be further extended to include diagnostic information. At the present, the detector is able to predict potential failure in the processor but provides no diagnostic information as to the location of the fault that causes the anomalous behaviour or the type of fault that causes the anomalous behaviour. Additional diagnostic data will assist in ensuring proper corrective action is taken.

6.2.2 Recovery

Another important area that has been identified is to design corrective action that will be applicable and suitable to be taken to address the anomalies detected in the system. While the detector is able to detect in advance the signs of a system behaving anomalously, the detector does not address the corrective action that needs to be taken. In other words, there is no fault correction attempt by the detector besides halting the application. This extreme corrective action of halting the system may be practical or applicable for some non-critical embedded systems, but for some safety-critical systems, this action may be deemed to extreme and impractical to be applied as it would cause catastrophic results. Therefore, the technique of applying suitable correction is still very much dependent on the nature of the application itself. However, as the prediction, detection and classification process for a single point performed by the detector only takes up about 2.138μs, there is at least 2.5μs available for corrective action to be taken.

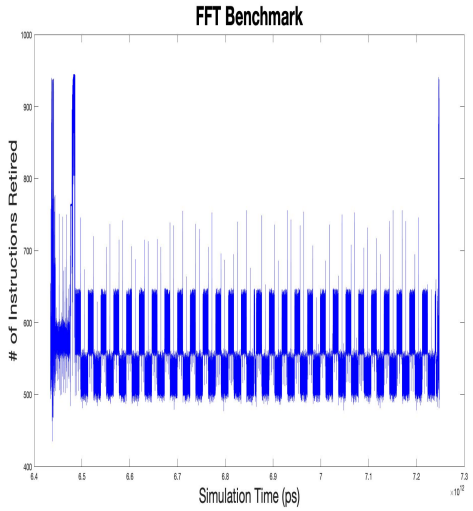
6.2.3 Implementation of the Inter-Core Communication Pipeline

Current technology allows core-to-core communication via shared memory space between cores. However, shared memory communication involves coherence invalidation and cache misses, which means this could affect how the HPC data is collected by the main core. The goal of this research is to be able to predict potential failure in real-time using HPC where data recorded truly depicts the behaviour of the system. Implementation of a dedicated, inter-core communication pipeline as suggested will ensure the reliability of the data collected and analysed by the detector. This dedicated pipeline will ensure the HPC data sent from main core to detector core is not compromised or changed and the HPC data will provides a real picture of the behaviour of the main core running the specific application.

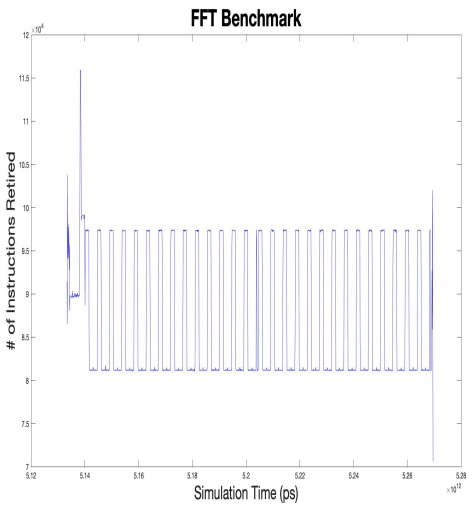
Appendix A

Execution Profiles for FFT, Stringsearch and QSort Benchmarks

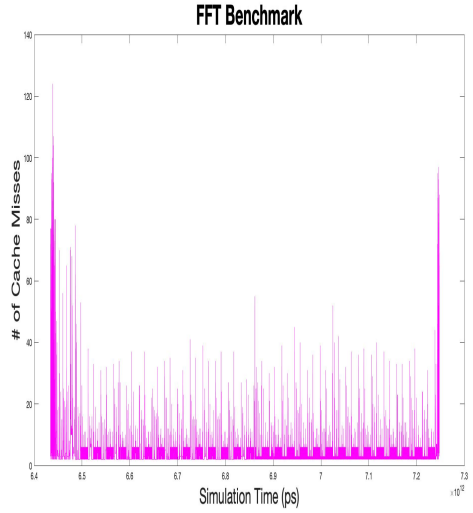
Figure A.1, Figure A.2 and Figure A.3 shows the execution profiles obtained from FFT, StringSearch and QSort benchmark where the execution profiles are plotted using two different PMEs, namely instructions retired PME and cache misses PME.



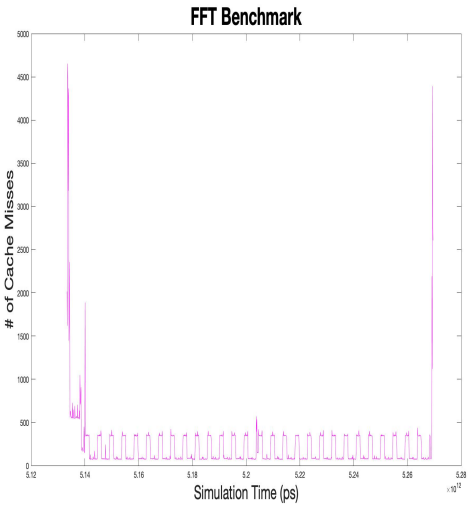
(a) Clock Speed = 250MHz



(b) Clock Speed = 2GHz

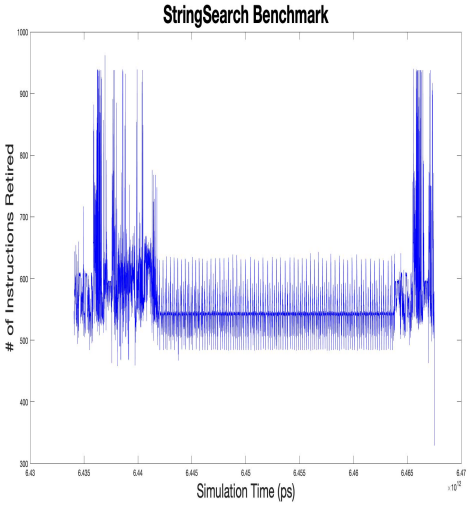


(c) Clock Speed = 250MHz

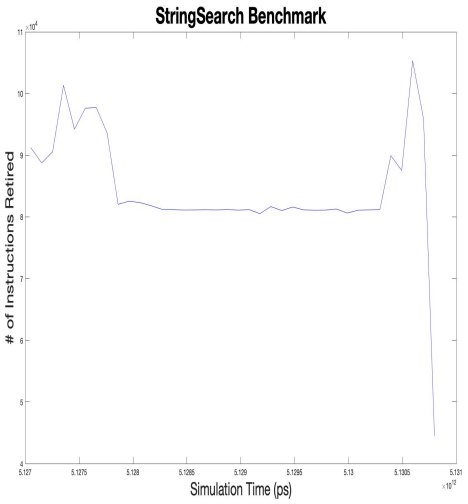


(d) Clock Speed = 2GHz

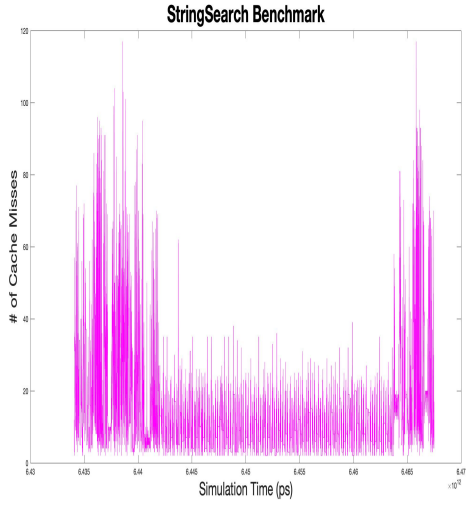
FIGURE A.1: Execution profiles using Number of Instructions Retired and Number of Cache Misses for FFT benchmark running at 250MHz and 2GHz clock speed



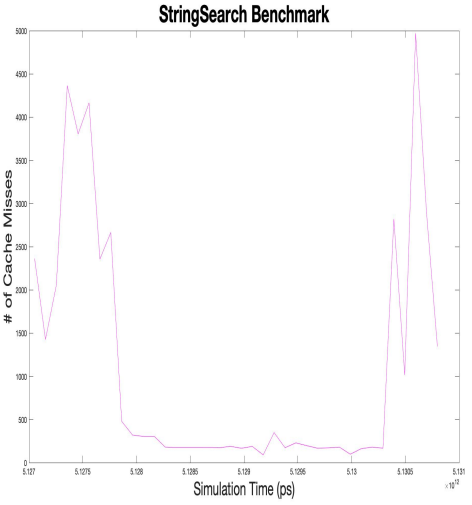
(a) Clock Speed = 250MHz



(b) Clock Speed = 2GHz

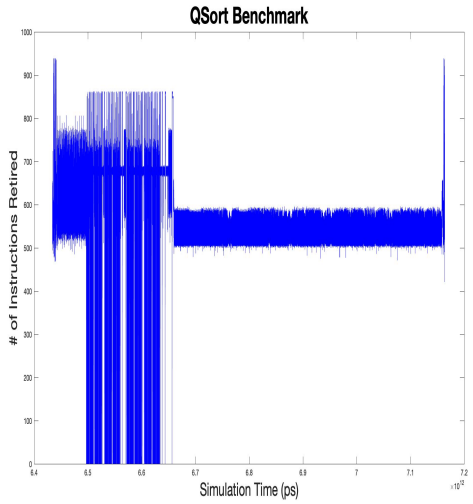


(c) Clock Speed = 250MHz

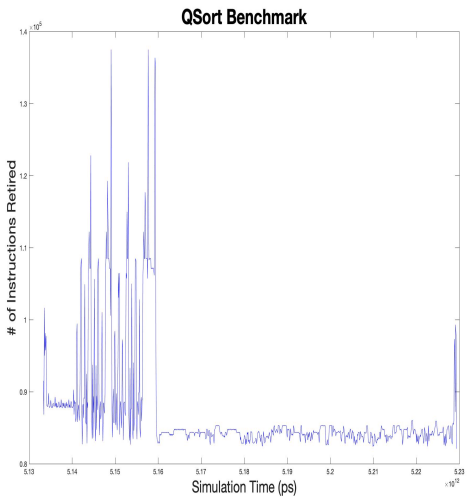


(d) Clock Speed = 2GHz

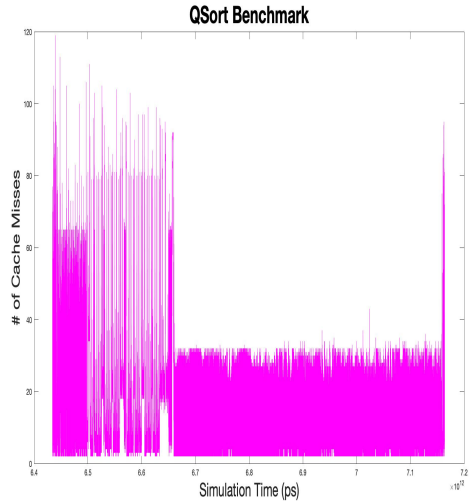
FIGURE A.2: Execution profiles using Number of Instructions Retired and Number of Cache Misses for StringSearch benchmark running at 250MHz and 2GHz clock speed



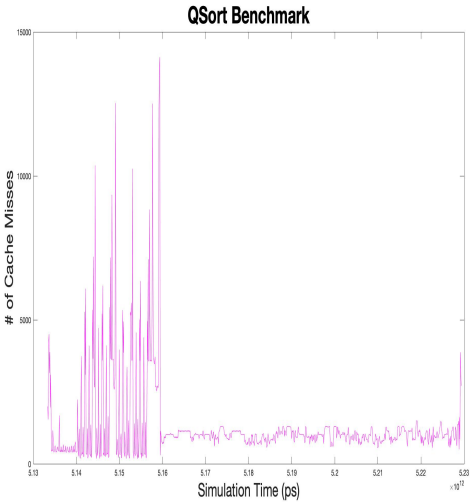
(a) Clock Speed = 250MHz



(b) Clock Speed = 2GHz



(c) Clock Speed = 250MHz



(d) Clock Speed = 2GHz

FIGURE A.3: Execution profiles using Number of Instructions Retired and Number of Cache Misses for QSort benchmark running at 250MHz and 2GHz clock speed

Appendix B

Error Distribution for Dijkstra, FFT, Bitcount and StringSearch Benchmarks

TABLE B.1: Statistics on error distribution for Dijkstra benchmark

Pipeline Stage	Type of Failure	Segmentation Fault	Invalid Opcode	Kernel Panic	NULL Pointer	Bad Paging	Assertion Error	Bad Trap	General Protection Fault	Stack Overflow	No Error	Total
Fetch	**Not Manifested**	0	0	0	0	0	0	0	0	0	24	**24**
	Fail Silence Violation	0	0	0	0	0	0	0	0	0	3	**3**
	Hang	0	0	0	0	0	0	2	0	0	0	**2**
	Crash	9	1	0	0	0	0	0	0	1	0	**11**
Decode	**Not Manifested**	0	0	0	0	0	0	0	0	0	0	**0**
	Fail Silence Violation	0	0	0	0	0	0	0	0	0	0	**0**
	Hang	0	0	0	0	0	0	0	0	0	0	**0**
	Crash	0	0	0	0	0	0	0	0	0	0	**0**
Execute	**Not Manifested**	0	0	0	0	0	0	0	0	0	24	**24**
	Fail Silence Violation	0	0	0	0	0	0	0	0	0	1	**1**
	Hang	0	0	0	0	0	0	0	0	0	0	**0**
	Crash	4	0	0	0	0	0	0	0	1	0	**5**
Load/Store	**Not Manifested**	0	0	0	0	0	0	0	0	0	18	**18**
	Fail Silence Violation	0	0	0	0	0	0	0	0	0	9	**9**
	Hang	0	0	0	0	0	0	0	0	0	0	**0**
	Crash	3	0	0	0	0	0	0	0	0	0	**3**

TABLE B.2: Statistics on error distribution for FFT benchmark

Pipeline Stage	Type of Failure	Segmentation Fault	Invalid Opcode	Kernel Panic	NULL Pointer	Bad Paging	Assertion Error	Bad Trap	General Protection Fault	No Error	Total
Fetch	**Not Manifested**	0	0	0	0	0	0	0	0	24	**24**
	Fail Silence Violation	0	0	0	0	0	0	0	0	1	**1**
	Hang	0	0	1	0	0	0	2	0	0	**3**
	Crash	2	0	0	0	0	0	0	0	0	**2**
Decode	**Not Manifested**	0	0	0	0	0	0	0	0	30	**30**
	Fail Silence Violation	0	0	0	0	0	0	0	0	0	**0**
	Hang	0	0	0	0	0	0	0	0	0	**0**
	Crash	0	0	0	0	0	0	0	0	0	**0**
Execute	**Not Manifested**	0	0	0	0	0	0	0	0	17	**17**
	Fail Silence Violation	0	0	0	0	0	0	0	0	4	**4**
	Hang	0	0	0	1	4	0	1	0	0	**6**
	Crash	1	0	1	0	0	0	0	1	0	**3**
Load/Store	**Not Manifested**	0	0	0	0	0	0	0	0	21	**21**
	Fail Silence Violation	0	0	0	0	0	0	0	0	2	**2**
	Hang	0	0	0	0	3	0	0	0	0	**3**
	Crash	4	0	0	0	0	0	0	0	0	**4**

TABLE B.3: Statistics on error distribution for Bitcount benchmark

Pipeline Stage	Type of Failure	Segmentation Fault	Invalid Opcode	Kernel Panic	NULL Pointer	Bad Paging	Assertion Error	Bad Trap	General Protection Fault	No Error	Total
Fetch	Not Manifested	0	0	0	0	0	0	0	0	23	**23**
	Fail Silence Violation	0	0	0	0	0	0	0	0	2	**2**
	Hang	0	0	0	0	0	0	0	0	0	**0**
	Crash	4	1	0	0	0	0	0	0	0	**5**
Decode	Not Manifested	0	0	0	0	0	0	0	0	30	**30**
	Fail Silence Violation	0	0	0	0	0	0	0	0	0	**0**
	Hang	0	0	0	0	0	0	0	0	0	**0**
	Crash	0	0	0	0	0	0	0	0	0	**0**
Execute	Not Manifested	0	0	0	0	0	0	0	0	26	**26**
	Fail Silence Violation	0	0	0	0	0	0	0	0	2	**2**
	Hang	0	0	0	0	0	0	0	0	0	**0**
	Crash	2	0	0	0	0	0	0	0	0	**2**
Load/Store	Not Manifested	0	0	0	0	0	0	0	0	20	**20**
	Fail Silence Violation	0	0	0	0	0	0	0	0	2	**2**
	Hang	0	0	1	1	0	0	1	0	0	**3**
	Crash	5	0	0	0	0	0	0	0	0	**5**

TABLE B.4: Statistics on error distribution for StringSearch benchmark

Pipeline Stage	Type of Failure	Segmentation Fault	Invalid Opcode	Kernel Panic	NULL Pointer	Bad Paging	Assertion Error	Bad Trap	General Protection Fault	Stack Overflow	No Error	Total
Fetch	**Not Manifested**	0	0	0	0	0	0	0	0	0	23	**23**
	Fail Silence Violation	0	0	0	0	0	0	0	0	0	3	**3**
	Hang	0	1	0	1	2	0	0	0	0	0	**4**
	Crash	0	0	0	0	0	0	0	0	0	0	**0**
Decode	**Not Manifested**	0	0	0	0	0	0	0	0	0	30	**30**
	Fail Silence Violation	0	0	0	0	0	0	0	0	0	0	**0**
	Hang	0	0	0	0	0	0	0	0	0	0	**0**
	Crash	0	0	0	0	0	0	0	0	0	0	**0**
Execute	**Not Manifested**	0	0	0	0	0	0	0	0	0	21	**21**
	Fail Silence Violation	0	0	0	0	0	0	0	0	0	1	**1**
	Hang	0	0	3	0	0	0	1	1	0	0	**5**
	Crash	3	0	0	0	0	0	0	0	0	0	**3**
Load/Store	**Not Manifested**	0	0	0	0	0	0	0	0	0	21	**21**
	Fail Silence Violation	0	0	0	0	0	0	0	0	0	1	**1**
	Hang	0	2	0	0	3	0	0	0	1	0	**6**
	Crash	2	0	0	0	0	0	0	0	0	0	**2**

Appendix C

Matlab Code

C.1 One-Step Ahead Prediction

C.1.1 Single Exponential Smoothing

```
function y = singleSmoothed(data, model, alpha)
    y = (alpha * data) + ((1-alpha) * model);
end
```

C.1.2 Autoregressive Moving Average

```
function y = arma(data, diff, const, ar, ma, p, q, i)
    delta = 0;
    theta = 0;
    if p ~= 0
        for a = 1:p
            if (i - a) <= 0
                e = 1;
            else
                e = i - a;
            end
            delta = delta + (ar{a} * data(e));
        end
    end
    if q ~= 0
        for a = 1:q
            if (i - a) <= 0
                e = 1;
            else
                e = i - a;
            end
            theta = theta + (ma{a} * diff(e));
        end
    end
    y = const + delta + theta;
end
```

C.1.3 Single-Layer Linear Network

```
function model = linearNetwork(data, position, window)
    sum_weight = 0;
    result = 0;
    for i = 1:window
        weight = window - i + 1;
        sum_weight = sum_weight + weight;
        point = position - i;
        if (point <= 0)
            result = result + 0;
        else
            result = result + (data(point) * weight);
        end
    end
    model = result / sum_weight;
end
```

References

[1] Intel, *Intel ®64 and IA32 Architectures Performance Monitoring Events*, Intel.

[2] ARM, *Cortex-A9 Technical Reference Manual*, ARM.

[3] W. A. Fuller, *Introduction to Statistical Time Series*. Wiley, 1976.

[4] P. C. Anderson, F. J. Rich, and S. Borisov, "Mapping the South Atlantic Anomaly continuously over 27 years," *Journal of Atmospheric and Solar-Terrestrial Physics*, vol. 177, pp. 237 – 246, 2018, dynamics of the Sun-Earth System: Recent Observations and Predictions.

[5] S. Teoh, "RM142m RazakSAT faulty after just one year, says federal auditor," *The Malaysian Insider*, October 2011.

[6] N. G. Leveson and C. S. Turner, "An investigation of the Therac-25 accidents," *Computer*, vol. 26, no. 7, pp. 18–41, July 1993.

[7] J.-L. Lions, "ARIANE 5 – flight 501 failure," Independent Inquiry Board, Failure Report, 1996. [Online]. Available: https://esamultimedia.esa.int/docs/esa-x-1819eng.pdf

[8] A. DeHon, N. Carter, and H. Quinn, "Final report of CCC cross-layer reliability visioning study," Computing Community Consortium (CCC) Visioning Study, United States, Full Report of Computing Community Consortium (CCC) Visioning Study, 2011. [Online]. Available: http://www.relxlayer.org/FinalReport?action=AttachFile&do=view&target=final_report.pdf

[9] N. Wehn, "Reliability: A cross-disciplinary and cross-layer approach," *Asian Test Symposium*, pp. 496–497, 2011.

[10] P. Marwedel, *Embedded System Design: Embedded Systems Foundations of Cyber-Physical Systems, and the Internet of Things*. Springer, 2018.

[11] A. Armoush, "Design patterns for safety-critical embedded systems," Ph.D. dissertation, RWTH Aachen University, 2010.

[12] MarketsAndMarkets.com, "Embedded systems market by hardware (MPU, MCU, application specific IC / application specific standard product, DSP, FPGA, and memory), software (middleware and operating system), application, and geography - global forecast to 2023," United States, 2017. [Online]. Available: https://www.marketsandmarkets.com/Market-Reports/embedded-system-market-98154672.html

[13] M. D. P. Emilio, *Features of Embedded Systems*. Switzerland: Springer International Publishing, 2015, pp. 25–31. [Online]. Available: https://doi.org/10.1007/978-3-319-06865-7_2

[14] H. Psaier and S. Dustdar, "A survey on self-healing systems: Approaches and systems," *Computing*, vol. 91, no. 1, pp. 43–73, Jnauary 2011.

[15] A. Avizienis, J. C. Laprie, B. Randell, and C. Landwehr, "Basic concepts and taxonomy of dependable and secure computing," *IEEE Transactions on Dependable and Secure Computing*, vol. 1, no. 1, pp. 11–33, Jan 2004.

[16] E. Dubrova, *Fault-Tolerant Design*. Springer, 2013.

[17] N. R. Storey, *Safety Critical Computer Systems*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1996.

[18] J. C. Knight, "Safety critical systems: challenges and directions," in *24th International Conference on Software Engineering (ICSE)*, May 2002, pp. 547–550.

[19] ITRS, "International technology roadmap for semiconductors, 2013 edition - process integration, devices and structures summary," The International Technology Roadmap for Semiconductors: 2013, International Technology Roadmap for Semiconductors, 2013 Edition, 2013. [Online]. Available: http://www.itrs2.net/itrs-reports.html

[20] J. Henkel, L. Bauer, N. Dutt, P. Gupta, S. Nassif, M. Shafique, M. Tahoori, and N. Wehn, "Reliable on-chip systems in the nano-era: Lessons learnt and future trends," in *50th ACM/EDAC/IEEE Design Automation Conference (DAC)*, May 2013, pp. 1–10.

[21] A. Rahimi, L. Benini, and R. K. Gupta, "Variability mitigation in nanometer cmos integrated systems: A survey of techniques from circuits to software," *Proceedings of the IEEE*, vol. 104, no. 7, pp. 1410 – 1448, July 2016.

[22] S. Borkar, "Designing reliable systems from unreliable components: The challenges of transistor variability and degradation," *IEEE Micro*, vol. 25, no. 6, pp. 10–16, 2005.

[23] A. Avizienis, "Fundamental concepts of dependability," *Computers and Operations Research*, pp. 1–20, 2012.

[24] K. Chakraborty and P. Mazuder, *Fault Tolerance and Reliability Techniques for High-Density Random-Access Memories*. Prentice Hall, 2002.

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

[26] 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.

[27] F. E. Grubbs, "Procedures for detecting outlying observations in samples," American Statistical Association and American Society for Quality, Tech. Rep. 1, 1969. [Online]. Available: http://www.jstor.org/stable/1266761

[28] M. Usman, V. Muthukkumarasamy, and X. W. Wu, "Mobile agent-based cross-layer anomaly detection in smart home sensor networks using fuzzy logic," *IEEE Transactions on Consumer Electronics*, vol. 61, no. 2, pp. 197 – 205, May 2015.

[29] A. DeOrio, Q. Li, M. Burgess, and V. Bertacco, "Machine learning-based anomaly detection for post-silicon bug diagnosis," in *Design, Automation Test in Europe Conference Exhibition (DATE)*, March 2013, pp. 491–496.

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

[31] 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.

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

[33] G. Galvas, "Time series forecasting used for real-time anomaly detection on websites," Master's thesis, Faculty of Science, Vrije Universiteit, 10 2016.

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

[35] 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.

[36] R. S. Hammer, D. T. McBride, and V. B. Mendiratta, "Comparing reliability and security: Concepts, requirements and techniques," *Bell Labs Technical Journal*, vol. 12, no. 3, pp. 65 – 78, Fall 2007.

[37] M. F. B. Abbas, S. P. Kadiyala, A. Prakash, T. Srikanthan, and Y. L. Aung, "Hardware performance counters based runtime anomaly detection using SVM," in *TRON Symposium (TRONSHOW)*, Dec 2017, pp. 1–9.

[38] K. Ott and R. Mahapatra, "Hardware performance counters for embedded software anomaly detection," in *EEE 16th International Conference on Dependable, Autonomic and Secure Computing, 16th International Conference on Pervasive Intelligence and Computing, 4th International Conference on Big Data Intelligence and Computing and Cyber Science and Technology Congress*, Aug 2018, pp. 528–535.

[39] M. Stanisavljević, A. Schmid, and Y. Leblebici, *Reliability of Nanoscale Circuits and Systems – Methodologies and Circuit Architectures*. Springer, 2011.

[40] J. Lienig and H. Bruemmer, *Fundamentals of Electronic Systems Design*, 1st ed. Springer Publishing Company, Incorporated, 2017.

[41] A. Avizienis, "The four-universe information system model for the study of fault tolerance," in *12th International Symposium on Fault-Tolerant Computing (FTCS-12)*, 1982, pp. 6–13.

[42] M. Zwoliński, *Digital System Design with SystemVerilog*, ser. Prentice Hall modern semiconductor design series. Addison-Wesley, 2010.

[43] G. K. Saha, "Software fault tolerance through run-time fault detection," *Ubiquity*, vol. 2005, no. December, pp. 2–2, Dec 2005.

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

[45] J. H. Barton, E. W. Czeck, Z. Z. Segall, and D. P. Siewiorek, "Fault injection experiments using FIAT," *IEEE Transactions on Computers*, vol. 39, no. 4, pp. 575–582, Apr 1990.

[46] F. Cristian, "Understanding fault-tolerant distributed systems," *Commun. ACM*, vol. 34, no. 2, pp. 56–78, Feb. 1991.

[47] D. P. Siewiorek, J. J. Hudak, B. H. Suh, and Z. Segal, "Development of a benchmark to measure system robustness," in *23rd International Symposium on Fault-Tolerant Computing (FTCS-23)*, June 1993, pp. 88–97.

[48] P. Koopman and J. DeVale, "The exception handling effectiveness of POSIX operating systems," *IEEE Transactions on Software Engineering*, vol. 26, no. 9, pp. 837–848, Sep 2000.

[49] M. Kaliorakis, S. Tselonis, A. Chatzidimitriou, N. Foutris, and D. Gizopoulous, "Differential fault injection on microarchitectural simulators," in *IEEE International Symposium on Workload Charaterization (IISWC)*, Oct 2015, pp. 172 – 182.

[50] D. J. Sorin, "Fault tolerant computer architecture in synthesis lectures on computer architecture," 2009. [Online]. Available: http://www.cslab.ntua.gr/~nkoziris/presentations/ACACES2010/Fault%20Tolerant%20Computer%20Architecture.pdf

[51] A. Kamran and Z. Navabi, "Hardware acceleration of online error detection in many-core processors," *Canadian Journal of Electrical and Computer Engineering*, vol. 38, no. 2, pp. 143 – 153, Spring 2015.

[52] M. Kaliorakis, M. Psarakis, N. Foutris, and D. Gizopoulous, "Accelerated online error detection in many-core microprocessor architectures," in *IEEE 32nd VLSI Test Symposium (VTS)*, April 2014, pp. 1 – 6.

[53] D. Gizopoulos, M. Psarakis, S. V. Adve, P. Ramachandran, S. K. S. Hari, D. Sorin, A. Meixner, A. Biswas, and X. Vera, "Architectures for online error detection and recovery in multicore processors," in *Design, Automation and Test in Europe (DATE)*. IEEE, 2011, pp. 1–6.

[54] M. Goldstein and S. Uchida, "A comparative evaluation of unsupervised anomaly detection algorithms for multivariate data," *PLOS One*, pp. 1–31, April 2016.

[55] Y. Kawachi, Y. Koizumi, and N. Harada, "Complementary set variational autoencoder for supervised anomaly detection," in *IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, April 2018, pp. 2366–2370.

[56] H. Song, Z. Jiang, A. Men, and B. Yang, "A hybrid semi-supervised anomaly detection model for high-dimensional data," *Computational Intelligence and Neuroscience*, vol. 2017, pp. 1–9, 2017.

[57] 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.

[58] M. Toledano, I. Cohen, Y. Ben-Simhon, and I. Tadeski, "Real-time anomaly detection system for time series at scale," in *Workshop on Anomaly Detection in Finance*, ser. Proceedings of Machine Learning Research, vol. 71. PMLR, 14 Aug 2018, pp. 56–65.

[59] M. S. Islam, W. Khreich, and A. Hamou-Lhadj, "Anomaly detection techniques based on kappa-pruned ensembles," *IEEE Transactions on Reliability*, vol. 67, no. 1, pp. 212–229, March 2018.

[60] V. Vercruyssen, W. Meert, G. Verbruggen, K. Maes, R. Bäumer, and J. Davis, "Semi-supervised anomaly detection with an application to water analytics," in *IEEE International Conference on Data Mining (ICDM)*, Nov 2018, pp. 527–536.

[61] J. Dromard, G. Roudiére, and P. Owezarski, "Online and scalable unsupervised network anomaly detection method," *IEEE Transactions on Network and Service Management*, vol. 14, no. 1, pp. 34–47, March 2017.

[62] Y. Sasaka, T. Ogawa, and M. Haseyama, "A novel framework for estimating viewer interest by unsupervised multimodal anomaly detection," *IEEE Access*, vol. 6, pp. 8340–8350, 2018.

[63] N. H. Duong and H. D. Hai, "A semi-supervised model for network traffic anomaly detection," in *17th International Conference on Advanced Communication Technology (ICACT)*, July 2015, pp. 70–75.

[64] L. Song, H. Liang, and T. Zheng, "Real-time anomaly detection method for space imager streaming data based on HTM algorithm," in *IEEE 19th International Symposium on High Assurance Systems Engineering (HASE)*, Jan 2019, pp. 33–38.

[65] K. Worden, G. Manson, and N. R. J. Fieller, "Damage detection using outlier analysis," *Journal of Sound and Vibration*, vol. 229, no. 3, pp. 647–667, January 2000.

[66] D. G. Pascual, *Artificial Intelligence Tools Decision Support Systems in Condition Monitoring and Diagnosis*, 1st ed. CRC Press, 2015.

[67] M. D. Anis, "Towards remaining useful life prediction in rotating machine fault prognosis: An exponential degradation model," *2018 Condition Monitoring and Diagnosis (CMD)*, pp. 1–6, 2018.

[68] C. Oriol, J. Clapes, A. Elyamani, J. Lana, C. Seguí, A. Martín, and P. Roca, "Damage detection using principal component analysis applied to temporal variation of natural frequencies," in *16th European Conference on Earthquake Engineering*, 06 2018.

[69] A. Alizadeh, "Fatigue crack monitoring of helicopter fuselage through sensor network," Master's thesis, Faculty of Industrial Engineering, Politecnico di Milano, 2014.

[70] M. Wei, B. Qiu, Y. Jiang, and X. He, "Multi-sensor monitoring based on-line diesel engine anomaly detection with baseline deviation," in *Prognostics and System Health Management Conference (PHM-Chengdu)*, Oct 2016, pp. 1–5.

[71] Y. Chen, B. Wang, W. Liu, and D. Liu, "On-line and non-invasive anomaly detection system for unmanned aerial vehicle," in *Prognostics and System Health Management Conference (PHM-Harbin)*, July 2017, pp. 1–7.

[72] D. Liu, J. Pang, G. Song, W. Xie, Y. Peng, and X. Peng, "Fragment anomaly detection with prediction and statistical analysis for satellite telemetry," *IEEE Access*, vol. 5, pp. 19 269–19 281, 2017.

[73] M.-L. Li, P. Ramachandran, S. K. Sahoo, S. V. Adve, V. S. Adve, and Y. Zhou, "SWAT: An error resilient system," World Wide Web, 2008. [Online]. Available: rsim.cs.uiuc.edu/Pubs/08SELSE-Li.pdf

[74] S. K. S. Hari, M.-L. Li, P. Ramachandran, B. Choi, and S. V. Adve, "mSWAT: Low-cost hardware fault detection and diagnosis for multicore systems," in *42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Dec 2009, pp. 122 – 132.

[75] N. K. A. Paschalis, D. Gizopoulos, and G. Xenoulis, "Software-based self-testing of embedded processors," *IEEE Transactions on Computers*, vol. 54, no. 4, pp. 461 – 475, April 2005.

[76] H. Al-Asaad and M. Shringi, "On-line built-in self-test for operational faults," in *IEEE Autotestcon Proceedings. IEEE Systems Readiness Technology Conference. Future Sustainment for Military Aerospace*, Sep. 2000, pp. 168–174.

[77] L. Chen and S. Dey, "Software-based self-testing methodology for processor cores," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 20, no. 3, pp. 369–380, March 2001.

[78] T.-F. Hsieh, J.-F. Li, K.-T. Wu, J.-S. Lai, C.-Y. Lo, D.-M. Kwai, and Y.-F. Chou, "Software-hardware-cooperated built-in self-test scheme for channel-based DRAMs," in *International Test Conference in Asia (ITC-Asia)*, Sep. 2017, pp. 107–111.

[79] M. Kaliorakis, N. Foutris, D. Gizopoulous, M. Psarakis, and A. Paschalis, "Online error detection in multiprocessor chips: A test scheduling study," in *IEEE 19th International On-Line Testing Symposium (IOLTS)*, July 2013, pp. 169 – 172.

[80] P. Bernardi, R. Cantoro, S. D. Luca, E. Sánchez, and A. Sansonetti, "Development flow for on-line core self-test of automotive microcontrollers," *IEEE Transactions on Computers*, vol. 65, no. 3, pp. 744–754, March 2016.

[81] U. Schiffel, "Hardware error detection using AN-codes," Ph.D. dissertation, Dresden University of Technology, 2011.

[82] K. Nørvåg, "An introduction to fault tolerant systems," Norwegian University of Science and Technology, 7034, Trondheim, Norway, IDI Technical Report 6/99, 2000. [Online]. Available: http://www.idi.ntnu.no/~noervaag/IDI-TR-6-99/IDI-TR-6-99.pdf

[83] V. Izosimov, P. Pop, P. Eles, and Z. Peng, "Synthesis of fault-tolerant schedules with transparency/performance trade-offs for distributed embedded systems," in *Design, Automation and Test in Europe Conference (DATE)*, vol. 1, March 2006, pp. 1–6.

[84] N. Oh, P. P. Shirvani, and E. J. McCluskey, "Error detection by duplication instructions in super-scalar processors," *IEEE Transactions on Reliability*, vol. 51, no. 1, pp. 63 – 75, Mar 2002.

[85] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. I. August, "SWIFT: Software implemented fault tolerance," in *International Symposium on Code Generation and Optimization*, March 2005, pp. 243 – 254.

[86] T. A. Alves, S. Kundu, L. A. J. Marzulo, and F. M. G. Franca, "Online error detection and recovery in dataflow execution," in *20th IEEE International On-Line Testing Symposium (IOLTS)*, July 2014, pp. 9–104.

[87] V. Thati, J. Vankeirsbilck, N. Penneman, D. Pissoort, and J. Boydens, "An improved data error detection technique for dependable embedded software," in *IEEE 23rd Pacific Rim International Symposium on Dependable Computing (PRDC)*, 12 2018, pp. 213–220.

[88] Y. Nezzari and C. Bridges, "Compiler extensions towards reliable multicore processors," in *IEEE Aerospace Conference*, June 2017.

[89] S. Hukerikar, K. Teranishi, P. C. Diniz, and R. F. Lucas, "An evaluation of lazy fault detection based on adaptive redundant multithreading," in *IEEE High Performance Extreme Computing Conference (HPEC)*, Sept 2014, pp. 1–6.

[90] K.-H. Chen, J.-J. Chen, F. Kribel, S. Rehman, M. Shafique, and J. Henkel, "Task mapping for redundant multithreading in multi-cores with reliability and performance heterogeneity," *IEEE Transactions on Computers*, vol. 65, no. 11, pp. 3441–3455, November 2016.

[91] J. Soman, N. Miralaei, A. Mycroft, and T. M. Jones, "REPAIR: Hard-error recovery via re-execution," in *IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFTS)*, Oct 2015, pp. 76–79.

[92] J. Soman, "A performance-efficient and practical processor error recovery framework," University of Cambridge, 15 JJ, Thomson Avenue, Cambridge CB3 0FD, UK, UCAM Technical Report 931, 2019. [Online]. Available: https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-931.pdf

[93] A. Meixner and D. J. Sorin, "Error detection using dynamic dataflow verification," in *16th International Conference on Parallel Architecture and Compilation Techniques (PACT)*, Sept 2007, pp. 104–118.

[94] C. A. Lisboa, C. Grando, A. de Freitas Moreira, and L. Carro, "Using software invariants for dynamic detection of transient errors," in *10th Latin American Test Workshop*, March 2009, pp. 1–6.

[95] A. Meixner, M. E. Bauer, and D. J. Sorin, "Argus: Low-cost, comprehensive error detection in simple cores," *IEEE Micro*, vol. 28, no. 1, pp. 52 – 59, Jan 2008.

[96] A. Carelli, A. Vallero, and S. D. Carlo, "Performance monitor counters: Interplay between safety and security in complex cyber-physical systems," *IEEE Transactions on Device and Materials Reliability*, vol. 19, no. 1, pp. 73–83, March 2019.

[97] L. Uhsadel, A. Georges, and I. Verbauwhede, "Exploiting hardware performance counters," in *5th Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, vol. 00, 08 2008, pp. 59–67.

[98] W. Mathur and J. Cook, "Toward accurate performance evaluation using hardware counters," in *ITEA Modeling and Simulation Workshop*, Dec 2003.

[99] C. Malone, M. Zahran, and R. Karri, "Are hardware performance counters a cost effective way for integrity checking of programs," in *6th ACM workshop on Scalable Trusted Computing*. ACM, 2011, pp. 71–76.

[100] H. Sayadi, H. M. Makrani, S. M. P. Dinakarrao, T. Mohsenin, A. Sasan, S. Rafatirad, and H. Homayoun, "2SMaRT: A two-stage machine learning-based approach for run-time specialized hardware-assisted malware detection," in *Design, Automation Test in Europe Conference Exhibition (DATE)*, March 2019, pp. 728–733.

[101] M. B. Bahador, M. Abadi, and A. Tajoddin, "Hpcmalhunter: Behavioral malware detection using hardware performance counters and singular value decomposition," in *4th International Conference on Computer and Knowledge Engineering (ICCKE)*, Oct 2014, pp. 703–708.

[102] M. Chiappetta, E. Savas, and C. Yilmaz, "Real time detection of cache-based side-channel attacks using hardware performance counters," *Appl. Soft Comput.*, vol. 49, no. C, pp. 1162–1174, Dec 2016.

[103] S. Behling, R. Bell, P. Farrell, H. Holthoff, F. O'Connell, and W. Weir, *The POWER4 Processor Introduction and Tuning Guide*, IBM.

[104] S. Microsystems, *UltraSparc^{TM} User Manual*, SUN Microsystems.

[105] J. C. Foreman, "A survey of cyber security countermeasures using hardware performance counters," *CoRR*, vol. abs/1807.10868, 2018.

[106] S. Wang, W. Zhang, T. Wang, C. Ye, and T. Huang, "VMon: Monitoring and quantifying virtual machine interference via hardware performance counter," in *IEEE 39th Annual Computer Software and Applications Conference*, vol. 2, July 2015, pp. 399–408.

[107] S. Rasoolzadeh, M. Saedpanah, and M. R. Hashemi, "Estimating application workload using hardware performance counters in real-time video encoding," in *7th International Symposium on Telecommunications (IST)*, Sept 2014, pp. 307–311.

[108] J. Demme, M. Maycock, J. Schmitz, A. Tang, A. Waksman, S. Sethumadhavan, and S. Stolfo, "On the feasibility of online malware detection with performance counters," *SIGARCH Comput. Archit. News*, vol. 41, no. 3, pp. 559–570, June 2013.

[109] 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 *29th International Conference on VLSI Design and 15th International Conference on Embedded Systems (VLSID)*, Jan 2016, pp. 587–588.

[110] X. Wang, C. Konstantinou, M. Maniatakos, R. Karri, S. Lee, P. Robison, P. Stergiou, and S. Kim, "Malicious firmware detection with hardware performance counters," *IEEE Transactions on Multi-Scale Computing Systems*, vol. 2, no. 3, pp. 160–173, July 2016.

[111] S. A. Musavi and M. R. Hashemi, "HPCgnature: A hardware-based application-level intrusion detection system," *IET Information Security*, vol. 13, no. 1, pp. 19–26, 2019.

[112] X. Wang and R. Karri, "Reusing hardware performance counters to detect and identify kernel control-flow modifying rootkits," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 35, no. 3, pp. 485–498, March 2016.

[113] K. Appiah, X. Zhai, S. Ehsan, W. M. Cheung, H. Hu, D. Gu, K. McDonald-Maier, and G. Howells, "Program counter as an integrated circuit metrics for secured program identification," in *4th International Conference on Emerging Security Technologies (EST)*, Sept 2013, pp. 98–101.

[114] X. Zhai, K. Appiah, S. Ehsan, H. Hu, D. Gu, K. McDonald-Maier, W. M. Cheung, and G. Howells, "Application of ICmetrics for embedded system security," in *4th International Conference on Emerging Security Technologies (EST)*, Sept 2013, pp. 89–92.

[115] Z. Wang and A. Chattopadhyay, *High-level Estimation and Exploration of Reliability for Multi-Processor System-on-Chip*. Springer, 2017.

[116] 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 *IEEE International Workshop, Proceedings of the Workload Characterization (WWC-4)*, ser. WWC '01. Washington, DC, USA: IEEE Computer Society, 2001, pp. 3–14.

[117] A. Akram and L. Sawalha, "A comparison of x86 computer architecture simulators," Western Michigan University, Computer Architecture and Systems Research Laboratory Report, 11 2016. [Online]. Available: http://scholarworks.wmich.edu/casrl_reports/1

[118] ——, "A survey of computer architecture simulation techniques and tools," *IEEE Access*, pp. 1–1, 2019.

[119] 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.

[120] R. Ubal, B. Jang, P. Mistry, D. Schaa, and D. Kaeli, "Multi2sim: A simulation framework for cpu-gpu computing," in *21st International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '12. New York, NY, USA: ACM, 2012, pp. 335–344.

[121] M. T. Yourst, "PTLsim: A cycle accurate full system x86-64 microarchitectural simulator," in *IEEE International Symposium on Performance Analysis of Systems Software*, April 2007, pp. 23–34.

[122] A. Patel, F. Afram, S. Chen, and K. Ghose, "MARSSx86: A full system simulator for x86-CPUs," in *Design and Automation Conference (DAC)*, June 2011.

[123] 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 *44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, June 2014, pp. 622–629.

[124] O. Wirth, J. Slaven, and M. A. Taylor, "Interval sampling methods and measurement error: A computer simulation," *Journal of Applied Behavior Analysis*, vol. 47, no. 1, pp. 83–100, 2014.

[125] A. Fog, "4. Instruction Tables," Technical University of Denmark, Software Optimization Resources, 2018. [Online]. Available: https://www.agner.org/optimize/instruction_tables.pdf

[126] J. Liedtke, "On μ-Kernel construction," in *15th ACM Symposium on Operating System Principles (SOSP)*, December 1995.

[127] H. Härtig, M. Hohmuth, J. Liedtke, S. Schönberg, and J. Wolter, "The performance of μ-kernel-based systems," in *17th ACM Symposium on Operating System Principles (SOSP)*, October 1997.

[128] R. Iyer, Z. Kalbarczyk, and W. Gu, *Benchmarking the Operating System against Faults Impacting Operating System Functions.* John Wiley & Sons, Inc., 2008, pp. 311–339. [Online]. Available: http://dx.doi.org/10.1002/9780470370506.ch15

[129] R. Leveugle, A. Calvez, P. Maistri, and P. Vanhauwaert, "Statistical fault injection: Quantified error and confidence," in *Design, Automation Test in Europe (DATE) Conference Exhibition*, April 2009, pp. 502–506.

[130] P. J. Brockwell and R. A. Davis, *Introduction to Time Series and Forecasting*, ser. Springer Texts in Statistics. Springer International Publishing, 2016. [Online]. Available: https://books.google.co.uk/books?id=P3fhDAAAQBAJ

[131] D. C. Montgomery, C. L. Jennings, and M. Kulahci, *Introduction to Time Series Analysis and Forecasting*, ser. Wiley Series in Probability and Statistics. Wiley, 2011. [Online]. Available: https://books.google.co.uk/books?id=-qaFi0oOPAYC

[132] R. Adhikari and R. K. Agrawal, "An introductory study on time series modeling and forecasting," *CoRR*, vol. abs/1302.6613, 2013.

[133] R. J. Hyndman, A. B. Koehler, R. D. Snyder, and S. Grose, "A state space famework for automatic forecasting using exponential smoothing methods," *International Journal of Forecasting*, vol. 18, no. 3, pp. 439–454, 2002.

[134] S. G. Makridakis, S. C. Wheelwright, and R. J. Hyndman, *Forecasting: Methods and Applications*, ser. Wiley Series in Management. Wiley, 1998.

[135] G. Shmueli and K. C. L. Jr., *Practical Time Series Forecasting with R - A Hands-On Guide*. Axelrod Schnall Publishers, 2018.

[136] S. Makridakis, E. Spiliotis, and V. Assimakopoulos, "Statistical and machine learning forecasting methods: Concerns and ways forward," PLoS ONE, PLoS ONE 13(3): e0194889, 2018. [Online]. Available: https://journals.plos.org/plosone/article/file?id=10.1371/journal.pone.0194889&type=printable

[137] F. Petropoulos, S. Makridakis, V. Assimakopoulos, and K. Nikolopoulos, "'Horses for Courses' in demand forecasting," *European Journal of Operational Research*, vol. 237, pp. 152–163, 2014.

[138] R. J. Hyndman and G. Athanasopoulos, *Forecasting: Principles and Practice 2nd Edition*. OTexts: Melbourne, Australia, 2019. [Online]. Available: https://otexts.com/fpp2

[139] NIST, "Nist/sematech e-handbook of statistical methods," 2013.

[140] J. Zhao, L. Xu, and L. Liu, "Equipment fault forecasting based on ARMA model," in *International Conference on Mechatronics and Automation*, Aug 2007, pp. 3514–3518.

[141] H. Akaike, "A new look at the statistical model identification," *IEEE Transactions on Automatic Control*, vol. 19, no. 6, pp. 716–723, December 1974.

[142] M. E. Lehr, "Maximum likelihood estimates of non-gaussian arma models," University of California Riverside, Tech. Rep., 1996.

[143] 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.

[144] T. D. Sanger, "Optimal unsupervised learning in feedforward neural networks," Master's thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, 1 1989.

[145] G. Serpen and Z. Gao, "Complexity analysis of multilayer perceptron neural network embedded into a wireless sensor network," *Procedia Computer Science*, vol. 36, no. 192-197, pp. 493–500, 2014.

[146] Z. Tang and P. A. Fishwick, "Feedforward neural nets as models for time series forecasting," *ORSA Journal on Computing*, vol. 5, no. 4, pp. 374–385, 1993.

[147] K. Krishnamoorthy, *Handbook of Statistical Distributions with Applications*, ser. Statistics: A Series of Textbooks and Monographs. CRC Press, 2006. [Online]. Available: https://books.google.co.uk/books?id=FEE8D1tRl30C

[148] N. S. Pillai and X.-L. Meng, "An unexpected encounter with Cauchy and Lèvy," *Ann. Statist.*, vol. 44, no. 5, pp. 2089–2097, 10 2016.

[149] C. Chatfield, *Prediction Intervals for Time-Series Forecasting.* Boston, MA: Springer US, 2001, pp. 475–494. [Online]. Available: https://doi.org/10.1007/978-0-306-47630-3_21

[150] T. P. S. University, "Stat 501: Regression methods," 2018. [Online]. Available: https://newonlinecourses.science.psu.edu/stat501/node/274/

[151] "t-distribution table." [Online]. Available: http://www.sjsu.edu/faculty/gerstman/StatPrimer/t-table.pdf

[152] E. Chavis, H. Davis, Y. Hou, M. Hicks, S. F. Yitbarek, T. Austin, and V. Bertacco, "SNIFFER: A high-accuracy malware detector for enterprise-based systems," in *IEEE 2nd International Verification and Security Workshop (IVSW)*, July 2017, pp. 70–75.

[153] C. Cullmann, C. Ferdinand, G. Gebhard, D. Grund, C. M. (Burguiére), J. Reineke, B. Triquet, and R. Wilhelm, "Predictability considerations in the design of multi-core embedded systems," in *Embedded Real Time Software and Systems Conference*, May 2010, pp. 36–42.