7-5-2016

High Performance Embedded Systems

Ahmed Abdullah Alsheikhy
University of Connecticut - Storrs, ahmed.alsheikhy@uconn.edu

High Performance Embedded Systems

Ahmed A. Alsheikhy, PhD

University of Connecticut, 2016

Embedded systems exist almost everywhere since 95% of the current market chips contains embedded devices; they can be seen as the brain of such systems. They control everything in their platforms such as access, store and processing of data. They are typically dedicated to perform specific tasks. They range from portable devices such as smart phones and MP3 players to a very complex one such as systems controlling air planes or automotive. Their complexity varies from a single system, which includes a single processing unit, to very complicated one with multiple units. They are designed to be used in many applications in our daily life such as educational, industrial and medical. Due to increases in the complexity of those systems with their tightened constraints on time and power dissipation as well as the scope of environment where they operate comes the need to estimate their performance metrics which include delay, for both processing and communication, and power consumption. Thereby achieving high quality of performance estimations is crucial and critical as well. Designing performance model and evaluation approaches to find system performance metrics is considered essential at an early stage of implementation for an efficient design especially real-time systems. Constructing performance models and evaluation techniques of a given system requires a significant effort. Therefore, it becomes crucial to develop a framework that is able to estimate response time and power consumption in the early stage of

design and implementation to avoid unexpected things such as increasing in the project costs, reducing in the productivity and delaying in the schedule.

In this work, we developed a framework to be able to perform analytical analysis to estimate performance metrics "*response time and power consumption*" for any embedded system during design phase. In this research we refer to the response time "delay" as the combination of the computation delay of software processes and communication as one of the different software architectures and hardware platforms. In order to achieve this, Hierarchical Performance Modeling (HPM) as a technique is used to find the expected average system performance for different layers of abstraction. HPM has been proven to be a powerful tool in terms of estimating delay or power consumption since it involves four layers of abstraction which can be summarized as follows: 1. System Level, 2. Task Level, 3. Module Level and 4. Operation Level. We are proposing a *Hierarchical Generic Finite State Machine (HGFSM)* which is used to link (map) between functional modeling analysis approaches such as FSM, Petri-Net and UML with the HPM. We also investigated the performance metrics (in terms of response time and power consumption) for an Android platform. Several hardware platforms are used to estimate the expected average value of both metrics and show the difference between it and the average actual values. The designing framework can be used to determine the bottleneck(s) in a system under investigation as we used it in the Android platforms. The output from the framework is performance equations which can be seen as Objective Functions. Then, we minimized the response time in Android platforms using

a parallelization approach with GPUs invocation and monitoring the consequences in code size and power consumption. In addition, we developed a method to minimize the response time in embedded systems during run-time phase by scheduling their aperiodic tasks in an appropriate way to reduce their average waiting and turn-around times while maintaining system stability. Furthermore, we developed a scheme to 1) improve response time if possible and 2) ensure that all tasks (processes or jobs) meet their deadlines for periodic tasks in real-time system using Worst-Case Execution Time "WCET" as a factor to decide which task or a set of tasks must be chosen first among several processes or sets exist in a system under investigation. Moreover, we used different probability distributions (pdf) to schedule periodic tasks in real-time systems. In many real-time applications such as multimedia, both audio and data processing and transmission offer a great variation. So using WCET as a factor may lead to undesirable results. Using different probability distributions (pdfs) to estimate the remaining time dynamically is called the moving average remaining time since the computed remaining time changes as a used distribution changes too whenever a task is added or removed from it. The purpose from previous approach is to ensure that all tasks meet their deadlines while maintaining system stability. Lastly, we utilize the designing framework to estimate response time in fire and pollution detection systems.

High Performance Embedded Systems

Ahmed A. Alsheikhy

B.Sc., King Abdulaziz University, **2004**

M.Sc., King Abdulaziz University, **2010**

A Dissertation

Submitted in Partial Fulfillment of the

Requirements for the Degree of

Doctor of Philosophy

at the

University of Connecticut

2016

APPROVAL PAGE

Doctor of Philosophy Dissertation

High Performance Embedded Systems

Presented by

Ahmed A. Alsheikhy, B.Sc., M.Sc.

Major Advisor _____

Reda A. Ammar

Associate Advisor _____

Sanguthevar Rajasekaran

Associate Advisor _____

Song Han

University of Connecticut

2016

ACKNOWLEDGEMENTS

All praise and thanks are due to Allah, to whom belongs all that is in the creation. Praise is firstly to Him who knows that which goes into the magnificent earth and that which comes forth from it. This thesis is the product of many years of research in which the completion would be impossible without the help of Allah throughout every stage. Also the assistance from a variety of people inside and outside of the department was a major contribution to my amazing success. I would like to immensely thank my advisor, Professor **Reda. Ammar**, for his professional, academic, and personal guidance. I could not have asked for a better mentor and I really feel very blessed to have him in my life. I admire him truly for being my role model and for helping me keeping a work-life balance with enduring patience. In addition, this project would not be possible without the critical feedback and mentorship from my committee members, Professors **Sanguthevar Rajasekaran and Song Han** and I hope all the best for them. Working alongside such wonderful faculty members has profoundly influenced my academic and personal development, and I hope that my relationship with them increases as my journey as a scholar continues. I extend my deepest thank to Professor **Ali Rushdi**, Professor **Mosleh Alharthi** and Professor **Raafat Elfouly** for their times and valuables recommendations. Furthermore, my achievement would be impossible without the generous financial support from (**Northern Borders University**). I conclude my acknowledgements by extending my greatest respect and special thanks to my family, wife and children for their endless support. I am eternally indebted to my father and mother. I could not have made it this far without their emotional and physical help. They faithfully stood by me even though they were a thousands of miles away. They are the ones who

encouraged me to strive higher than I could ever imagine. I also would like to relay my gratitude to my friends in Connecticut for their endless support and motivations.

DEDICATION

Seham (my wife)

Who deserves special thanks for sharing all moments

My parents

For their support and motivations. Their inspiration has guided me to achieve my desired goal

My children

They stand by my side in all times and believe in me

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

Overview

1.1 Embedded Systems

An embedded system can be defined as a special-purpose computing system which is integrated into a larger product to perform specific tasks in a selected application domain. This system is designed to execute a few applications and they are not programmable by an intended user. Embedded systems must be reliable and efficient in terms of size, cost, power consumption and delays which refer to response times. Moreover, they must be highly dependable which means any malfunction is not acceptable.

In many embedded systems, the availability and correctness of the computations are relevant with the timeliness of the computed results. That systems with precise timing requirements are called Real-Time embedded systems. Their behaviors in terms of computation times and latencies are functional system requirements. Real-time systems exist everywhere in our daily life such as aviation, medical, communications and even entertainment industries. Due to heavy demands on embedded systems where a potential error could lead to a catastrophic disaster, the construction of a fault-free dependable system becomes essential. Several requirements for performance analysis are required which can be summarized as:

A. Accuracy: the estimated calculations should be closer to the Worst-Case Estimated Time (WCET), which is the maximum allowed time for a task to be executed; also known as the deadline time, and Best-Case Estimated Time (BCET), referred to

the minimum execution time for a task, which can be considered as the upper and lower bounds of the system.

B. Correctness: performance analysis must produce correct results. In other words, violating the bounds (upper or lower) must be avoided; so there are no reachable states such that the estimated bounds are exceeded.

C. Reusability: performance analysis scheme must be easy to refine an existing system. In particular, it must allow designers to model and analyze any system with different levels of abstraction. So reusing that analysis is valid across different level.

There are several performance metrics for the evaluation of the efficiency of an embedded system and can be stated as following: 1) Latency (Delay), 2) Power Consumption, 3) Cost and 4) Code size. However, the focus of performance analysis methods for Real-Time embedded systems is on the analysis of timing aspects [1,2,3]. In particular, a designer is intended to estimate the Best-Case Execution Time "BCET" and WCET to make sure that the system meets the real-time requirements. The following three approaches exist and are widely used these days for performance analysis on the embedded systems:

I. Simulation Based Methods: a verification process of performance analysis is required to prove that the results lay in an acceptable range (between WCET and BCET). Several tools are available for this kind of analysis such as *Cycle Accurate Hardware-Software Co-design and SystemC* platform which is commonly used for system-level modeling. Trace-Based is considered the most used method in the

many simulation approaches where a designer provides traces of input data to derive the simulation of the system under investigation. Ability to simulate large modeling scope is one of the advantages of using Simulation Based Methods [4,5,6].

II. Analytical (Mathematical) Based Methods: it is also known as Modular Analysis which is exhaustive in sense that all possible behaviors are taken into consideration. In general, the analytical methods don't scale with the complexity of the embedded systems whereas the simulation methods are used more often [4,5].

III. Direct Measurements: since approach is very costly based on time and finances. In particular, there is a need to buy special equipment to perform a desired analysis and time is wasting for them to arrive which may lead a project delay in scheduling [4,5,6].

A performance modeling scheme is required to evaluate the delays caused by communication and computation by distributed system architectures and existing software on different platforms [1,2,3]. Many application domains such as air-traffic control, e-commerce and medical systems require performance modeling and evaluation to estimate the delay before releasing them to the public [1]. Engineers rely on performance modeling to predict the expected latency before moving to the final stage of implementing. However, in the absence of a performance evaluation scheme, they must design and implement a system to predict the performance defects or bottleneck. Waiting to spot the performance defects or bottlenecks until the final stage of implementation and integration between different components results in increased project costs, reduced productivity and delays in

schedule [1,4,5,6]; applying performance modeling and evaluation from the first stage of design in any system exhibits better results than those using a “fix-it-later” approach [1].

1.2 Motivation and Related Work

Estimating performance characteristics in the final stage of implementation for any embedded system at an early level of design process is considered one of the most difficult task these days. Many designers face several questions related to system performance such as:

- What is the **CPU Utilization**?
- Where does the **bottleneck** occur?
- What are the **memory demands**?
- Do timing requirements **meet** the design requirements?

According to the designers, answering those questions is very hard and having right predictions is also not an easy task. Computing accurate performance characteristics for embedded systems is a must for several reasons which include:

I. Performance analysis plays a significant role in the design level process. To be more specific, it is required to derive the design space exploration. Different implementations in terms of *partitioning and allocation* are evaluated on the basis of reliable and accurate prediction of system performance.

II. It is crucial in the domain of real-time applications where provable guarantees of that analysis are fundamental elements.

III. The high demands on embedded systems products put a pressure on the designers to maximize the system performance and minimize the prices; so the need for accurate estimating rises up.

Functional Modeling techniques and Analytical Modeling ones are used to estimate the system performance metrics at an early stage if possible. Queueing schemes have been used since the 1970s to model performance of any software systems [1]. An FSM is used to evaluate system level performance [7,8, 9,10,11]; however, that FSM was not applicable to any system since it was designed for a specific system. So designing a hierarchical generic FSM "HGFSM" to be used in evaluating performance for any embedded system is proposed in section 3. In [12], a definition for Software Performance Engineering (SPE) was defined by Smith. She emphasized the importance of quantitative methods which should be used at the start of the software development lifecycle to detect the spot of performance defects. In [7,9,11], the performance evaluation was done at the system level only and that didn't include the task level, module level and operations level. Smith and Williams in [13,14] used synchronization nodes and presented an advanced model to estimate the performance model for distributed software architectures. That model is similar to HPM since it utilizes queueing networks to predict the delays of software architectures and hardware devices [1]. However, it fails to generate performance models based on primitive operations for several hardware platforms [1,15].

Different schemes for performance analysis are found in the literature. However, various schemes can be classified as Heterogeneous in terms of modeling scope, tools support, modeling effort, scalability and accuracy. Analytical Modeling approaches such

as Queueing Models and Hierarchical Performance Model (HPM) are used to compute different performance metrics such as Latency delays and power consumption in the embedded systems. Functional Modeling approaches such as Finite State Machine (FSM), Unified Modeling Language (UML) and Petri-Nets are used to represent different types of embedded systems in order to understand their behaviors and reactions for their environments. Applying HPM directly in any embedded system to estimate a desired performance metric is inapplicable since there is a gap between functional modeling methods with it, so developing a framework in order to use HPM as a tool rises. A designing framework as shown in figure 1 can be used in any embedded system since it has the capability to capture all required information about the system to estimate a desired metric.

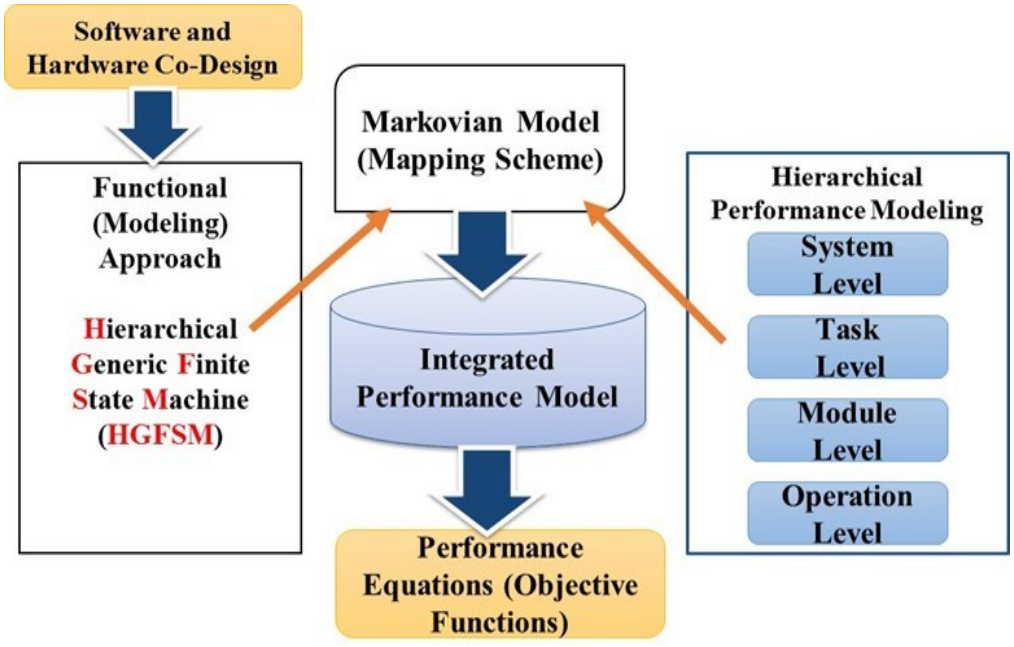


Figure 1: Developed framework

The output of this mapping scheme will be the performance metrics equations for a system under investigation. Figure 2 depicts the interactions between Abstraction Layers in order to display how the constraints propagate through the entire system and how the performance analysis is constructed.

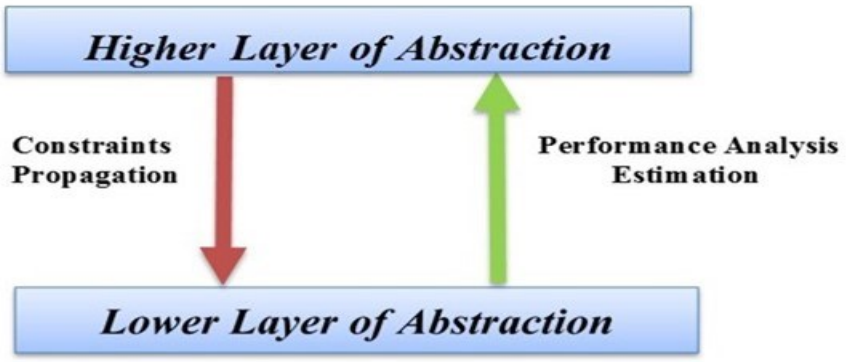


Figure 2: Typical overview of how constraints propagate and performance analysis is performed

1.3 Dissertation Roadmap

The remainder of the thesis is divided into the following chapters. Chapter 2 provides the related work in the area of performance metrics estimation methods. These schemes include functional modeling approaches and analytical modeling approaches. Also the limitations of each method is addressed as well. The details of the designing framework are presented in Chapter 3. This includes the highlighting of the operation mechanism of each component. In addition to that, the chapter details performance results to estimate response time and power consumption for several Android platforms. Furthermore, response time estimation for an embedded OPENWRT is presented in this chapter too. Chapter 4 conducts a complete performance analysis for response time minimization in several Android platforms using available resources inside the system

under consideration. Chapter 5 presents real-time scheduling algorithms for periodic and aperiodic tasks (jobs). The purposes of these methods are to minimize response time if possible and to ensure that all processes meet their deadlines without allowing any deadline miss to occur. Chapter 6 evaluates the presented work using real-time applications of fire and pollution detection systems to estimate their response time. Lastly, Chapter 7 concludes the main contributions of this thesis and suggests some directions for future work.

CHAPTER 2

Literature Survey/Related Works

2.1 Embedded Systems Challenges

Embedded systems have become very important in our lives; they pervade all fields in today's advanced technology. They are found in 95% of the current market in such things as home appliances, manufacturing, automotive and medical applications. Due to heavy demands on them, the use of data measurements and processing in embedded systems increased in the last decade [18]. Many fields, for instance, healthcare, transportation, military and automotive are real examples of where dramatic changes have happened in their products [18]. Data dependencies in many embedded systems influence analytical model solutions [18,19]. Furthermore, uncertainty in the models being used, their components and parameters affect a design methodology and can be treated in several methods such as using a probabilistic approach [18]. Technology and complexity play a significant impact on any design methodology. These two factors are tightly coupled together, as the technology being used goes up the complexity rises too. In modern cars, the increasing number of embedded systems with their sophisticated software escalates the system design complexity as well [19].

Today, around 95% of all innovations are driven and controlled by embedded electronics components and their software [19]. Modern cars include around 70 *Electronic Control Units* (ECU) which are connected by 5 system buses [19]. 2500 signals are transmitted and exchanged between these components which increases the cost in terms of development point of view [19]. Dealing with quantitative system constraints and concurrency can be

done using analytical model-based approaches [20]. However, these methods have difficulties with full or partial specifications as well as with computational complexity [20,21].

Analytical model-based schemes are used in hardware design, control theory, scheduling and performance evaluation. On the other hand, computational based-model methods deal with nondeterministic abstraction levels and a more rich theory of complexity [21]. In many real-time applications, their requirements, such as power consumption and lifetime depend on their environments in which they will be developed [21, 22]. Many real-time embedded systems rely on batteries to operate and perform their functions. In today's technology, many embedded systems are integrated using a lot of small components where their behaviors are totally different [4]. Different *Model of Computations (MOCs)* are used to describe different system behaviors [4]. They represent the states of their systems, a way computation occurs and also the communication takes place inside them. There are several types of MOCs such as:

- Discrete Events (DE).
- Finite State Machine (FSM).
- Synchronous Data Flow (SDF).
- Continuous Time (CT).

In this thesis, we will consider FSM as a model to analyze an embedded system in order to estimate several performance metrics such as response time and power consumption. Performance analysis can be considered one of the key challenges in the system analysis [23]. It influences by several parameters such as response time,

communication delays, throughput and the degree of parallelism in computation aspects [23]. This thesis considers Functional Modeling Approaches, such as *FSM, Unified Modeling Language (UML) and Petri-Net*s, and *Hierarchical Performance Modeling Approach (HPM)* to estimate several performance metrics. Only response time and power consumption are considered within this research. However, estimating different performance metrics can be achieved since the developed framework is capable of constructing different performance equations as needed. The designing framework in figure 1 consists of three components which are: a Functional Modeling Approach which is represented by Hierarchical Generic Finite State Machine (HGFSM), Markovian Model (MM) and Hierarchical Performance Modeling (HPM) Approach which represents analytical scheme. More information about each component will be discussed in the next sections.

2.2 Functional Modeling Approaches

Embedded systems can be seen as the brain of most of the electronic systems [5]; they control every aspect such as access, storage and data control [5]. They can be optimized to reduce the size and even cost since they are dedicated to perform specific tasks [5]. Furthermore, their reliability and performance can be estimated; hence, using a functional modeling method is one of the best available methods to achieve them. Only FSM is considered within this research.

2.2.1 Finite State Machine (FSM)

A typical FSM model is composed of 5-tuples $\{\sum, S, S_0, \delta, F\}$; where: $\sum$ represents a set of input alphabets. S represents a set of states in the model. S_0 represents an initial state or a set of states which are sub-elements of S. δ represents a state-transition function which maps between a current state to a next state, and F contains a final state or a set of states which belongs to S, which is distinguished by a double circles around it as depicted in figure 3.

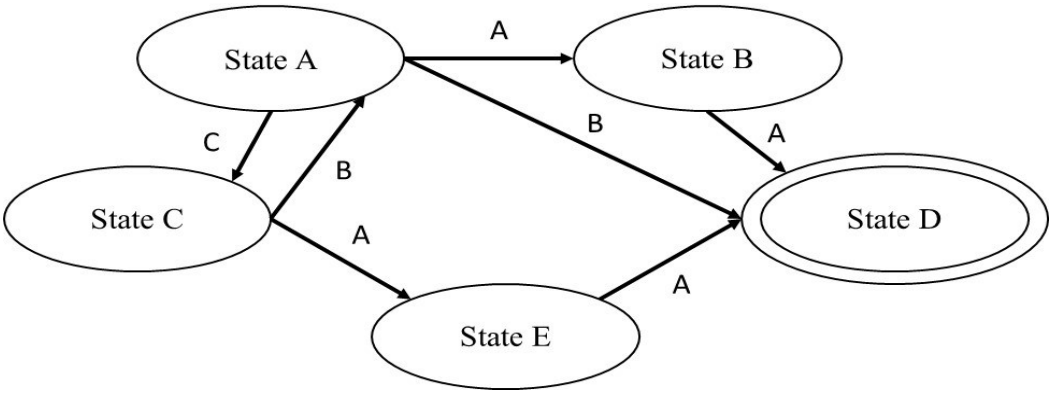


Figure 3: Typical finite state machine

In figure 3, each arrow represents a transition and is associated with an action which causes a movement from the current state to the next one. The current state represents the source and the next state represents the destination. In a hierarchical FSM, a state might be decomposed into another FSM which is called sub-FSM or the slave while the outer FSM is called the super state or the master [7]. That state is called also hierarchical one as shown in figure 4.

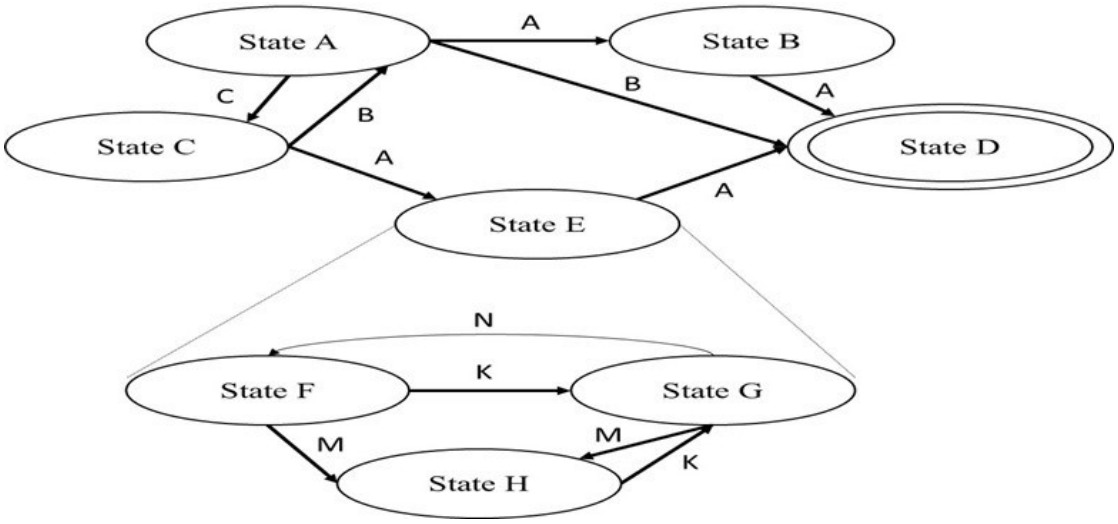


Figure 4: Typical hierarchical finite state machine

In [7], B. Lee and E. A. Lee used a *Hierarchical Concurrent Finite State Machine* "**HCFSM**" to show how an FSM can interact with concurrent models. They focused on three concurrent models which were:

- Synchronous Data Flow (SDF).
- Discrete Event (DE).
- Synchronous Reactive Models (SRM).

Their HCFSM supported heterogeneity which means that a slave state of a hierarchical one need not be another FSM [7]. In other words, the slave state is needed to have a well-defined terminating computation which asserts output events by reacting to input events or triggers [7]. HCFSM can be combined with any concurrency model. They used nested HCFSM, the depth and order of nesting FSM were arbitrary. Their reason behind using nested HCFSM was the ability to describe concurrent models being used or investigated.

Synchronous Data Flow (SDF)

In each SDF, a system includes a set of blocks interconnected by arcs [7]. Each block represents a set of computational functions that map input data with output data when they are fired. The arcs represent stream on data tokens which can be implemented in First-Come-First-Serve (FCFS) basis.

When an FSM describes a block of SDF graph, it should follow SDF semantics, it produces and consumes a fixed number of tokens on every input arc and output arc.

Discrete Events (DE)

DE models represent notations of global time that is known simultaneously throughout a system. Each input occurs at a point in time, it is needed to carry a time stamp which indicates the time at which the event happens. The time stamp is generated by its source block and determined by a latency of its source.

When a block of DE fires a token, its FSM performs one reaction which occurs when an event happens in one of its inputs. The delay or latency occurs from the DE block is considered to be zero.

When a DE represents an FSM, events passed by the inner DE carries the same time stamp provided by the outer DE. Hence, the present time stamps keep consistent through all DE in the FSM even a hierarchal one.

Synchronous/Reactive Models (SRM)

An SRM system represents a set of blocks interconnecting together through directed arcs. Execution of the system is done by a set of discrete instants. Most SRM

systems use strict functions which are always monotonic. However, causality problems are caused by a directed loop in strict functions.

Inside SRM systems, FSMs need to be treated as non-strict functions to get the best of all directed loops in each reaction. For example, if there are two outgoing transitions, labeled as follows: a $\wedge$ b /x and a $\wedge$ -b /x. For a state inside FSMs, a function maps inputs a and b with output x can be seen as $f_x(a,b) = (a \wedge b) \vee (a \wedge -b) = a$. In other words, the output x can be asserted as long as the input "a" is known to be present or absent regarding any knowledge about input b. This method simplifies a need to know which inputs should be known at each state to define output functions.

When an FSM system refines into an SRM, the semantics of SRM are exported to the outer model where the FSM is located. In addition, one slave of that SRM system is considered to be one instant.

The communications of each transition between two FSMs can be defined as micro steps within a macro step.

In [8], A. Stan et al used an FSM for embedded software development. The reasons for them to use that FSM were its flexibility to add, delete or change a flow of a program without impacting the overall code structure. Their FSM was formed from two sets, a set of state arrays which include all states and references to their transition arrays. The other set contains information about all possible transitions from every state. If an event occurs, the state of the FSM associated with that event will be updated which can be done by asserting the state changed flag. Once that event is processed, a function which is responsible for updating states will have existed even if there are other pending events.

They used that feature in complex systems to prioritize transitions based on application requirements. The developed FSM occupied around 250 bytes of memory. The ATMega family of microcontroller was used to evaluate the developed FSM. They targeted an 8-bit embedded environment. They used a sequence detector as a target application, this application is an abstraction for many practical problems that might be solved with help from FSM. The sequence detector is used to detect a sequence of binary values (0, 0, 1} at its input. If that sequence exists, then the output is set to 1, otherwise, it will be 0.

In each state, a table of transitions is defined, each element of that table has two fields which are: 1. A pointer to a function which evaluates a condition of an event and 2. A destination state if that condition is true and has occurred. Every output function has two paths, one for actions that are executed only once if the event is true and the other path for actions that are executed continuously as long as the FSM stays in the same state. They used cycles count taken by a program to be executed as a performance metric to evaluate their model. Furthermore, they compiled the model in four different configurations which can be summarized as follows:

A. FSM information stored in SRAM as data and compiler optimization set to a low level.
B. FSM information stored in SRAM as data and compiler optimization set to a high level.
C. FSM information stored in Flash is set to a low level.
D. FSM information stored in Flash is set to a high level.

A basic script for an IAR simulator was used. Their results showed that the execution time on Flash was slightly larger than one on SRAM due to the fact that reading data from Flash consumed more cycles than SRAM.

In [10], B. Lee and A. E. Lee used HCFSM in the Ptolemy software environment to decouple it from concurrency models. They decoupled HCFSM from SDF and DE concurrent models. Ptolemy is a software environment which was developed to support heterogeneous system design in order to allow diverse models of computation to coexist and interact. **Ptolemy** is constructed by interconnecting blocks which are a *Star and a Galaxy*. The Star is a fundamental block which often contains code segments for a simulation purpose. The Galaxy internally contains Stars and possibly other Galaxies. Each state in their FSM represents a fundamental block which was implemented as a Star. Galaxy was used to represent a diagram of interconnected Stars. A scheduler was used to manage execution of a subsystem within their model. A game called "**Reflex**" was used to test the model. It was a version for two players. The game measures a reflex time of player 2 by *estimating the elapsed time between Go and Stop events*. They used the DE domain to simulate the real-time behavior of that game. Counting number of ticks generated by a clock being used was conducted in several states to measure the elapsed time.

L. Yuan et al in [24] used an FSM re-engineering performance framework method to sequential circuit synthesis by state splitting conception. Their framework starts with the traditional FSM synthesis procedures then proceeds to re-construct a prototype model with different topology according to an optimization objective. It ends with another re-constructed FSM synthesis which allowed them to explore a larger space that consists of a

set of FSMs. The proposed model was developed to minimize power and area for a system under consideration. They developed a heuristic algorithm and a generic one to re-construct their model. In addition, the model was encoded by an encoding algorithm. To show the validation and effectiveness of that model, a benchmark called "*MCNC91*" was used. The benchmark is sequential circuit which is synthesized in an SIS environment. The proposed model gave around 5% reduction for power consumption with 1.3% increasing in area and the same amount in delay. Lastly, an Integer Linear Programming (ILP) was used to achieve an optimal low power state encoding for benchmarks of small size. The optimal low power was found to be around 1% to 8% better than the optimal solutions in the original FSM in terms of power reduction. Sequential circuits, play a significant impact in digital systems, can be modeled by FSMs.

A **standard State Transition Graph (STG)** was used to represent the encoding FSMs with G = {V, E), where a node $v_i \epsilon v$ which represents a state s_i. A directed edge (v_i, v_j) ϵ E represents a transition from a current state s_i to state s_j. The STG then was transformed to a undirected weighted graph $G' = \{V, E, \{C_i\}, \{w_{ij}\}\}$, where C_i refers to a code segment and w_{ij} refers to the weighted graph. Hamming distance between different edges is used to estimate a weighted edge between two states. The weighted edge is defined as the total probability transition between its states; in order to estimate it, input distribution(s) at each state should be present and can be obtained using a simulation.

The proposed algorithm in [24] suffers from high overhead due to the fact that the probability transitions need to be estimated, then the weighted edge is computed too and

finally hamming distance between every two ending states is calculated too. We refer the readers to [24] for more information about the described approach.

N.P. Dash et al in [25] used an FSM model to give a brief overview of event driven programs and their relationship with FSMs. The FSM model was developing in C programming language for Microcontroller Units (MCUs). The proposed FSM based-model with event-driven programming techniques are very useful in handling concurrent events that usually occur.

Events are often generated when a user action is done on a system, that action can be any one of the following form:

- A press on a push button or a key pad.
- Touch a touch-screen.
- Move or a click of a mouse.
- Message packets through a physical interface.
- Timeout or a software exception

The proposed FSM model was used to describe *a power key in a system problem*. Initially, the system is either **On** or **Off**. If the power key is pressed for two seconds, the system switches to the opposite state. Otherwise, it remains in its current state if the pressed was less than two seconds. The proposed FSM model consumed less CPU cycles and memory space. However, it is useful and very suitable if the states and events combinations are handled in tiny embedded systems where resources are very scarce. Otherwise, more memory space will be consumed without doing any activity. It takes around 18 instructions cycles.

2.3 Markov Model Approach

For any given system, a Markov Model (MM) includes all possible states of that system, also all possible transition paths with their rates as depicted in figure 5 [3].

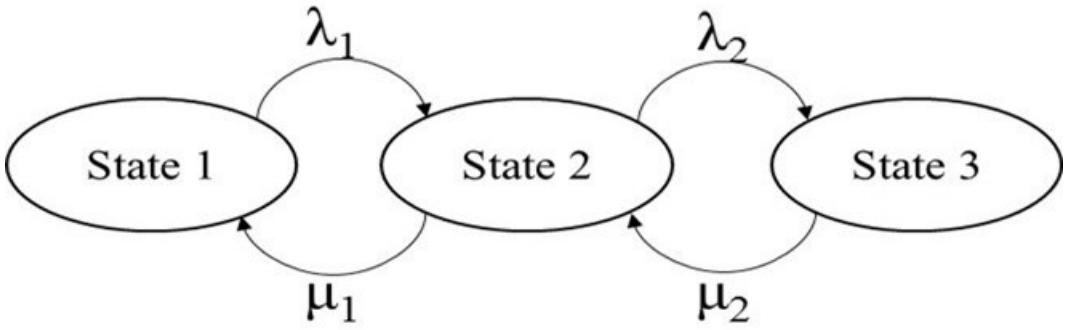


Figure 5: Typical Markov model

λ indicates the arrival rate parameter while **μ** indicates the departure rate parameter of transitions between different states. **Markovian Model** "*MM*" is a stochastic one which is used to describe randomly changing systems where a transition from a current state to the next one depends only on the current state. MM is considered to be a useful tool to represent the core of the most performance analysis models. It can be used to evaluate queueing networks or Petri-nets [6]. Any MM consists of 3-tuples {S, A, P}, where S denotes to all states that represent a system; A refers to initial probability transitions and P refers to probability transitions matrix between all states. Note that $\sum_i^n P_{ij} = 1$, Where i refers to the source and j refers to the destination. MMs can be treated as an FSM, all transitions are annotated with the probability of going from a state to another one. Readers are referred to read [3] for more information about MM.

J, Happe in [6] used a MM to *predict mean service execution times of software components*. These components' performance cannot be considered constant due to the fact that they can be easily deployed in different contexts.

He developed a model to compute the excepted service time execution time using the mathematical features of MM. FSMs consider one of the available methods to describe an infinite set of call functions which are invoked to be used later in the analysis. MM is used to enable computations of *Quality of Service (QoS)* attributes of a required service. It represents the core of the model being used. Several assumptions were taken into consideration which can summarized as follows:

- Any transition from a source state into a destination one depends only on the current "source" state.
- The execution times of different components are independent, which is not true in all cases.
- The execution times are not influenced by external sources such as interruptions or other services which run in the background.
- All execution times are independent of their input parameters.
- The execution times should be given as expected values, which are easy to develop and specify a software architect.
- Works only on single thread systems and doesn't include the influences of concurrency.

Two case studies were performed using a web server developed system to evaluate the assumptions mentioned earlier. However, the proposed scheme works only on single thread

systems where many applications run on multiple threads systems. Furthermore, it didn't include the influences of communication on the computed service time which really has a significant impact on the execution time if added to the analysis procedures.

M. Choy and M. N. Laik in [26] used a Markov chain method to estimate an optimal performance period and bad credit score to help banks decide who deserve to be loaned. Bad credit card scores are a bad sign indicator which tells banks that this is a big risk. This tool is considered to be the most important tool box in the banking industry. It was developed to determine whether a customer will be 30 days past due in the next couple of months or not. The performance period in the banking industry is typically measured using a type of analysis called **Ever Delinquency Curves Analysis** "*EDCA*". It works by analyzing the ever delinquency curves trend and attempts to spot or determine a point of time which shows the rate of increment in the delinquency becomes slow as depicted in figure 6 from [26].

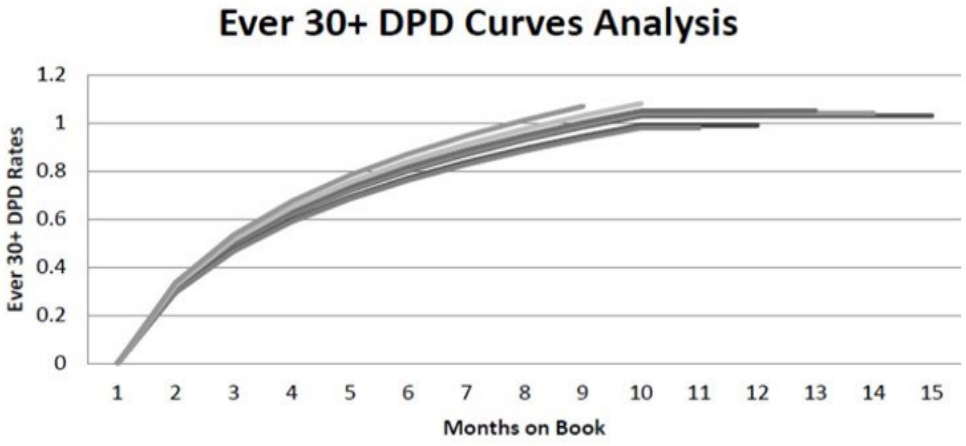


Figure 6: Ever delinquency curves analysis

DPD stands for Day Past Due.

It is a very difficult task to identify that point from fig. 6 since it is not clear from which point the flattening started. Also the EDCA analysis requires to preset the delinquency which will be used in order to proceed. MM was used to filter customers who have never been delinquent in their accounts lifetime. This approach decides which account will go to write off state, hence, the accounts which have delinquent history can be easily targeted and it becomes easy to know when they will reach the point of no return. In order to identify that point, a canonical form is required to transform the matrix form obtaining from MM.

Table 1: Mean transition time matrix

States	Current	X	30	60	90
Current	5.9	7.6	1.2	0.6	0.4
X	6.2	10.0	1.7	0.8	0.6
30	1.4	2.5	1.2	0.6	0.5
60	0.9	1.4	0.8	0.9	0.6
90	0.4	0.7	0.2	0.9	0.9

By just summing any row in table 1 from [26] it becomes easy to determine the point of no return. For example, 60 DPD from the current state “time”, 5.9 + 7.6 + 1.2 + 0.6 = 15.3; which tells that the average performance period to reach 60 DPD is nearly 15 months.

W. Lu in [27] estimated average system performance using Ergodic Markov chain based on long-run of time between two consecutive actions in Lossy Channel. A Reward function was used with Ergodic Theorem, each time a state is visited, a reward value is obtained from the reward function. The value is either 1 for true or 0 for false. An equilibrium distribution method is used in the analytical scheme in order to use the ergodic theorem.

In [9], A. Nandi used MM to estimate system-level power consumption and performance for embedded system design. An SAN method is used in the analysis to model loosely and strongly coupled communicating concurrent processes. The SAN, refers to Stochastic Automate Network, is a very powerful Markovian formalism belonging to a class of processing algebra equations. The model proposed is a process-level functional model which is free of most architectural details. It was used on the MPEG-2 Video Decoder application to evaluate its power consumption and performance based on response time which also can be seen as delay for different input parameters. The model requires to have its steady-state behavior to be known after observing it over an extended period of time. From MM, a probability vector is obtained using several numerical iterative methods such as Gaussian elimination method and Jacobian method. A true rate of an activity is estimated in order to estimate performance metrics such as throughput, utilization and average response time. The true rate is obtained by multiplying the given rate of such activity with its probability.

The performance estimation is achieved using several steps which can be summarized as follows:

I. System Specification: MM was used to represent MPEG2 Decoder. Matlab platform was chosen to do this step since it uses semantics of state carts. Matlab features help to describe the behavior of complex concurrent system components which are characterized by event driven scheme.

II. Application Modeling: A process graph was used to model an application of interest which is MPEG 2 Decoder in this case. In process graph, each

component is associated with a process in the application. The communication delays between different components were estimated by event and wait synchronization signals. The SAN model specifies the embedded system that translates into a network of automata. Each process was assumed to be run in its own space, hence, it did not compete for any computing resource.

Architecture Modeling: starts with an abstract specification of the platform which provides SAN model with the behaviors of that particular specification. Figure 7 from [9] represents an architecture modeling block for different components in the MPEG application

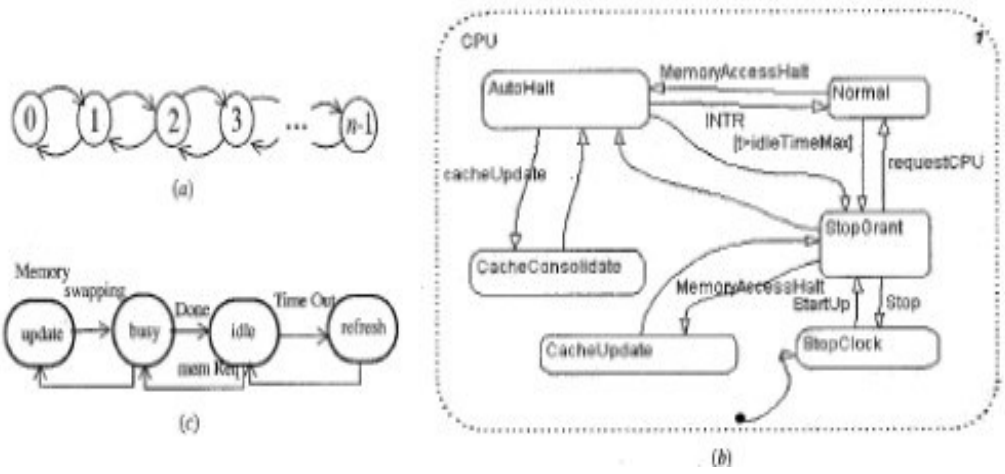


Figure 7: Models of architecture components a) buffer, b) CPU c) memory

In fig. 7 (a), 0 state refers to an empty buffer while n-1 state indicates that the buffer is full. Whenever a request of a new insertion occurs, the current state changes one position to the right and vice versa when a request of deletion occurs. In (b), the block refers to a power saving architecture while (c) describes a typical memory architecture block model.

A scheduler was used to map between different architectural components with various concurrent processes of the MPEG-2 application.

The proposed model was evaluated under 2 scenarios using the analytical procedures using an **SF2SAN tool** and an **SAN analyzer tool** that were developed for this particular purpose. The SF2SAN tool works within matlab environment which constructs matrices corresponding to each automata in the MM diagrams. The SAN analyzer tool reads the matrices and uses a power method to obtain the steady-state transition matrix. The input bit rate was assumed to be similar in both scenarios. A simulation method was used to obtain values for the model's input parameters. Readers are referred to [9] for more information about the analytical results for the proposed model.

2.4 Analytical Modeling Approach

Performance modeling and evaluation techniques such as analytical methods are considered to be essential and crucial when designing and developing embedded systems. Developing performance models for those systems require a significant effort and time [1]. The ability to estimate performance metrics such as response time (delay or latency) at an early stage of final implementation in any embedded system is essential for efficient design especially real-time systems. Queueing models are one model of several models used to evaluate performance in early 1970s [1]. They were used along with Layered queueing models to provide a framework in order to model contention for hardware and software abstraction layers [1]. Later, Angio traces were developed and used as performance traces at an early lifecycle to generate a mechanism for combining heuristical performance modeling techniques [1].

In [12], a definition for Software Performance Engineering (SPE) was defined by Smith. She emphasized the importance of quantitative methods which should be used at the start of the software development lifecycle to detect the spot of performance defects. D. Smarkusky et al in [1] developed Hierarchical Performance Modeling (*HPM*) for distributed system architectures. Performance models were defined as abstractions of the functional and performance characteristics of a system that are used together to determine if the system under consideration satisfies performance requirements based on a user's demands and hardware architecture. The proposed model was tested and evaluated on distributed system architectures. It provided a high level of accuracy that cannot be reached with only a single layer. The performance analysis includes the computation time for software processes and also the communication time between different distributed processes and the hardware platform [1]. However, the authors did not specify which type of the functional modeling scheme they used.

C.P. Rosiene and R. A. Ammar in [2] developed a date modeling framework for the performance analysis of sequential and parallel software. The SPE model incorporates both models "functional and analytical" into the development of high performance systems such as parallel or distributed or even the real-time ones [2]. The proposed data framework was developed to aide SPE in achieving high performance evaluation. It uses Object-Oriented Paradigm (OOP) with modeling to represent semantic present in the performance models [2]. The authors formalized semantic aggregation relationship first in order to develop their framework. Starting by defining the data model and its components objects and their relationships type specifications was their next step. They encapsulated all the information

related to a single application in a single data model. That model includes a set of object types and a set of relationships between different objects. Each object was classified under a certain object type with its relationships. Every object included a three-tuples of fields which were: 1. Name, 2. A set of attributes and 3. A set of methods. The same thing applied on the relationship. The data model was classified as atomic or non-atomic one in the proposed model. Every atomic object includes the operations and condition nodes to construct which is known as a **Computation Structure Model** (*CSM*). More information about the CSM is provided in chapter 3 section 3.4. Two examples were tested to show the validation of the proposed framework model. The two cases were bubble sort and parallel adder computations. However, no information about the functional modeling approach was given nor specific details about the analytical procedures. Nevertheless, the proposed data framework was a part of an ongoing procedures to create a modeling environment to support the analysis of performance models. More information about the proposed framework can be found in [2].

R. A. Ammar and T. L. Booth in [15] developed a software optimization using user models to achieve more comprehensive design methodology in order to get high performance software. The authors included user performance in their model as an integrated part of the development process. The proposed scheme was used to study the design of a text editor to measure its response time. They considered it as the most important performance specification in human-computer interaction; designers of computer systems put their focuses on quickness of response time to the user actions. The authors saw the performance specifications as boundaries that take the form "$\boldsymbol{C \leq K}$", where

C refers to the time cost (response time) and K refers to the upper boundary associated with that response time. The system model includes an abstraction representation of a system that describes in an organized fashion. The flow of data and control at the computation level was provided by the CSM. The performance was generated as a function of one or more parameters that represent the randomness of the input at both levels using the proposed model. The authors validated their approach on a **VAX** machine running **UNIX** in two stages. The first stage was aiming to identify performance constraints by using a set of controlled experiments. Then collecting data to model the user behavior during regular editing session was done in the second stage. The collected data was used to evaluate the average time cost "response time" of the system over a range several conditions. Three different implementations of the file being edited were investigated to find an optimal one based on the average response time. The experiments that were performed included twelve students with only one semester of experience with the text editor "*Xedit*" and had no experience with another text editor "*EE*". They were given written documentation that explained the editor, necessary instructions and the experimental procedures. From the first implementation, the performance equation obtained was as follows:

$C_{average} = 0.008\ C_{bp} + 66\ C_{fp} + 0.42\ C_{gn} + 0.23\ C_s$, where C_{bp} refers to the response time associated with traveling backward by a page, C_{fp} refers to the response time associated with traveling forward by a page, C_{gn} refers to the response time associated with go to operation instruction and C_s refers to the response time associated with the search operation. For the remaining of the performance equations for other implementations, readers are referred to [15]. The optimal solution was found by using a performance

equation that gives the average response time. Average value of several parameters in terms of the file size and number of lines in the screen were used to construct the optimal solution. The number of lines in the screen varied from 1 to 24, the authors did not mention any reason for that. However, the file size could take any value. The proposed data model provided the designers with performance estimations which can steer the design with a specific goal. User characteristics were incorporated in the analysis and showed the trade-off that can be made between different design alternatives.

G. K. Reddy et al in [11] evaluated software performance of a Polar Satellite Antenna Control Embedded System. The purpose from their paper was to list out various performance evaluations that were unexplored design methodologies for the improvement of throughput. Polar Satellite Antenna Control Systems belong to soft real-time systems which are used mainly in earth observatory systems. They are leveraged in the setup of delivering real-time data transmission and communication. They are equipped with motors through rotating belts. A sensor called “home sensor” that detects the co-ordinates of the antenna is attached with the rotating belts. Any satellite coverage takes one of the following two types: 1. Ascending and 2. Descending according to its orbit design and lifecycle. One way to improve the throughput is to adjust the position of the antenna for maximum possible values of azimuth and elevation ones. It can be achieved by adding a new mode to the controller to estimate the next position value before or when the expected position is reached. Improving the drive belt design helped reaching better response time by restricting the tooth of the belt to be increased or decreased as needed. The proposed performance

analysis was simulated to various initial boundary conditions. More information about the performance analysis can be found in [11].

S. L. Tsao and S. Y. Lee in [28] estimated a performance evaluation of inter-processor communication for an embedded heterogeneous multi-core processor. They referred to Inter-Processor Communication as "**IPC**", several comprehensive experiments were conducted to evaluate the IPC performance for multi-core processor under different design strategies. This multi-core processor was a general purpose processor (**GPP**) and it used a static IPC concept to exchange data between different components in a system under consideration. The authors suggested a dynamic adjusted of IPC to improve the performance. They improved the performance of a Voice over IP (**VoIP**) for phones around 35% while decreasing the GPP workload. In addition, the proposed method was applied on an embedded media gateway system and simulation results showed an accepted improvement in the performance when compared with the static version of IPC.

To evaluate the proposed approach, a Texas instrument (*TI*) Da Vince **DM6446** was used. DM6446 has an *ARM926-EJS* processor, *TI C64 DSP* and embedded Linux as an Operating System (OS). DSP stands for Digital Signal Processor. The IPC requests were handling by the DSP Bios which includes DSP libraries. Modifying the Linux kernel and the bios was conducted to track the IPC procedures in order to measure the latency occurred. Four different sizes of packets were implemented to estimate the response time from IPC, the four different sizes were 128 Bytes, 1 KB, 16 KB and 32 KB, the measured response time was in microsecond "μs". The authors also measured the number of IPC data copies through internal and external shared memory to see how much it influenced the

performance. However, the proposed approach can work only on ARM processors to estimate the response time occurs from IPC only and does not consider the response time caused by other components. Readers are suggested to refer to [28] for more information.

L. Chen et al in [29] estimated the performance of an embedded system based on behavior expression. A digital oscilloscope was developed to be used to estimate the performance based on behavior expressions according to the voltage magnitude and its associated frequency. The authors focuses on the speed of analog-digital converter since its speed influences the system performance. Transfer function W(s) was used to estimate the response time according to the behavior expressions resulted from different voltage amplitudes. The behavior expressions represent the whole process of voltage being processed inside the system. The estimated transfer function from the proposed model was as follows:

$W\alpha(s) = \frac{1}{1-2s} * \frac{x}{x-s} * \frac{1}{1-4s} * \frac{1}{1-4s} * \frac{1}{1-5s} * \frac{1}{1-5s}$, where α represents the total cycle behavior expression. The estimated total response time was computed as follows:

$T = \frac{\partial W\alpha(s)}{\partial s}\Big|_{s=0.} = x^* (19 + 2x)$

x represents the time of data transmission and its associated voltage from digital processing.

Y. Y. Cho et al in [30] proposed a performance evaluation system for embedded software. It consists of code analyzer, data analyzer, report viewer and lastly testing agents. Code analyzer was used to insert additional code dependent on a target system into source code and compiles it. Data analyzer translates the raw level results to high level of APIs for reporting viewer. Report viewer provides graphical user interface to view the reports

while testing agents execute the performance tests. The proposed scheme works only on pure software and ignores any presence of hardware components. Figure 8 from [30] depicts a general overview of the proposed architecture of performance evaluation system.

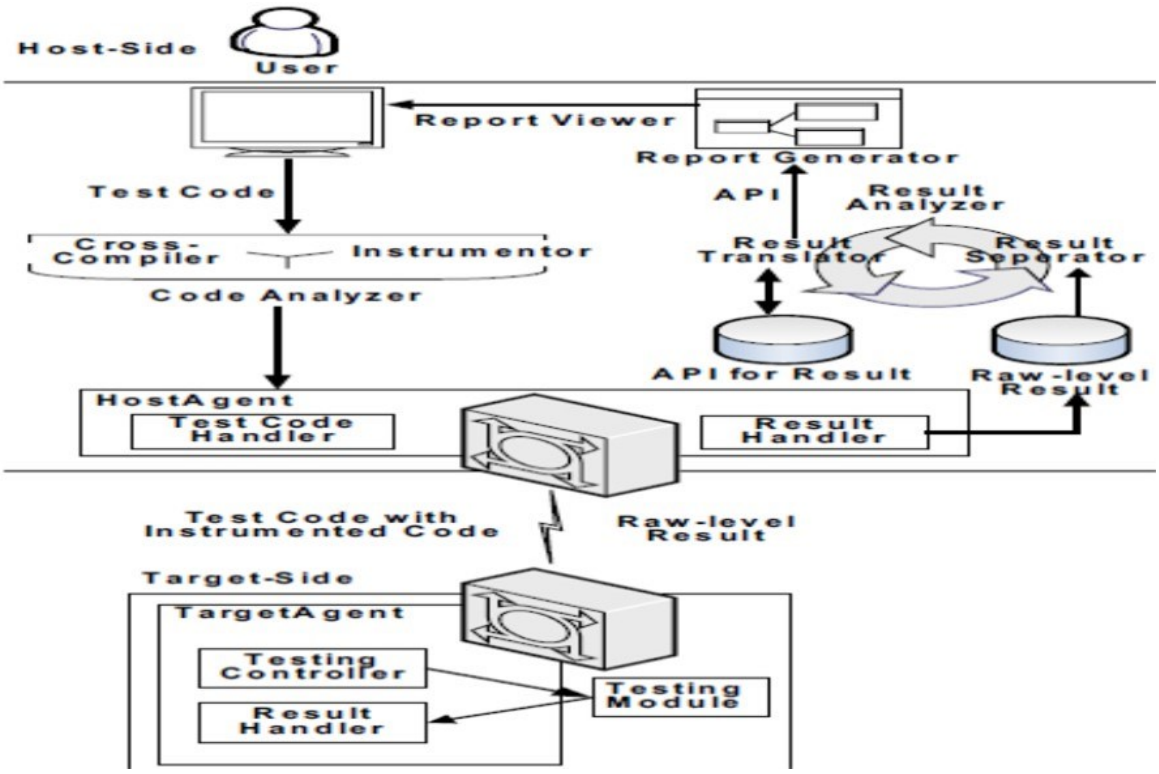


Figure 8: General overview of the proposed model

According to fig. 8, the proposed model is just a client/server based in host-target architecture. Since the embedded systems suffer from small memory capacity, the proposed scheme was placed on both ends. On the host side, it provided users with the convenient GUI and on the target side, it executed the performance system. The proposed model gave the authors ability to trace what functions were being executed to distinguish between what were necessary and what were not. The report viewer displayed the results in UML diagram files. That results are converted by result translator into API classes or XML files and are

stored into API data base. The proposed approach was developed in Java and had ability to evaluate C programs. They contained about 500 lines of coding and three modules. The model developed in [30] focused mainly in memory evaluation by providing the obtained results in graphical interfaces which made easy to read and more understandable. However, it was used mainly in ARM systems and neglected the influences of hardware components on the system performance.

D. Pimentel in [31] proposed a model to evaluate performance of embedded systems at system level only. The model is called **Artemis Workbench** which was developed to provide modeling and simulation methods with supported tools for efficient performance evaluation. It focused only on heterogeneous embedded multimedia systems. It allowed the author for architectural exploration at different levels of abstraction. It was applied on a motion-JPEG application as a case study to illustrate the modeling aspects and show its validation and correctness.

The Artemis Workbench is composed of a set of tools and schemes which are integrated to form a framework that allows designers to model applications and SoC architectures at a high level of abstraction. Figure 9 from [31] depicts a flow of operations inside the Artemis Workbench.

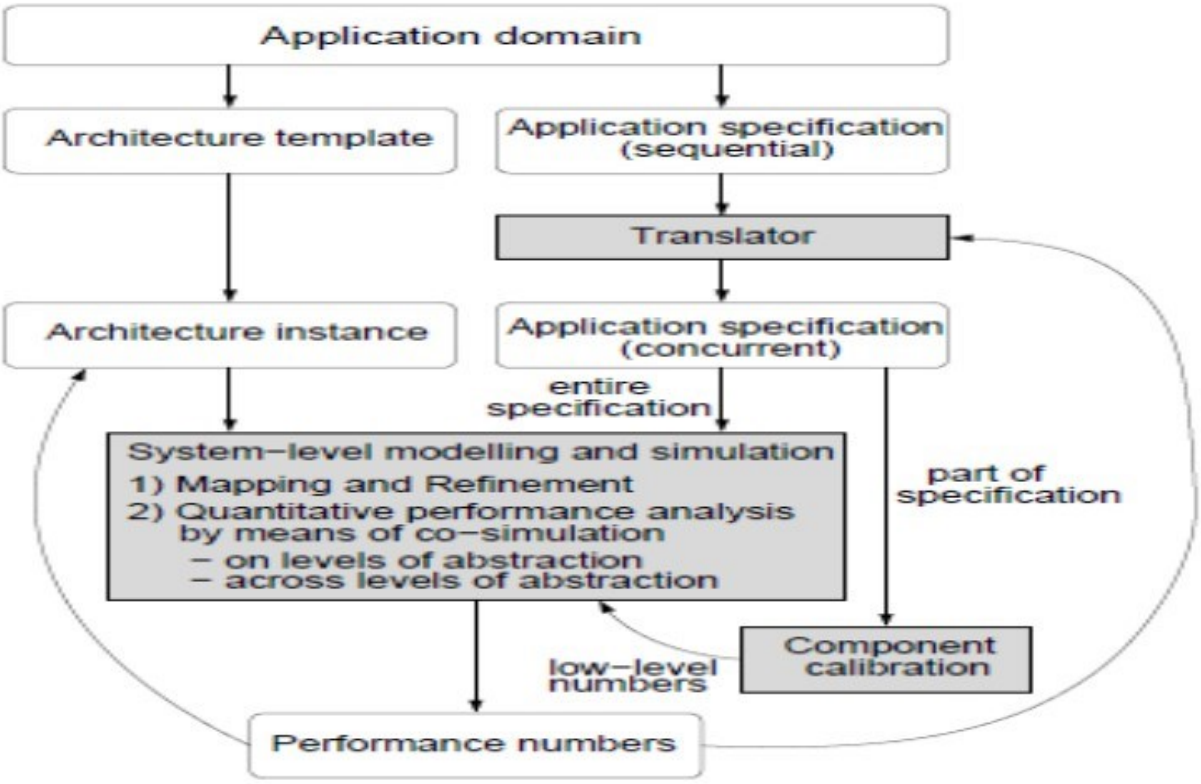


Figure 9: General overview of flow direction inside Artemis Workbench

The grey areas in fig. 9 indicate a set of various tools that embody the proposed model.

A sequential application specification is transformed into Kahn Process Network (KPN) by a translator called Compaan. The performance analysis on all levels of abstraction was done by a mapping layer which laid between the application and architecture layers. The system level modeling of Artemis is called Sesame, which includes three components. The three components are as follows:

1. Application Model which is represented as KPN.
2. Mapping Layer which shows the data flow direction.
3. Architecture Model which is built as discrete event method.

Figure 10 from [31] gives an overview of how all three components are connected together.

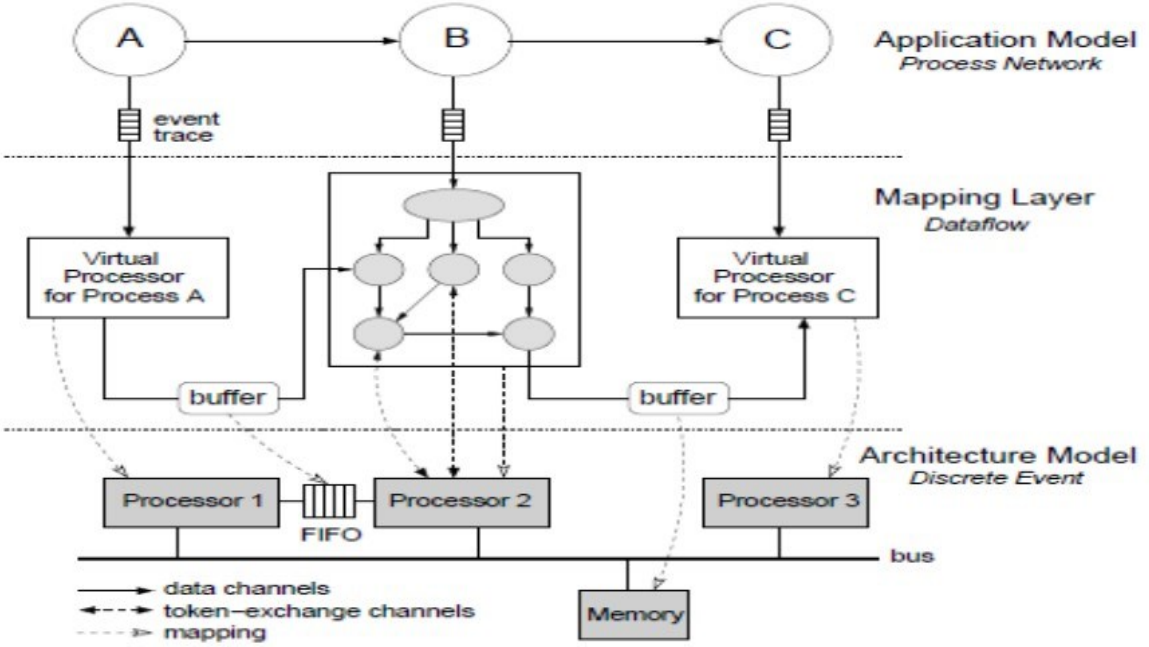


Figure 10: Sesame components

The performance analysis in the proposed model was estimated using a Y-chart design approach. Y-chart design approach includes application and architecture models along with the system simulation to form the Y shape. The application model described the functional behaviors of an application being run; the architecture model described and defined the architecture resources and captured their performance constraints. The system simulation was developed based on trace-driven approach. More information about the proposed scheme can be found in [31]. That model considered only the performance evaluation at system level and neglected other levels such as task, module and operation levels.

G. Madi et al in [32] proposed a method to estimate performance of distributed real-time embedded systems (**DRE**) by discrete event simulations. The method represented

DRE systems as discrete event systems (**DES**) in continuous time first, then provided an automated formulation for the performance evaluation. The proposed approach was applied on a *synchronous DRE system* using fixed priority scheduling policy. Hence, a resultant scheme was non preemptive scheduling model. The proposed model consisted of five components which were as follows:

1. A set of Tasks T.
2. A set of Machines M.
3. A set of Communication channels "C ⊆ T.
4. A set of Timers TR ⊆ T.
5. A set of dependency relationship D ⊆ T * T.

Every task was executed only one time on a machine and was given an execution interval [BCET, WCET] rather than using a constant time value. BCET refers to Best-Case Estimated Time. All tasks were scheduled using FIFO policy, where FIFI stands for First-In First-Out. Each computation task had three states which were: 1. Initial, 2. Wait and 3. Run. Whenever a task receives a trigger "event" from another task, its status then is changed from initial to wait which means the task has been enabled and ready to be executed. The changing from wait to run state was handled by the scheduler.

In order to estimate and evaluate the proposed model, an **Event Order Tree** "*EOT*" was used. It had two traces for the execution and they were equivalent. For the equivalence, only the order of events were the same but not their execution time interval. The EOT is a directed tree representation inside the DRE to capture all valid traces. Every node in it

represents an event with its time constraints. Each path from a root to a node represents a possible number of equivalent execution traces.

The proposed model was applied on **Boeing Bold Stroke** execution framework which was developed based on a real-time **Corba Avionic application**. The framework had 98 tasks and 57 dependencies between them. The performance measurements were estimated on two metrics; the first one was response time and the other metric was Schedulability as if any task may miss its deadline. Around 20 million of the non-equivalent traces were obtained from running the model for a week. All of them had a different execution order. For more information about the proposed model, readers can refer to [32].

In [33], A. Abdel-raouf et al proposed a model to analyze and evaluate performance of **Distributed Object-Oriented Software** "*DOOS*". The scheme provided a methodology based on a performance-based model to estimate the performance of a system under investigation while preserving the **Object-Oriented** "*OO*" features such as encapsulation, information hiding and inheritance. The proposed approach considered the communication overhead between different nodes and added to the estimated response time. It had two stages, one for the execution process and the other one for the communication process. The model was hierarchical one to model the distinct abstraction levels of the DOOS. Different arrival times were considered in the model to capture all behaviors observed in order to estimate the response time.

The proposed model was applied on a transaction process in banking system as a case study. The model used the HPM as a technique to estimate the average response time. More information about HPM can be found in chapter 3. The scheme starts from estimating the

response time that occurs from the communication process then it estimates the response time occurring from the computation process. The results from both components is the estimated average response time.

The developed model was not applied on embedded systems, however, it showed how important it is to include the communication overhead in the procedures to get a good estimation for the average response time.

2.5 Applications

Embedded systems have become very important in our lives; they pervade all fields in today's advanced technology. Embedded systems are found in 95% of the current market such as home appliances, industrial, automotive and medical applications. In this section we conclude with some possible applications that are applicable for our designing framework.

2.5.1 Controller Area Network (CAN)

A controller area network (CAN bus) is a vehicle bus standard designed to allow microcontrollers and devices to communicate with each other in applications without a host computer. It is a message-based protocol, designed originally for automotive applications, but is also used in many other contexts [107,108]. Development of the CAN bus started in 1983 at Robert Bosch GmbH. The protocol was officially released in 1986 at the Society of Automotive Engineers (SAE) congress in Detroit, Michigan. The first CAN controller chips, produced by Intel and Philips, came on the market in 1987. There is no addressing scheme used in controller area networks, as in the sense of conventional addressing in

networks (such as Ethernet) [107]. Rather, messages are broadcast to all the nodes in the network using an identifier unique to the network. Based on the identifier, the individual nodes decide whether or not to process the message and also determine the priority of the message in terms of competition for bus access. This method allows for uninterrupted transmission when a collision is detected, unlike Ethernets that will stop transmission upon collision detection.

The modern automobile may have as many as 70 electronic control units (ECU) for various subsystems. Typically the biggest processor is the engine control unit. Others are used for transmission, airbags, antilock braking/ABS, cruise control, electric power steering, audio systems, power windows, doors, mirror adjustment, battery and recharging systems for hybrid/electric cars, etc. Some of these form independent subsystems, but communications among others are essential. A subsystem may need to control actuators or receive feedback from sensors. The CAN standard was devised to fill this need.

Architecture

CAN is a **multi-master serial bus** standard for connecting Electronic Control Units (ECUs) also known as nodes [107,108,109]. Two or more nodes are required on the CAN network to communicate. The complexity of the node can range from a simple I/O device up to an embedded computer with a CAN interface and sophisticated software. The node may also be a gateway allowing a standard computer to communicate over a USB or Ethernet port to the devices on a CAN network [107,108,109,110]. All nodes are connected to each other through a two wire bus. The wires are 120 Ω nominal twisted pair as shown in figure 11 from [107].

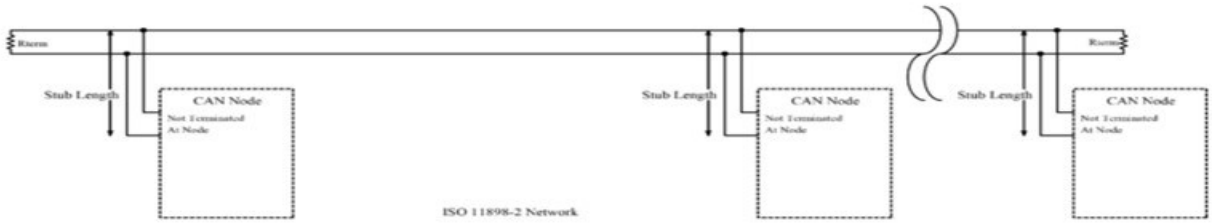


Figure 11: High speed CAN network. ISO 11898-2

High speed CAN is usually used in automotive and industrial applications where the bus runs from one end of the environment to the other. Fault tolerant CAN is often used where groups of nodes need to be connected together [107,108,109]]. The ISO specifications require the bus be kept within a minimum and maximum common mode bus voltage, but do not define how to keep the bus within this range.

Each node requires a:

- **Central processing unit**, a microprocessor, or a host processor
    - The host processor decides what the received messages mean and what messages it wants to transmit.
    - Sensors, actuators and control devices can be connected to the host processor.
- **CAN controller; often an integral part of the microcontroller**
    - Receiving: the CAN controller stores the received serial bits from the bus until an entire message is available, which can then be fetched by the host processor (usually by the CAN controller triggering an interrupt).

    - Sending: the host processor sends the transmit message(s) to a CAN controller, which transmits the bits serially onto the bus when the bus is free.

- **Transceiver <u>Defined by ISO 11898-2/3 Medium Access Unit [MAU] standards</u>**
    - Receiving: it converts the data stream from CAN bus levels to levels that the CAN controller uses. It usually has protective circuitry to protect the CAN controller.
    - Transmitting: it converts the data stream from the CAN controller to CAN bus levels.

Each node is able to send and receive messages, but not simultaneously [107]. A message or Frame consists primarily of the ID (identifier), which represents the priority of the message, and up to eight data bytes. A CRC, acknowledge slot [ACK] and other overhead are also part of the message. The improved CAN FD extends the length of the data section to up to 64 bytes per frame. The message is transmitted serially onto the bus using a non-return-to-zero (NRZ) format and may be received by all nodes.

The devices that are connected by a CAN network are typically sensors, actuators, and other control devices. These devices are connected to the bus through the host processor, a CAN controller, and a CAN transceiver. Figure 12 from [107] shows a typical structure of a CAN node.

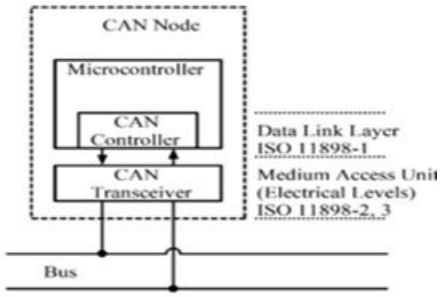


Figure 12: A CAN node structure

CAN Benefits

- **Low-Cost, Lightweight Network:** CAN provides an inexpensive, durable network that helps multiple CAN devices communicate with one another. An advantage to this is that electronic control units (ECUs) can have a single CAN interface rather than analog and digital inputs to every device in the system [107,107]. This decreases overall cost and weight in automobiles.
- **Broadcast Communication:** Each of the devices on the network has a CAN controller chip and is therefore intelligent. All devices on the network see all transmitted messages [107,109]. Each device can decide if a message is relevant or if it should be filtered. This structure allows modifications to CAN networks with minimal impact. Additional non-transmitting nodes can be added without modification to the network.
- **Priority:** Every message has a priority, so if two nodes try to send messages simultaneously, the one with the higher priority gets transmitted and the one with the lower priority gets postponed. This arbitration is non-destructive and results in

non-interrupted transmission of the highest priority message. This also allows networks to meet timing constraints.

- **Error Capabilities:** The CAN specification includes a Cyclic Redundancy Code (CRC) to perform error checking on each frame's contents. Frames with errors are disregarded by all nodes, and an error frame can be transmitted to signal the error to the network. Global and local errors are differentiated by the controller, and if too many errors are detected, individual nodes can stop transmitting errors or disconnect itself from the network completely.

Figure 13 from [107] depicts the benefit of using CAN.

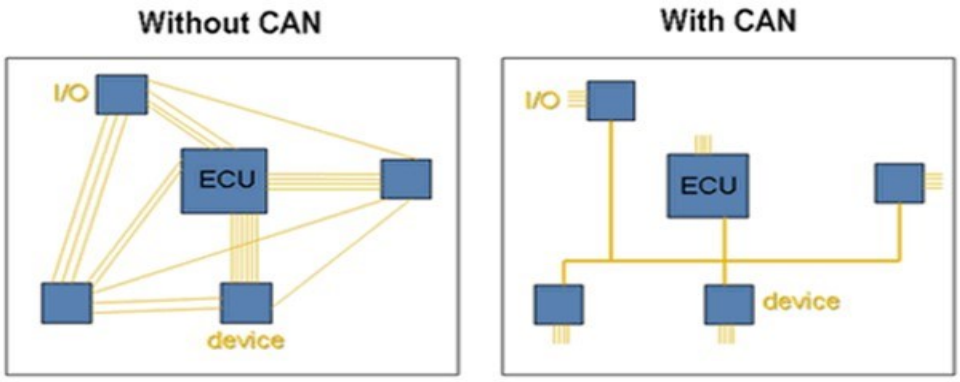


Figure 13: CAN benefits

From fig. 13, CAN significantly reduces the wiring. Figure 14 from [107] depicts CAN blocks as related to OSI layers and features

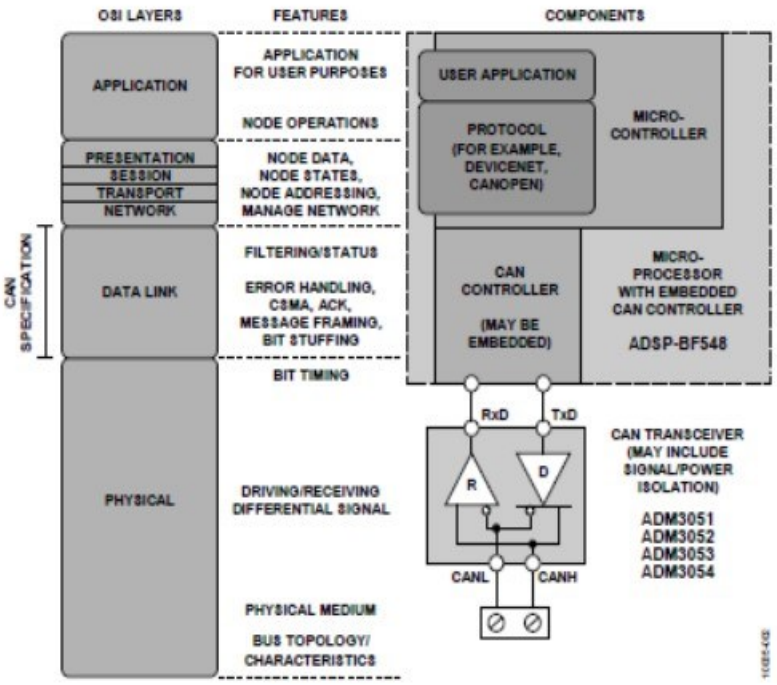


Figure 14: CAN implementation Blocks

SAE J1939

Society of Automotive Engineers **SAE J1939** is a vehicle bus recommended practice used for communication and diagnostics among vehicle components, originally by the car and heavy-duty truck industry in the United States [107]. SAE J1939 is used in the commercial vehicle area for communication throughout the vehicle. With a different physical layer, it is used between the tractor and trailer. This is specified in *ISO 11992*.

SAE J1939 defines five layers in the seven-layer OSI network model, and this includes the **Controller Area Network** (*CAN*) *ISO 11898* specification (using only the 29-bit/"extended" identifier) for the physical and data-link layers [107]. Under J1939/11 and J1939/15, the data rate is specified as 250 kbit/s, with J1939/14 specifying 500 kbit/s. The session and presentation layers are not part of the specification. Originally, CAN was not

mentioned in J1939, which covered cars and tractor-trailer rigs, and with some dual and triple use 8-bit addresses assigned by the SAE J1939 board. CAN was not originally free, but its instruction set did fit in the custom instruction format of J1939. This was true as of 2000. Since then, CAN has been included, the chipset for J1939 has been clocked faster [clarification needed], and 16-bit addresses (PGN) have replaced 8-bit addresses. J1939, ISO 11783 and NMEA 2000 all share the same high level protocol [107].

All J1939 packets, except for the request packet, contain eight bytes of data and a standard header which contains an index called Parameter Group Number (PGN), which is embedded in the message's 29-bit identifier [107]. A PGN identifies a message's function and associated data. J1939 attempts to define standard PGNs to encompass a wide range of automotive, agricultural, marine and off-road vehicle purposes. A range of PGNs (00FF0016 through 00FFFF16, inclusive) is reserved for proprietary use. PGNs define the data which is made up of a variable number of Suspect Parameter Number (SPN) elements defined for unique data. For example, there exists a predefined SPN for engine RPM.
SAE J1939 can be considered the replacement for the older SAE J1708 and SAE J1587 specifications. SAE J1939 has been adopted widely by diesel engine manufacturers. One driving force behind this is the increasing adoption of the engine Electronic Control Unit (ECU), which provides one method of controlling exhaust gas emissions within US and European standards. Consequently, SAE J1939 can now be found in a range of diesel-powered applications: vehicles (on- and off-road), marine propulsion, power generation and industrial pumping. Applications of J1939 now include off-highway, truck, bus, and even some passenger car applications [107].

2.5.2 Android

Android software architecture is designed and built as a stack structure as shown in figure 15 [34]

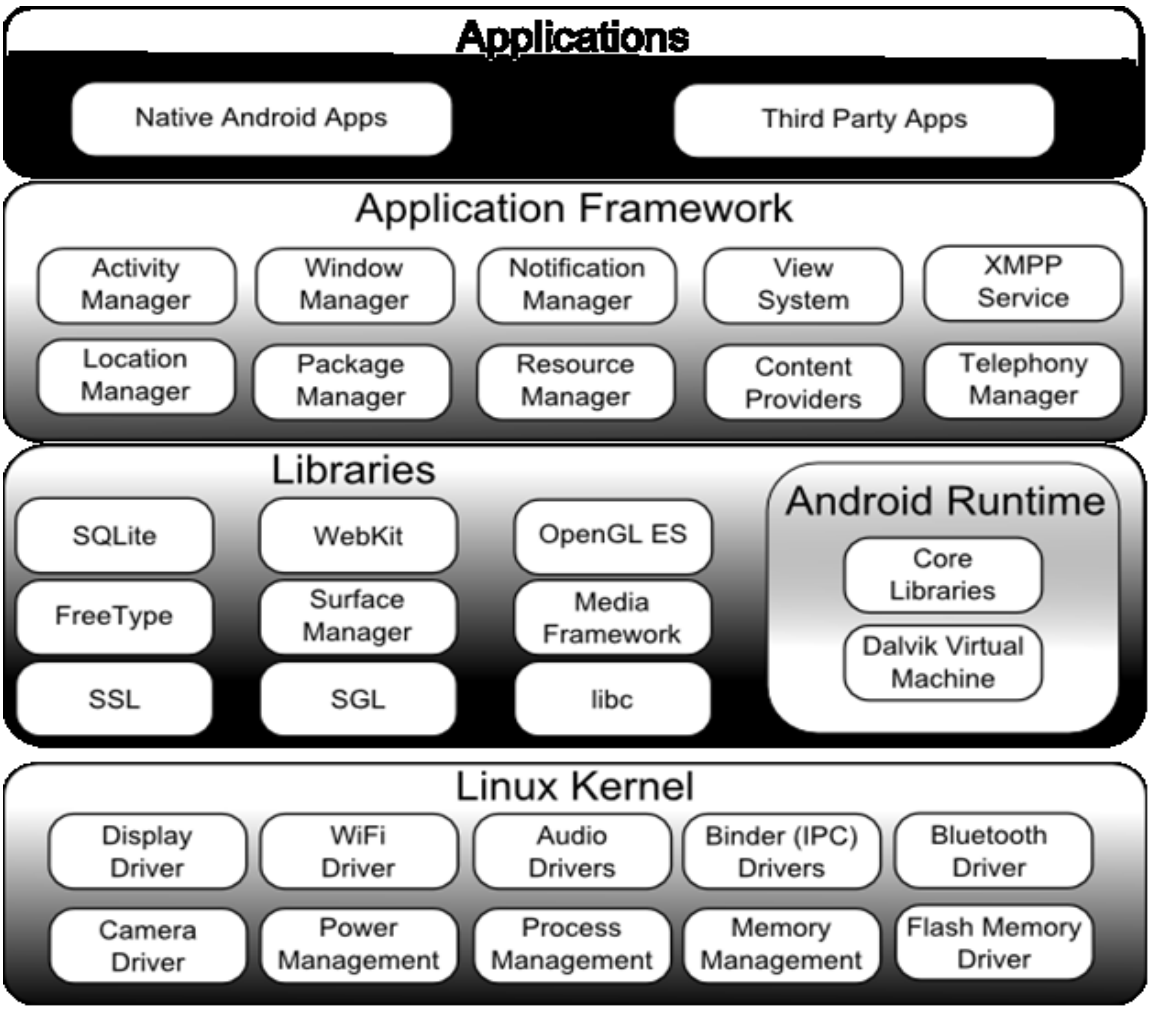


Figure 15: Typical Android software structure

Figure 15 shows 4 layers of components integrate with each other to form what is known today as ***ANDROID***. Each layer contains some components combine together to perform a set of specific jobs [34,35,36].

1- **LINUX KERNEL**: is considered as the basic layer which interacts with Hardware elements and includes all necessary hardware drivers for its designated system. Drivers are programs which control and communicate with the hardware. It uses the kernel to perform its all core functionality such as process management, memory management, security settings and etc.

2- **LIBRARIES AND ANDROID RUNTIME**: enable a device to handle different types of data. They contain a place (Dalvic Virtual Machine) where applications are run and optimized for low processing power and memory environments.

3- **APPLICATION FRAMEWORK**: manages the basic functions of the device like voice call management. It is the block where our applications directly interact with.

4- **APPLICATIONS**: it is the top layer in the architecture and the place where our applications fit.

Any application in Android is built based on 4 different components which are

- Activity.
- Content provider.
- Service.
- Broadcast receiver.

In the Android, **a task can be defined as an activity or a set of activities. The transitions between different activities are initiated by using intents**. A typical lifecycle for any activity has 7 states which are:

I. **OnCreate**: when an application is launched, it first enters oncreate state; it initializes data elements. It is provided with a Bundle object as parameter to restore the UI state. Assigns a thread to execute the specific task(s).

II. **OnStart**: this state is called before the Activity is being visible to the User. The task is not running yet.

III. **OnResume**: this state is also known as *running (active) state*; where the application is visible and running.

IV. **OnPause**: this state is called when another activity is being running and interacting with the user.

V. **OnRestart**: it is called when the user navigates back to the previous activity which leads it to the OnStart state.

VI. **OnStop**: this state is called when the activity is no longer visible and the user cannot interact directly with it. It means that the application runs in the background or when the task is done with the P.U.

VII. **OnDestroy**: this is the final state and called when the user finishes using the application and the task was either successfully executed or failed.

The following diagram shows the activity lifecycle in Android system. More information about Android can be found in [34,35,36].

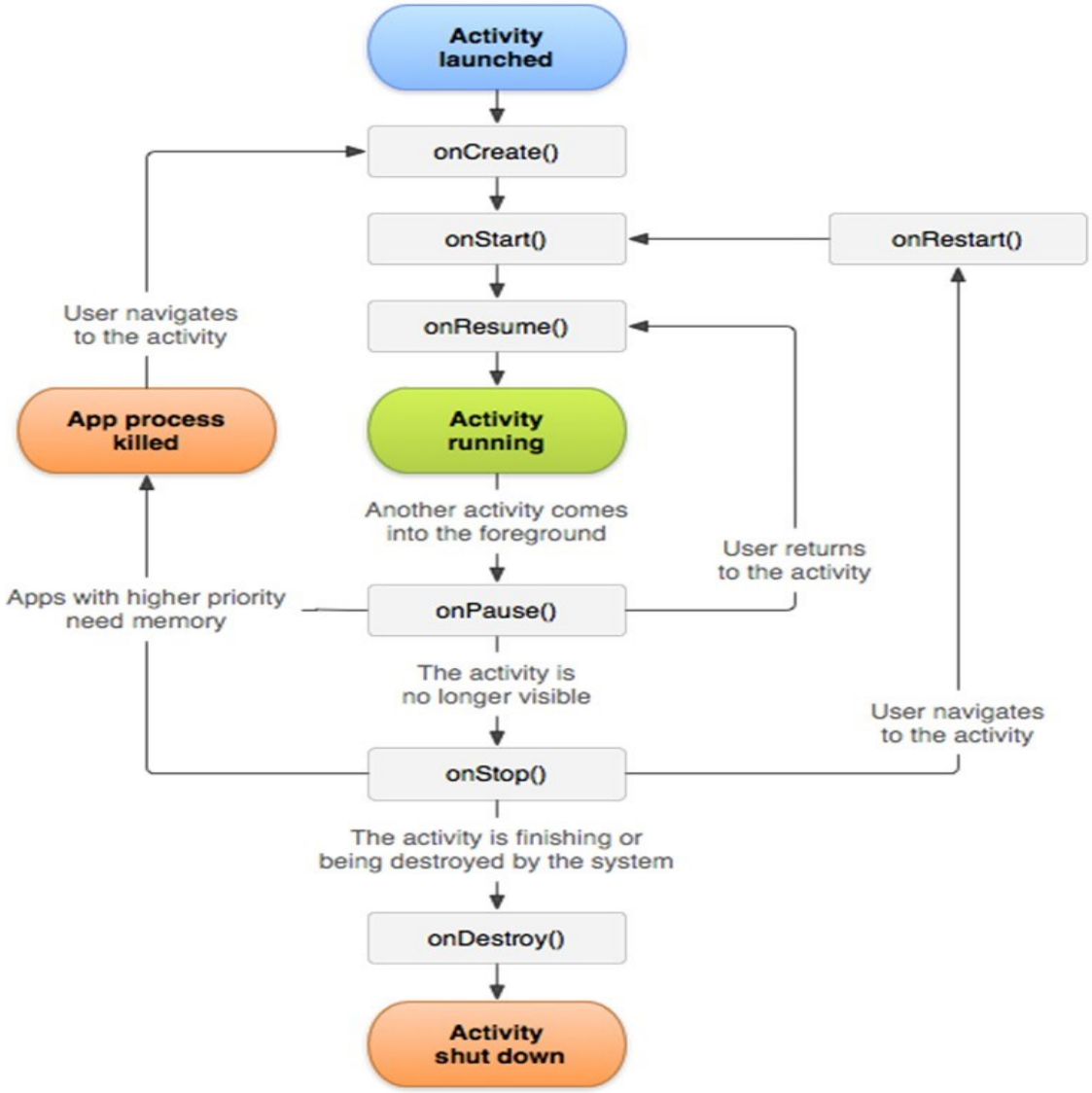
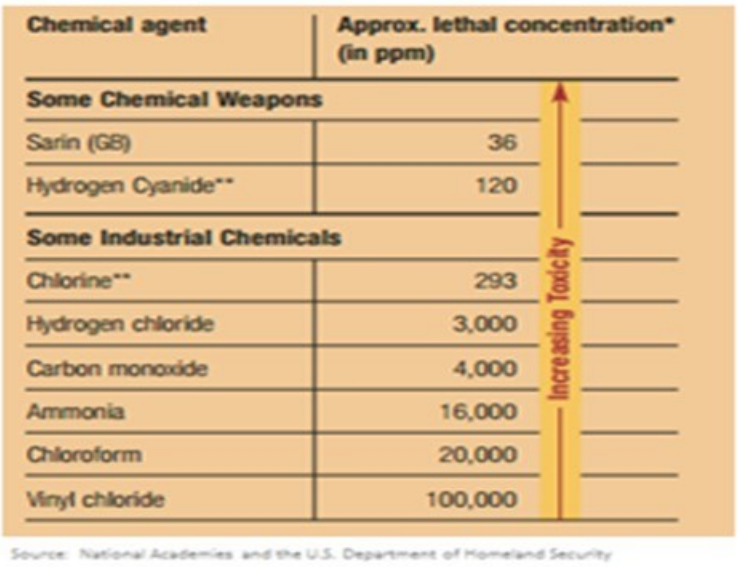


Figure 16: Android lifecycle

2.5.3 Pollution Detection Systems

According to homeland security, Chemical detection have become an important problem since modern chemical detection technologies that are utilized in the defense field suffer from poor performance accuracy [38]. Performance refers to the ability to detect all threats and also inability to in terms of high confidence identification [38]. Chemical

detection is considered to be an intensive problem in time and data. A pioneering miniaturized chemical sensor technology which delivers multi-threat detection with high confidence identification has been developed and design by a few organizations [38]. That technology is integrated into small devices such as handheld devices and also can be mounted on vehicles to achieve high capability against about 95% of known chemical threats [38]. **Chemical warfare** (*CW*) is considered to be one of the weapons of mass destruction "WMDs". Figure 17 from [38] displays the list of chemical threats and their lethal concentration.



Chemical agent	Approx. lethal concentration* (in ppm)
Some Chemical Weapons	
Sarin (GB)	36
Hydrogen Cyanide**	120
Some Industrial Chemicals	
Chlorine**	293
Hydrogen chloride	3,000
Carbon monoxide	4,000
Ammonia	16,000
Chloroform	20,000
Vinyl chloride	100,000

Increasing Toxicity

Source: National Academies and the U.S. Department of Homeland Security

Figure 17: List of chemical threats and their lethal concentration

Governments worldwide have focused on the use of chemical and improved explosives in warfare and terrorism to protect civilians and militaries. Militaries carry detectors to a field to find harmful chemicals or bombs before they fall into the wrong hands [38]. Those detectors must be lightweight, portable, reliable and easy to use. They must have a high

confidence identification to detect harmful materials. Otherwise, the militaries' live will be in danger.

Modern chemical sensors uses Micro or Nano technology to fabricate and provide improved sensitivity with reduced size, power consumption and cost to accurately detect threat chemicals [38]. The sensors, with high detection accuracy, save time and even live and are considered more useful among other sensors which might provide false alarms [38]. They can be deployed in high traffic places such as malls or airports where evacuation becomes the highest priority in case of anything goes wrong.

Modern chemical sensors uses chemical mobility as a measure to differentiate chemicals. Chemical mobility can be defined as the measure of how quickly an ion of a chemical moves through an electric field which is generated by the sensors [38]. The sensors detect the chemical of interest according to their mobility characteristics by filtering out their background [38]. Using ionization methods is the most sensitive and reliable technique available today to detect harmful chemicals.

2.5.4 Fire Detection Systems

Because of the speed and totality of the destructive forces of fire, it constitutes one of the more serious threats [39,40]. Items destroyed by fire, however, are gone forever [39]. Any building can be completely obliterated and burned by a fire within a few hours.

The functions of fire detection systems are as follows:

- Identify an incident upon occurring either manually or automatically.
- Raise occupants' alarm in order to evacuate a premises as fast as possible.
- Send a notification to emergency response experts about it.

Many existing types of fire detection systems depend mainly on the characteristics of the protected premises [40]. Identifying a developing fire emergency in a timely manner is a key aspect of fire protection. Several factors determine the choice of fire alarm systems; the factors are summarized as follows [39,42]:

A. Building structure.

B. Current legislation law.

C. Purpose(s) and use of a premises.

Modern fire detection systems typically operate on a same principle, if a detector identifies a smoke or heat or someone operates a manual break point, then the detector sounders operate to alarm and warn others on the premises that there is a fire and they need to evacuate immediately [41]. Furthermore, it sends an alert signal to a central station or emergency response experts to notify them about the incident.

Fire detection systems are categorized into the following:

1) Conventional systems: a simpler technology is used and suited for small or even medium applications. They are very effective in terms of cost and maintenance [41,43].
2) Addressable Systems: a system is configured to provide a wide range of flexibility with components controlling a variety of devices [44]. Each device in the system is assigned with a unique address which allows a control panel to monitor and control the status of each individual device connected to it [43].

Wireless systems: very effective alternative to traditional wired fire detection systems for all types of applications.

CHAPTER 3

Developed Framework

3.1 Introduction

In this chapter, we aim to introduce the designing framework, shown in fig. 1, which is used to estimate the average performance metric such as response time, power consumption, reliability and/or availability. The designing framework can be seen as a multidimensional design methodology to estimate multiple performance metrics. Multidimensional means that it is capable of estimating several performance metrics. However, this research focuses only on response time, known as delay or latency, and power consumption.

In [45], we developed the designing framework which is composed of three components which are: 1. **Hierarchical Generic Finite State Machine** "*HGFSM*" which represents the functional modeling approach, 2. **Markovian Model** and 3. **Hierarchical Performance Model** "*HPM*" which represents the analytical approach which is also known as the mathematical scheme. In section 3.2, detailed information about HGFSM is presented. Section 3.3 describes the Markovian Model which is used to map between other components. In section 3.4, comprehensive details about HPM are provided. Two case studies are presented in section 3.5 to show how the designing framework is used to estimate average response time and power consumption in Android and response time in OPENWRT.

3.2 Hierarchical Generic Finite State Machine "HGFSM"

A task in any embedded system can be classified as either completed or failed. A set of states exists among those two states to form a hierarchical generic finite state machine "HGFSM" [45]. Figure 18 depicts the HGFSM for an execution cycle of any task inside a system.

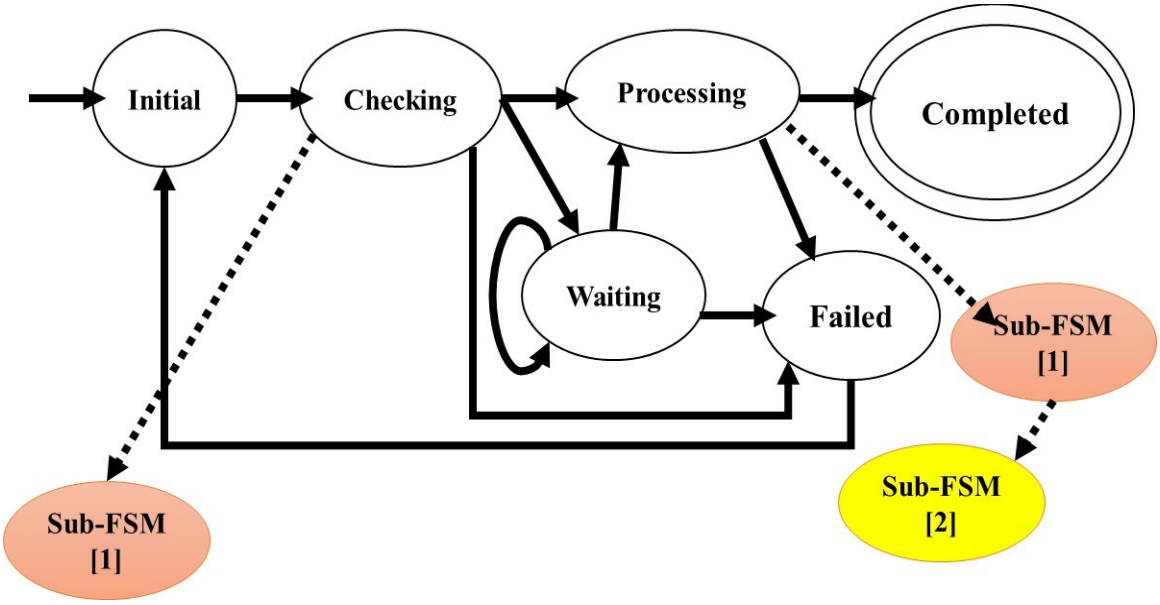


Figure 18: Hierarchical generic finite state machine

Fig. 18 shows that there are three levels in the developed HGFSM which are differentiated in three different colors white, red and yellow. The 6 states in white represent the higher abstraction level. The 2 sub-states in red represent the second level, the number between square brackets indicates that both states are in the same level. However, they are totally internally different since each sub-state has its own function. The sub-state in yellow represents the last level in the hierarchical generic FSM [45,46]. The HGFSM model is composed of 5-tuples $\{\sum, S, S_0, \delta, F\}$, where

1- ∑ represents a set of input alphabets.

2- S represents a set of states in the model.

3- S_0 contains initial states in which they are an element or sub-elements of S.

4- δ represents a state-transition function which maps between an input state(Current state) with input alphabet(s) to a new state (next state).

5- F contains a final state or a set of final states which belong to S.

∑ contains a set of input alphabets = {Y, N, T} where Y and N stand for Yes and No respectively which refer to a condition result in the proposed model. While T stands for a task or a set of tasks being executed. Input alphabets are used to cause a movement from current state S_i to next state S_j

S contains 6 states in the proposed HGFSM; they are named as follows ***Initial State, Checking State, Executing State, Waiting State, Failed State and Completed State***. So S = {Initial State, Checking State, Suspend State, Executing State, Waiting State, Failed State, Completed State}.

S_0 contains only one state; so S_0 = {Initial State}.

δ maps between the current state with its transition function to the next state as mentioned earlier so $\delta = S_i * Y \rightarrow S_j, T$ or $\delta = S_i * N \rightarrow S_j, T$

F contains only one state so F = {Completed State}. This state is denoted by two circles around it as shown in fig. 18 [45].

Initial state: each task is provided with an arrival time ($\mathbf{t_a}$) and a deadline ($\mathbf{t_d}$) time which refers to one of the constraints in the system. *There is no transition when a system is idle which means there is no incoming task.*

Checking state: its jobs are:

- Checks if the task deadline can be met or not; if not, it forwards task into failed state to restart its cycle again. Otherwise, move to next condition checking.
- Checks available resources for execution; if none are present, sends tasks to failed state. Otherwise, performs the next test.
- Checks if the queue in the execution state is full or not; if not, then forwards the task into execution state. Otherwise, forwards it into waiting state.

Execution state: represents the place where the task completes its cycle. If the execution succeeds, the task is sent to the completed state. Otherwise, it sends it to the failed state. The execution is done successfully if the execution time (t_e) <= deadline time (t_d).

Waiting state: the task waits its turn to be executed once the queue of execution state is not full or the processing unit becomes available when the deadline can be met; otherwise, the task is sent to failed state to restart its cycle again.

Completed State: represents the last point for the task before it goes to another part in a system under investigation.

Failed State: all unfinished tasks are sent to this state to restart their cycles if possible.

The following data flow chart depicts the flow of tasks inside the designing framework.

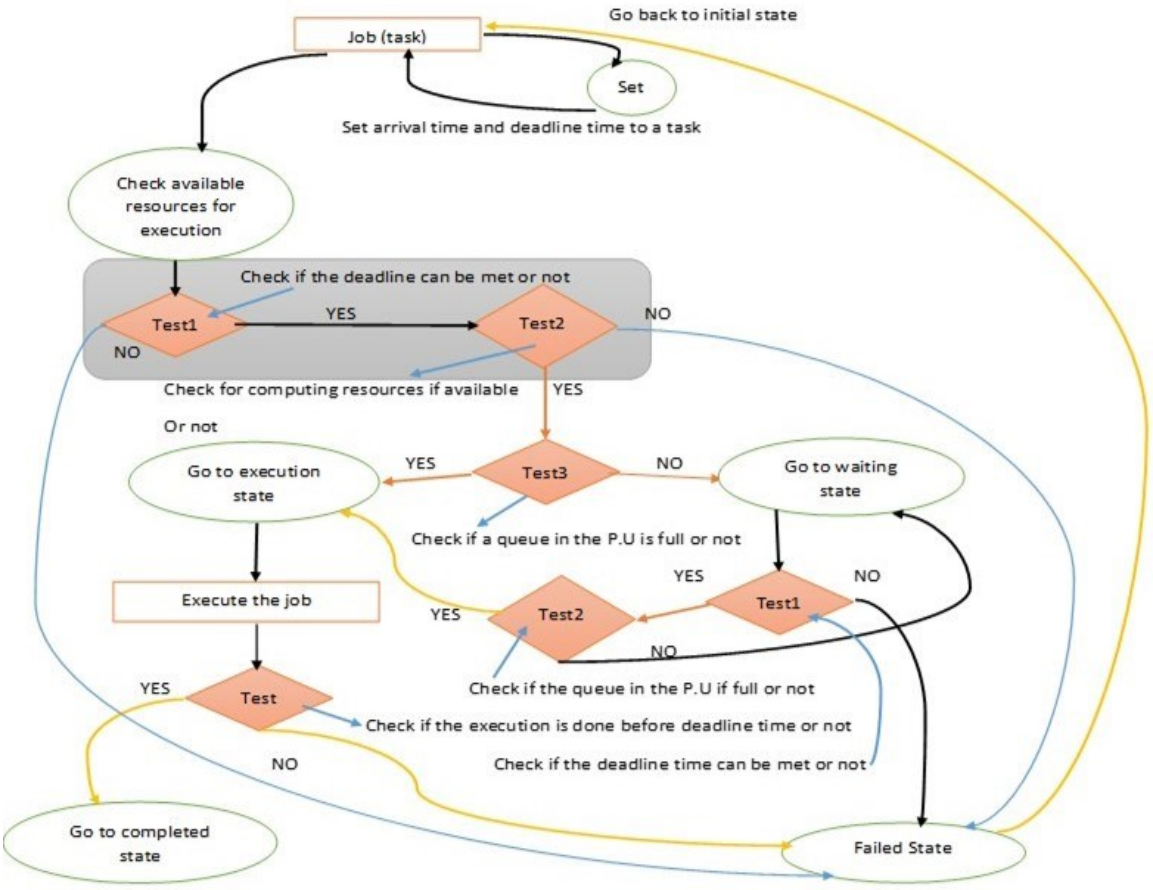


Figure 19: Data flow graph for task inside the developed HGFSM

<u>The two test nodes inside the grey square indicate they exist inside the checking state</u>

The checking state has its own sub-state where there are three sub-states which are:

Receiving and sorting State: receives incoming tasks from the higher level state which is initial state; sorts them according to an implementing scheduling algorithm such as FIFO, LIFO or EARLY DEADLINE FIRST. Checks for the deadline time first if it can be met or not. If yes, it sends tasks to the decision state; otherwise, it sends them to the failed state.

Decision State: dispatches a task to failed or waiting or execution (processing) state based on test condition result which is done to determine the availability of the required resources

after making sure the deadline can be met. If the deadline cannot be met, the task is sent to the failed state. If yes, another test will be performed to determine where the task will be sent. If the result is yes, the task is sent to the processing state. Otherwise, it is sent to the waiting state. In the meantime, it sends a notification to the Recording state to tell whether the task is sent to failed or processing or waiting state.

Recording State: acts as a storing one. It keeps track of a status of incoming tasks which one can be executed and which one will be forwarded to the failed state to restart its cycle. Fig. 20 shows the flow of operations performed inside the checking state.

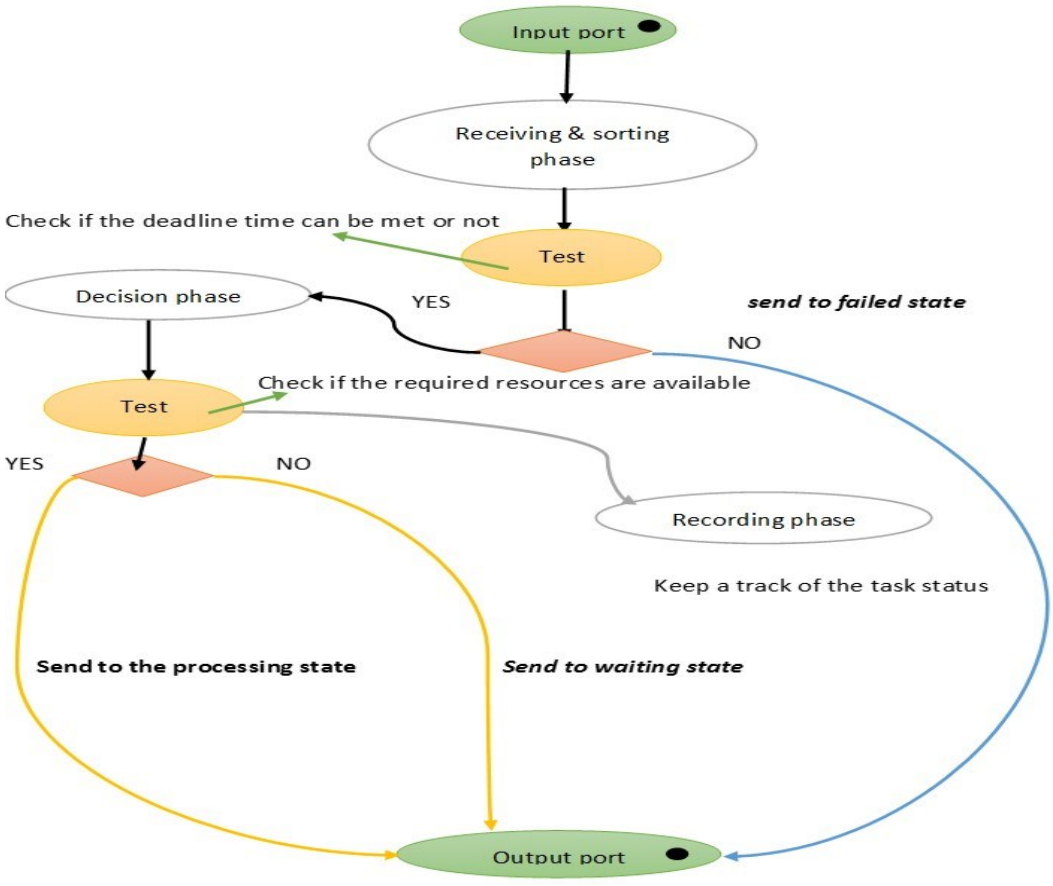


Figure 20: Data flow chart for operations inside the checking state

The Execution “processing” State is decomposed into another sub-FSM model; the model is shown in fig. 21. The sub-state (handling) is also decomposed into another sub-FSM model as depicted in fig. 22.

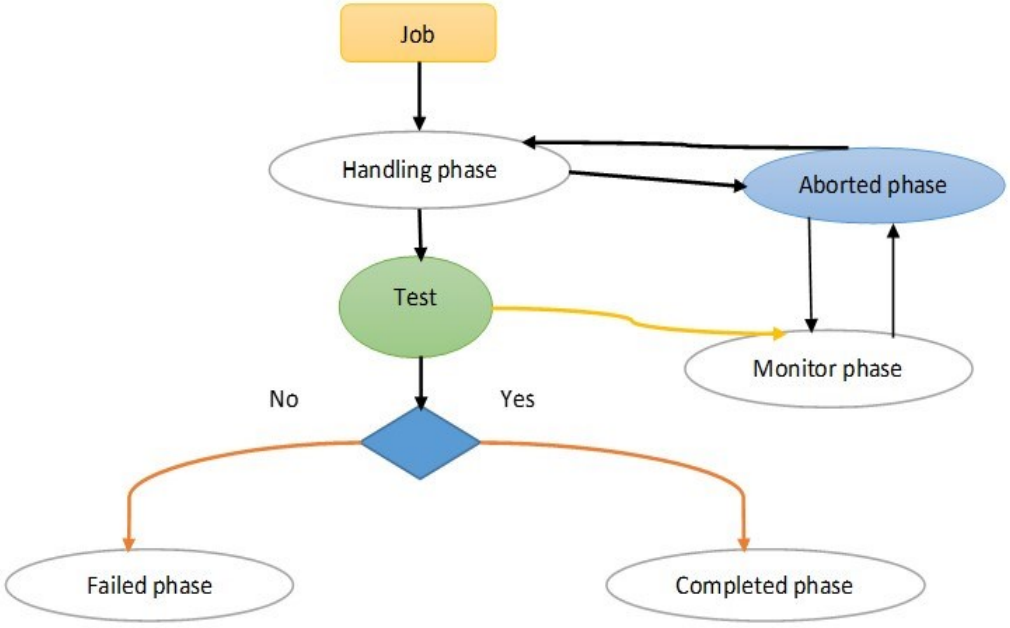


Figure 21: Data flow chart for operations inside the processing state

In test, if the execution time (t_e) ≤ the deadline time (t_d) then the task was completed successfully and is sent into the completed state. Otherwise, the task is sent into the failed state.

Handling State: receives a task from other states (Checking or Waiting), prepares all required computing resources and executes the task if possible.

Monitoring State: monitors a status of all processing tasks as Aborted, Completed or Failed. This step is done simultaneously while the task is being executed.

Aborted State: if a task is aborted "blocked" for any reason in the handling state, it is sent to the aborted state. It acts actually as a temporary memory. While the task is being held, a notification is sent to monitoring state to alert it about the new status of the task. Upon releasing from the state, another alert is sent to the monitoring state for the same purpose.

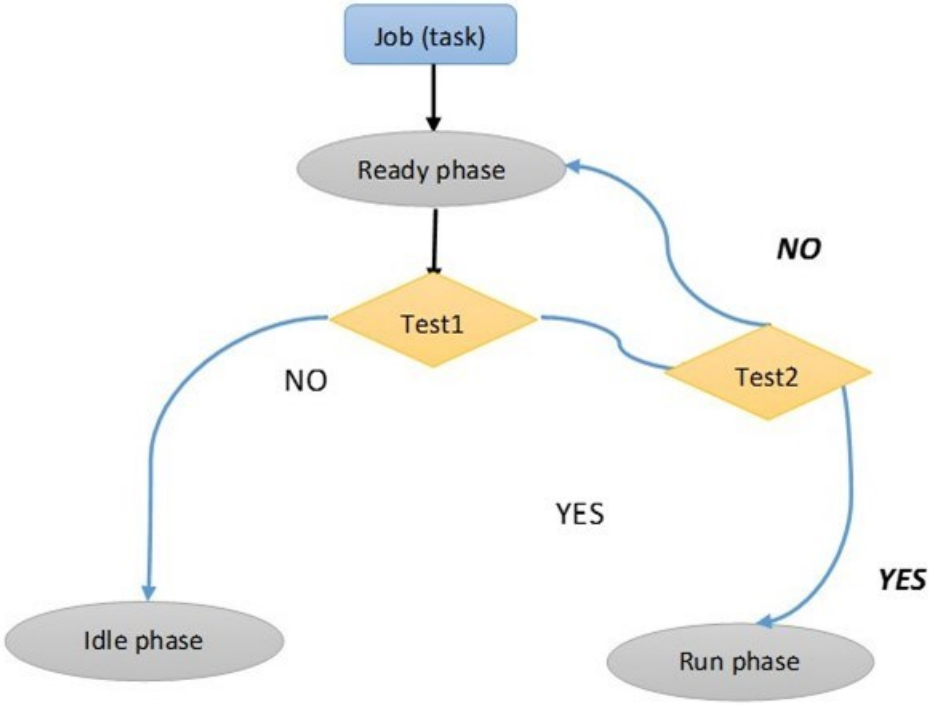


Figure 22: Data flow chart for operations inside the handling state

Tests1 indicates that the system checks if the current task will acquire the P.U. or not, if no then the current task will be sent into the idle state. If yes, then the system will check if the P.U. is available or not. If yes, the task is sent to the run state to complete the processing operation. Otherwise, the task is sent into the ready state to wait its turn when the P.U. becomes free. In the same time, checking deadline time is performed simultaneously.

Ready State: where a task is ready to be executed and waiting its turn when processing unit "P.U." becomes available.

Idle State: contains tasks which no longer need the processing unit and there is a high possibility they will go to P.U. (Processing Unit) again.

Run State: this is the place where a task is being executed by the P.U.

The general structure for the developed model is shown in fig. 23 [45].

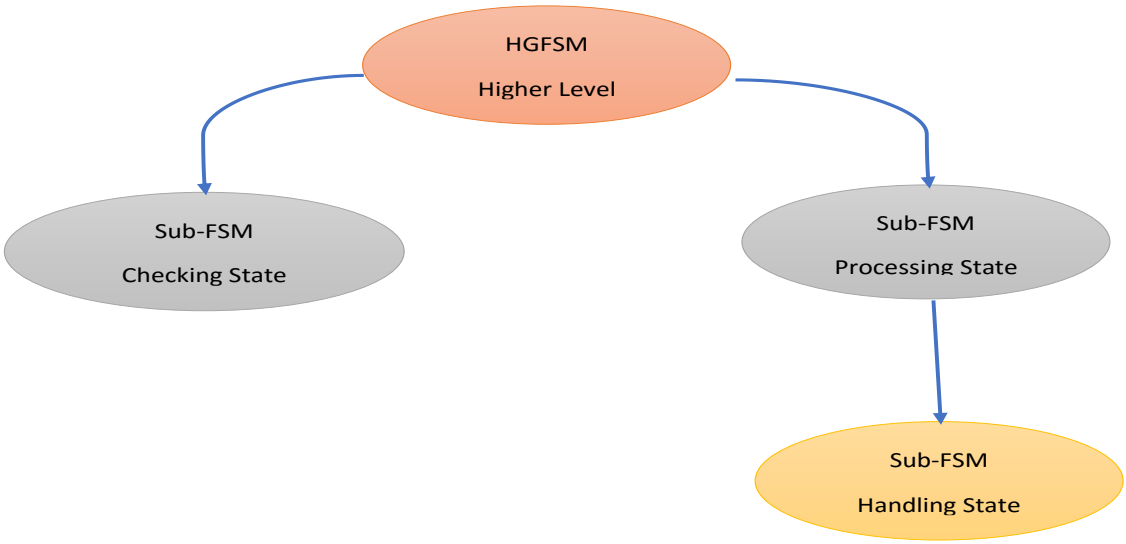


Figure 23: General structure of the HGFSM

Table 2 shows the relation transition between states according to a result of condition in the system which represents the input alphabet in a particular state. Subscript denotes the number and order of tests that have been done to decide which state should be the next one. Y and N stand for Yes and No respectively. Letter N in the initial state means that there is a malfunction in the system under consideration and no tasks can be sent into the next state. In this research we assume that the system is reliable and no malfunction occurs.

Table 2: Relation transition between different states

FROM - TO	Initial State	Checking State	Execution State	Waiting State	Failed State	Completed State
Initial State	N_1	Y_1	-	-	-	-
Checking State	-	-	Y_1, Y_2, Y_3	Y_1, Y_2, N_3	N_1	-
Execution State	-	-	-	-	N_1	Y_1
Waiting State	-	-	Y_1, Y_2	Y_1, N_2	N_1	-
Failed State	Y_1	-	-	-	-	-
Completed State	-	-	-	-	-	-

3.3 Markovian Model

Any Markovian model has 3-tuples {S, A, P}, where "S" represents a set of states that existed in the HGFSM model. "A" denotes a vector of initial probabilities values for all states in the model. While "P" contains a matrix that represents the transition probabilities between states according to some circumstances that existed in the developed model. HGFSM is converted to the Markovian model as follows [45,46]:

- Every state in HGFSM is mapped to a state in the markovian model.
- Each edge in HGFSM is converted to a transition arrow q_{ij} which represents flow direction from a current state (i) to a next state (j).
- Each a transition arrow is associated with a parameter k_{ij} which represents a number of tasks (jobs) that go from state S_i to state S_j. That parameter is used to calculate value of P_{ij}.

- Every a transition arrow has a probability value P_{ij} which denotes possibility to move from the current state to the next state and it is calculated using the following equation:

  $P_{ij} = K_{ij} / N$ (number of total tasks in state S_i).

- Each FSM graph is associated with its CSM (Computation Structure Model) to show data flow graph in it. CSM helps in constructing performance equations.
- If applicable, a state is decomposed into another an FSM and Markovian model to create a hierarchy approach.

Fig. 24 and table 3 show the Markovian model graph and probability transition matrix for the developed HGFSM model [45,46].

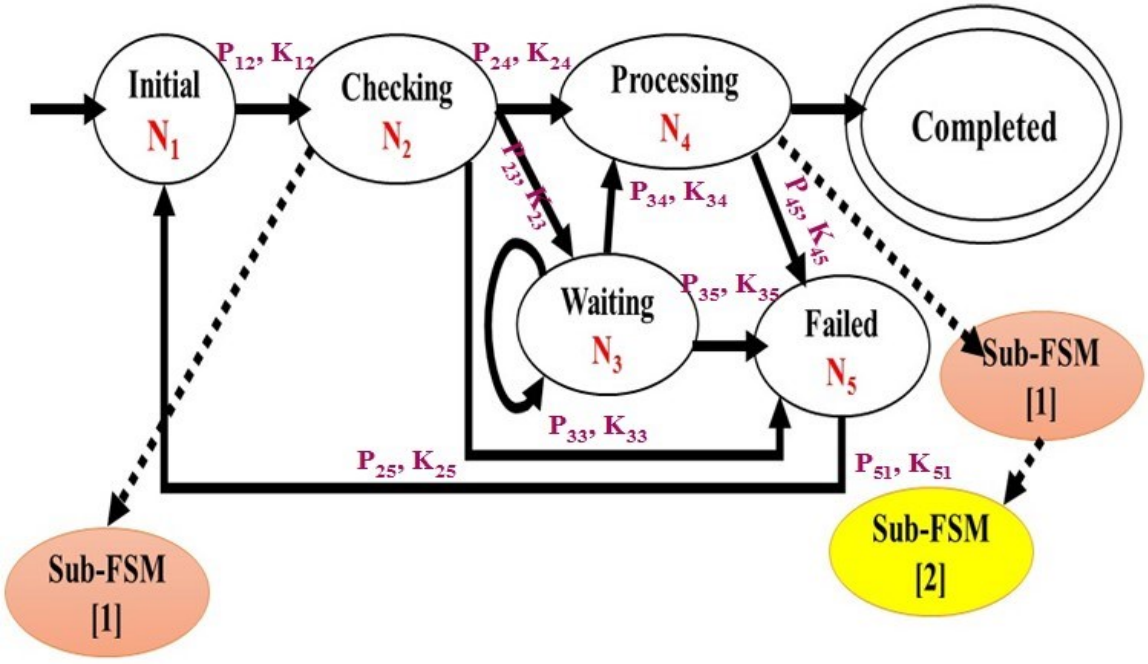


Figure 24: Markovian model graph for the HGFSM

Table 3: Probability transition matrix

FROM - TO	$INITIAL_1$	$CHECKING_2$	$WAITING_3$	$EXECUTION_4$	$FAILED_5$	$COMPLETED_6$
$INITIAL_1$	----------	$\mathbf{P_{12} = 1}$	----------	----------	----------	----------
$CHECKING_2$	----------	----------	$\mathbf{P_{23} = k_{23}/N_2}$	$\mathbf{P_{24} = k_{24}/N_2}$	$\mathbf{P_{25} = k_{25}/N_2}$	----------
$WAITING_3$	----------	----------	$\mathbf{P_{33} = k_{33}/N_3}$	$\mathbf{P_{34} = k_{34}/N_3}$	----------	----------
$EXECUTION_4$	----------	----------	----------	----------	$\mathbf{P_{45} = k_{45}/N_4}$	$\mathbf{P_{46} = k_{46}/N_4}$
$FAILED_5$	----------	$\mathbf{P_{52} = 1}$	----------	----------	----------	----------
$COMPLETED_6$	----------	----------	----------	----------	----------	----------

3.4 Hierarchical Performance Modeling

The **H**ierarchical **P**erformance **M**odel "*HPM*" can be illustrated as shown in fig.25 [45].

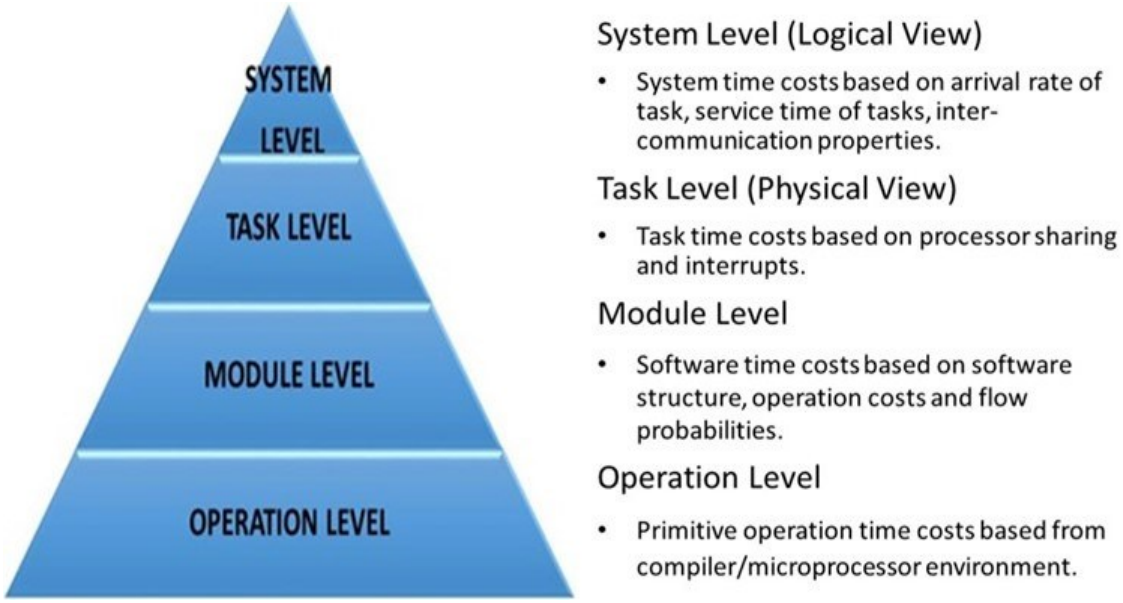


Figure 25: Hierarchical performance modeling

A. **SYSTEM LEVEL:** is located at the top layer and represents a logical view of the system (both hardware and software components); can be represented using a

queueing model. Its length is assumed to be infinite and is composed of two essential views (elements), 1. An application view and 2. A node view. The application view shows a global picture of the software system under investigation and represents communications and interactions between software processes which can be illustrated as shown in the following figure [1,2,3].

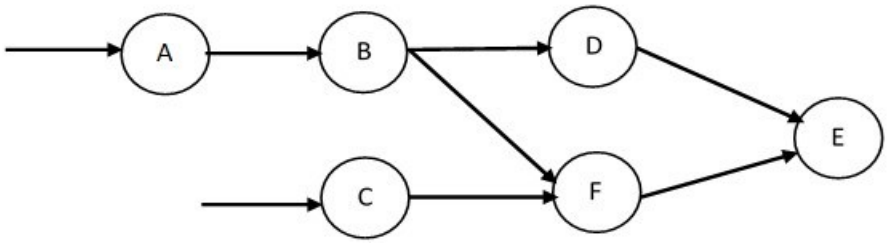


Figure 26: Application view

Each circle represents an Activity of the application and the links represent a flow of information from one software process to another one; software processes might be allocated to the same processor or different processors based on whether it is a single processor or multiprocessors. Whereas the node view presents more detailed information about the queueing properties associated with each software process such as the arrival rates and the deadline times. The arrival rates for each task, software process service rates, message multipliers (indicate number of messages departing for each message processed), number of classes and flow probabilities for each class are performance parameters which will be used to derive performance equations. The following graph shows a general detailed overview of the Node View [1,3].

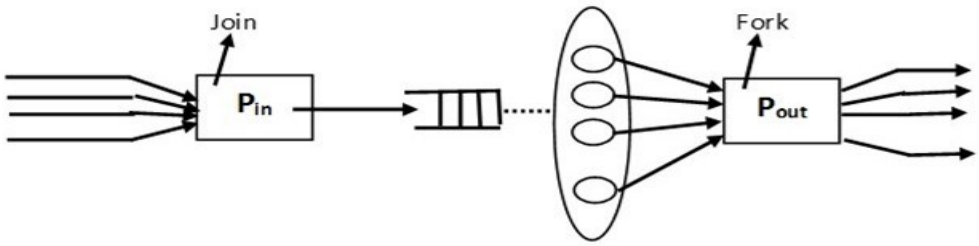


Figure 27: Node view

P_{in} and P_{out} boxes represent the flow probabilities of messages between processes for designated message classes. The queue and servers node structure represent the combined computation service delay times and communication waiting times [1].

B. TASK LEVEL: represents the physical view of the system under investigation and mainly concentrates on the interaction between software modules being executed [1]. Each software module is assumed to be an independent process (task) which synchronizes and communicates with other processes to complete a desired job. Interruption cost is also considered and modeled in this level since they affect the computation and communication cost. The following graph shows a general overview of the task level structure [1,3].

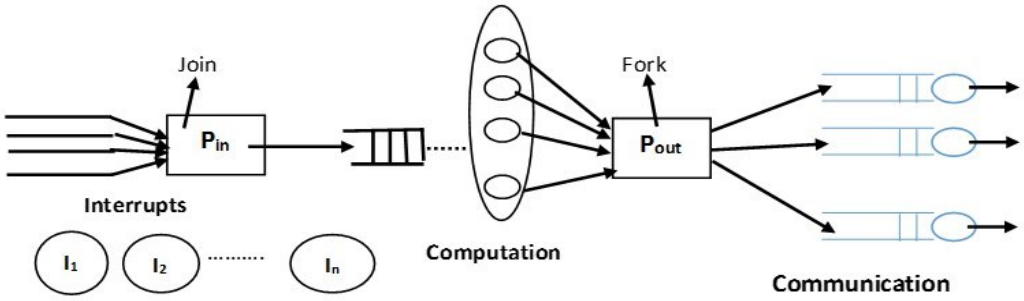


Figure 28: Task level

$\mathbf{P_{in}}$ and $\mathbf{P_{out}}$ structures represent the arriving and departing flow probabilities for the message classes on physical channels for communication purpose. This level requires usage of **processing power** which is defined to be the utilization percentage of the processor available for the execution of the assigned tasks which include interruptions and other factors and **its value is assumed to be unity** [1,2,3].

C. **MODULE LEVEL:** allows the designers to have a closer view for the specification of the software components, procedures and functions; this is known as ***Computation Structure Method (CSM)*** [1,3]. This scheme contains two essential graphs, one for ***Data Flow Graph (DFG) which is the same as data flow chart and Control Flow Graph (CFG) which shows the direction of flow inside any system***. Performance equation or a set of performance equations is generated for each CSM during generation of the performance analysis [1]. Figure 29 shows the control flow graph of the designing framework.

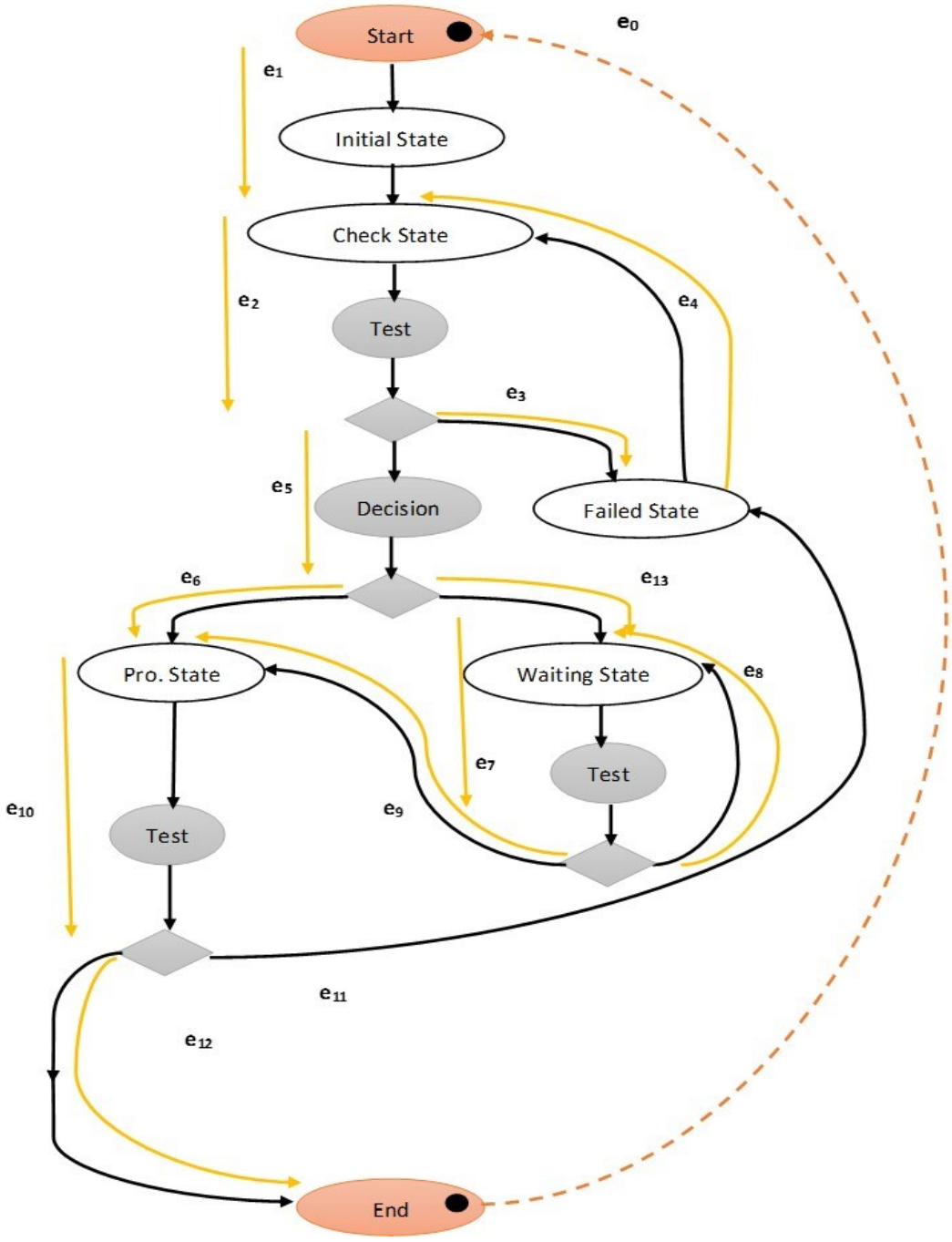


Figure 29: Control flow graph of the HGFSM

Each state in the previous CFG is associated with its flow variable(s) which is denoted by ***e***; each flow variable represents a value of moving through a path from

a start node to an end node in the CFG [45]. Each flow takes a value between {0, 1, …., ∞} and mainly depends on a type of distribution. The flows also represent the data dependent aspects of the computation time [3]. They are discrete random variables and are modeled using probability distribution and statistics methods [45,46]. Several probability distributions exist which are summarized as follows:

1. *Bernoulli*
2. *Binomial*
3. *Geometric*
4. *Modified Geometric*
5. *Poisson*

Given the probability distribution type of e, several characteristics such as Expected value E(e), second moment $E(e^2)$, Variance Var(e) and the coefficient of variation C^2 are easily obtained [45,46]. More information is found in [3].

D. OPERATION LEVEL: provides time cost measurements for the primitive operations such as addition, subtraction, multiplication and so on [3]. It also provides the time cost measurements for built-in functions such as sin, cos, sqrt and also function calls and arguments passing as specified in each software component [1]. All values are obtained from hardware manufacturer's specifications or through actual experiments [1]. The primitive operations depend on different factors such as:

- ✓ **Compiler**

- ✓ **Optimization settings**
- ✓ **Operating System**
- ✓ **Platform Profile Parameters**

Fig. 30 and fig. 31 depict the CFG for the sub-FSM inside the checking and the processing.

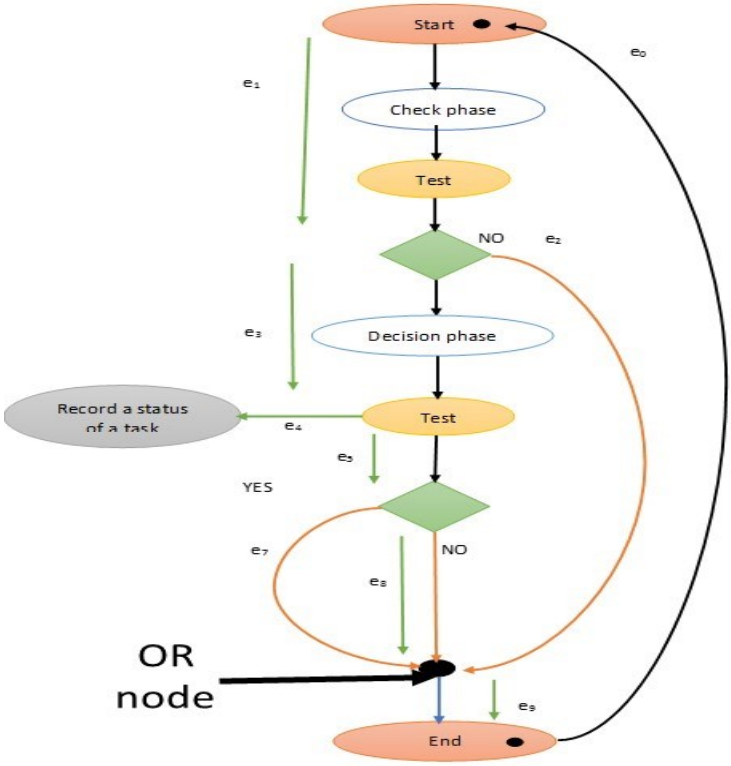


Figure 30: Control flow graph of sub-FSM inside the checking state

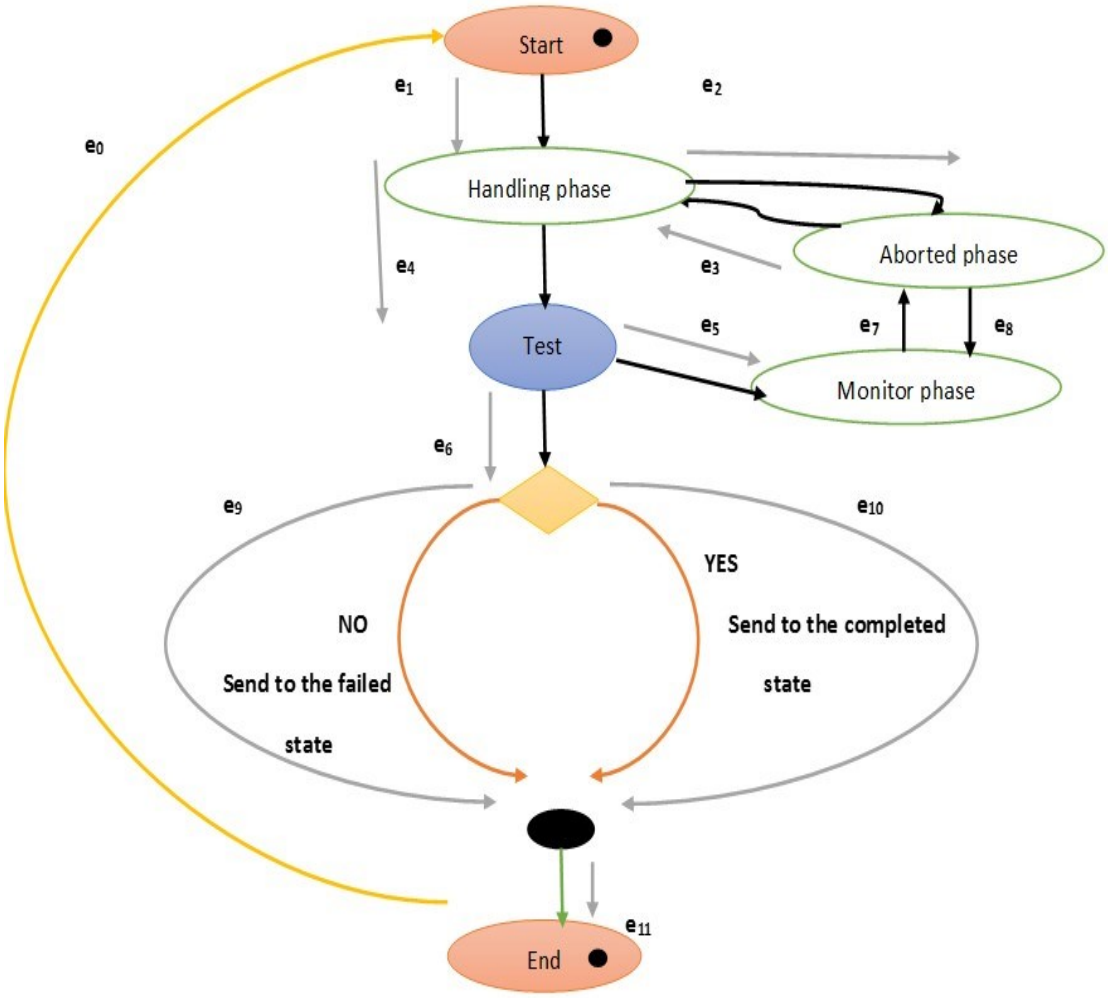


Figure 31: Control flow graph of sub-FSM inside the processing state

Fig. 32 depicts the CFG chart inside the handling state.

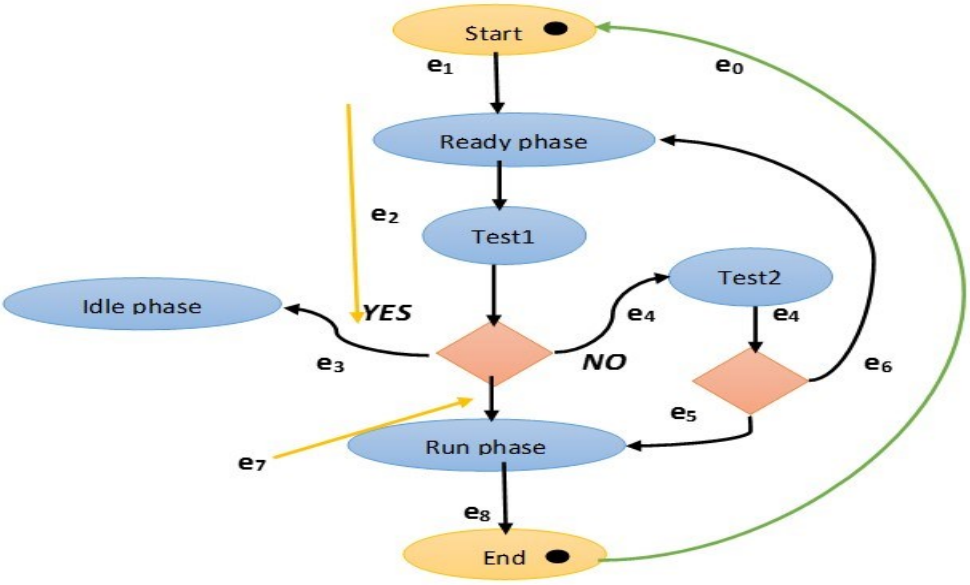


Figure 32: Control flow graph inside the handling state

3.5 Case study

In this section, the designing framework was applied on two embedded systems which were Android and OPENWRT. The objectives from both cases are to 1. Estimate the average system performance metrics “response time and power consumption” and 2. Show the validation or effectiveness of the designing framework. The results obtained from it are close to the average actual results. Maximum error is about 12% which is considered acceptable since many factors affected the results.

For Android, JAVA eclipse was used to determine several performance parameters for several primitive operations after conducting multiple experiments in order to compute the expected average performance metrics. In addition, a profiler called Trepn was also used to estimate the expected average power consumption. Over 100 runs were performed with the average duration time being about 45s for each application on each platform.

For OPENWRT, a simulator named OPNET was used to find the average response time and also to determine several performance parameters for different primitive operations.

Several Android platforms were used in the experiments to estimate the average performance metrics as listed in table 4 [46]. The platforms range from smartphones to a tablet [46].

Table 4: List of Android platforms

Device name	CPU	GPU
Galaxy Note3 N9000	2 Cortex-A15 (1.9Ghz) and 2 Cortex-A7 (1.3Ghz)	Mali-T628 MP6 (480Mhz)
Galaxy Tab3 “7 inches” PXA986	Cortex-A9 (1.2Ghz)	PowerVR SGX540 (200Mhz)
Galaxy S4 I9500	2 Cortex-A15 (1.6Ghz) and 2 Cortex-A7(1.2Ghz)	PowerVR SGX544MP3 (544Mhz)
Galaxy S4 mini I9190	Snapdragon 400(1.7Ghz)	Adreno 305 (450Mhz)

Four different applications were used to profile the performance parameters in order to compute the desire metrics, the applications were as follows:

- Audio recording.
- Calculator.
- Mobibench.
- Norvigtorious.

The first two applications are self-programmed while the remaining applications are the benchmark ones used to profile system performance on Android devices.

3.5.1 Android

The developed HGFSM with a small adjustment can be partitioned to 3 parts as shown in the following diagram. Each part is associated with its state or a set of states existed in Android activity lifecycle.

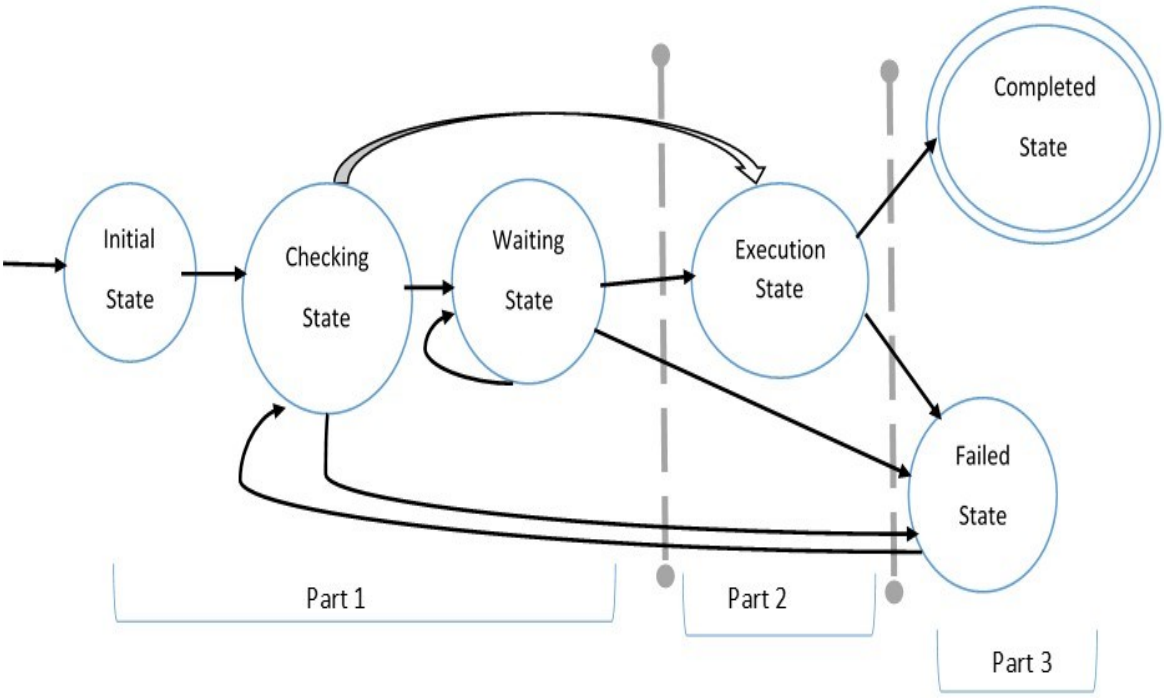


Figure 33: Partitioning of the designing HGFSM

Part 1 is linked to the following states in Android activity lifecycle:

- OnCreate.
- OnStart.
- OnRestart

Surrounded by rectangle in Android activity lifecycle diagram in fig. 34.

Part 2 is linked to the following states in the activity lifecycle:

- OnResume.
- OnPause.
- OnStop (task is still running but in the background).

Surrounded by Oval in fig. 34.

While part 3 is linked with the following state in the activity lifecycle (surrounded by diamond in Android activity lifecycle diagram in fig. 34.

- OnDestroy.
- OnStop (this state overlaps between part 2 and part 3) and it means here the task will restart; “the task goes from Failed State to Checking State” which is indicated by an oval shape in the same diagram with red background color.

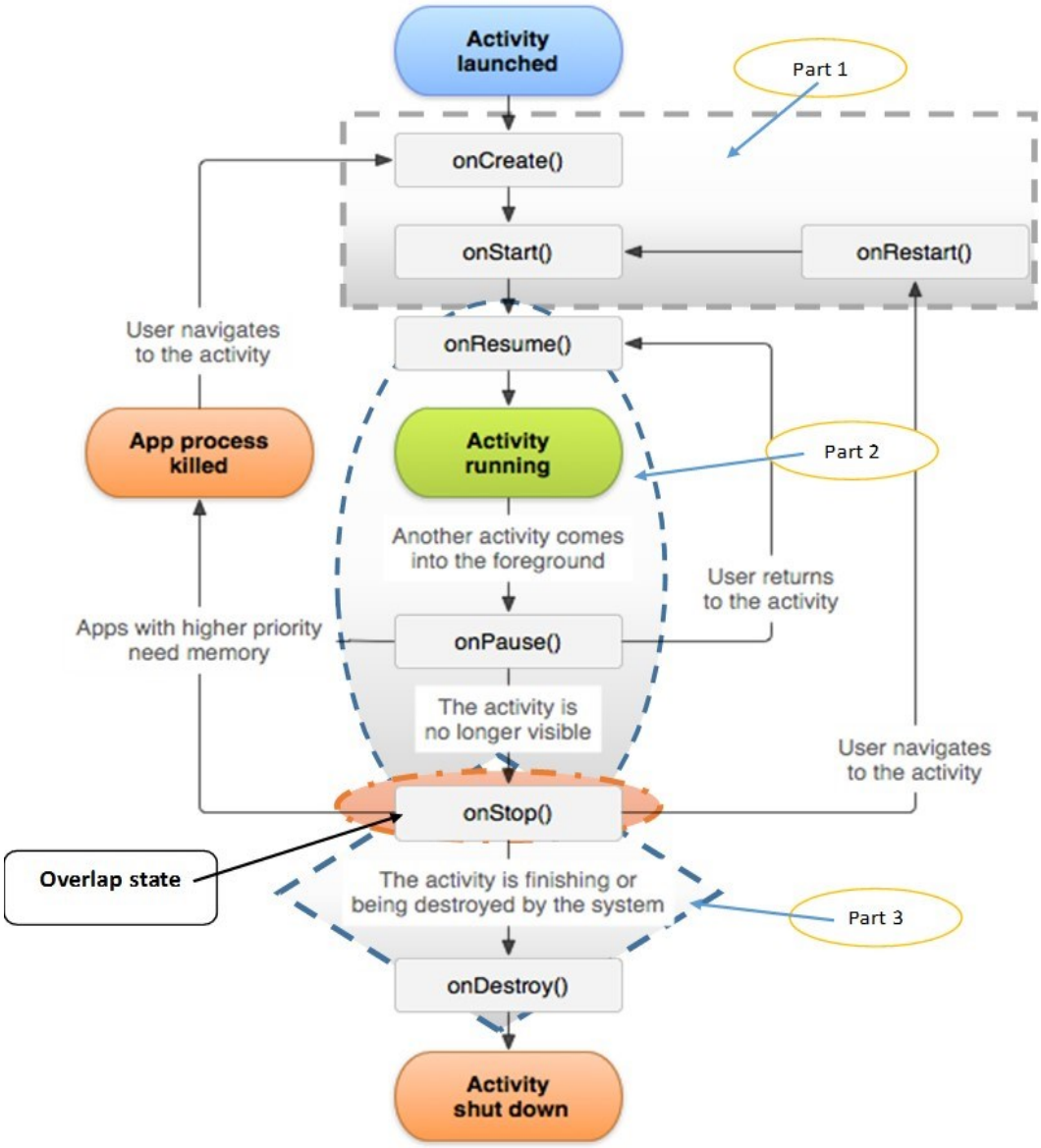


Figure 34: Linking the developed HGFSM with Android lifecycle

Table 5 shows the relation between the developed HGFSM and Android activity lifecycle model and also the possible next states in the both schemes [45].

Table 5: Mapping state between Android lifecycle with the developed HGFSM

	Current State		Next State	
	Activity lifecycle	FSM	Activity lifecycle	FSM
PART 1	OnCreate	Initial	OnStart	Checking
	OnStart	Checking	OnResume or OnStop	Failed or Waiting or Execution
	OnRestart	Suspend or Failed	OnStart	Checking or Execution or Failed
PART 2	OnResume	Execution	OnPause	Execution (Aborted)
	OnPause	Execution	OnResume or OnStop	Execution (Ready/Run) or Failed
	OnStop	Execution	OnRestart or OnDestroy	Completed or Failed
PART 3	OnDestroy	Execution	-----	Completed or Failed
	OnStop	Execution	OnRestart	Failed

Fig. 35 shows a picture of how an application starts on an Android device, either a smartphone or a tablet. Abbreviations used: T = task, Init = initial, check = checking, Exe = execution or processing and complete = completed. The application starts when a user presses on to launch it [45].

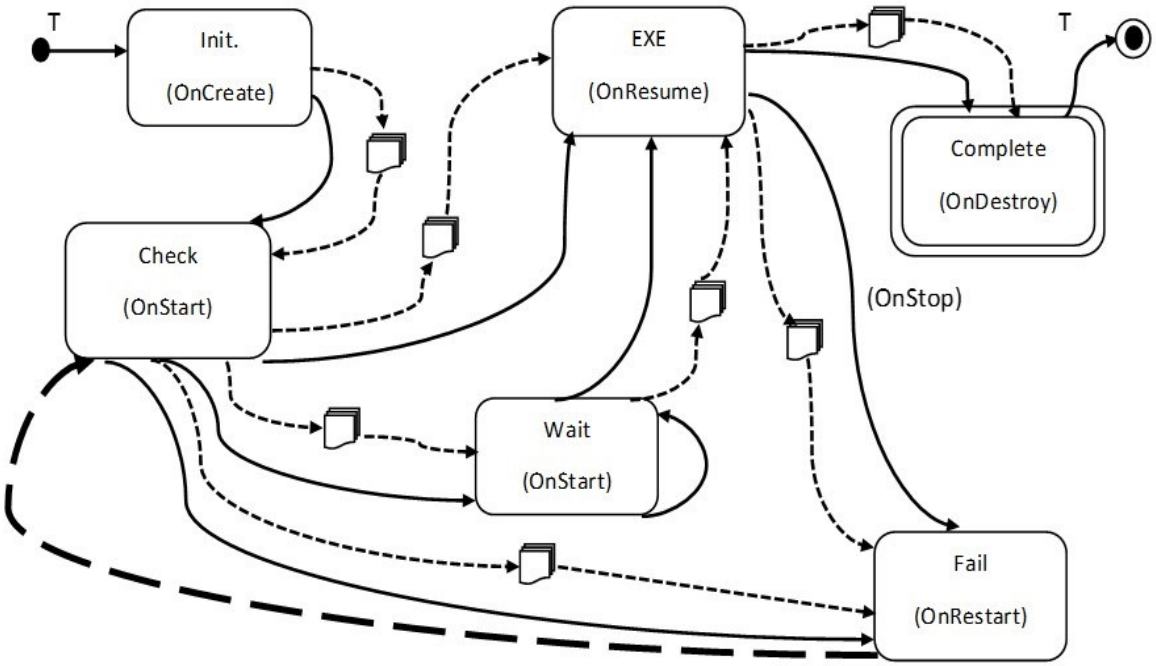


Figure 35: Typical diagram for applications on Android

In fig. 35, solid lines indicate the control flow between the states whereas the dashed lines indicate a message that is sent among states. *The bold dashed line (from Fail to check) indicates that the user tries to restart the app*. once the software processes, which are displayed as the states, and the interface messages between all states are known, our next step is to determine the performance parameters associated with the graph [45]. These parameters are:

I. Tasks arrival rates "λ_i", where i is a current state index.

II. Number of tasks "N_i" exist in each state before processing them and number of tasks "K_{ij}" move from a current state (S_i) to a new state (S_j).

III. Flow probabilities "P_{ij}" between different states.

IV. Message multipliers "β_i" which are assumed to be unity.

V. The computation and communication cost (service) times "*E[s]*" and "μ_i".

All values for the previously mentioned parameters can be obtained from a system designer or through several experiments. To utilize the performance parameters, at the early stage, we identify the input(s), output(s) and divide system into different components if possible as shown in fig. 36 [45].

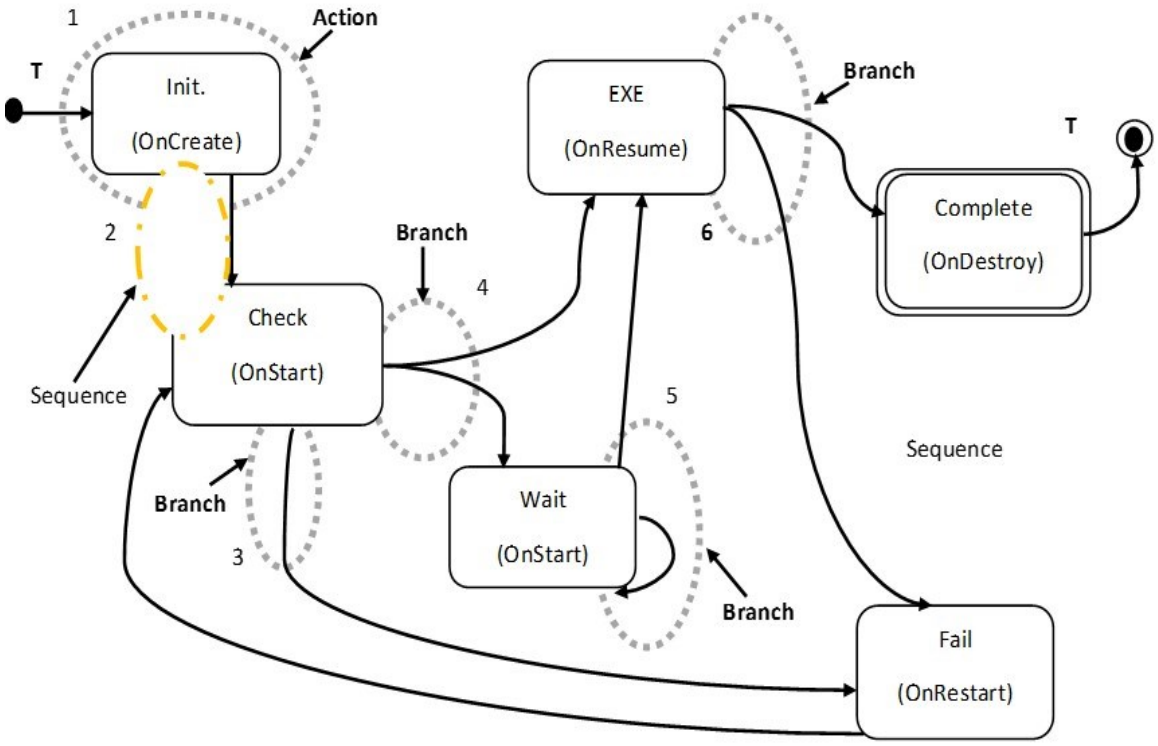


Figure 36: System components

Fig. 36 shows there is **one input, one output and six components** (*one action, one sequence and four branches*).

To determine the probability values we need to know how many tasks (N) exist first in each state and then how many tasks (k_{ij}) out of N are sent from state S_i to state S_j; all these

numbers should be known in advance either by obtaining them from actual tests (simulation) or given by the designers as stated earlier [45,46].

The probability value P_{ij} is computed as follows:

$$P_{ij} = K_{ij} / N_i \qquad (1)$$

Table 3 displays the probability equations between different states; the subscript indicates the number of the state [45].

SOFTWARE STRUCTURE SPECIFICATION

Our next step is to specify the details of the methods used to derive the performance equation [45]. The software structure indicates the order in which the operations are executed in order to complete a desired task or computation. The software structure can be seen as the ***Computation Structure Method (CSM)*** which consists of Data Flow Graph "**DFG**" and Control Flow Graph "**CFG**". The following two diagrams (flow charts) show the DFG and the CFG respectively of the Android application.

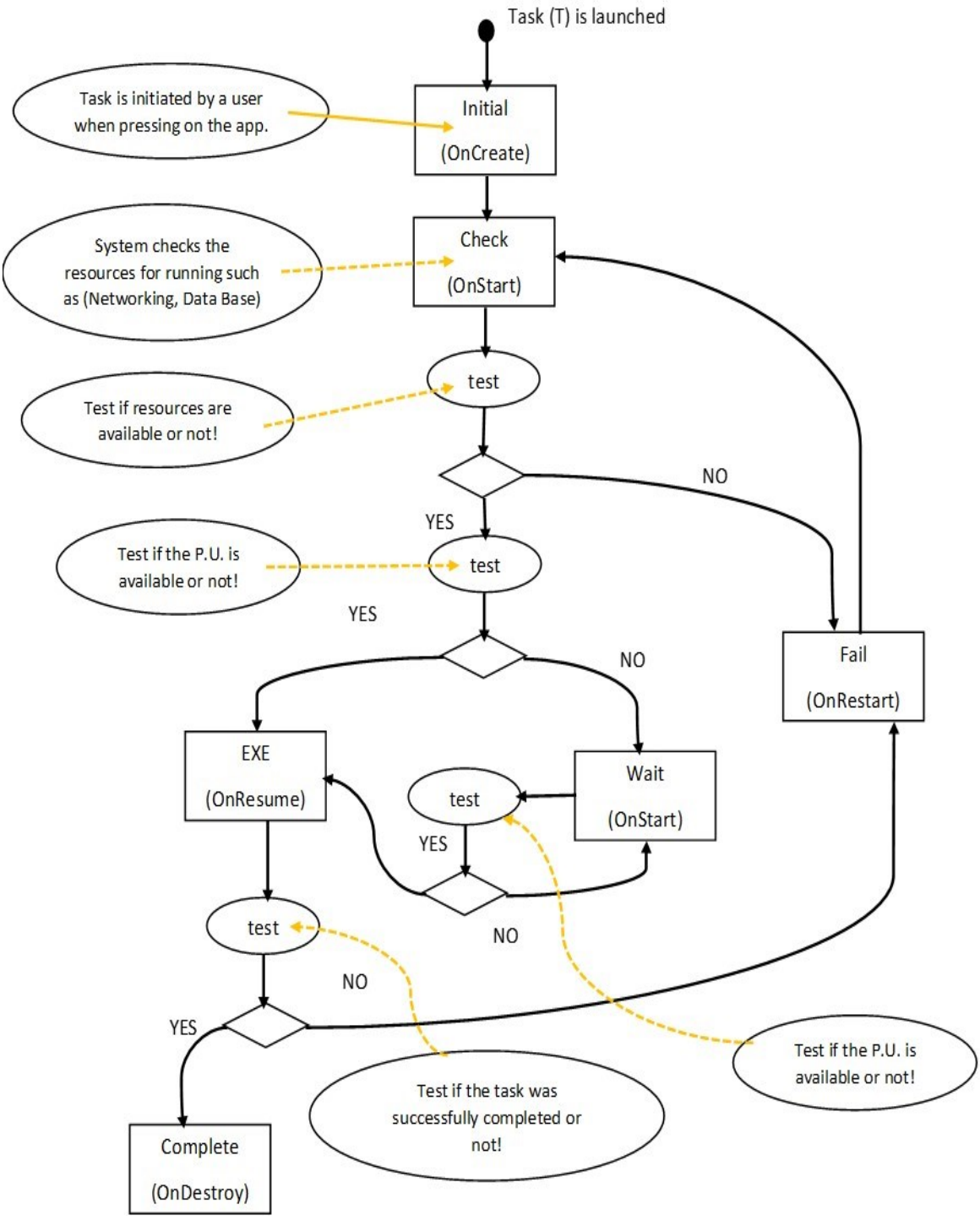


Figure 37: Data flow graph for applications on Android

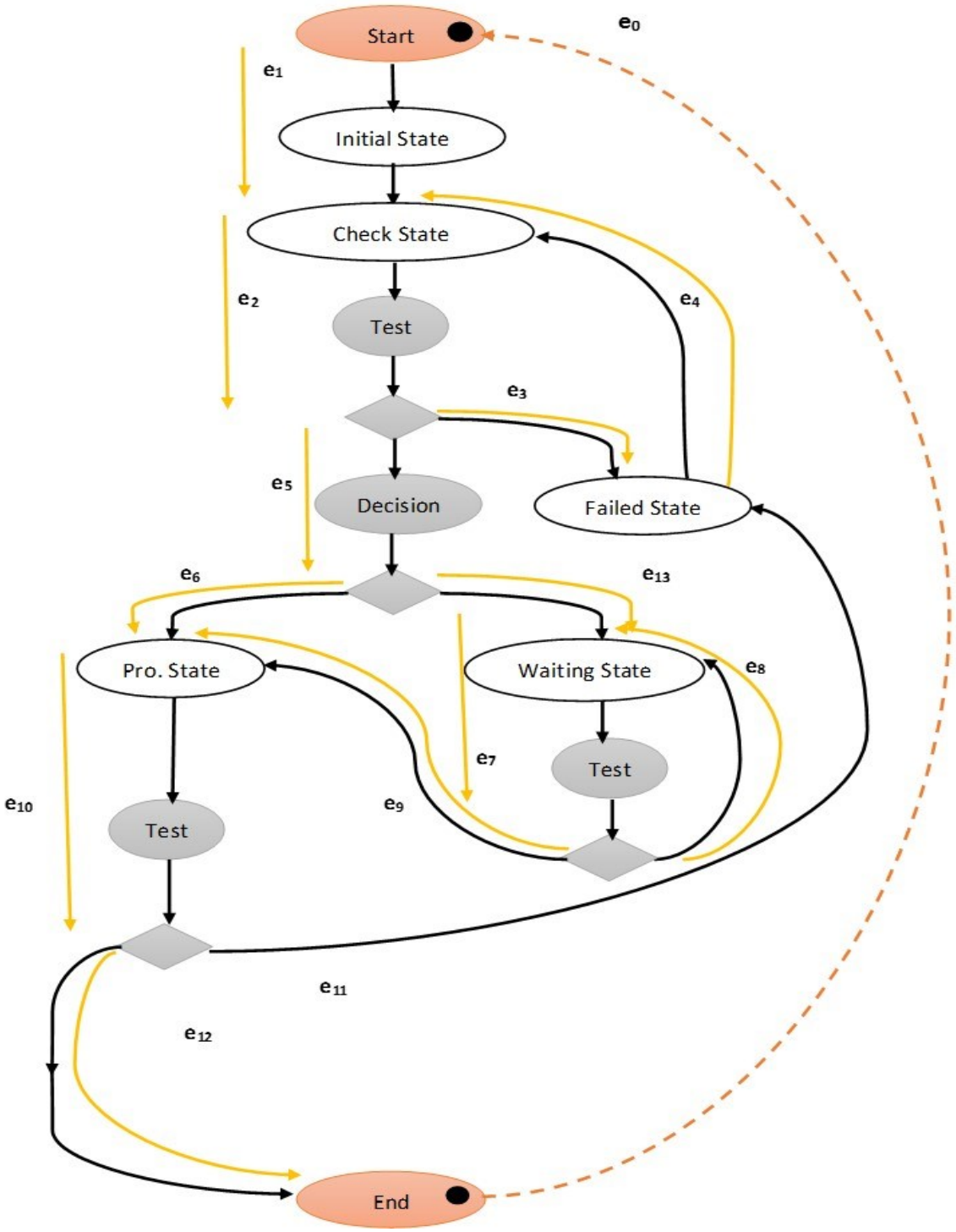


Figure 38: Control flow graph for applications on Android

From fig. 38, the obtained performance equation is as follows:

$$Cost = C = (e_1 * C_{initial}) + (e_2 * (C_{check} + C_{test})) + (e_3 * 0) + (e_4 * 0) + (e_5 * C_{decision}) + (e_6 * 0) + (e_{13} * 0) + (e_7 * (C_{wait} + C_{test})) + (e_8 * 0) + (e_9 * 0) + (e_{10} * (C_{exe} + C_{test})) + (e_{11} * 0) + (e_{12} * 0) \quad (2)$$

Eq. (2) can be written as follows after removing all zeros parts:

$$Cost = C = (e_1 * C_{initial}) + (e_2 * (C_{check} + C_{test})) + (e_5 * C_{decision}) + (e_7 * (C_{wait} + C_{test})) + (e_{10} * (C_{exe} + C_{test})) \quad (3)$$

Each cost "either as response time or power consumption" is associated with its flow(s) parameter(s). The flows parameters are categorized as either dependent or independent [1,3]. The dependent flows are recognized as the ones which complete loops while the independent flows are the remaining ones [3]. From the CFG in fig. 38, we can obtain the following equations:

$$e_2 = e_1 + e_4 \text{ -------- } (4)$$

$$e_3 = e_2 + e_5 = e_4 - e_{11} \text{ ------- } (5)$$

$$e_5 = e_6 + e_{13} \text{ ----- } (6)$$

$$e_6 = e_{10} - e_9 \text{ ------ } (7)$$

$$e_{13} = e_7 - e_8 \text{ ------- } (8)$$

$$e_7 = e_8 + e_9 \text{ ------- } (9)$$

$$e_{10} = e_{11} + e_{12} \text{ ------ } (10)$$

$$e_{12} = e_0 \text{ --------- } (11)$$

$$e_1 = e_0 = 1 \text{ ------- } (12)$$

Substitute the equations from (4) till (12) in equation (3) to find that

$$Cost = C = (e_0 * C_{initial}) + ((e_1 + e_4) * (C_{check} + C_{test})) + ((e_6 + e_{13}) * C_{decision}) + ((e_8 + e_9) * (C_{wait} + C_{test})) + ((e_{11} + e_{12}) * (C_{exe} + C_{test})) \quad (13)$$

The relation between independent flows ($e_0, e_4, e_8, e_9, e_{11}$) and dependent ones ($e_1, e_2, e_3, e_5, e_6, e_7, e_9, e_{10}, e_{12}$) can be determined using a spanning tree technique. The spanning tree method is tool used to obtain the relationships between dependent and independent flows [3]. Fig. 39 depicts how relations between the independent and dependent flows are constructed using the spanning tree technique. More information about the spanning tree method can be found in [3].

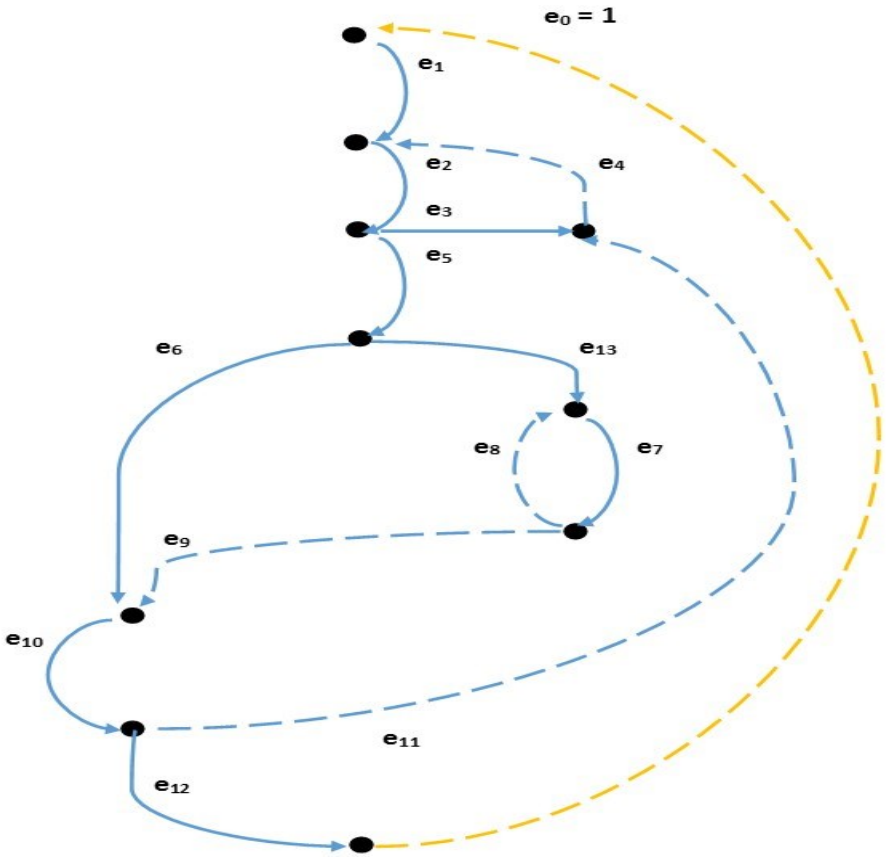


Figure 39: Spanning tree

In fig. 39, solid lines indicate dependent flows while dashed lines indicate independent flows. The spanning tree can be reduced more by removing the dashed lines, which form loops, so it becomes as shown in fig. 40. The purpose from removing the independent flows is to determine the relationships between different flows.

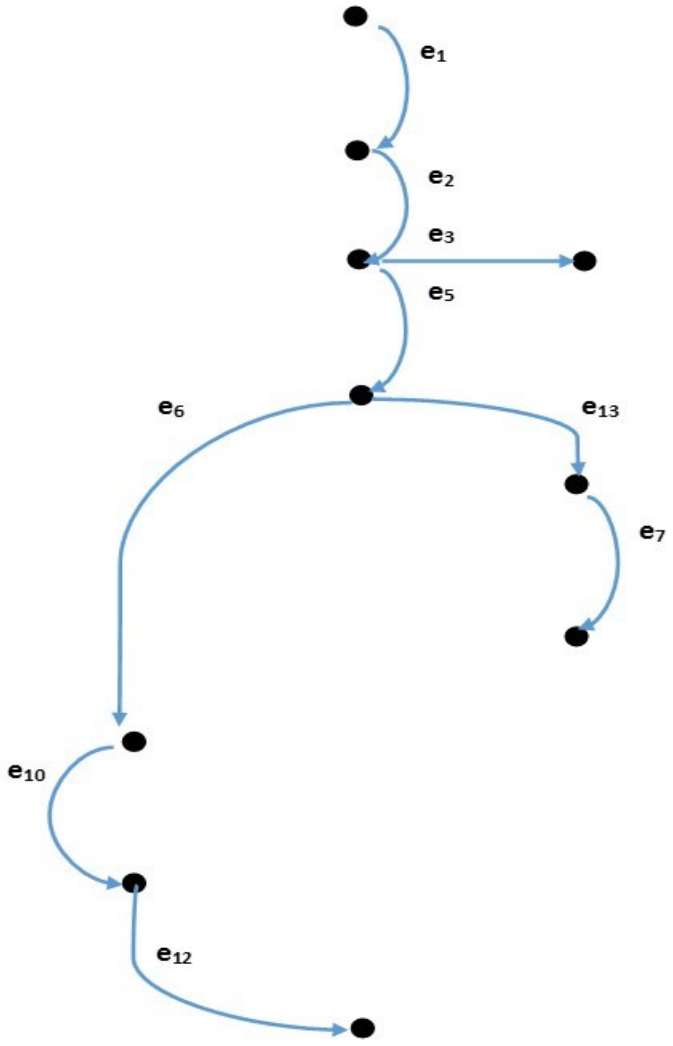


Figure 40: Modified spanning tree

The following table shows the relations between all flows “dependent and independent”.

Table 6: Relations between all flows

Dependent flows	Independent flows					Equations
	e_0	e_4	e_8	e_9	e_{11}	
e_1	e_0					$e_1 = e_0 = 1$
e_2	e_0	e_4				$e_2 = 1 + e_4$
e_3		e_4			$-e_{11}$	$e_3 = e_4 - e_{11}$
e_5	e_0				e_{11}	$e_5 = e_{11} + 1$
e_6				$-e_9$	e_{11}	$e_6 = e_{11} - e_9$
e_7			$-e_8$	e_9		$e_7 = e_9 + e_8$
e_{10}	e_0				e_{11}	$e_{10} = e_{11} + e_0$
e_{12}	e_0					$e_{12} = e_0 = 1$
e_{13}				e_9		$e_{13} = e_9$

Substitute in equation (13) to find that:

$$Cost = C = (1 * C_{initial}) + ((1 + e_4) * (C_{check} + C_{test})) + ((e_{11} + 1) * C_{decision}) + ((e_8 + e_9) * (C_{wait} + C_{test})) + ((e_{11} + 1) * (C_{exe} + C_{test})) \quad (14)$$

To find the cost of every quantity in equation (14), we construct its CSM.

1- **Initial State (OnCreate):** this is where initialization of data elements is done; assigning a thread to run the task and creating the GUI (Graphical User Interface) for the task. The DFG and CFG are and shown in the following graphs respectively.

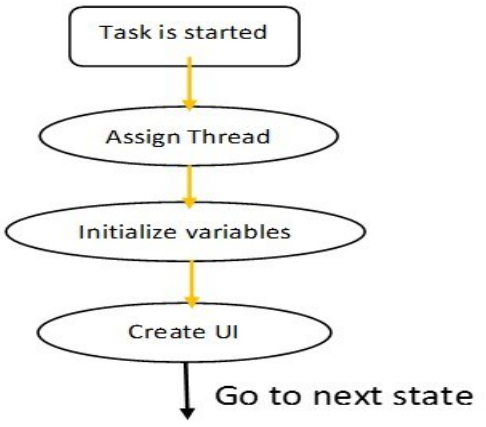


Figure 41: Data flow graph for operations inside the initial state

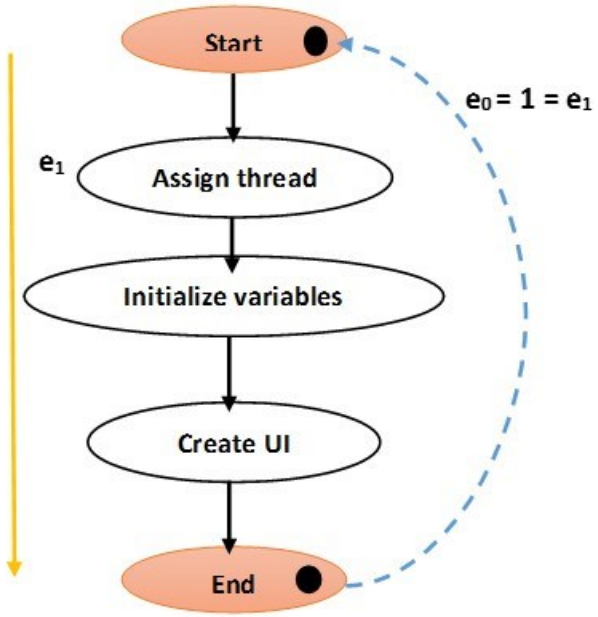


Figure 42: Control flow graph for operations inside the initial state

From fig. 42, the cost for three operations take place in the initial state is as follows:

$$C_{Initial} = (e_1 * [C_{create\ UI} + C_{Initializations} + C_{assigning\ thread}]);$$ since $e_1 = 1$; so

$$C_{initial} = C_{create\ UI} + C_{Initializations} + C_{assigning\ thread} \qquad (15)$$

2- Checking State (OnStart): checks the system resources such as Networking, Data Base inquiries and Processing Unit (P.U.) to run the task.

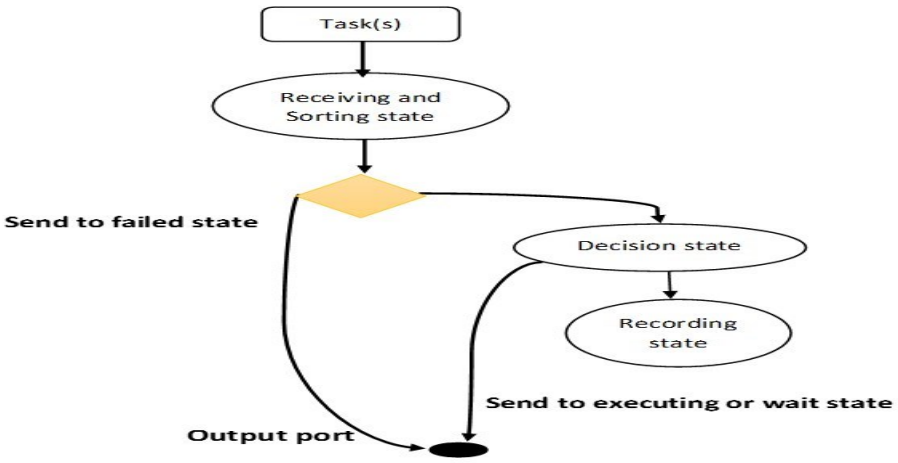


Figure 43: Data flow graph for operations inside the checking state

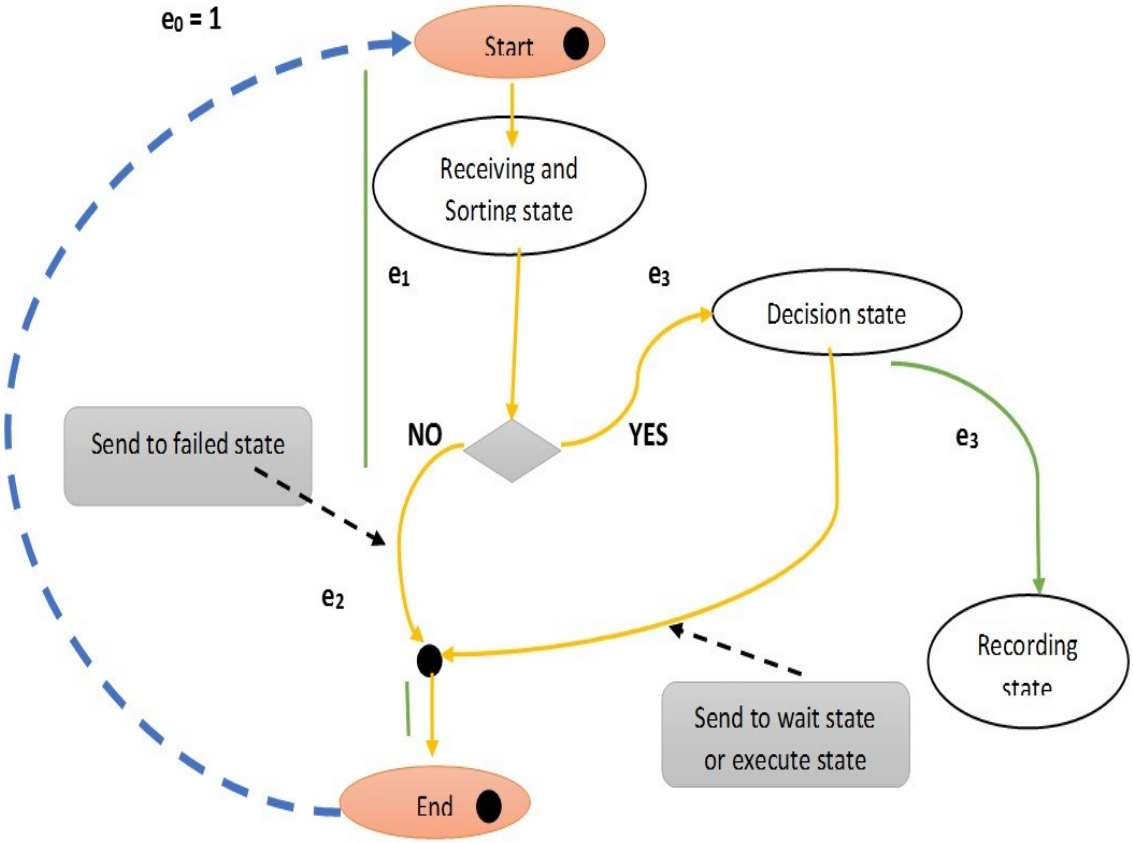


Figure 44: Control flow graph for operations inside the checking state

$$C_{check} = [e_1 * (C_{receiving\ and\ sorting} + C_{test})] + (e_2 * 0) + [e_3 * (C_{decision} + C_{test})] + (e_4 * 0) + (e_5 * 0) = [e_1 * (C_{receiving\ and\ sorting} + C_{test})] + [e_3 * (C_{decision} + C_{test})] \quad (16)$$

To find the relations between all flows, the spanning tree method is applied. The spanning tree for the previous CFG is as shown in fig. 45.

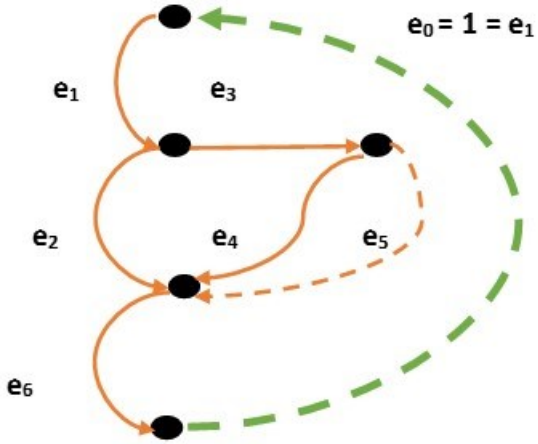


Figure 45: Spanning tree for operations in the checking state

The spanning tree in fig. 45 reduces even more by removing all independent flows as shown in the following chart.

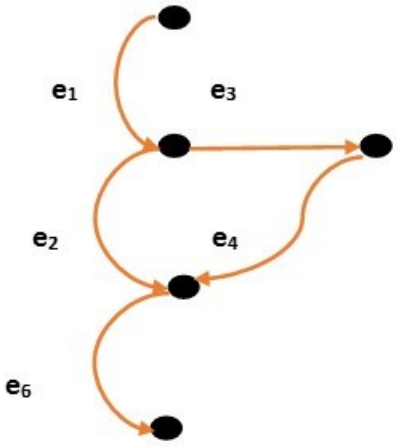


Figure 46: Reduced spanning tree of operations in the checking state

From the reduced spanning tree in fig. 46, we find that

$$e_1 = e_0 = 1$$

$$e_3 = e_0 = 1$$

Substitute the previous two equations in eq. (16) to find that:

$$C_{check} = [1 * (C_{receiving\ and\ Sorting} + C_{test})] + [1 * (C_{decision} + C_{test})] \qquad (17)$$

3 - Waiting State (OnStart): the tasks wait their turn to be executed. Its DFG and CFG are shown in the following two charts respectively.

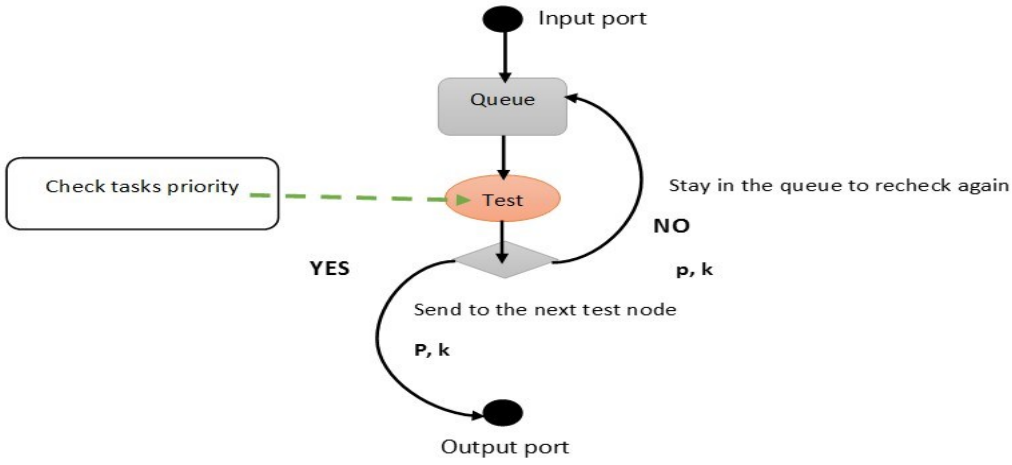


Figure 47: Data flow graph for the waiting state on Android

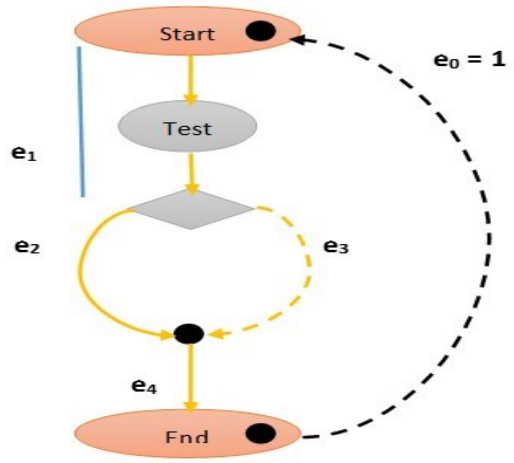


Figure 48: Control flow graph for the waiting state on Android

From CFG, the independent flows are **e_0 and e_3** while the dependent flows are **e_1, e_2 and e_4.**

$$e_1 = e_0 = 1$$

$$e_1 = e_2 + e_3 = 1$$

$$e_4 = e_0 = 1$$

$$e_2 = e_0 - e_3$$

$$C_{wait} = (e_1 * C_{test}) + (e_2 * 0) + (e3 * 0) + (e4 * 0) = C_{test} \qquad (18)$$

4- Execution State (OnResume): the activity is running and visible to the user. Its CFG is shown in fig. 49.

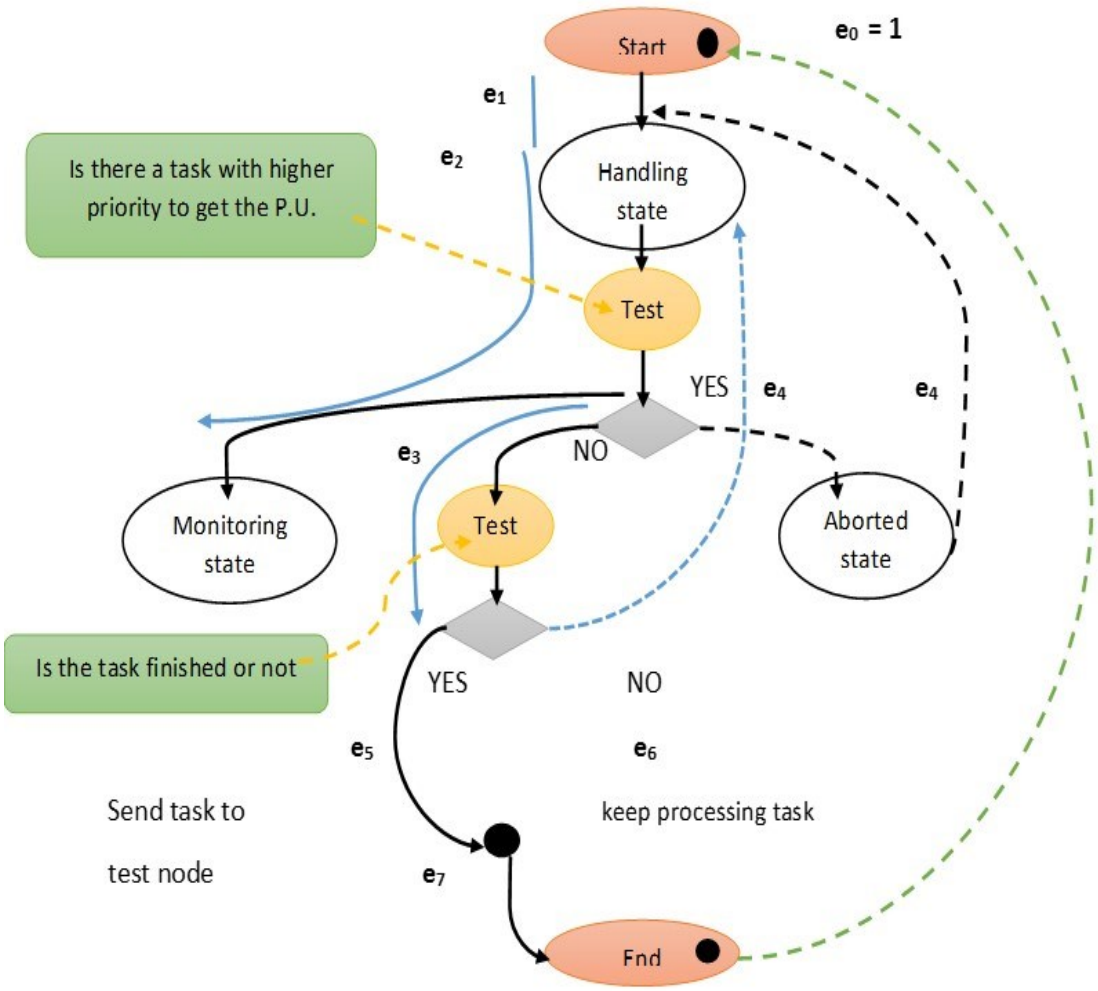


Figure 49: Control flow graph of the processing state on Android

$$C_{execution} = (e_1 * 0) + [e_2 * (C_{Handling\ state} + C_{test})] + (e_4 * C_{aborted}) + (e_3 * C_{test}) + (e_5 * 0) + (e_6 * 0) + (e_7 * 0) = [e_2 * (C_{processing\ state} + C_{test})] + (e_4 * C_{aborted}) + (e_3 * C_{test})$$

From the CFG in fig. 49,

$$e_1 = e_0 = 1$$

$$e_2 = e_1 + e_4 = e_3 + e_4$$

$$e_3 = e_5 + e_6$$

$$e_7 = e_5$$

$$e_3 = e_0 + e_4$$

$$C_{execution} = [(e_1 + e_4)*(C_{Handling\ state} + C_{test})] + [e_4 * C_{aborted}] + [(e_0 + e_4) * C_{test}] \qquad (19)$$

The CFG for the sub-FSM "Handling state" inside the processing state is shown in fig. 50.

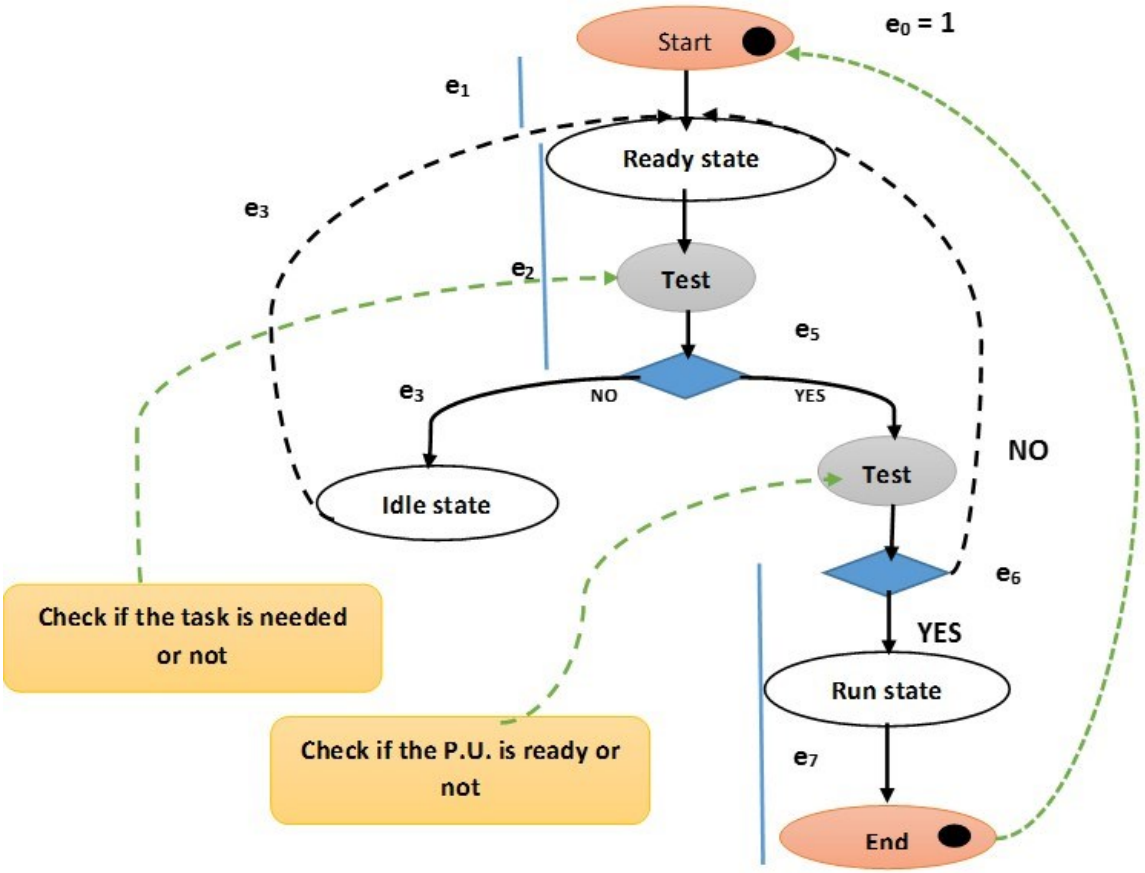


Figure 50: Control flow graph of the Handling state on Android

$$C_{Handling} = (e_1 * 0) + (e_2 * (C_{ready} + C_{test})) + (e_3 * C_{idle}) + (e_5 * C_{test}) + (e_6 * 0) + (e_7 * (C_{run}$$

$$)) = (e_2 * (C_{ready} + C_{test})) + (e_3 * C_{idle}) + (e_5 * C_{test}) + (e_7 * (C_{run}))$$

Only required flows are determined.

$$e_1 = e_0 = 1$$

$$e_2 = e_1 + e_3 + e_6$$

$$e_5 = e_6 + e_7$$

$$e_7 = e_0 = 1$$

The cost associated with the processing state can be computed as follows after substituting the cost associated with the handling state.

$$C_{Handling} = [(1 + e_3 + e_6) * (C_{ready} + C_{test})] + (e_3 * C_{idle}) + [(e_6 + 1) * C_{test}] + (1 * (C_{run})) \quad (20)$$

Now substitute equations (15), (16), (17), (18), (19) and (20) into eq. (14) to derive the performance equation which become as follows:

$$C = [C_{create\ UI} + C_{Initializations} + C_{assigning\ thread}] + [(1 + e_4) * (C_{test} + [1 * (C_{receiving\ and\ sorting} + C_{test})] + [1 * (C_{decision} + C_{test})])) + ((e_8 + e_9) * (C_{test} + C_{test})) + ((e_{11} + 1) * C_{test}) + (1 * C_{complete}) + ((e_{11} + 1) * ([[(e_1 + e_4) * ([[(1 + e_4 + e_6) * (C_{ready} + C_{test})] + (e_3 * C_{idle}) + [(e_6 + 1) * C_{test}] + (1 * (C_{run}))] + C_{test})] + [e_4 * C_{aborted}] + [(e_0 + e_4) * C_{test}]] + C_{test})) \quad (21)$$

Now finding number of visits to each existing state; it is computed using the following equation:

$$[V] = (I - P)^{-1} \quad (22)$$

Where [V] is a matrix whose elements indicate number of visits to each state; the number of its entries is equal to the number of states exist in the system. I is the identity matrix and P is the matrix of transition probabilities between all states in the steady state. The steady state probability is computed using the following formula:

$$[P] = [P] * P^0 \quad (23)$$

P^0 refers to initial probability values. Solving previous equation using linear algebra gives the value of steady state probability for each state; substitute it in eq. (21), we obtain the values of [V]. Matlab is used to obtain the values of [V]. So

$$The\ Average\ performance = \sum (V_i * C_i) \qquad (24)$$

i = 1 , …….. ,6 which is number of states in the designing framework model; C_i indicates the value of cost associated with each state. By substituting eq. (24) into eq. (21) we can determine the average performance metrics.

APPLICATION PROFILING

The profiling was done in three parts according to the developed HGFSM which contains:

- Initial part (part one in the developed HGFSM): represents the first stage toward finding the execution time for a task. This stage contains "Initial state" in the HGFSM and "OnCreate" in Android Activity Lifecycle.
- Check part (part one in the developed HGFSM): represents the second stage and contains two states which they are (Checking State and Waiting state in the HGFSM) and (OnStart) in the Android Activity Lifecycle.
- Run part (part two and three in the HGFSM): represents the last stage and contains the following states: Execution, Completed and Failed.

The aims of this profile are to spot the bottleneck(s) of Android applications in different architectures using the same applications and to compute the average response time and/or power consumption. All applications were tested several times (about 30 to 45 times) and then the average time is determined using equations (21) and (24). In each platform, all four applications were installed and then the profiling started by launching them one by one.

Two different tracing schemes exist in the Android Developing tools which they are:

A. TraceView: is a graphical tool for execution log app. which is created by debugging class in order to trace the performance of the execution code. Two approaches are available in the TraceView tool:

- Timeline Panel: describes when each thread and method starts and stops; the following figure shows how the timeline panel looks like after the execution.

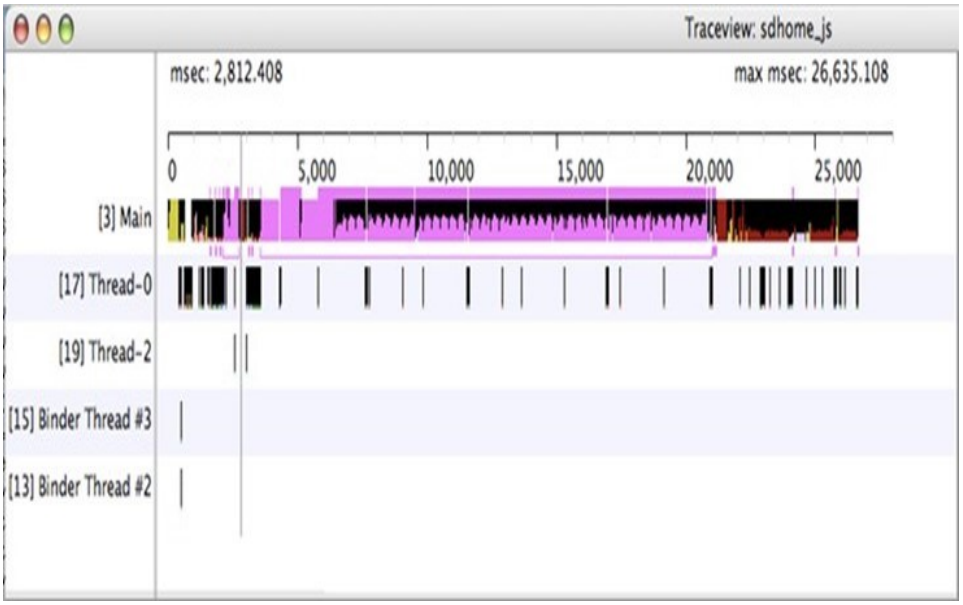


Figure 51: Timeline panel in Android debugging tool

- Profile Panel: provides a summary of what happens inside each method. The following figure shows how the profile panel looks like.

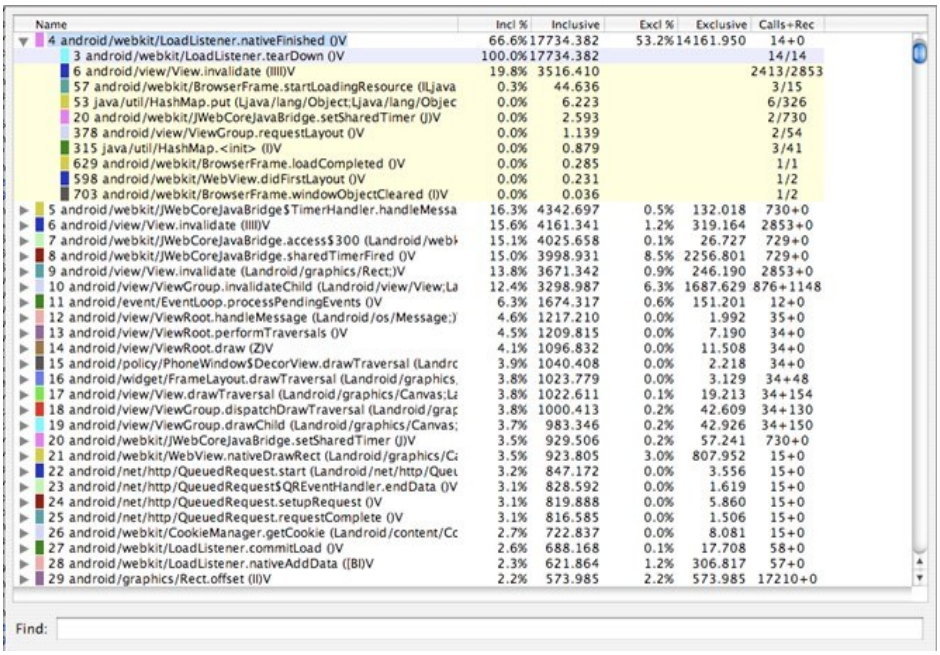


Figure 52: Profile panel in Android debugging tool

Both Trace Views were used in the conducted experiments to get the results the performance parameters.

B. Dmtracedump: is a tool that provides an alternative way to show trace log files. It represents the trace as a tree diagram; it shows the trace flow from a parent node to it's a child node using arrows as shown in the following figure.

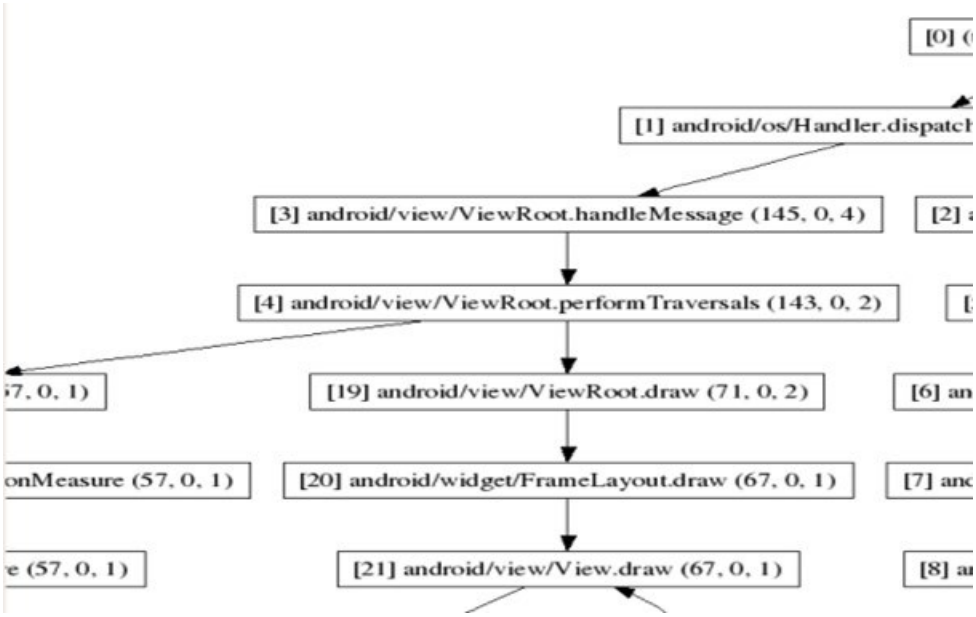


Figure 53: Dmtracedump view

Each node contains several fields which represent: 1. Call Reference Numbers as used in the trace file. 2. Inclusive Elapsed time in milliseconds spent in a method and all child methods are included as well. 3. Exclusive Elapsed time in milliseconds spent in the method without including all child methods. 4. Number of Calls for each method.

We will do analytical analysis for single task on Galaxy Note 3 on Audio Recording application to estimate the average response time for simplicity in order to make it easy for readers to understand procedures. The following performance parameters are assumed as follows:

$\lambda = 1$ task, $E[s] = 1/CPU = 1/1.3Ghz = 0.000769 * 10^{-6}$ ms; there are 4 CPUs already existed in Galaxy Note 3. 2 CPUs are dedicated for heavy computations while the remaining 2 are dedicated for regular computations. In normal mode, only 1 CPU takes control of every

operation in order to save power and energy, the second CPU works when multiple tasks option is enabled.

Message size = M = 2000 B, R = Bandwidth = BUS Speed * Bus Width (Number of bits) = 32 (bits) * 1300MB/s = 41600 MB/s = 41,600,000,000 B/s. $\beta = 1$, PP (Processing Power) = 1, C_{test} = 0.0341 ms. JAVA eclipse indicated that a value for C_{test} is slightly different between all four platforms, so we assume that it is equal in all platforms. We assume equally likely for a branch to be taken so $p = q = 0.5$ since $p + q = 1$.

Several performance values for different primitive operations were obtained from the conducting experiments as shown in table 7 in ms.

Table 7: Elapsed time for primitive operations

Primitive Operation	Elapsed Time
Calling function	2.748
Passing argument	2.294
Addition	167.389
Subtraction	166.357
Multiplication	491.337
Division	431.819
Power	498.995

Inside initial state: three operations take place as stated earlier. JAVA eclipse determined that:

Assigning thread took 3%, variables initializations took 22% and GUI took 75% of a total time assigned to the initial state. **Dmtracedump showed that the first two operations occurred only once while the last operation occurred 4 "this number indicates number of calls to start the GUI element"**.

$C_{initial} = C_{create\ UI} + C_{Initializations} + C_{assigning\ thread} = (4 * 7.875 = 31.5) + 9.24 + 1.26 = 42$ *ms.*

Inside checking state:

$C_{check} = [\ 1\ * (C_{receiving\ and\ Sorting} + C_{test})\ ] + [\ 1\ * (C_{decision} + C_{test})\ ];$ each quantity in the checking equation was ***called only once*** as determined by the JAVA eclipse. The software profiler was unable to distinguish between the two values of $C_{receiving\ and\ Sorting}$ *and* $C_{decision}$. We use the summation of both quantities together.

$C_{check} = [\ 1\ * (C_{receiving\ and\ Sorting} + C_{test})\ ] + [\ 1\ * (C_{decision} + C_{test})\ ] = 8.243 + (2$*0.0341 ms) = 8.3112 ms.

Inside waiting state: no operation took place in this state since the task was sent to the processing state immediately.

Inside processing state:

$C_{execution} = [(e_1 + e_4) * (C_{Handling\ state} + C_{test})] + [e_4 * C_{aborted}] + [(e_0 + e_4) * C_{test}];$

$C_{aborted} = 0$, *since the task was not aborted.*

$C_{test} = 0.0341$

$e_0 = 1$ *"determined from CFG in fig. 49.*

$e_1 = 1$ *"determined from CFG in fig. 49.*

$e_4 = 0,$ this flow occurs from returning a task from failed state into the checking state. However, the task was completed successfully so $e_4 = 0$ as shown in fig. 38. Now to find the cost associated with the Handling state.

$C_{Handling} = [(1 + e_3 + e_6) * (C_{ready} + C_{test})] + (e_3 * C_{idle}) + [(e_6 + 1) * C_{test}] + (1 * (C_{run}))$

From fig. 50, $e_3 = 0$ since it occurs from going to the idle state and returning back to the ready state. The task was not sent to the idle state due to the fact that no other task requested the CPU.

$e_6 = 0$ since the P.U. was ready and took control of the task. So cost equation for the Handling state becomes as follows:

$C_{Handling} = [(1) * (C_{ready} + C_{test})] + [(1) * C_{test}] + (1 * (C_{run})) = 3792.074\ ms$ "represents the duration time for recording".

By substituting the value of $C_{Handling}$ into the cost processing state equation to find that:

$C_{execution} = [(e_1 + e_4) * (C_{Handling\ state} + C_{test})] + [(e_0) * C_{test}];$

$e_4 = 0$

$C_{execution} = 3792.074 + 0.0341 + 0.0341 = 3792.1422\ ms.$

Table 8 shows the values for probability transition between all states.

Table 8: Probability between different states on Galaxy Note 3

FROM - TO	$INITIAL_1$	$CHECKING_2$	$WAITING_3$	$EXECUTION_4$	$FAILED_5$	$COMPLETED_6$
$INITIAL_1$	0	**1**	0	0	0	0
$CHECKING_2$	0	0	**$P_{23} = 0$**	**$P_{24} = 1$**	**$P_{25} = 0$**	0
$WAITING_3$	0	0	**$P_{33} = 0$**	**$P_{34} = 0$**	0	0
$EXECUTION_4$	0	0	0	0	**$P_{45} = 0$**	**$P_{46} = 1$**
$FAILED_5$	0	**$P_{52} = 0$**	0	0	0	0
$COMPLETED_6$	0	0	0	0	0	0

To get the number of visits in each state, we substitute in eqs. (22) and (23), where $[V] = (I - P)^{-1}$, we use Matlab to obtain the results as shown in table 9 as shown below:

Table 9: Number of visits in each state

FROM - TO	INITIAL$_1$	CHECKING$_2$	WAITING$_3$	EXECUTION$_4$	FAILED$_5$	COMPLETED$_6$
INITIAL$_1$	1	**1**	0	1	0	1
CHECKING$_2$	0	1	**0**	**1**	**0**	1
WAITING$_3$	0	0	**1**	**0**	0	0
EXECUTION$_4$	0	0	0	1	**0**	**1**
FAILED$_5$	0	**0**	0	0	1	0
COMPLETED$_6$	0	0	0	0	0	1

Substitute into eq. (24) for each quantity in eq. (21) to get the average value which will be substituted into eq. (21) to get that:

$$C = [C_{create\ UI} + C_{Initializations} + C_{assigning\ thread}] + [(1 + e_4) * (C_{test} + [\ 1 * (C_{receiving\ and\ sorting} + C_{test})\ ] + [\ 1 * (C_{decision} + C_{test})\ ]\ ) + ((e_8 + e_9) * (C_{test} + C_{test})) + ((e_{11} + 1) * C_{test}) + (1 * C_{complete}) + ((e_{11} + 1) * ([[(e_1 + e_4) * ([[(1 + e_4 + e_6) * (C_{ready} + C_{test})] + (e_3 * C_{idle}) + [(e_6 + 1) * C_{test}] + (1 * (C_{run}))] + C_{test})] + [e_4 * C_{aborted}] + [(e_0 + e_4) * C_{test}]] + C_{test})) = 42\ ms + 0.0341\ ms + 8.3112\ ms + 0 + 0.0341 + 1 + 3792.1422\ ms + 0.0341\ ms + 0.0341\ ms = 3834.5898\ ms.$$

The same procedures are applied to determine the average power consumption. More information about obtaining the results for analytical analysis can be found in [3]. Table 10 shows the average actual and estimated response time respectively on all platforms for several tasks "between 15 to 24 jobs in average" on each application while figures 54 to 57 illustrate the average estimated and actual response time after applying several jobs.

Table 10: Average actual and estimated response time

Application Name	RESPONSE TIME IN MSEC							
	Note 3		S 4		S 4 Mini		Tab 3 "7 inches"	
	ACT	EST	ACT	EST	ACT	EST	ACT	EST
Audio Recording	19258	20389	20901	22250	23491	24114	20012	21974
Calculator	20134	22054	22569	23198	27629	29943	21692	22692
Mobibench	50289	54190	52891	57329	55478	58997	51335	56689
Norvigtorious	78000	84386	79338	83661	81452	89119	78893	85398

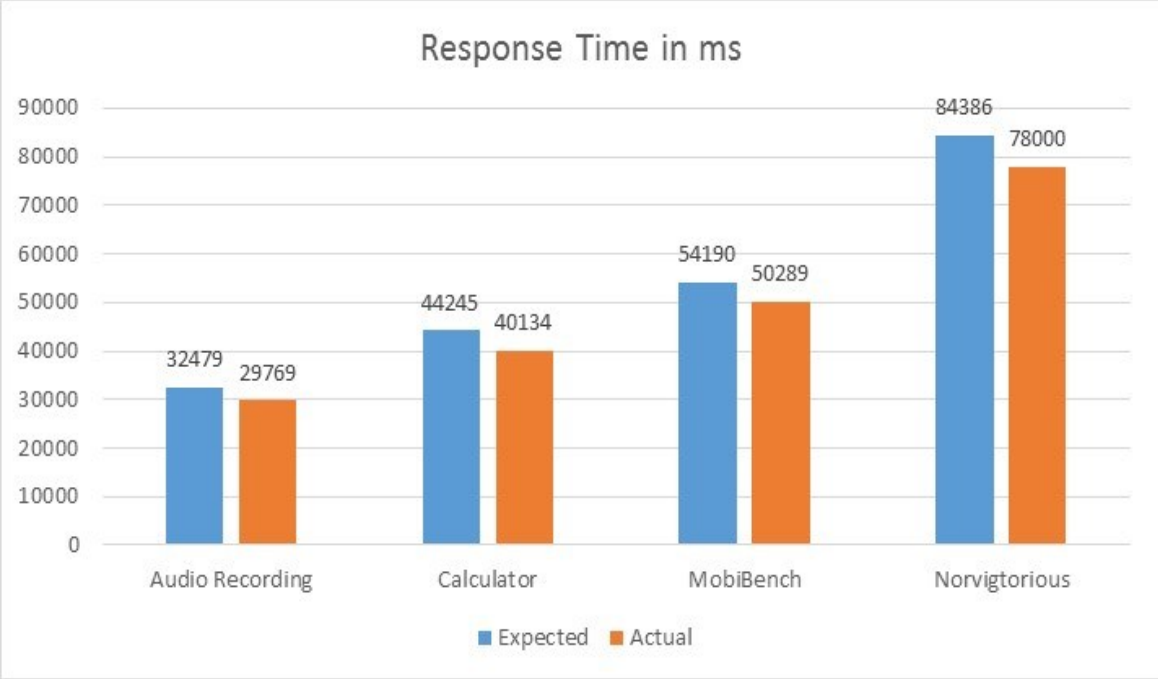


Figure 54: Average estimated and actual response time on Note 3

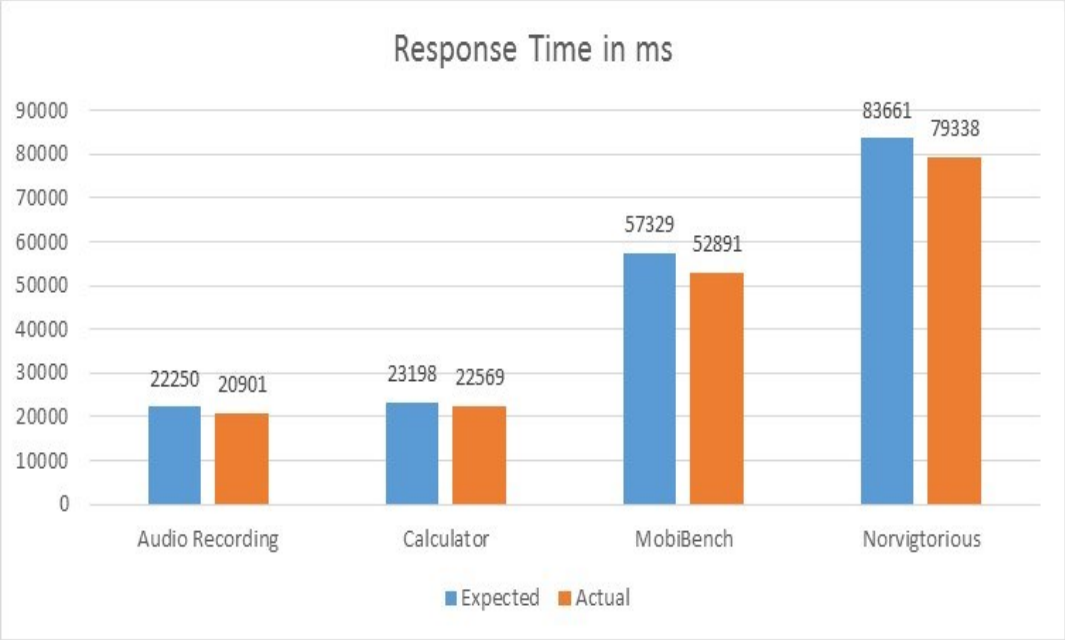


Figure 55: Average estimated and actual response time on S 4

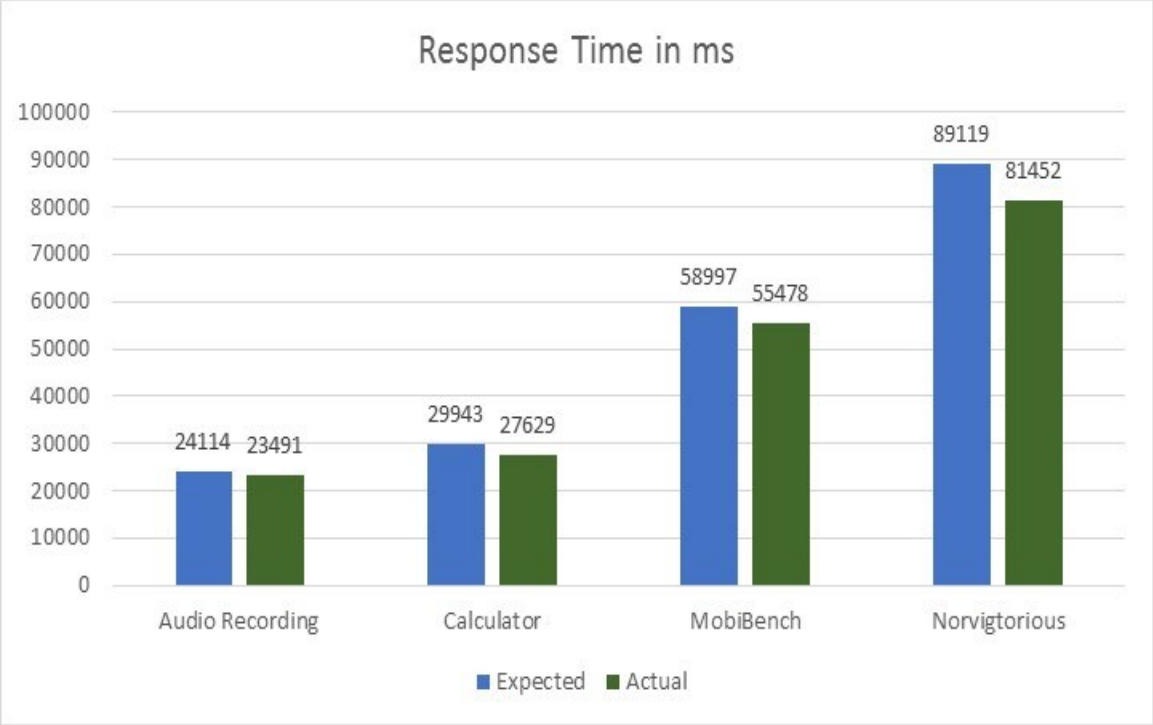


Figure 56: Average estimated and actual response time on S 4 mini

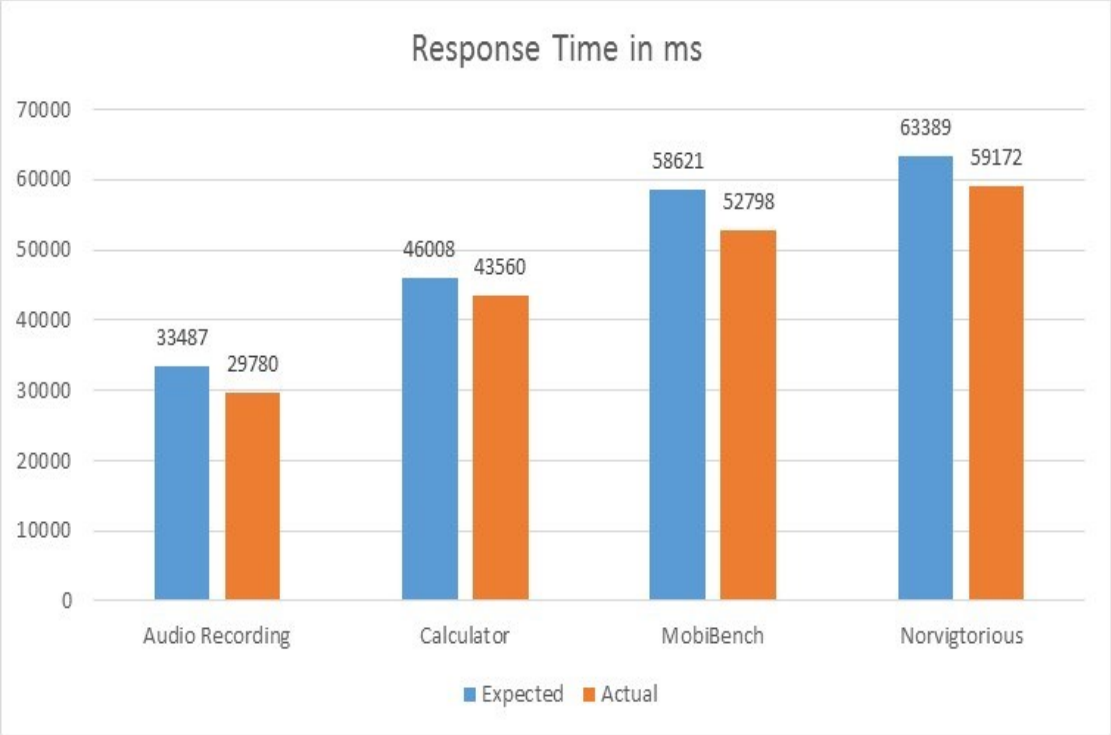


Figure 57: Average estimated and actual response time on Tab 3 "7 inch"

Some observations from the profiling method can be summarized as follows:

- ✓ The running time varies from architecture to another one due to:
    1. Each device comes with a unique processor even though they were produced by the same company "Samsung".
    2. Several applications were running in the background which affect the performance of the P.U.
- ✓ It was hard to keep the time exactly the same among all architecture when executing 4 applications. So there is a difference in the execution time among all 4 applications.

- ✓ In each application, around “15 to 25” tasks were executed to determine the change in the execution time and the recording of the average.
- ✓ Android Developing Bridge tool (ADB) was unable to determine the execution time in a case of failure. The results indicate that all tasks were executed successfully.
- ✓ Creating the GUI of any Android application is considered one of the bottlenecks that exist in the system. Significant reduction in the execution time could be achieved if a powerful tool such as GPU is used and synchronized with CPU in the Initial stage. Another method is assigning another thread to do this part instead of using only one; so two threads working simultaneously will give better performance in terms of the execution time.

For power consumption in all platforms, the same steps are applied to determine the average estimated power consumption and to compare the results with the average actual ones. Fig. 58 displays the results of the average actual and estimated on only Note 3 and S 4, the power saving mode was off, since the profiler we used is available only on two platforms. Estimating the average power consumption is slightly different than response time.

In each application, we ran a task and monitored its generated trace file for several primitive operations, recorded each value and then determined the average value. The traces files included different elements for power consumption and differentiating desired components which consumed time. To reduce displayed elements, adjusting the setting on the software profiler was performed. At the same time we had to clean the cache of each device since the applications in the background were affecting the results. We also performed the

experiments of each platform for two modes, the first mode indicated that the power saving mode was ON while the second mode implied that the power saving mode was OFF. The purpose of doing that was to monitor the difference in power consumption when that setting was ON and OFF. Galaxy Note 3 consumed less power than Galaxy S 4 even though it was the main device for personal use while S 4 was a backup device. Furthermore, many applications were installed and running on Note 3 whereas a few applications were installed on S 4. Nevertheless, Note 3 performed better in terms of response time and power consumption. The difference in power consumption in both modes was very small.

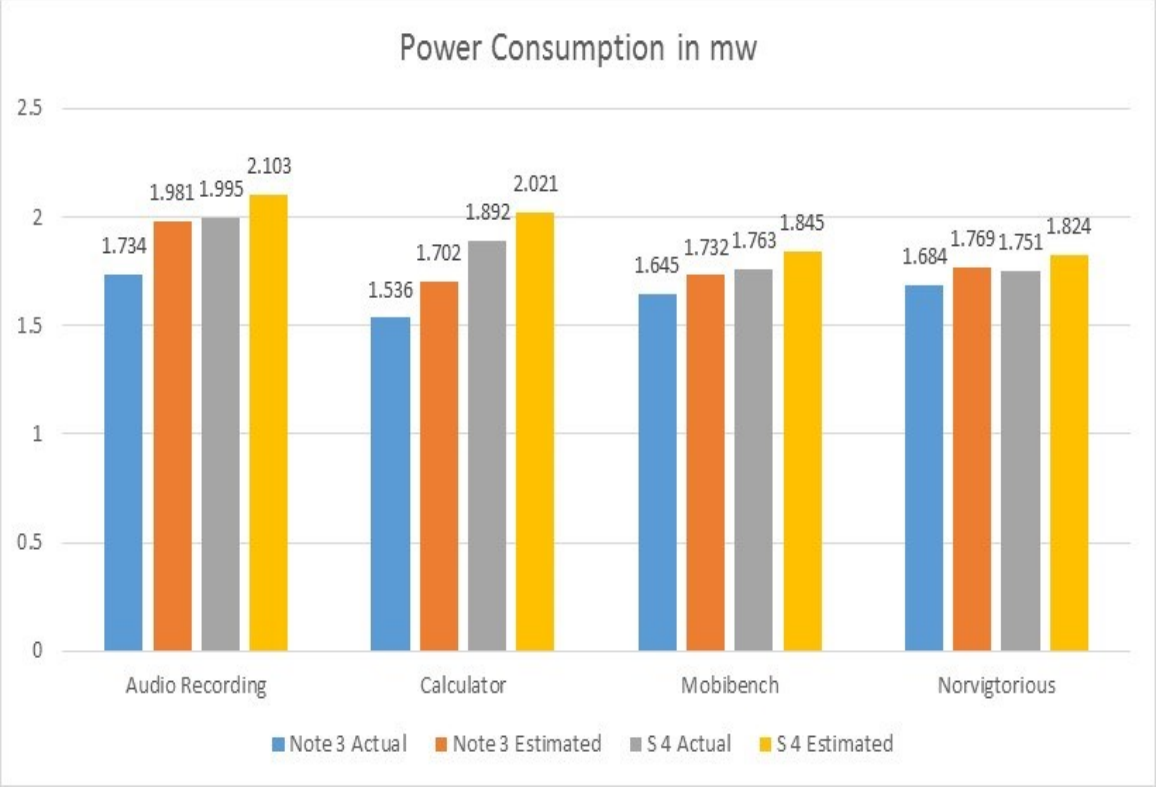


Figure 58: Average actual and estimated power consumption

The first two bars indicate the power consumption on Note 3 while the remaining two bars for S 4. The first bar in each platform represents the average actual results while the other bar represents the average estimated results.

To estimate the produced energy, the following equation is used:

$$Energy = Power * Time = PW (mw) * T (ms) \qquad (25)$$

Fig. 59 displays the average energy produced that is "estimated" in two platforms during the experiments. The first bar refers to the average estimated energy on Note 3 while the second bar refers to the average estimated energy on S 4. From the experiments we performed, the average error was about 12%, several factors led to the obtained result such as: Message size "M", Bandwidth, Bus Width; the values for the previous two factors that were taken from manufacturer web site. Also in the analytical analysis we assumed only one channel existed for communication between different components. In addition, no interruption occurred during the experiments. Interested readers are referred to [3] and [46] for more information about performance parameters.

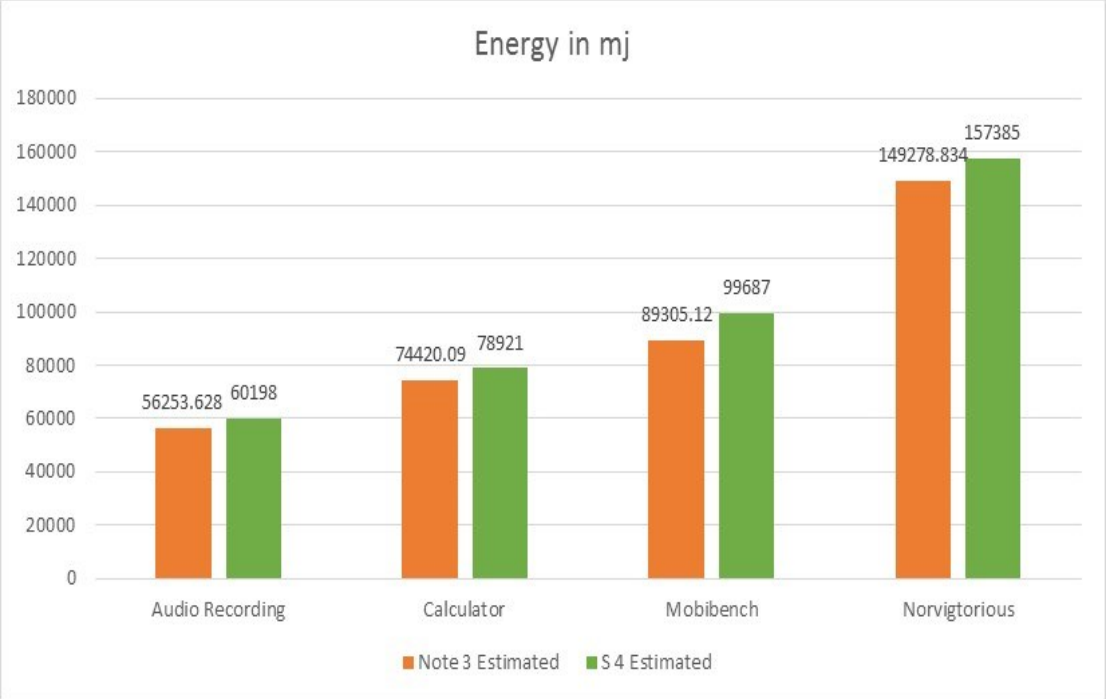


Figure 59: Average estimated energy consumption

3.5.2 OPENWRT

Openwrt is an embedded operating system based on Linux kernel; its primary function is to route network traffic. It exists on most of the routers available in the market these days. The HGFSM for OPENWRT is shown in fig. 60.

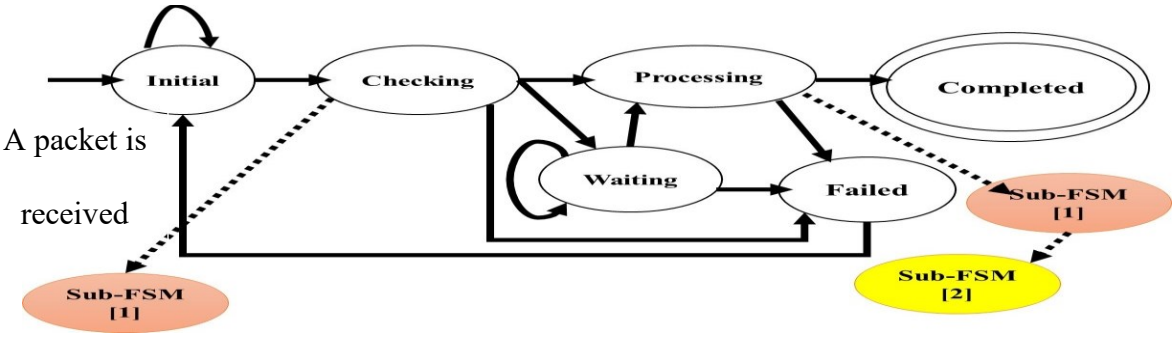


Figure 60: Hierarchical generic FSM for OPENWRT

Initial state: A packet is received through the incoming packet port (input terminal); self-loop indicates that the router is unable to run properly because there is a malfunction in it. Otherwise, the packet is sent to the next state after finishing the initializations such as creating the directory for accessing purpose, configuring boot source and lunching Kernel files. This initialization process is done only when the router is turned on (turn the power on). After that, the packet is sent to the next state.

Check state: The router checks its routing table to determine the best match between a destination IP address and one of the network addresses existing in the table. If the destination exists in the table, then, the packet is sent either to the execution "processing" state or the waiting state according to circumstances whether such as the P.U. is free or not and the priority level of the packet. If the destination address is not in the table, then, the router decides the best path to the destination.

Waiting state: Represents the place where the packet waits its turn to be sent when multiple packets are presented in the system. In the meantime, the packets are checked to determine which one should be sent first based on its level of priority. No arrow to the failed state exists here since the router forwards all packets.

Execution "Processing" state: Represents the place where the packet is sent to the destination address.

Failed state: Means the packet was not delivered correctly due to any reason such as the destination is unreachable (i.e. the device is offline) or an unknown error occurs during transmission process.

Completed state: Represents the final state and means the packet was sent and delivered successfully when an acknowledgment message was received from the destination address. The general system level overview is shown in fig. 61; it gives a pictorial picture of how OPENWRT works in normal mode. If there is a malfunction in the router, self-loop occurs in the initial state to indicate that the router is unable to process any packet(s).

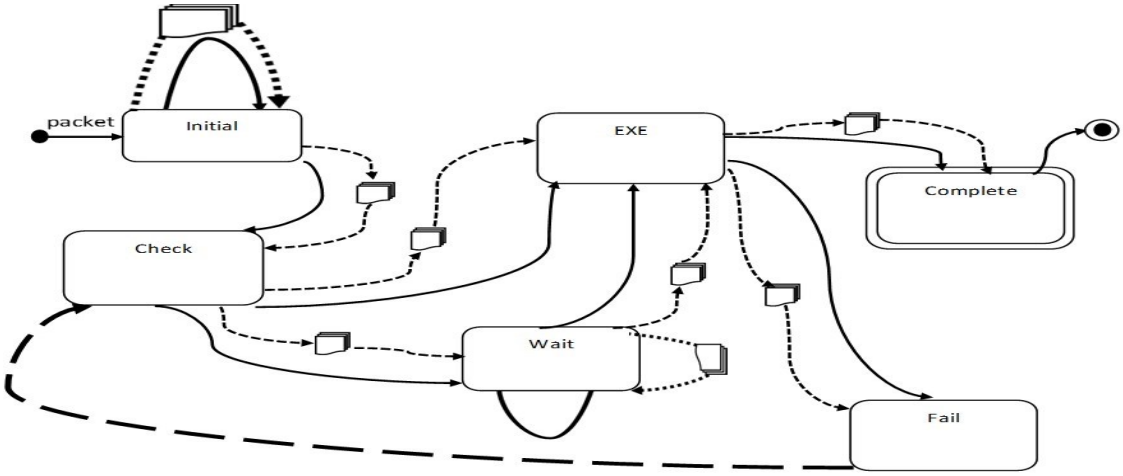


Figure 61: System level overview for OPENWRT

Solid lines indicate the control flow while the dashed lines indicate a class of message being sent. To utilize the performance parameters, we identify system components, input(s) and output(s) as shown in fig. 62.

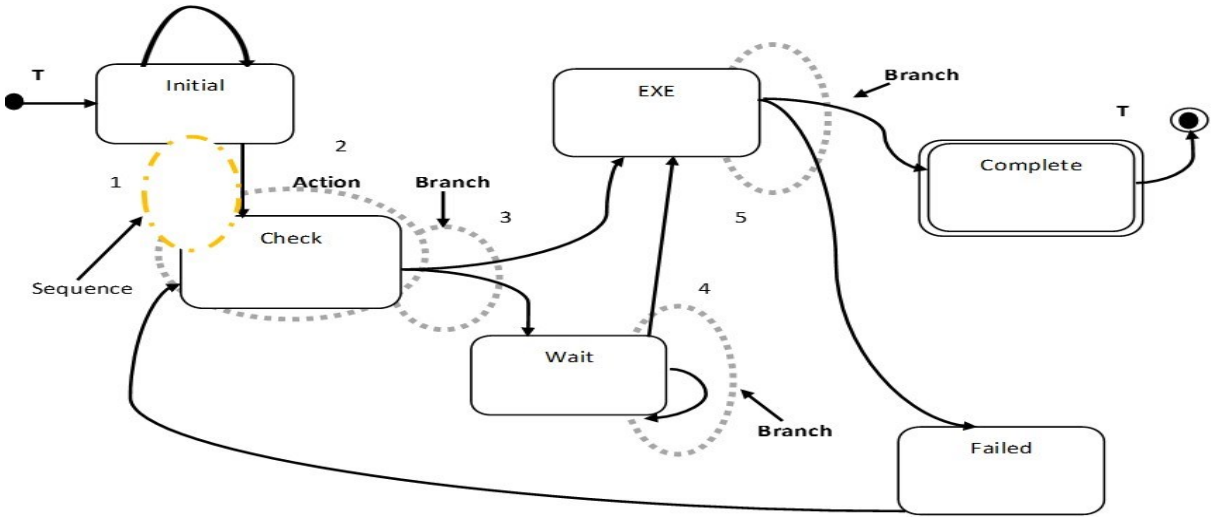


Figure 62: System components for OPENWRT

Fig. 62 shows that there is one input, one output and five components (one action, one sequence and three branches). The action takes place at the checking state where the routing table is searched to determine where to send a packet. Also checking a priority in each packet is also done to decide which packet should be sent first.

The Markov Model for the HGFSM is constructed using the same steps we applied on Android. Table 11 displays the probability transition equations between all states.

Table 11: Probability transition equations in OPENWRT

FROM - TO	$INITIAL_1$	$CHECKING_2$	$WAITING_3$	$EXECUTION_4$	$FAILED_5$	$COMPLETED_6$
$INITIAL_1$	$\mathbf{P_{11}=k_{11}/N_1}$	$\mathbf{P_{12}=k_{12}/N_1}$	----------	----------	----------	----------
$CHECKING_2$	----------	----------	$\mathbf{P_{23}=k_{23}/N_2}$	$\mathbf{P_{24}=k_{24}/N_2}$	----------	----------
$WAITING_3$	----------	----------	$\mathbf{P_{33}=k_{33}/N_3}$	$\mathbf{P_{34}=k_{34}/N_3}$	----------	----------
$EXECUTION_4$	----------	----------	----------	----------	$\mathbf{P_{45}=k_{45}/N_4}$	$\mathbf{P_{46}=k_{46}/N_4}$
$FAILED_5$	----------	$\mathbf{P_{52}=1}$	----------	----------	----------	----------
$COMPLETED_6$	----------	----------	----------	----------	----------	----------

Computation structure model "***CSM***" for the developed model of OPENWRT is shown in fig. 63 and fig. 64.

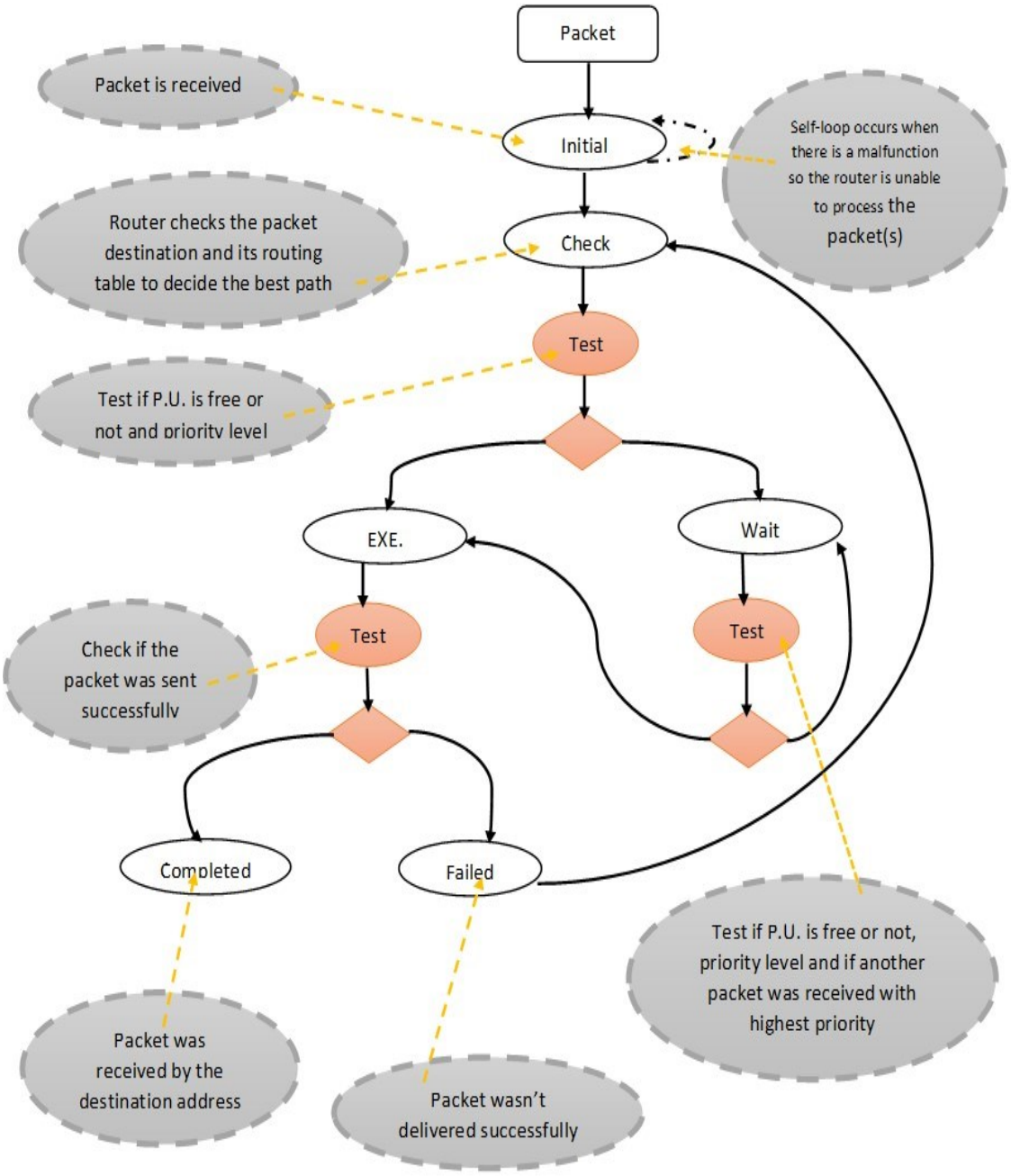


Figure 63: Data flow graph for OPENWRT

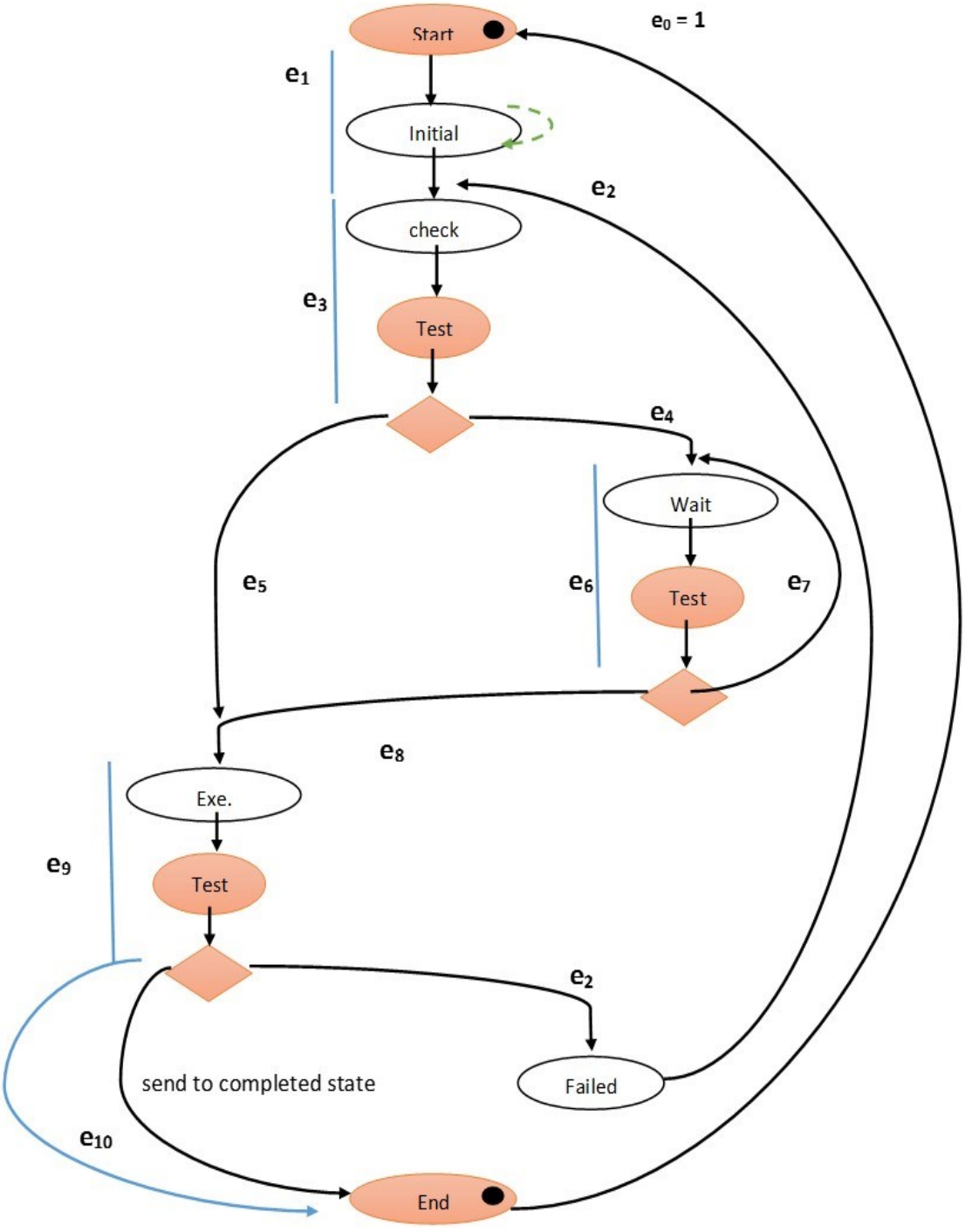


Figure 64: Control flow graph for OPENWRT

From fig. 64, we can obtain the following equation to determine the performance metrics which are response time and power consumption.

$$Cost = C = (e_1 * C_{Initial}) + (e_3 * (C_{check} + C_{test})) + (e_4 * 0) + (e_5 * 0) + (e_8 * 0) + (e_6 * (C_{wait} + C_{test})) + (e_9 * (C_{Exe} + C_{test})) + (e_2 * C_{failed}) + (e_{10} * 0) = (e_1 * C_{Initial}) + (e_3 * (C_{check} + C_{test})) + (e_6 * (C_{wait} + C_{test})) + (e_9 * (C_{Exe} + C_{test})) + (e_2 * C_{failed}) \quad (26)$$

To find the relationships between dependent and independent flows, the spanning tree technique is used as shown in fig. 65.

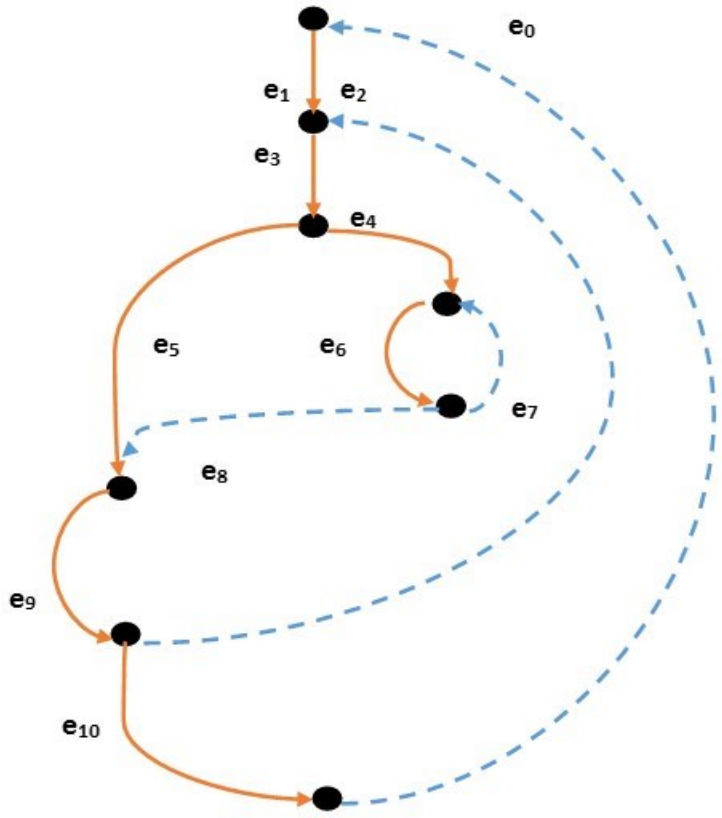


Figure 65: Spanning tree for OPENWRT

Solid lines indicate dependent flows (***e1 , e3 , e4 , e5 , e6 , e9 and e10***) whereas dashed lines indicate independent flows (e_0 , e_2 , e_7 *and* e_8). The previous spanning tree is reduced as shown in fig. 66 in order to find the relations between different flows.

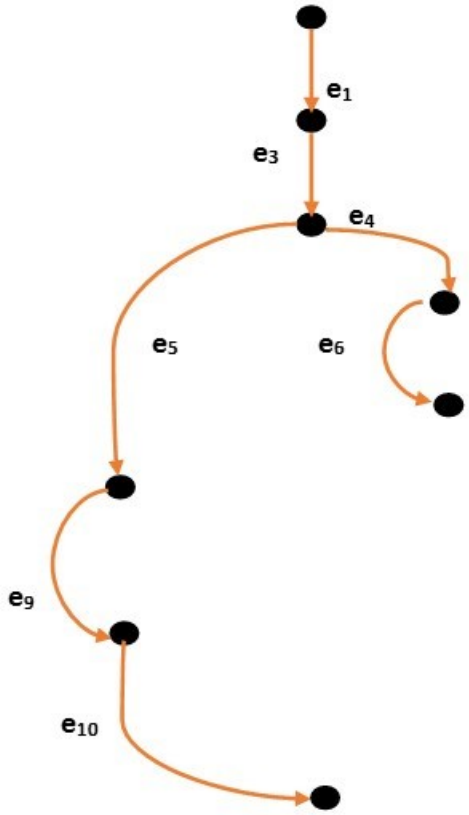


Figure 66: Reduced spanning tree for OPENWRT

From fig. 64, 65 and 66, we can obtain the following relations:

$$e_1 = e_0 = e_{10} = 1$$

$$e_3 = e_1 + e_2 = 1 + e_2 = e_4 + e_5$$

$$e_6 = e_4 + e_7$$

$$e_9 = e_5 + e_8 = e_2 + e_{10} = 1 + e_2$$

$$e_3 = e_9$$

$$e_4 + e_5 = e_5 + e_8$$

Table 12 illustrates the relations between dependent and independent flows.

Table 12: Relations between different flows in OPENWRT

Dependent flows	Independent flows					Equations
	e_0	e_2	e_7	e_8		
e_1	e_0					$e_1 = e_0 = 1$
e_3	e_0	e_2				$e_3 = 1 + e_2$
e_4	e_0	e_2				$e_4 = e_0 + e_2$
e_5		e_2		$- e_8$		$e_5 = e_2 - e_8$
e_6		e_2	e_7			$e_6 = e_7 + e_2$
e_9	e_0	e_2				$e_9 = e_0 + e_2$
e_{10}	e_0					$e_{10} = e_0 = 1$

Substitute all obtained equations in table 12 into eq. (26) to get;

$$Cost = C = (1 * C_{Initial}) + [\ (1 + e_2) * (C_{check} + C_{test})\ ] + [e_7 * (C_{wait} + C_{test})\ ] + [\ (1 + e_2) * (C_{Exe} + C_{test})\ ] + (e_2 * C_{failed}) \qquad (27)$$

In order to find the cost of each quantity in eq. (27), we need to find operations take place in each state by using its CSM.

1 – Initial State: The packet is forwarded to the checking state. Only initializations lead to significant time consumption if we assume that passing packet consumes a little bit of time which can be neglected. So

$$C_{Initial} = 0 \text{ (in normal mode of operation). Otherwise, } C_{Initial} \neq 0 \qquad (28)$$

2 – Check State: checks the packet header to determine where to send the packet; then, searches the routing table to find the best path to a final destination. Upon deciding the best path, another test is performed to determine if the P.U. is free or not is done. Its CSM is shown in fig. 67 and 68.

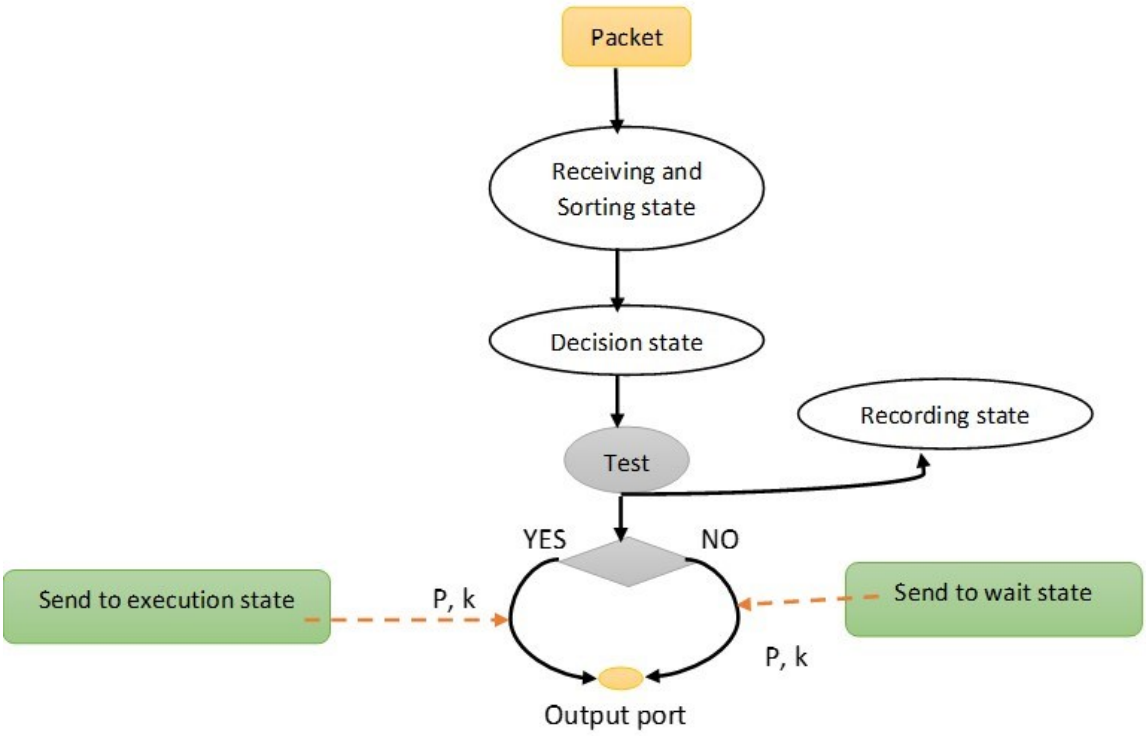


Figure 67: Data flow graph for the Checking state in OPENWRT

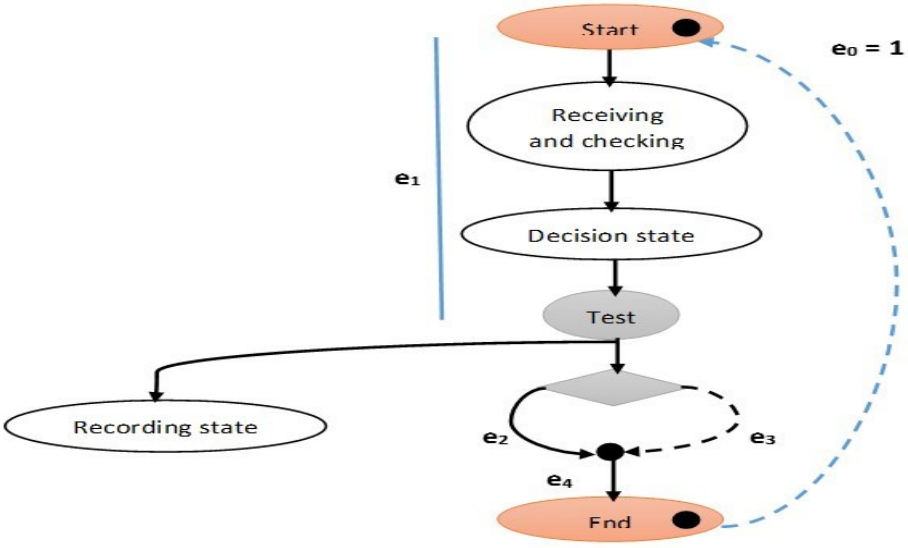


Figure 68: Control flow graph for the Checking state in OPENWRT

$$C_{check} = (e_1 * (C_{receiving\ and\ Sorting} + C_{decision} + C_{test})) + (e_2 * 0) + (e_3 * 0) + (e_4 * 0) = (e_1 * (C_{receiving\ and\ checking} + C_{decision} + C_{test}))$$

$$since\ e_1 = e_0 = 1;\ so$$

$$C_{check} = C_{receiving\ and\ checking} + C_{decision} + C_{test} \quad (29)$$

3 – Wait State: packets are checked to determine which one should be sent first to the P.U. and then P.U. is examined to determine whether it is available or not. If the P.U. is free, the packet is sent to it. Otherwise, the packet remains in its state. DFG and CFG for the waiting state are shown in the following two figures respectively.

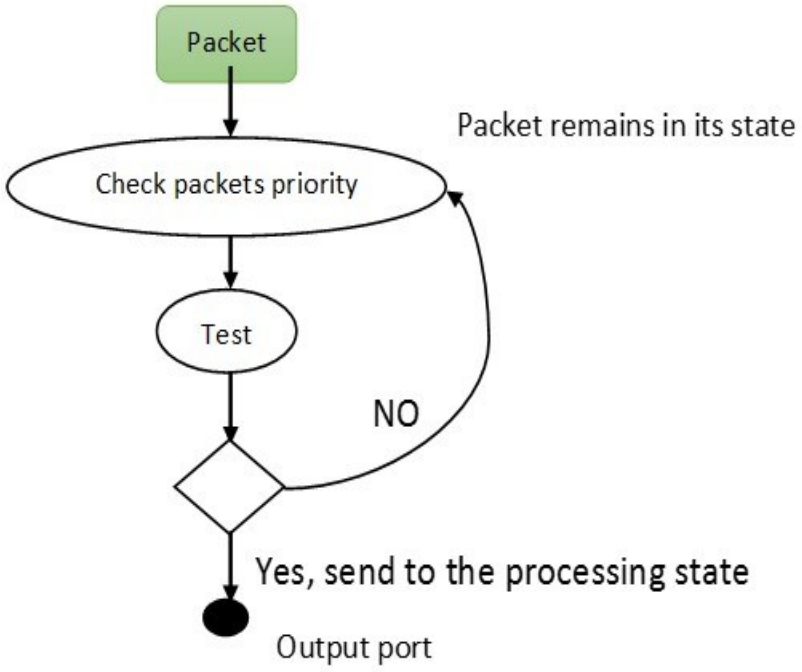


Figure 69: Data flow graph for the waiting state in OPENWRT

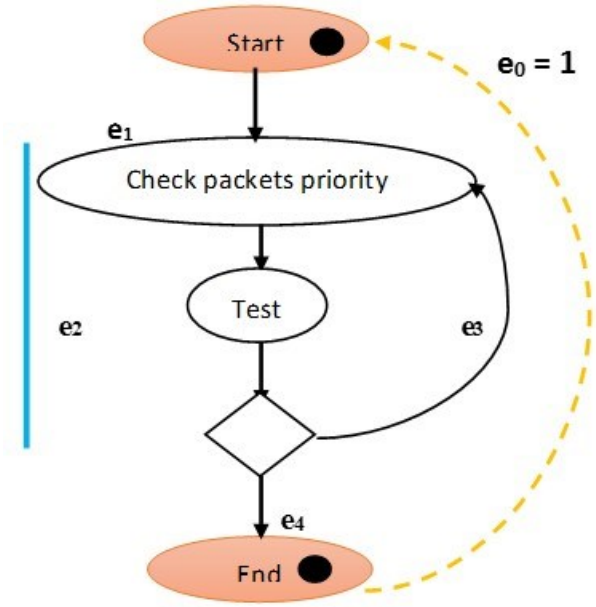


Figure 70: Control flow graph for the Waiting state in OPENWRT

From fig. 70, we can obtain the following relations between different flows:

$$e_1 = e_0 = 1$$

$$e_2 = e_1 + e_3 = 1 + e_3$$

$$e_4 = e_2 - e_3 = 1 + e_3 - e_3 = 1$$

$C_{wait} = (e_1 * 0) + (e_2 * (C_{check\ packet\ priority} + C_{test})) + (e_3 * 0) + (e_4 * 0)$; so

$$C_{wait} = ([1 + e_3] * (C_{check\ packet\ priority} + C_{test})) \qquad (30)$$

4 – Execution State: Packets are sent to their destination address. Before that, the P.U. screens if there is a packet with higher priority to process. In the meantime, it keeps track of every packet status as successfully sent or failed and this process is done by receiving an acknowledgment message from the destination. Its CSM is shown in the following two graphs.

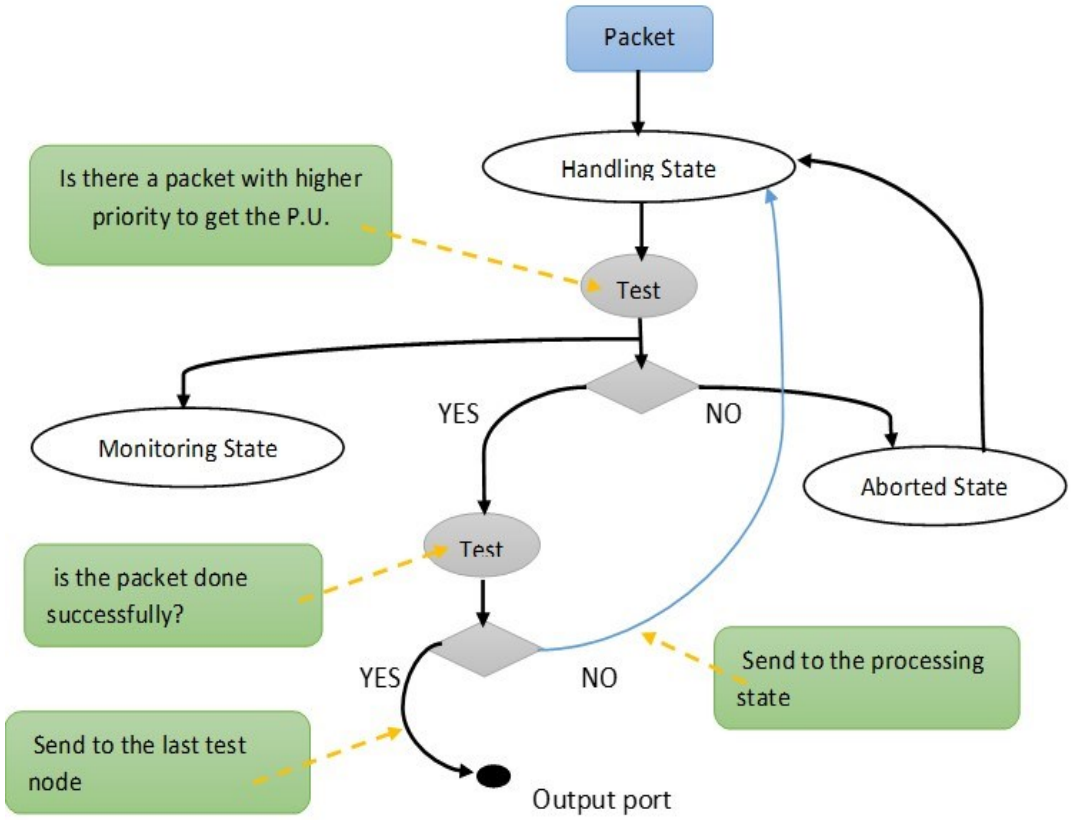


Figure 71: Data flow graph for the Processing state in OPENWRT

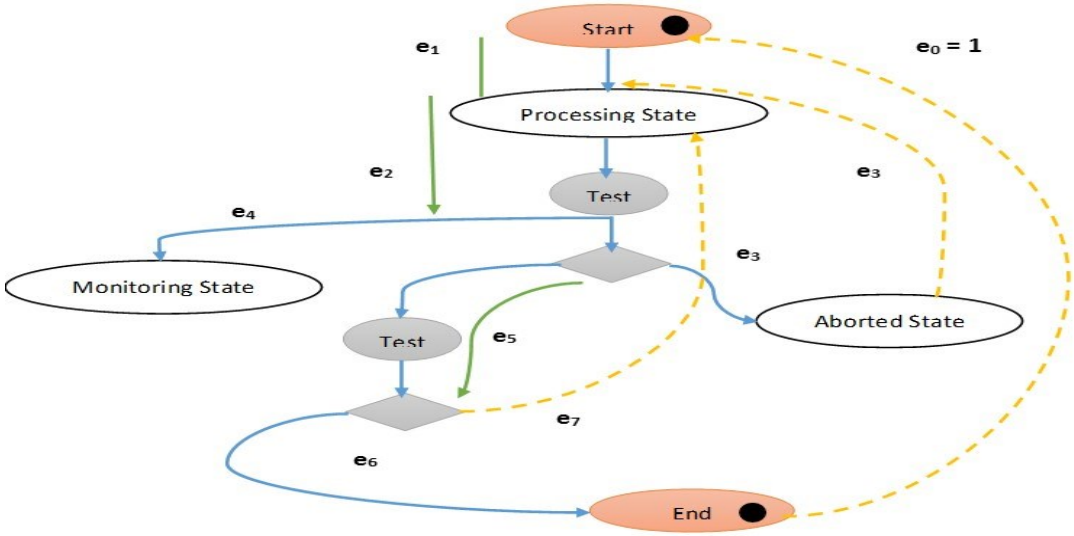


Figure 72: Control flow graph for the Processing state in OPENWRT

$$e_1 = e_0 = 1$$

$$e_2 = e_1 + e_3 + e_7 = 1 + e_3$$

$$e_5 = e_6 + e_7 = 1 + e_7$$

$$C_{Execution} = [(1 + e_3 + e_7)*(C_{Handling} + C_{test})] + (e_3 * C_{aborted}) + ((1 + e_7) * C_{test}) \quad (31)$$

The CFG for the Handling state inside the processing state is shown in fig. 73.

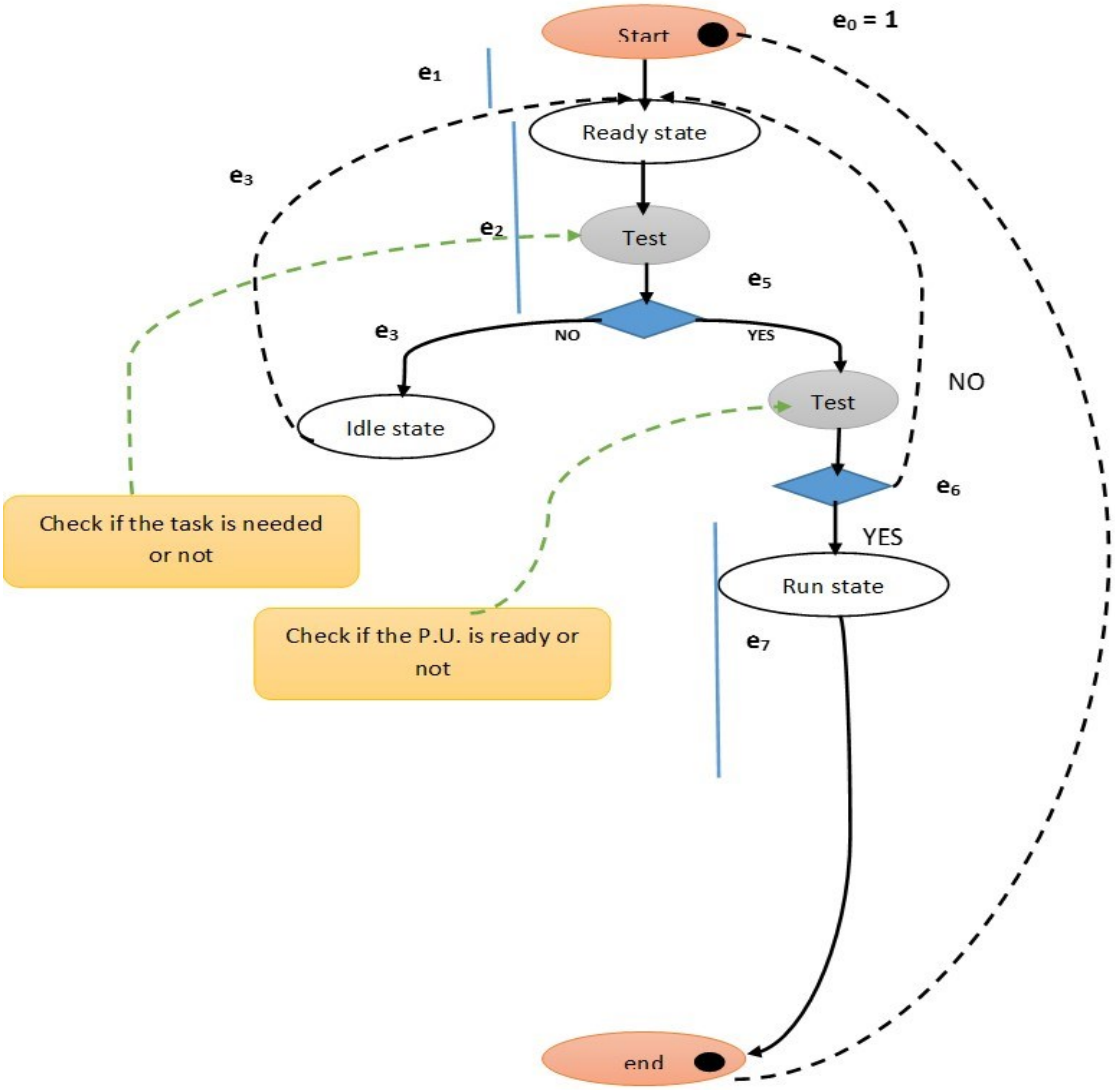


Figure 73: Control flow graph for the Handling state in OPENWRT

$$e_1 = e_0 = 1$$

$$e_2 = e_1 + e_3 + e_6$$

$$e_5 = e_6 + e_7$$

$$e_7 = e_0 = 1$$

$$C_{Handling} = [(1+e_3+e_6)*(C_{ready} + C_{test})] + (e_3 * C_{idle}) + [(e_6 + 1) * C_{test}] + (1 * (C_{run})) \quad (32)$$

5 – Failed State: Passing the packet to the checking state again for the resend process. Also an error message occurs so

$$C_{failed} = C_{passing\ packet} + C_{display\ error\ message} \quad (33)$$

The cost for passing the packet is very small and can be neglected by our assumption so

$$C_{passing\ packet} = 0 \quad (34)$$

Substitute equations (28), (29), (30), (31), (32) and (33) into eq. (27) to get that:

$$Cost = C = [\ (1 + e2) * ([C_{receiving\ and\ Sorting} + C_{decision} + C_{test}] + C_{test})\ ] + [e7 * ([[(1 + e_4) * (C_{check\ packet\ priority} + C_{test})]] + C_{test})\ ] + [\ (1 + e2) * ([[(1 + e_3 + e_7) * ([[(1 + e_3 + e_6) * (C_{ready} + C_{test})] + (e_3 * C_{idle}) + [(e_6 + 1) * C_{test}] + (1 * (C_{run} + C_{test}))] + C_{test})] + (e_3 * C_{aborted}) + (1 * C_{test})] + C_{test})\ ] + (e2 * C_{display\ error\ message}) \quad (35)$$

PROFILING WITH OPNET SIMULATOR

Opnet is a software used to determine the ***performance metrics for Computer Networks and Applications***. It was built and designed in 1986 and went public in 2000. Riverbed Company acquired Opnet in 2012. The commercial version of Opnet is quite expensive. However, there is a free version which is an Academic edition (Guru Academic Edition). Several drawbacks of the Guru Edition exist which can be summarized as:

- A few number of nodes (work stations) < 55 can be used in the simulation.

- Only 1 router is allowed to be used in the simulation.
- Only 1 server station is allowed to be used with just one service such as email, ftp and etc.
- A few number of performance metric parameters are supported.
- The simulation works only in an office mode; other modes are not supported.

Another network simulator software such as ***NS3*** exists and can be used. However, ***Opnet*** has more capabilities related to performance metrics and that why it is preferred. The profiling scheme is used to depict the variance of delay time across different implementations.

Several scenarios (4) were developed and implemented in order to find the **expected average delay time**, *a time spent between receiving a packet at a port until forwarding it to its destination*, in any router with OPENWRT embedded operating system. In all implemented scenarios, a common network topology was used which was **STAR**. To profile the delay time, some assumptions were made and several parameters were estimated to compute the expected average delay time. The assumptions are summarized as follow:

- The starting time, when the packet is received, is considered to be current time in a router; for simplicity, we assume it is = 0 and the time between receiving the packet and starting the checking stage is negligible as stated earlier.

- The time for checking the routing table and deciding the path for forwarding the packet to its destination is also negligible, since the number of components "clients or work station" is very small, as proved by the simulation.

Single run is used during the experiments and equally likely assumption is considered for flows to be taken in either True or False conditions. Same procedures are applied as we applied them on the Android case study.

SCENARIO 1

2 switches with 16 ports, 1 server station, 1 router (Cisco brand), 29 nodes (work stations) with link speed = 100 MBPS between all nodes, server and switches. 24618 packets were simulated

Table 13: Elapsed time

DELAY TIME IN µs			
Initial Time	Checking Time	Waiting Time	Forwarding Time
0	0	0	55

All packets were forwarded successfully; there was no packet in the waiting state since the router forwarding capacity is 30000 packets per second. Forwarding time refers to the processing time. The probability matrix table becomes as follows:

Table 14: Probability transition values for scenario 1

FROM - TO	$INITIAL_1$	$CHECKING_2$	$WAITING_3$	$EXECUTION_4$	$FAILED_5$	$COMPLETED_6$
$INITIAL_1$	**0**	**$P_{12} = 1$**	**0**	**0**	**0**	**0**
$CHECKING_2$	**0**	**0**	**$P_{23} = 0$**	**$P_{24} = 1$**	**0**	**0**
$WAITING_3$	**0**	**0**	**$P_{33} = 0$**	**$P_{34} = 0$**	**0**	**0**
$EXECUTION_4$	**0**	**0**	**0**	**0**	**0**	**$P_{46} = 1$**
$FAILED_5$	**0**	**$P_{52} = 0$**	**0**	**0**	**0**	**0**
$COMPLETED_6$	**0**	**0**	**0**	**0**	**0**	**0**

To find the number of visits to each state, we use matlab to compute the inverse of the following quantity $V = [I - P]^{-1}$, the result is shown in table 15:

Table 15: Number of visits in scenario 1

FROM - TO	$INITIAL_1$	$CHECKING_2$	$WAITING_3$	$EXECUTION_4$	$FAILED_5$	$COMPLETED_6$
$INITIAL_1$	**0**	**1**	**0**	**0**	**0**	**0**
$CHECKING_2$	**0**	**0**	**0**	**1**	**0**	**1**
$WAITING_3$	**0**	**0**	**0**	**0**	**0**	**0**
$EXECUTION_4$	**0**	**0**	**0**	**0**	**0**	**1**
$FAILED_5$	**0**	**0**	**0**	**0**	**0**	**0**
$COMPLETED_6$	**0**	**0**	**0**	**0**	**0**	**1**

So the expected average delay time $T_d = \sum (V_i * T_i)$, where "i" represents the state number from 1 to 6 "number of states in the system". So the average $T_d = 55 * 1.5 = 82.5$ µs.

Fig. 74 shows the average the response time in scenario 1 using OPNET simulator.

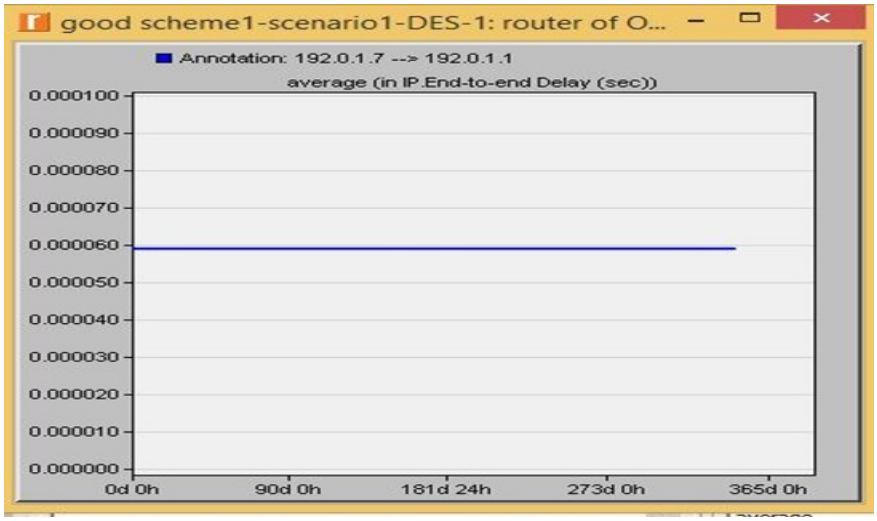


Figure 74: Response time in scenario 1

Figure 75 depicts the average actual and estimated response time in scenario 1.

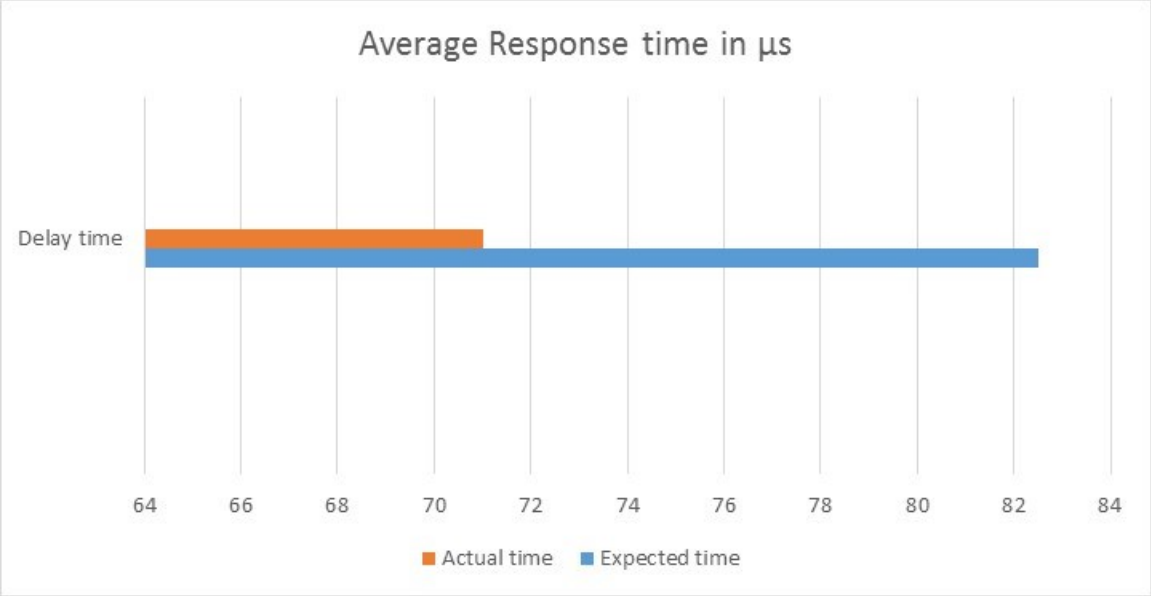


Figure 75: Average actual and estimated response time in scenario 1

SCENARIO 2

3 switches (2 switches were connected together and 1 linked to the router), 41 nodes with link speed = 100 MBPS and 37417 packets were simulated.

Table 16: Response time in scenario 2

DELAY TIME IN µs			
Initial Time	Checking Time	Waiting Time	Forwarding Time
0	0	38	23, 29.2

30000 packets were in the processing state while the remaining packets equaling 7417 were in the waiting state. so P_{24} = 30000 / 37417 = 0.802 and P_{23} = 7417 / 37417 = 0.198. P_{34} = 1 = 7417 / 7417. The probability matrix table becomes as follows:

Table 17: Probability transition values in scenario 2

FROM - TO	$INITIAL_1$	$CHECKING_2$	$WAITING_3$	$EXECUTION_4$	$FAILED_5$	$COMPLETED_6$
$INITIAL_1$	**0**	**$P_{12} = 1$**	**0**	**0**	**0**	**0**
$CHECKING_2$	**0**	**0**	**P_{23}=0.198**	**$P_{24} = 0.802$**	**0**	**0**
$WAITING_3$	**0**	**0**	**$P_{33} = 0$**	**$P_{34} = 1$**	**0**	**0**
$EXECUTION_4$	**0**	**0**	**0**	**0**	**0**	**$P_{46} = 1$**
$FAILED_5$	**0**	**$P_{52} = 0$**	**0**	**0**	**0**	**0**
$COMPLETED_6$	**0**	**0**	**0**	**0**	**0**	**0**

To find the number of visits to each state, we use matlab to compute the inverse of the following quantity $V = [I - P]^{-1}$, the result is shown in the following table:

Table 18: Number of visits in scenario 2

FROM - TO	$INITIAL_1$	$CHECKING_2$	$WAITING_3$	$EXECUTION_4$	$FAILED_5$	$COMPLETED_6$
$INITIAL_1$	**0**	**1**	**0.198**	**0**	**0**	**0**
$CHECKING_2$	**0**	**0**	**0.198**	**1**	**0**	**0**
$WAITING_3$	**0**	**0**	**0**	**1**	**0**	**0**
$EXECUTION_4$	**0**	**0**	**0**	**0**	**0**	**1**
$FAILED_5$	**0**	**0**	**0**	**0**	**0**	**0**
$COMPLETED_6$	**0**	**0**	**0**	**0**	**0**	**1**

So the expected average delay time $T_d = \sum (V_i * T_i)$; since there were 2 switches connected directly with each other which implies that their delay time is bigger than other switch.

Average $T_{d1} = (38 * 0.5) + (1.5 * 29.2) = 19 + 43.8 = 62.8$ µs.

$T_{d2} = 19 + 34.5 = 53.5$µs

T_{d1} refers to the response time from 2 switches which were connected together to the remaining switch only. T_{d2} refers to the response time for the switch which was connected to the router directly. Fig. 76 displays the waiting time and average delay in forwarding packets in scenario 2 while fig. 77 shows the average actual and estimated response time in scenario 2.

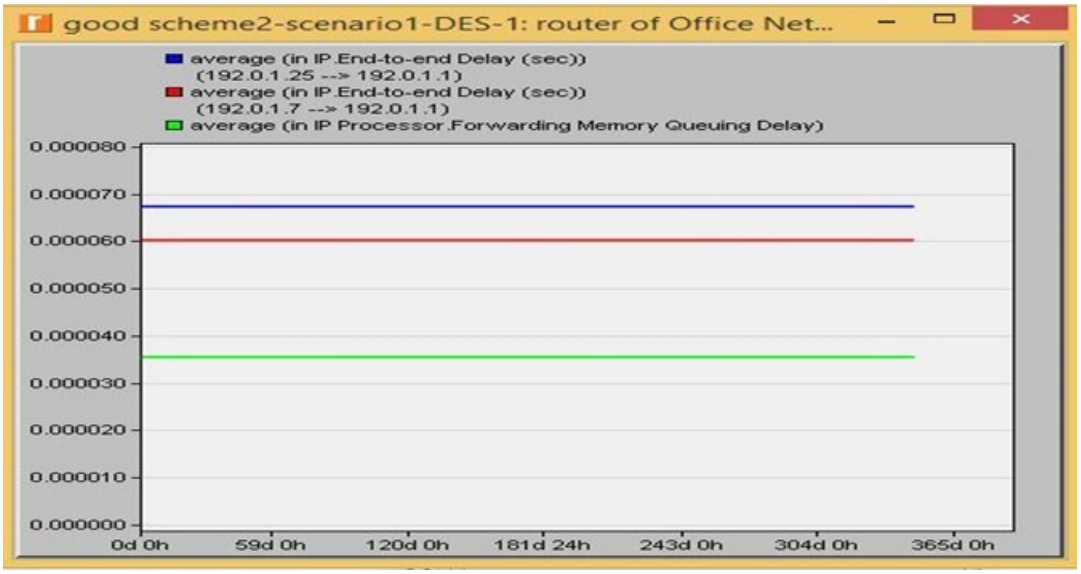


Figure 76: Estimated waiting time and average response time in scenario 2

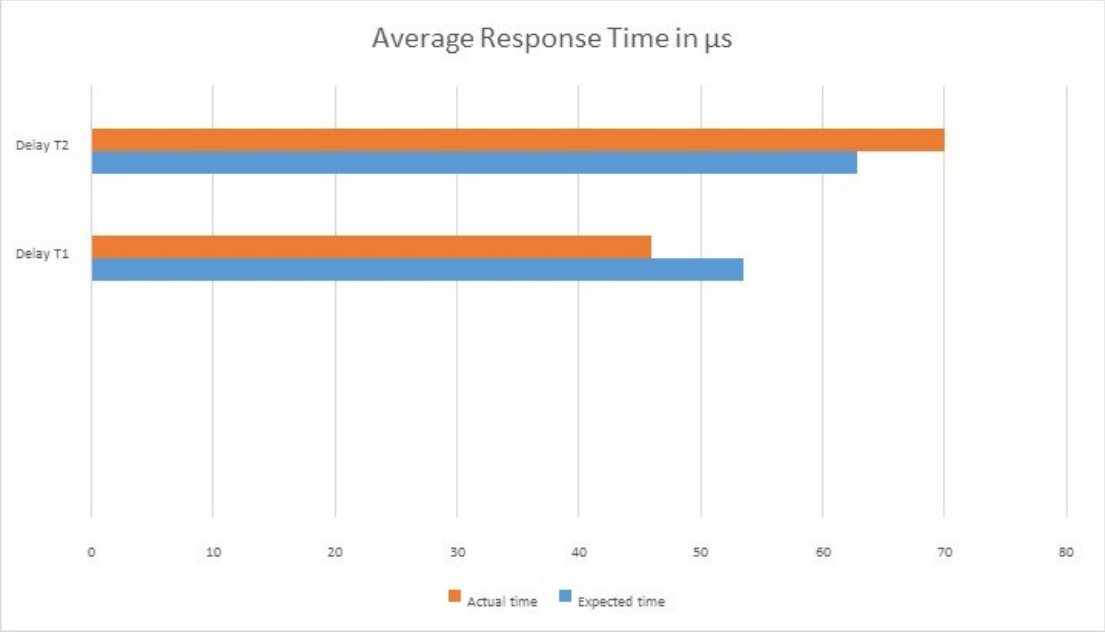


Figure 77: Average actual and estimated response time in scenario 2

The first bar in fig. 77 for delay T_1 and delay T_2 refers to the actual average response time while the second bar refers to the estimated average response time.

SCENARIO 3

4 switches where each 2 switches were connected directly, 1 router, 1 server and 51 nodes with the speed link = 100 MBPS. 48198 packets were simulated. Table 19 displays the estimated delay time in all states.

Table 19: Response time in scenario 3

DELAY TIME IN µs			
Initial Time	Checking Time	Waiting Time	Forwarding Time
0	0	37	27, 30.4

P_{23} = 18198 / 48198 = 0.378 and P_{24} = 30000 / 48198 = 0.622, P_{34} = 18198 / 18198 = 1

The probability matrix table becomes as follows:

Table 20: Probability transition values in scenario 3

FROM - TO	$INITIAL_1$	$CHECKING_2$	$WAITING_3$	$EXECUTION_4$	$FAILED_5$	$COMPLETED_6$
$INITIAL_1$	**0**	**P_{12} = 1**	**0**	**0**	**0**	**0**
$CHECKING_2$	**0**	**0**	**P_{23}=0.378**	**P_{24} = 0.622**	**0**	**0**
$WAITING_3$	**0**	**0**	**P_{33} = 0**	**P_{34} = 1**	**0**	**0**
$EXECUTION_4$	**0**	**0**	**0**	**0**	**0**	**P_{46} = 1**
$FAILED_5$	**0**	**P_{52} = 0**	**0**	**0**	**0**	**0**
$COMPLETED_6$	**0**	**0**	**0**	**0**	**0**	**1**

To find the number of visits to each state, we use matlab to compute the inverse of the following quantity $V = [I - P]^{-1}$, the result is shown in the table 21:

Table 21: Number of visits in scenario 3

FROM - TO	$INITIAL_1$	$CHECKING_2$	$WAITING_3$	$EXECUTION_4$	$FAILED_5$	$COMPLETED_6$
$INITIAL_1$	**0**	**1**	**0**	**0**	**0**	**0**
$CHECKING_2$	**0**	**0**	**0.378**	**1**	**0**	**0**
$WAITING_3$	**0**	**0**	**0**	**1**	**0**	**0**
$EXECUTION_4$	**0**	**0**	**0**	**0**	**0**	**1**
$FAILED_5$	**0**	**0**	**0**	**0**	**0**	**0**
$COMPLETED_6$	**0**	**0**	**0**	**0**	**0**	**1**

So the expected average response time $T_d = \sum (V_i * T_i)$; so

$T_{d1} = 19 + 40.5 = 59.5$ µs

$T_{d2} = 19 + 45.6 = 64.6$ µs

Each value refers to the response time in each switch, T_{d1} refers to the 2 switches which were connected to the router whereas T_{d2} refers to the remaining 2 switches as shown in fig. 78 and 79.

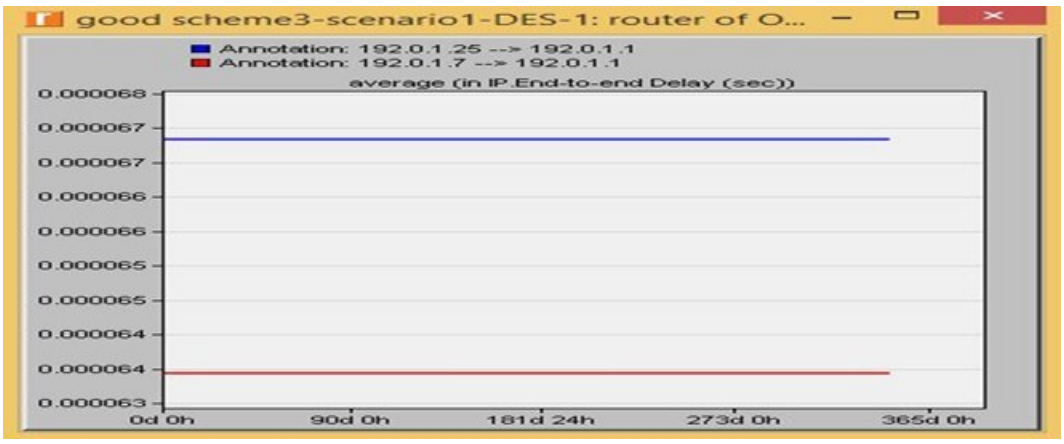


Figure 78: Average estimated forwarding time in scenario 3

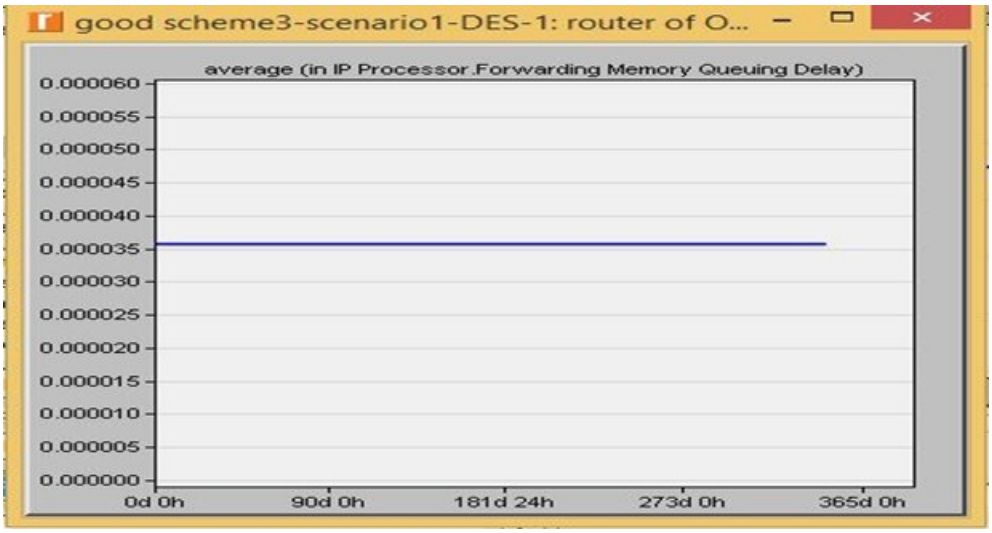


Figure 79: Average waiting time in scenario 3

Fig. 80 displays the average actual and estimated response time in scenario 3

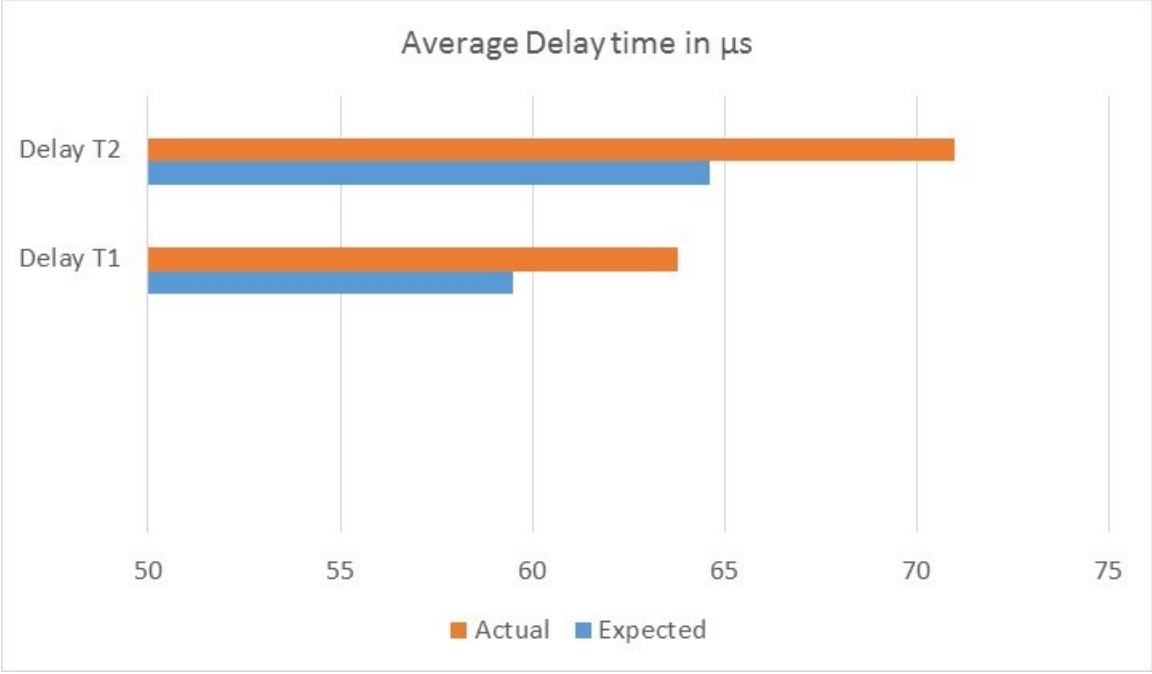


Figure 80: Average actual and estimated response time in scenario 3

SCENARIO 4

4 switches, 1 server, 1 router and 51 nodes with link speed = 100 MBPS and 10 MBPS between the router and all switches. 31835 packets were simulated.

Table 22: Response time in all states in scenario 4

DELAY TIME IN µs			
Initial Time	Checking Time	Waiting Time	Forwarding Time
0	0	38	85

P_{23} = 1835 / 31835 = 0.058, P_{24} = 30000 / 31835 = 0.942 and P_{34} = 1835 / 1835 = 1. The probability matrix table becomes as follows:

Table 23: Probability transition values in scenario 4

FROM - TO	$INITIAL_1$	$CHECKING_2$	$WAITING_3$	$EXECUTION_4$	$FAILED_5$	$COMPLETED_6$
$INITIAL_1$	**0**	**P_{12} = 1**	**0**	**0**	**0**	**0**
$CHECKING_2$	**0**	**0**	**P_{23}=0.058**	**P_{24} = 0.942**	**0**	**0**
$WAITING_3$	**0**	**0**	**P_{33} = 0**	**P_{34} = 1**	**0**	**0**
$EXECUTION_4$	**0**	**0**	**0**	**0**	**0**	**P_{46} = 1**
$FAILED_5$	**0**	**P_{52} = 0**	**0**	**0**	**0**	**0**
$COMPLETED_6$	**0**	**0**	**0**	**0**	**0**	**1**

To find the number of visits to each state, we use matlab to compute the inverse of the following quantity $V = [I - P]^{-1}$, the result is shown in the following table:

Table 24: Number of visits in scenario 4

FROM - TO	$INITIAL_1$	$CHECKING_2$	$WAITING_3$	$EXECUTION_4$	$FAILED_5$	$COMPLETED_6$
$INITIAL_1$	**0**	**1**	**0**	**0**	**0**	**0**
$CHECKING_2$	**0**	**0**	**0.198**	**1**	**0**	**0**
$WAITING_3$	**0**	**0**	**0**	**1**	**0**	**0**
$EXECUTION_4$	**0**	**0**	**0**	**0**	**0**	**1**
$FAILED_5$	**0**	**0**	**0**	**0**	**0**	**0**
$COMPLETED_6$	**0**	**0**	**0**	**0**	**0**	**1**

So the expected average delay time $T_d = \sum (V_i * T_i)$;

$T_d = 19 + 127.5 = 146.5$ µs.

Fig. 81 displays the average actual and estimated response time in scenario 4.

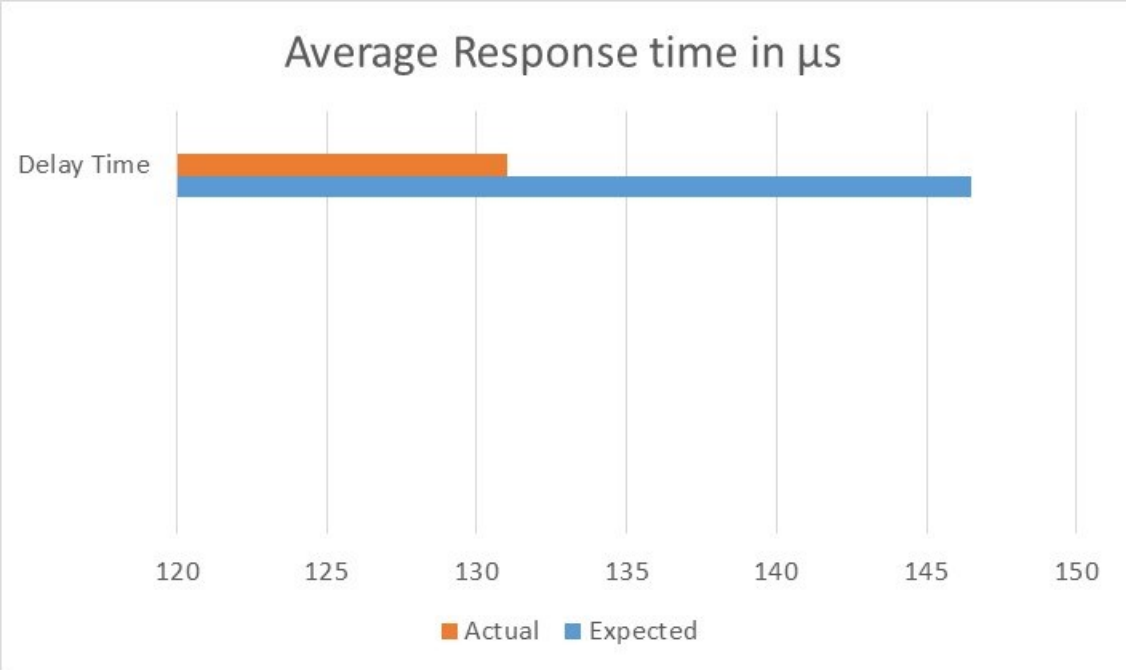


Figure 81: Average actual and estimated response time in scenario 4

OBSERVATIONS FROM PROFILING STAGE

- ✓ Increasing number of nodes in a network affects End-to-End "ETE" Delay time.
- ✓ In a large network, the time spent for checking the routing table plays a significant role.
- ✓ Speed of used cables between nodes in the network affects the ETE Delay time.

Using the Full Version of OPNET, where all features are allowed, will give more reasonable and acceptable results.

CHAPTER 4

Performance Analysis of Improved Response Time

4.1 Introduction

In this chapter, we conduct performance analyses of improved response time only; the power consumption is left as future work. We use the analytical approach to derive performance parameters for several Android platforms in order to estimate the average response time. Parallelization with optimization schemes along with an invocation of GPUs are used to minimize the response time.

Using GPUs and parallelization schemes together show a promising sight to enhance the delay in a system under investigation with a trade-off in power consumption and code size [46].

4.2 Parallelization and GPUs Schemes

Embedded systems have become a key factor of technological components for all kinds of complex systems ranging from smart devices, PDAs, aircraft to weapons and intelligence systems [46,47,48,49]. Ability to estimate a correct performance metric is critical and essential. Figure 82 shows the average states time for a single task taken on the Galaxy Note 3 in milliseconds “ms”; nevertheless, the average states time is different in all hardware platforms that we used in the experiments.

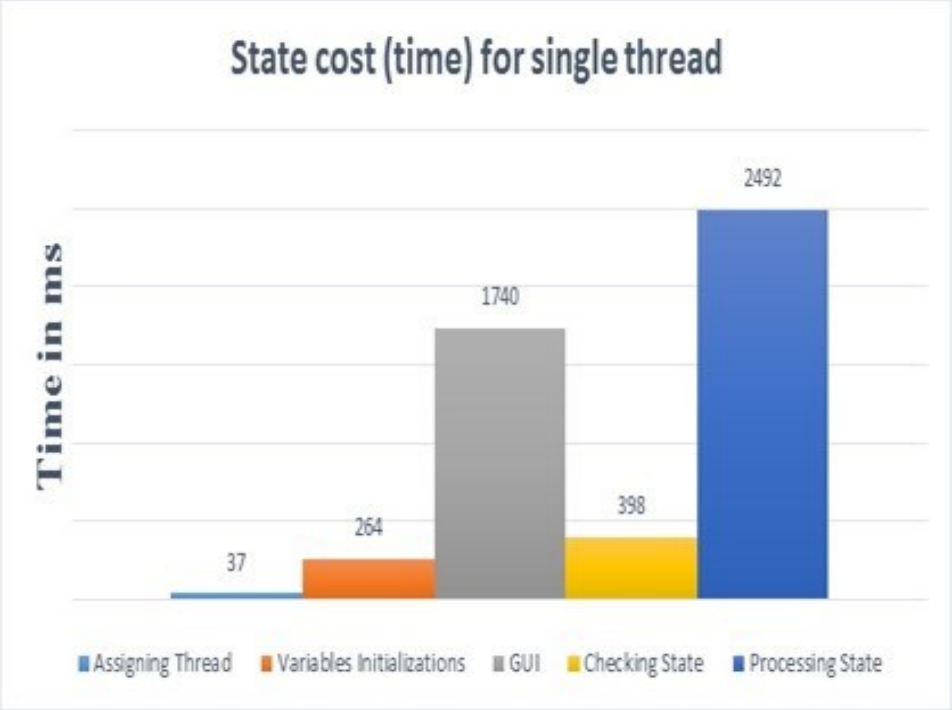


Figure 82: Average state cost on Note 3

The “Waiting state time” is omitted in figure 82 since its value is zero due to the fact that no task went into it. Our previous work in [45] shows that an application always starts with a single thread on the Android platform. Typically, a task starts when a user presses on a key, it enters into the initial state where three operations are performed. Then, it goes to the checking state to check the availability of required resources. If these resources are not available, it is sent to the failed state to restart its cycle again and an error message pops up on the screen. If the resources are available, then another test is performed to decide where to send the task either to the waiting or the processing state. If it is sent to the waiting state, it will stay and wait its turn to gain control of the CPU. In the processing state, the task is executed to perform the desired action by showing a result on the screen as depicted in fig. 38.

In [47], Y. H. Jung and L. P. Carloni developed a framework to accelerate concurrent simulations for several virtual platforms using GPUs on a host machine. Their approach worked by leveraging the physical presence of GPUs that existed on the host machine. The developed method improved the speed up without affecting the optimization code on the host machine. Furthermore, two techniques to speed up the simulation were developed by them. The idea of the proposed framework is to execute GPU code on multiple virtual GPU models on the host machine using the multiplexing method as a tool to achieve their objectives as depicted in fig. 83 from [47].

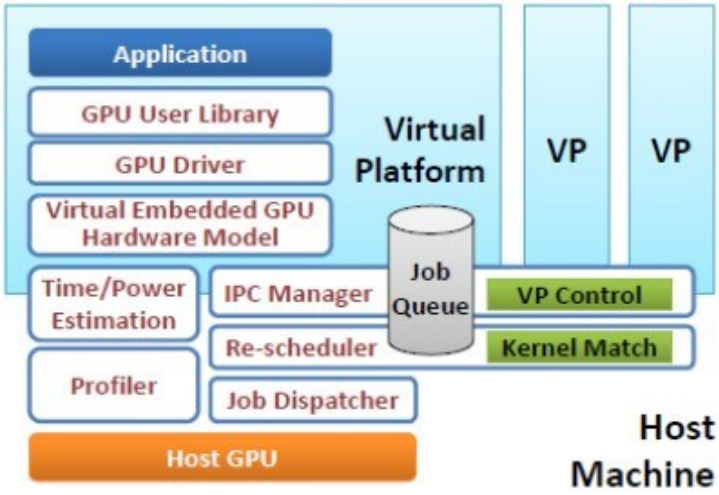


Figure 83: The developed simulation framework

The developed framework aims to reduce time and power consumption as well. The performance analysis for two metrics was done based on Profile-Based Execution Analysis which was developed by the authors. For timing analysis, three refined models were used to estimate the number of clock cycles needed to execute Kernel code on the host machine. A CUDA platform was used to demonstrate the developed framework for the performance metrics analysis.

S. Nomura et al in [48] proposed a novel multi-GPU system with an ExpEther. ExpEther is a virtualization technique used to extend PCIe of the host CPU to the Ethernet. They assumed that all devices connected by ExpEther are treated as if they were connected directly to the host machine. Two applications were evaluated without concerning GPU-GPU communication. The developed model consisted of a single CPU and multiple GPUs "around 4", this model is named GPU-Box. A micro-benchmark was used for the performance analysis with several practical application programs. The model could achieve a better performance with a limited data exchange between GPUs. Latency was measured using small Kernel functions described in Cuda. However, there was no interaction between CPU and GPUs.

ExpEther tool enables designers to connect a host node that controls them using an Ethernet network switch. The used GPU-Box provided 8 slots with two 10GB Ethernet ports for each slot. It had 3000W for a power supply which was enough to operate with 8 GPUs connected together. ***Data transfe***r between different GPUs was considered as a performance metric in the proposed scheme. Initially, *a certain amount of time was kept constant until a specific size of data transfer was reached*. This time can be seen as the minimum latency needed to set up the data transfer. Three different sets of tests were performed, ***the first set included only one GPU, the second set included four GPUs while the last set included four clusters of GPUs***.

Grasso I. et al in [49] determined the possibility of using embedded GPUs for **High Performance Computing** (*HPC*). They came up with 9 benchmark applications and executed them on an ARM Mali-T604 GPU to estimate the performance "throughput" and

compared results with ARM Cortex-A15 cores. They performed analysis for performance and energy for embedded GPUs for HPC. The obtained results from HPC depicted that the improved speed up was about 9% over a single Cortex-A15 core.

The first embedded GPU with OpenCL, which is an open industry standard for programming on heterogeneous systems, had full profile support that resulted from the proposed approach. In addition, the importance of using OpenCL software optimization tools was identified in order to utilize ARM Mali GPU architecture for best efficiency. Mali GPU architecture is designed to become fully complicit with OpenCL for a high precision purpose. The versions of OpenCL used within [49] were developed in the Open Computing Language in order to allow a parallel execution on a GPU. All experiments were performed on Samsung Exynos 5 Dual Arndale Board which was equipped with the Samsung embedded system-on-chip “SOC” (Exynos 5250), 2GB of DDR3L-1600 memory and dual-core ARM Cortex-A15. The speed of the ARM Cortex-A15 is 1.7 Ghz with a cache size of 32KB.

During the experiments, a problem size was maintained constant so that all benchmarks performed the same amount of work. Furthermore, they were repeated around 20 times and mean values were collected. All parallel regions on each benchmark were the interested element, the initializations and finalization phases were excluded from the analysis.

In [50], Glenis A. and Petridis A. evaluated the performance metrics Frame Per Second "FPS and Thermal Design Power (TDP)" for several embedded systems. They

evaluated performance gains of GPUs vs CPUs. They used Harris Corner Detection algorithms in the area of detecting and tracking.

Authors considered using integral image computations as a method to evaluate their model. The integral image computation is performed by a prefix scan followed by a matrix transpose, followed again by the prefix scan and the matrix transpose in order to fix the orientation of an image. Different available libraries were used to optimize the baseline of GPUs with the help from CUDA. A developed library called CUDPP was used for high performance computation for a two-dimensional prefix scan. A platform used for the experiments was equipped with GeForce GTX480 that had 1.5 GB of RAM and GTS450 1 GB of RAM. An Intel Core2Due that runs at 3.6 Ghz was existed in it. The authors did not determine the effect of using GPUs with CPUs together on performance metrics since their focus mainly was on GPUs.

Huang M. and Lai C. in [51] conducted a comprehensive analysis on estimating the performance of using GPUs as accelerators on embedded systems. They analyzed the performance on GPUs and CPUs separately and proposed a hybrid scheme to integrate them together to get a better result. Their approach was to distribute the workload between parallel GPUs and sequential CPUs since it is known that GPU is a very powerful tool in parallel computations. Two different categories of benchmarks were used in their experiments, they were 1. A level 3 BLAS subroutines and 2. Computer vision algorithms such as the mean shift image segmentation and the scale-invariant feature transform "***SIFT***". All experiments were carried out on an Nvidia CARMA development kit which consisted of Nvidia Tegra 3 quad-core CPU and Nvidia Quadro 1000M GPU. The authors

adopted an empirical method in their work. Our scheme is different since GPUs take control of all graphical tasks for both parallel and sequential while CPUs perform other parallel and sequential jobs.

From [50] and [51], authors found that GPUs consumed much less power compared to the one produced by CPUs. Nevertheless, GPUs outperform CPUs in many application domains.

4.2.1 OpenGL

OpenGL was developed to get the maximum performance from GPUs; all APIs are defined as a set of functions which cooperate together to perform a job. It is used to draw 2D and 3D graphics. In our research, we use OpenGL ver. 2.0 which is compatible on any device that uses either OpenGL ver. 3.0 or ver. 1.0 and it is widely used. Android supports high performance 2D and 3D graphics using OpenGL. OpenGL APIs provide a standard software interface for either 2D or 3D graphics processing hardware.

To use GPUs properly, OpenGL functions have to be involved. OpenGL stands for Open Graphics Libraries which are open source codes and available for GPUs. It was developed by Silicon Graphics in the early 90s. These days, OpenGL has become the most widely used graphic library worldwide. Several versions of it exist which they are as follows:

I. OpenGL ver. 1.

II. OpenGL ver. 2.

III. OpenGL ver. 3.

4.3 Case Study

In this section, we perform performance analysis using the developed framework to minimize the response time on several Android platforms using available resources that are on them. We will use the same platforms mentioned in the previous chapter. Since it is difficult to improve the clock frequency due to higher power consumption and cost; the parallelization approach gives a promising solution for this issue. It reduces the delay and produces less power when compared to increasing the clock frequency. To minimize response time, Parallelization with optimization and GPU invocation are used. The parallelization scheme refers to software acceleration whereas GPU invocation refers to the hardware acceleration. For the parallelization scheme, several threads are applied to speed up the response time.

Deciding number of threads to be used is performed according to a method described in [60] which was developed by C. L. Rathbone in 1988. The more threads that are used, the more the dependency and the complexity rise. Our results show that the software acceleration method minimizes the response by about 10% whereas the hardware acceleration approach reduces the response time by around 28% for entire application.

4.3.1 Double-Thread Approach

Our previous work in [45] shows that any application on Android starts always with a single thread. Multiple cores require parallelism in a software platform [47, 48, 49]. So one might ask, is it ok to start with double threads? If so, is it safe? The answer to the previous questions is yes with more attention in programming side [46]. Any task starts

after it is assigned a thread; then the fork structure splits the thread and it becomes two threads that work simultaneously. Each thread takes control of performing desired operations in either the initial or the checking state as depicted in figure 84 [46]; the join structure is used to synchronize all threads together in order for the application to function properly.

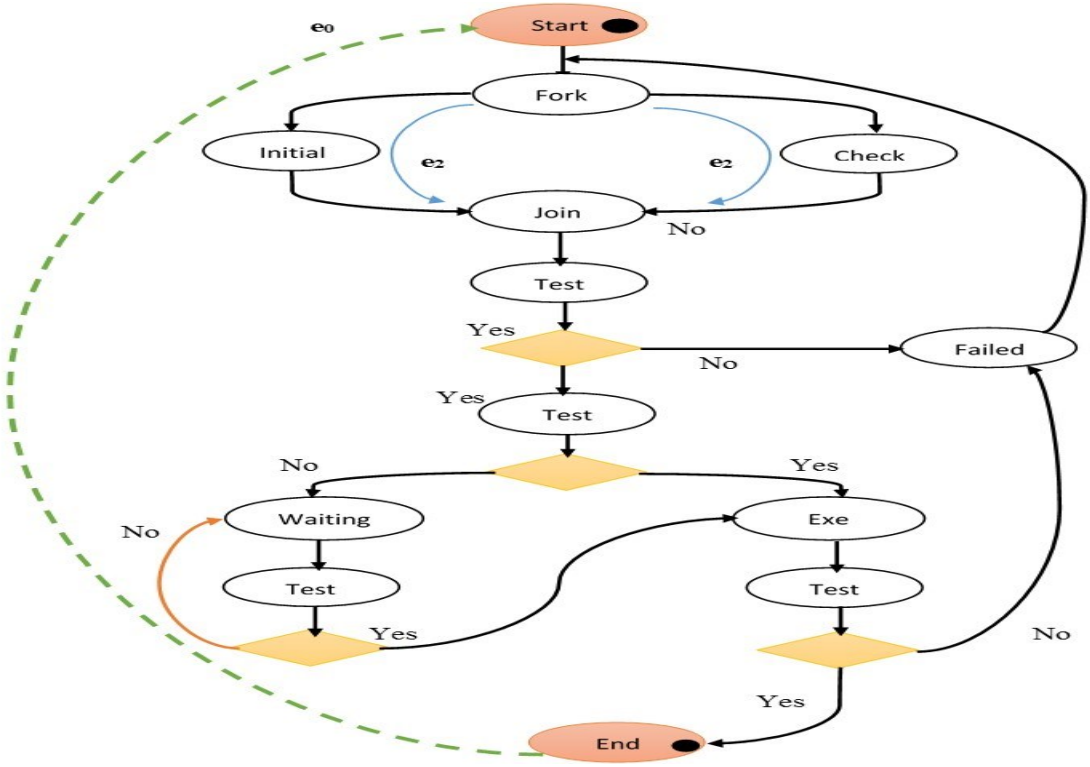


Figure 84: Double-thread approach

Fig. 84 can be modified for simplicity as shown in fig. 85 where ***p.s*** stands for **parallel structure** which includes two states (Initial and Checking) as obtained from our work in [46]. Using double threads improves the delay by less than ***10%*** as observed in the tests

since it eliminates the cost value associated with the checking state due to the fact that its value is smaller than the value of the initial state [46].

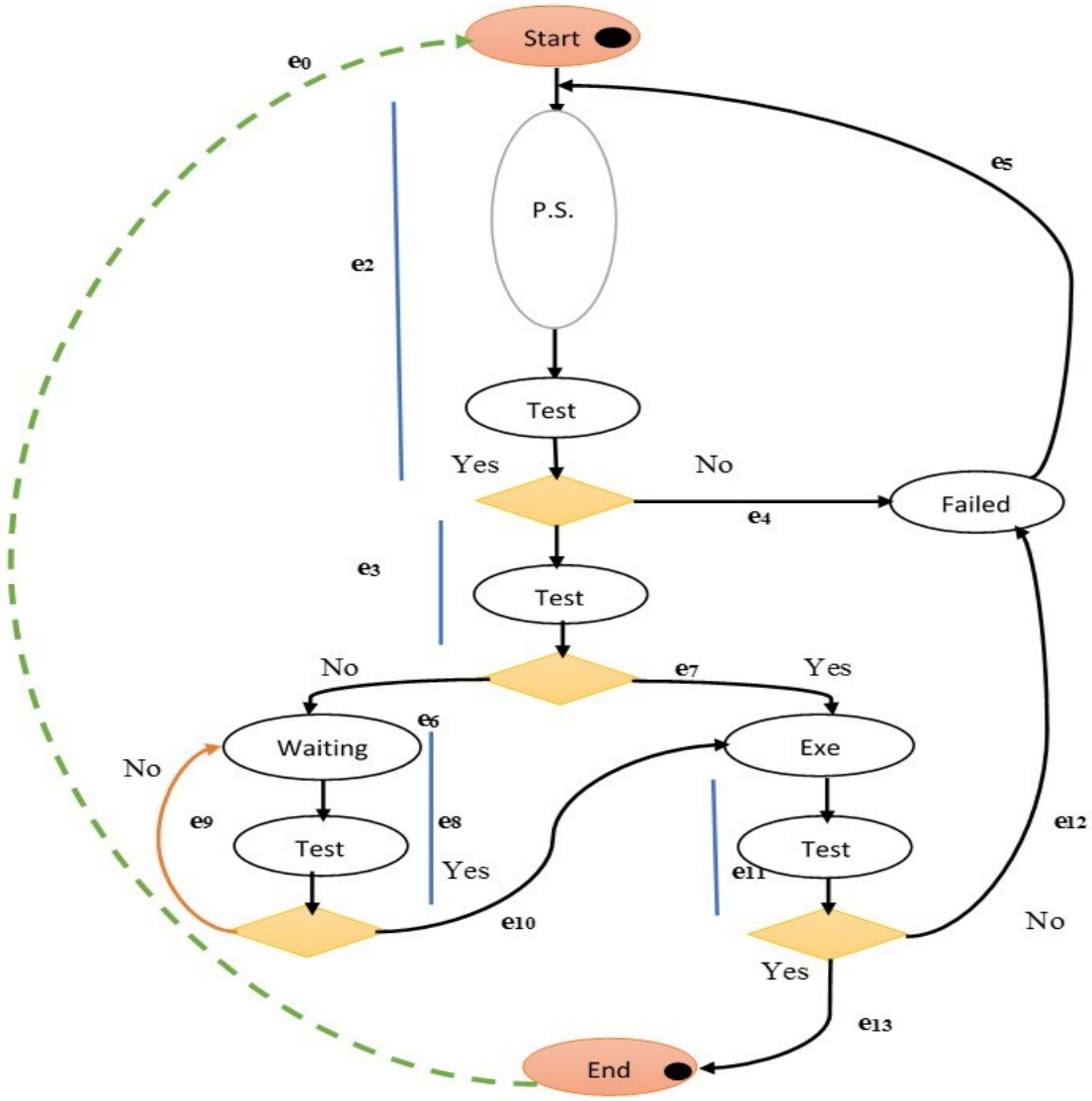


Figure 85: Modified double-thread approach

To derive a cost equation, we multiply each state cost with its associated flow parameter; then sum all results. After substituting all dependent flows with independent ones as

mentioned in chapter 3, the equation for the response time of the double threads shown in figure 85 becomes as follows:

$$Cost = delay = [(1 + e_5) * (C_{p.s.} + C_{test})] + [(e_0 + e_{12}) * C_{test}] + [(e_{10} + e_9) * (C_{waiting} + C_{test})] + [(1 + e_{12}) * (C_{Exe} + C_{test})] \quad (36)$$

In equation (36), each parameter, which represents a state cost, is associated with its flow variable(s) which is denoted by "*e*"; the *Cp.s.* is expressed as follows:

$$Cp.s. = C_{Fork} + C_{Join} + MAX\{(e_2 * C_{Initial}), (e_2 * C_{Checking})\} \quad (37)$$

The expected response time is estimated by substituting eq. (37) into eq. (36).

4.3.2 Triple-Thread Approach

Typically, there are three operations that take place in the initial state which are: 1) Creating a Thread, 2) Variables Initialization and 3) Starting the GUI. The first operation takes an average between 3% and 5% whereas the other two operations take about 95% of the total cost of the initial state [46]. By using the parallelization method, we eliminate about an average of 25% of the initial cost. Figure 86 from [46] shows the control flow graph for the sequential operation inside the initial state on the left whereas the parallel method is on the right.

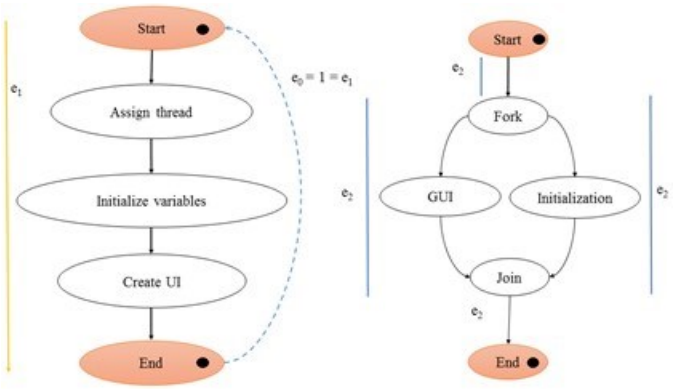


Figure 86: Control flow graph of sequential and parallel operations inside the initial state

The expected average cost for initial state using parallel approach inside it is expressed as follows:

$$C_{Initial\ State} = C_{Fork} + C_{Join} + MAX\{ (e_2 * C_{GUI}), (e_2 * C_{Variables\ Initializations}) \} \qquad (38)$$

By substituting eq. (38) into eq. (36), the expected average response time can be obtained.

4.3.3 GPUs Approach

GPUs are known today as powerful tools in parallel computation. They are highly efficient architectures in terms of performing hundreds of threads in parallel [46, 54]]. GPUs benefit comes from their parallel architectures [46,54,55,56] which make them sufficient tools for intensive operations to improve the desired performance [46]. In our research, we use OpenGL ver. 2.0 which is compatible in any device that uses either OpenGL ver. 3.0 or ver. 1.0 and it is widely used as stated earlier. Our previous study in [45] shows that the GUI operation takes about an average of 75% of the total cost of the initial state. By cooperating GPUs with the CPUs we gained about 40% speed up for a

single task while the speed up for the whole system was nearly 28% due to the fact that not all tasks require GPU to use [46].

In order to perform the analysis, we made the following assumptions: 1) Tasks arrival rates λ_i for each state which are assumed to be exponential, 2) Message multipliers β_i which are assumed to be *unity*, 3) the computation and communication cost (service) times $E[s_{ij}]$ and there is no an interruption so $\rho_i = 0$ where "i" represents the interruption source which is either communication "*c*" or execution "*e*". If there is interruption, then $\rho_i \neq 0$ and will be included in the analysis, 4) the system processing power, also known as CPU power, $\boldsymbol{PP}_s$ is *unity* where "s" represents the system. The system processing power refers to the ability of a computer to handle and manipulate data and 5) two communication channels exist in the analysis, one channel for computation of CPUs and the other channel for the computation of GPUs. The performance analysis consists two components, one for communication and the other one for computation. We will start with communication channel "side" first since it makes the analysis easiest and smooth.

4.3.4 Communication Analysis

Two common factors that affect the communication delay are the message size M_s and channel bandwidth R on which a message is sent. M_s is assumed to be 2000B on all platforms and R is found from Samsung web site; each platform comes with a unique bandwidth since several CPUs exist on each platform with different speeds. However, the communication channel bandwidth R are assumed to be fixed for all channels. We will analyze a system with *two* arrival rates λ_1 and λ_2 and *two* communication channels with

probability P_{ij} where "i" represents the source ID and "j" represents the destination channel ID. However, on Android, only one source for input which represents λ exists due to the fact that the input is initiated by a user. A system with multiple input sources and multiple communications channels is considered the most complex system for analysis and that is why we will do it. The same procedures are applied on any system with a single input source. Interested readers are referred to [3] for more information about different systems with their performance analysis approaches.

Two communication channels are used, one channel for CPU and another one for GPU. Since the message is sent from CPU to GPU, R will then be the bandwidth of CPU. The arrival rate λ_{c1} for the first channel, "dedicated to CPU" whereas the second channel is dedicated to GPU". It is computed using the following equation:

$$\lambda_{c1} = [P_{11} * \beta_1 * \lambda_1] + [P_{21} * \beta_2 * \lambda_2] \qquad (38)$$

Similarly, the arrival rate λ_{c2} for the second channel is computed in the same way. The **communication time** t_{ci} is expressed as follows:

$t_{ci} = M_i / R$; so **the communication service time** t_{mi} is estimated as follows:

$$t_{mi} = \sum \frac{flow * tci}{\lambda ci} \qquad (39)$$

Where flow = $P_{ij} * \beta_i * \lambda_i$; i represents the arrival source ID; j is the communication channel ID; the summation contains two channels. Now, the communication ρ_{ci}, which represents a percentage of CPU or GPU being busy in the communication aspect is computed using eq. (40), where "i" represents the channel ID, can be easily computed and will be used later in the analysis.

$$\rho ci = t_{mi} * \beta_i * \lambda_i \qquad (40)$$

For more than one channel, the total response communication time T_c is computed using the following equation where i is the total number of communication channels:

$$T_c = \sum \frac{tmi}{1 - \rho ci} \qquad (41)$$

4.3.5 Computation Analysis

Now it is easy and straightforward to estimate the computation response time T_e. The processing power becomes as follows:

$$PP = 1 - \sum \rho_{ci} \qquad (42)$$

Where i represents number of channels available; in our analysis, i = 2. A new expected service time $E\{sei'\}$ is computed as follows:

$$E\{sei'\} = \frac{E[se]}{PP} \qquad (43)$$

ρ_{ei}, which represents a percentage of the CPU being busy in the computation aspect that is computed using the following equation:

$$\rho_{ei} = \frac{\lambda i}{\mu i} = \frac{\lambda i}{E\{sei\prime\}} \qquad (44)$$

So the total computation delay is computed as follows:

$$T_e = \sum \frac{E\{sei\prime\}}{1 - \rho ei} \qquad (45)$$

Combining equations (45) and (41) **gives the expected average total delay T_d** which is expressed as follows:

$$T_d = T_e + T_c \qquad (46)$$

Note that $T_e \rangle\rangle T_c$.

4.3.6 Results and Discussion

In all applications, about **30 tasks** were applied more than **20 times**. All procedures were done with the *multitasking feature enabled* and *power saving mode was off*.

Double-Thread Approach Results

The following tables illustrate the results of parallel structure on all platforms.

Table 25: Results of double-thread on Note 3

Application Name	Galaxy Note 3			
	Variables Initializations	Create UI	Fork	Join
Calculator	216.72	1280	57.4	54
MobiBench	126.728	621	36.4884	31.5765
Norvigtorious	252.84	1745	66.8	63
Audio	169.764	1017	44.81	42.3

Table 26: Results of double-thread on Tab 3

Application Name	Samsung Tab 3 “7 inch”			
	Variables Initializations	Create UI	Fork	Join
Calculator	188	1342	48.88	42.3
MobiBench	146.286	678	42.12	36.45
Norvigtorious	133.644	1793	38.48	33.3
Audio	194.4	1072	50.544	43.74

Table 27: Results of double-thread approach on S 4

Application Name	Galaxy S 4			
	Variables Initializations	Create UI	Fork	Join
Calculator	239	1391	61.32	58.97
MobiBench	157.389	812	55.3	51.03
Norvigtorious	148.99	2031	65.9	64.32
Audio	274.5	1294	63.65	59.49

Table 28: Results of double-thread approach on S 4 mini

Application Name	Galaxy S 4 mini			
	Variables Initializations	Create UI	Fork	Join
Calculator	302.98	1592.1	73.01	70.68
MobiBench	231.34	1045.09	69.15	68.72
Norvigtorious	179.68	2067.44	58.56	53.9
Audio	349.12	1445.81	71.32	65.88

Tables 25, 26, 27 and 28 clearly show that the cost "time" of the fork operation takes more time than the join which is obvious since it includes the time of creating and starting another thread and also the starting time of synchronization between two threads. In our performance analysis, we will consider the average value of all performed runs and use it in the equations to estimate the average response time. Tables from 29 to 32 illustrate the average actual and expected "estimated" response times in all platforms being used.

Table 29: Average actual and expected response time on Note 3

Average Response Time (ms)		
Application Name	Actual	Expected
Audio Recording	29713	32410
Calculator	40087	44192
Mobibench	50232	54133
Norvigtorious	77943	84319

Table 30: Average actual and expected response time on Tab 3

Average Response Time (ms)		
Application Name	Actual	Expected
Audio Recording	29697	33429
Calculator	43414	45889
Mobibench	52733	58468
Norvigtorious	79005	84791

Table 31: Average actual and expected response time on S 4

Average Response Time (ms)		
Application Name	Actual	Expected
Audio Recording	29932	33857
Calculator	43891	46293
Mobibench	53563	58903
Norvigtorious	80476	86225

Table 32: Average actual and expected response time on S 4 mini

Average Response Time (ms)		
Application Name	Actual	Expected
Audio Recording	30692	34269
Calculator	45109	47322
Mobibench	53831	59879
Norvigtorious	80994	85246

Triple-Thread Approach Results

The following tables illustrate the results of parallel structure on all platforms.

Table 33: Results of Triple-thread approach on Note 3

Application Name	Galaxy Note 3			
	Create UI	Variables Initializations	Fork	Join
Calculator	1020	240	57.4	54
MobiBench	618	104.69	36.4884	31.5765
Norvigtorious	1740	206.02	66.8	63
Audio	1280	129.3	44.81	42.3

Table 34: Results of triple-thread approach on Tab 3

Application Name	Samsung Tab 3			
	Create UI	Variables Initializations	Fork	Join
Calculator	1052.89	70	48.88	42.3
MobiBench	681.3	257	42.12	36.45
Norvigtorious	1623	561.5	38.48	33.3
Audio	965	204	50.544	43.74

Table 35: Results of triple-thread approach on S 4

Application Name	Galaxy S 4			
	Create UI	Variables Initializations	Fork	Join
Calculator	1052.89	70	61.32	58.97
MobiBench	681.3	257	55.3	51.03
Norvigtorious	1623	561.5	65.9	64.32
Audio	965	204	63.65	59.49

Table 36: Results of triple-thread approach on S 4 mini

Application Name	Galaxy S 4 mini			
	Create UI	Variables Initializations	Fork	Join
Calculator	1004.34	98.03	73.01	70.68
MobiBench	531.45	213.79	69.15	68.72
Norvigtorious	1732	632.1	58.56	53.9
Audio	750	339.69	71.32	65.88

Tables 37 to 40 illustrate the average actual and expected “estimated” response times in all platforms being used.

Table 37: Average actual and expected response time on Note 3

Average Response Time (ms)		
Application Name	Actual	Expected
Audio Recording	29569	32102
Calculator	39931	43995
Mobibench	50068	54023
Norvigtorious	77830	84079

Table 38: Average actual and expected response time on Tab 3

Average Response Time (ms)		
Application Name	Actual	Expected
Audio Recording	29599	33118
Calculator	43286	45726
Mobibench	52636	58331
Norvigtorious	78942	84035

Table 39: Average actual and expected response time on S 4

Average Response Time (ms)		
Application Name	Actual	Expected
Audio Recording	29558	33739
Calculator	43693	45718
Mobibench	53104	58356
Norvigtorious	80098	85724

Table 40: Average actual and expected response time on S 4 mini

Average Response Time (ms)		
Application Name	Actual	Expected
Audio Recording	30450	33928
Calculator	45002	47075
Mobibench	53259	58912
Norvigtorious	80679	84546

Using more thread in computation reduced the response time as observed in the experiments; however, the power consumption went up by about 7% which is acceptable since the average reduction was between 10% - 14 %.

GPUs Approach Results

The actual and expected delays for all applications are shown in the following tables.

Table 41: Average actual and expected response time on Note 3

Average Response Time (ms)		
Application Name	Actual	Expected
Audio Recording	29142	30892
Calculator	39483	43225
Mobibench	49689	52137
Norvigtorious	77567	80274

Table 42: Average actual and expected response time on Tab 3

Average Response Time (ms)		
Application Name	Actual	Expected
Audio Recording	29389	31783
Calculator	43174	44301
Mobibench	52181	53682
Norvigtorious	78713	81731

Using GPU gives a significant observed reduction in the delay. However, the maximum improvement was about *600 ms* since not all tasks required GPU and the maximum utilization of it was nearly *28%*; both hardware platforms have a single GPU installed and running. Several challenges using OpenGL showed up during the conducting of experiments and can be summarized as follows:

- ✓ Compatibility: Not all devices are supported; (NDK) might be missing. More attention to a new API level 21 and up since many changes have been added and may affect the application behaviors.
- ✓ Availability: Only API ver. 11 and up.
- ✓ More RAM was required to run.
- ✓ Not all 2D operations are supported

- ✓ The more views the system has to draw, the slower it will be. This applies to the software rendering pipeline as well. Reducing views is one of the easiest ways to optimize UI.
- ✓ Poor performance if not handled carefully and improperly.
- ✓ Codes become too complex which requires a closer look at each part of the code.
- ✓ The size of the applications raised around 65% to 120%.

Advised to use it only if you have complex custom computations for scaling, rotating and translating of images, but do not use it for drawing lines or curves (and other trivial operations).

CHAPTER 5

Tasks Scheduling Algorithms

5.1 Introduction

In real-time embedded systems, scheduling policy is considered one of the main factors that affect their performance. It helps to choose which task should be selected first from ready queue to run [78]. Scheduling techniques have received much attention from researchers in the Computer Science and Engineering field [77]. Efficient CPU scheduling algorithms affect computer system performance and the need for them cannot be ignored [77]. Improper CPUs scheduling schemes degrade system performance such as response time and also increase waiting time which in turn affect the advantages of using modern processors with high speed [77]. Meeting deadlines can be reached by having an efficient task scheduler where CPU time is managed [77, 78].

In this chapter we aim to **minimize response time during run-time for periodic and aperiodic tasks in real-time embedded systems by deploying an efficient and an effective scheduling algorithm**. For aperiodic tasks, we introduce ***an improved dynamic Round Robin scheduling algorithm based on a variant quantum time*** [78].

In periodic tasks, we developed ***scheduling algorithms in real-time embedded systems based on either single value, such as WCET, or average dynamic calculations, also known as moving dynamic average, using different probability distributions.*** Since it becomes very hard to predict the WCET for tasks in many real-time applications such as multimedia applications where processing time mainly depends on the amount of data

which varies a lot. So predicting the WCET is not applicable. Thus using it as a factor to select a task or a set of tasks may give undesired scheduling results.

5.2 Real-Time and Non Real-Time

Real-time systems can be defined as those systems where their usefulness and correctness depend not only on the results of computations but also on their times at which the computations were performed and results were obtained [77]. Computations which occur in real-time systems can be known as real-time tasks [77]. Real-time tasks inherent timing constraints from their environments [77, 78]. Deadline time in real-time embedded systems is defined as a time where real-time tasks must be executed before it or at it. Non real-time systems aim to

1. Minimize the response time required to execute all tasks in their applications [77].
2. Maximize throughput of a system under consideration. [77, 78].
3. Maximize efficiency where all tasks must be executed and have a proper CPU time allocated to each task.

5.3 Real-Time Scheduling Algorithms

Two categories of real-time tasks in real-time embedded systems exist these days which are: 1. Aperiodic tasks and 2. Periodic tasks. In each category, two types of tasks exist which are: A) Preemptive: where a process “task” is blocked by another process which has a higher priority, and B) Non preemptive: any task completes its execution cycle even though there is another task with higher priority in the ready queue. The coming two subsections give a brief yet detailed explanation about each type.

5.3.1 Aperiodic Tasks

CPU performance is affected by scheduling algorithms; they provide methods for processes in the ready queue(s) to be executed [62, 63]. Tasks scheduling is considered as one of the most important areas for OS [77, 78]. It allocates time to the processes in the waiting queue; this procedure must be done in a fair way so all processes get a chance to be run and executed during their assigned time fashion for periodic tasks or in a reasonable time for aperiodic tasks. After the appearance of multitasking concept, it has become necessary to choose which job in the ready queue should be selected first to be run [63].

Several criteria determine the efficiency of the scheduling algorithms in aperiodic tasks such as 1) Average Waiting Time "AWT"; 2) Average Turnaround Time "ATT" which can be defined as the summation of the waiting time and the execution time of all tasks in the ready queue and 3) Number of Context Switches "NCS" between executed tasks. Multiple algorithms exist which can be summarized as follows:

1. First-Come-First-Serve (FCFS): a process that arrives first and is immediately allocated to the CPU. The major disadvantage of this algorithm is that a process with a small burst time takes long to be executed if another process with a long burst is chosen first as illustrated in the following table and chart respectively [64, 65].

Table 43: List of processes in ready queue

Task ID	Arrival Time	Execution Time
T_1	0	5
T_2	1	3
T_3	2	8
T_4	3	6

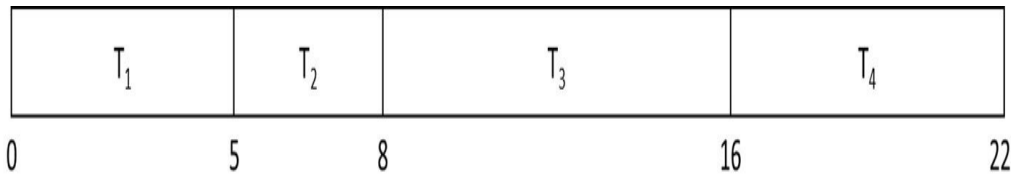


Figure 87: FCFS chart

2. Shortest Job First (SJF): this approach allocates processes with short bursts "execution" first from their ready queue [66]. It is more efficient than FSFC since it minimizes the average waiting time for a small burst. However, processes with a long burst time wait longer which cause a starvation for CPU resources as illustrated in table 43 and figure 88 [64, 67]. The drawback of this method is that it requires advance knowledge about CPU burst time which is impractical and difficult in most cases.

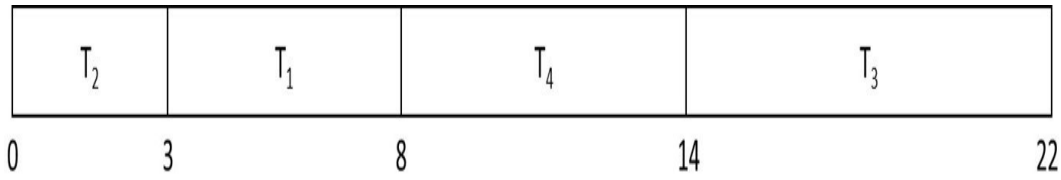


Figure 88: SJF chart

3. Round Robin (RR): each process gets its turn to be executed in a fair time slicing; this concept is known as Time Quantum (Q_t) and it is fixed for all processes [63,

64, 65]. When the quantum time for any process expires, it is temporarily blocked and placed at the end of the ready queue if its execution time is not finished or removed from the ready queue if it is done [77]. This procedure is applied on all available tasks until no more tasks exist in the ready queue [66, 67].

Round Robin algorithm efficiency depends totally on the quantum time; if it is a small amount then a frequent context switch occurs which causes too much of overhead [66,67,70]. On the other hand, quantum time that is too long increases the average waiting time and average turnaround time [67,69,70,71,72]. Figure 89 illustrates the concept of Round Robin scheduling based on processes that exist in table 43, the quantum time size is assumed to be 4 time units.

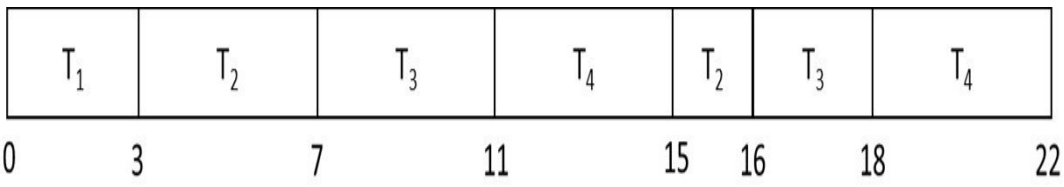


Figure 89: Round Robin scheduling algorithm

4. Earliest Deadline First (EDF): a process with shortest deadline time gets its turn first since it has the highest priority among all other processes [77]. This algorithm is considered optimal in uniprocessor environments since it utilizes 100% of the CPU and can be used for both types of tasks (periodic and a periodic) [63,65,68,69,77].
5. Fixed Priority Preemptive (FPP): every task and all its stances are assigned a fixed priority where processes with lower priority are blocked by incoming processes with higher priority [71,72,73] and it has a significant overhead. The AWT and

ATT mainly depend on the priority; processes with higher priorities get a chance to have small AWT and ATT. Otherwise, the AWT and ATT are higher for processes with low priorities. The starvation occurs since high priority processes acquire the resources and do not let other processes with lower priorities be executed [78].

6. Multilevel Feedback Queues (MFQ): a process moves between different queues; this action is characterized by the CPU burst time (B_t), also known as the execution time, if the process requires too much time then it moves to a queue where lower priority processes are placed. However, if it waits long time then it moves to the queue of the higher priority processes to prevent starvation from occurring.

In this research, we focus on the Round Robin algorithm for aperiodic tasks to minimize the response time by developing a new scheduling scheme based on a variant quantum time. The proposed scheme is discussed in detail in section 5.4.

5.3.2 Periodic Tasks

Most of the real-time systems applications such as monitoring, control loop and action planning have periodic activities and they represent the major computational aspects in the systems [77]. Those activities need to be scheduled properly in order to be executed at a specific rate. This rate can be derived from an application environment. Any periodic task is characterized by its four tuples T = {r,p,c,d}, where r represents the arrival time, p represents the period for the task, c is considered to be the worst-case execution time and d is the relative deadline time [77].

Given a set of periodic jobs (tasks) N, the processor factor utilization U, refers to the fraction of CPU being occupied in the execution of the job set, is computed as follows:

$$U = \sum_i^n \frac{Ci}{Ti} \qquad (47)$$

Equation (47) provides a measurement of the computational load on a CPU for the specific tasks set. There are several approaches that exist to schedule periodic tasks on a uniprocessor or multiprocessors and they can be summarized as follows:

1. Rate Monotonic (RM): is a preemptive fixed-priorities algorithm since it assigns a static priority to a task based on its request rate [77]. To be more specific, tasks with short periods get higher priorities; those priorities are assigned before the execution and do not change as time changes.
2. Deadline Monotonic (DM): is a static priority method where a fixed priority is assigned to each task [77].
3. Earliest Deadline First (EDF): is a dynamic approach and very simple where the earlier the deadline is, the higher the priority is. This method is considered optimal on single processor since it schedules all tasks correctly [77,79].
4. Least Slack Time First (LST): is a dynamic method where a smaller slack time gets higher priority and is assigned to the CPU. This approach is very effective on uniprocessor [77,79].

However, all previous methods do not provide maximum utilization on multiple processor environments since CPUs are idle in some times which lead to deadline misses in some cases. Hence a technique to give maximum utilization on each CPU is required and necessary. For this reason, we developed a method to keep all CPUs busy as much as possible; the purpose is to distribute a task execution load among existing processors.

Section 5.5 gives details of it using either a single value such as WCET or dynamic average estimation based on different probability distributions [77,79,80].

5.4 Developed Scheduling Algorithm in Round Robin for Aperiodic Tasks

An operating system is the interface between a user and a machine and it has many features to deliver an excellent service to the user. Scheduling is one of that fundamental features and it is responsible for deciding which job is selected and run from the ready queue [61,63,64,65,78]. Scheduling method affects CPU performance since it determines the CPU and resources utilizations [61,62,78]. The main purpose of scheduling policy is to ensure complete fairness between different tasks in the ready queue, maximizing the throughput, minimizing the average waiting, turn-around times and the overhead occurs from context switches and makes sure no starvation happens at all.

Several factors are used to determine whether a scheduling policy is good or not and can be summarized as follows:

A. Waiting time: the time between tasks that become available in the ready queue until the first time of their execution.

B. CPU Utilization: the percentage of the CPU being occupied.

C. Turn-around time: the summation of waiting and execution time for each task as mentioned earlier.

D. Fairness: which is dividing the CPU time equally among all available jobs [61,62,63,78].

In today's technology, many operating systems perform multitasking operations which mainly depend on scheduling algorithms to ensure that all processes meet their deadline times and execute fairly [62,78]. Multitasking can be defined as a concept of performing multiple operations at the same time. However, it does not imply that all tasks, also known as processes, are executed in parallel.

In Round Robin, a concept called time slicing is used where each process gets the same amount of time for its execution and this concept is known as quantum time [61,68,69,78].

In [61], A. R. Dash et al proposed an optimized Round Robin algorithm with dynamic quantum time. They claimed that their approach is the optimal one. However, our proposed scheme achieves better results in terms of the average waiting time, the average turn-around time and the number of context switches as proved by experiments performed under several circumstances [61,65,78]. They named their algorithm “DABRR” which stands for Dynamic Average Burst time Round Robin. In Round Robin algorithm. Burst term refers to the execution time. Their approach works based on finding the mean of burst times of all processes available in the ready queue. If there is only one process left in the ready queue, then the quantum time will be the burst time associated with that process. We conducted a comparison analysis between DABRR, our proposed method and also several versions of Round Robin are included in the comparison.

In [63], D. Maste et al proposed an intelligent dynamic Round robin algorithm for multilevel feedback queues. Each queue is assigned a time slice and it changes with each round of execution dynamically. Neural Networks (NN) were used to control the value of the quantum time to optimize the turn-around time. To find the dynamic quantum time,

they considered burst time span as a method to obtain the quantum time with help from average priorities and highest priority of a queue. The overhead that occurred from their approach was higher than expected whereas our scheme minimizes the overhead since the quantum size is big enough so that each process gets sufficient time to complete its execution time if possible which implies that the overhead will be minimum [78].

A. Noon et al in [64] proposed a new dynamic Round Robin scheduling algorithm using the mean average as a method to compute a new value for quantum time in each round [78]. The operating system decides the value of quantum time based on the burst time of the existing set of tasks in the ready queue. Their algorithm gives a better result in terms of the average waiting and turn-around times compared to the static Round Robin scheme [78]. However, those values are still high and more modification is needed in order to achieve better results. In addition, the number of context switches that occurs in their scheme is still high and causes too much overhead; our approach achieves a small number of context switches which implies that less overheard occurs.

I. S. Rajput and D. Gupta in [65] developed a priority based Round Robin scheduling algorithm for real-time systems. Their proposed architecture was implemented to gain a good performance in terms of context switch and the average waiting and turn-around times in the static version of Round Robin scheme. However, it did not provide more improvement in the context switches, the average waiting and turn-around times while our proposed scheme gives a better performance since all previous parameters are minimized as much as possible.

In many real-time systems, such as servers, Android, industrial plant monitoring, an alert might be produced after a set of readings from sensors reach a certain level of hazard detection. Anti-lock Brakes (ABS) in cars also have aperiodic tasks which need to be processed as fast as possible; which in turn means that their response time must be as minimum as possible [78]. The developed algorithm is suitable for any real-time system with aperiodic tasks [78]. We have chosen Android as a case study since some versions of it still use Round Robin as the scheduling policy beside other schemes. In the Android platform, the static Round Robin is used and the developed algorithm fits there and can produce a significant reduction in the response time. The average expected improvement in the response time is around 40% to 55% when compared to the static approach as shown in our research in [78]. Furthermore, we achieved the minimum number of context switches as proved by a simulation system we developed to test and show the validation of the developed method.

In Round Robin (RR) algorithm, the performance mainly depends on the size of its quantum time (Q_t); so a small size gives poor performance while a size that is too large tends to make the algorithm be "FCFS" [78]. So choosing the size of the quantum time is very critical to enhance the system performance and for this reason it was the motivation behind the developed approach. The size of the quantum was selected to be large enough in order to accommodate more processes to finish their execution times to minimize the overhead occurring from context switches [78]. However, that size must not lead the developed algorithm to degenerate like FCFS. So to choose the large time slice "quantum", the average of the two highest burst times was computed and then the average of the two

lowest arrival times was taken from that estimated value for one time only; later, we subtract the average of the arrival time for only the lowest process that existed in the ready queue. The reason behind that is to keep the quantum as large as possible while maintaining its properties as Round Robin method [78]. The following pseudo code describes how the developed scheme works.

Algorithm: Improved Dynamic Round Robin

1. *Burst Time B_i; Arrival Time Ar_i; Tasks T; Average Waiting Time AWT; Average Turn-Around Time ATT; Number of Context Switches NCS; Quantum Time Q_t*
2. ***Initialization: AWT = 0, ATT = 0 and NCS = 0***
3. ***While (ready queue != 0)***
4. *If (all arrival times Ar_i == 0), then*
5. *Sort all tasks in ascending order based on their burst time values*
6. ***Q_t = ∑ of the two highest B_i / 2***
7. *Assign Q_t to every process in the ready queue to proceed*
8. ***If ($[B_i(i) - Q_t] == 0$), then***
9. *Remove $B_i(t)$ from ready queue*
10. *else*
11. *continue proceeding with remaining tasks*
12. ***Q_t = ∑ of the highest remaining B_i with old Q_t / 2***
13. *Continue until no more processes in the ready queue*
14. *If a new process arrives, insert it at its right place*
15. ***If ($A_r(i)$ has different value), then***
16. ***Q_t = the first arrived task***
17. *Assign Q_t to available process in the ready queue to proceed; if more processes arrive in the ready queue, then,*
18. *Sort all tasks in ascending order based on their burst time values*
19. ***Q_t = ∑ of the highest remaining B_i with old Q_t / 2 - ∑ of the two lowest remaining Ar_i / 2; for once only.***
20. ***If ($[B_i(i) - Q_t] == 0$), then***
21. *Remove $B_i(t)$ from ready queue*
22. *else*
23. *For next rounds;*
24. ***Q_t = ∑ of the highest remaining B_i with old Q_t / 2 - the lowest remaining Ar_i / 2.***
25. *If new process arrives, just insert it at its right position*
26. **Until** (*the end of sorted processes is reached*)
27. *Calculate AWT, ATT and NCS*

Several examples are presented to demonstrate how the approach works, then a comparison analysis between the developed method with variant versions of Round Robin algorithms is presented.

Case 1 (in [64]): Same arrival time for four processes in the ready queue as shown in table 44.

Table 44: Available tasks in the ready queue for case 1

Tasks	Arrival Time (Ar_t)	Burst Time (B_t)
T_1	0	40
T_2	0	20
T_3	0	80
T_4	0	60

The developed approach works as follows:

a) Sort all processes according to their burst times as shown in table 45.

Table 45: Sorted processes in case 1

Task	Arrival Time (Ar_t)	Burst Time (B_t)
T_2	0	20
T_1	0	40
T_4	0	60
T_3	0	80

b) For round 1: $Q_t = [60 + 80] / 2 = 140 / 2 = 70$ time units

c) For round 2: $Q_t = [70 + 10] / 2 = 80 / 2 = 40$ time units

d) The Gantt chart for all processes is as follows:

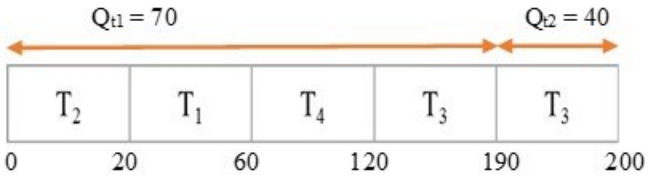


Figure 90: Gantt chart of case 1

Table 46 illustrates the AWT and ATT for all processes shown in table 46

Table 46: Results for the AWT and ATT for case 1

Tasks	Waiting Time	Turn-around Time
T_1	20	40
T_2	0	40
T_3	120	200
T_4	60	120
Average	50	100

Case 2 (in [67]): Same arrival times; table 47 lists five tasks with their arrival and burst times.

Table 47: List of 5 processes in case 2

Tasks	Arrival Time (A_{rt})	Burst Time (B_t)
T_1	0	80
T_2	0	45
T_3	0	62
T_4	0	34
T_5	0	78

The Gantt chart for processes in case after applying the proposed method is shown in figure 91

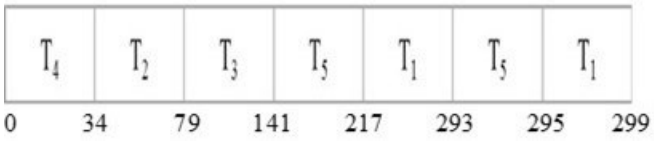


Figure 91: Gantt chart for case 2

Table 48 illustrates the results for both parameters "AWT and ATT" after repeating the previous procedures.

Table 48: Results for the AWT and ATT in case 2

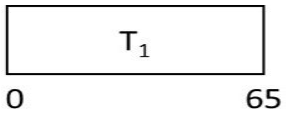
Tasks	Waiting Time	Turn-around Time
T_1	219	299
T_2	34	79
T_3	79	141
T_4	0	34
T_5	141	217
Average	94.6	154

Case 3 (in [67]): Different arrival times; five tasks are listed in table 49 with their arrival and burst times. The procedures are the same except that the quantum time Q_t for ***round 1*** is determined to be the burst time value associated with the first arrived task.

Table 49: List of available processes in case 3

Tasks	Arrival Time (Ar_t)	Burst Time (B_t)
T_1	0	65
T_2	1	72
T_3	4	50
T_4	6	43
T_5	7	80

a) For round 1: **Qt = 65 time units**. It is assigned to the first process and it is executed



b) For round 2: several processes are in the ready queue; repeats of the same procedures were performed in the previous example. Table 50 shows the obtained results "AWT and ATT" for all tasks. The Gantt chart is shown in fig. 92.

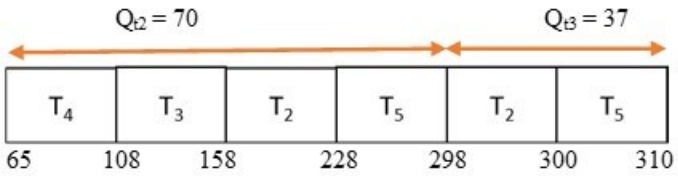


Figure 92: Gantt chart for case 3

Table 50: Results for the AWT and ATT in case 3

Tasks	Waiting Time	Turn-around Time
T_1	0	65
T_2	227	299
T_3	104	154
T_4	65	102
T_5	223	303
Average	122.6	184.6

Comparison Analysis

Several versions of Round Robin algorithm were used to perform a comparison analysis which includes static and dynamic as shown in our research in [78]. ***For the static version, the quantum time size was chosen to be 25 time units***. All tasks with their arrival

times and burst times were taken from [61] since it is the most recent paper in this field. For more information about several algorithms of Round Robin being used within this paper, the readers are referred to [61], [66] and [71] respectively. The comparison includes the algorithms DABRR in [61], static Round Robin S.R.R, DQRRR in [66] and SARR in [71] and lastly the developed approach. The comparison analysis was conducted based on the three performance parameters which are the Average Waiting Time (AWT), the Average Turn-around Time (ATT) and the Number of Context Switches (NCS). The objective of this comparison analysis is to show values of all three parameters.

Example 1:

Table 51: List of processes in example 1

Tasks	Arrival Time (A_{rt})	Burst Time (B_t)
T_1	0	40
T_2	0	55
T_3	0	60
T_4	0	90
T_5	0	102

Figures 93, 94 and 95 show the comparison analysis results for all several Round Robin algorithms mentioned earlier which were taken from [78].

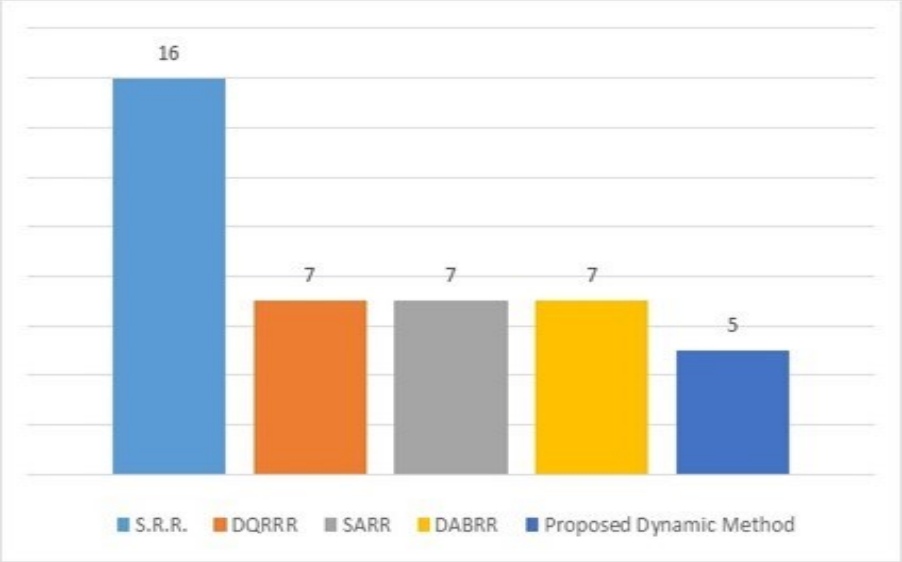


Figure 93: Number of context switches results in example 1

The developed algorithm produced the minimum number of context switches among other four algorithms [78].

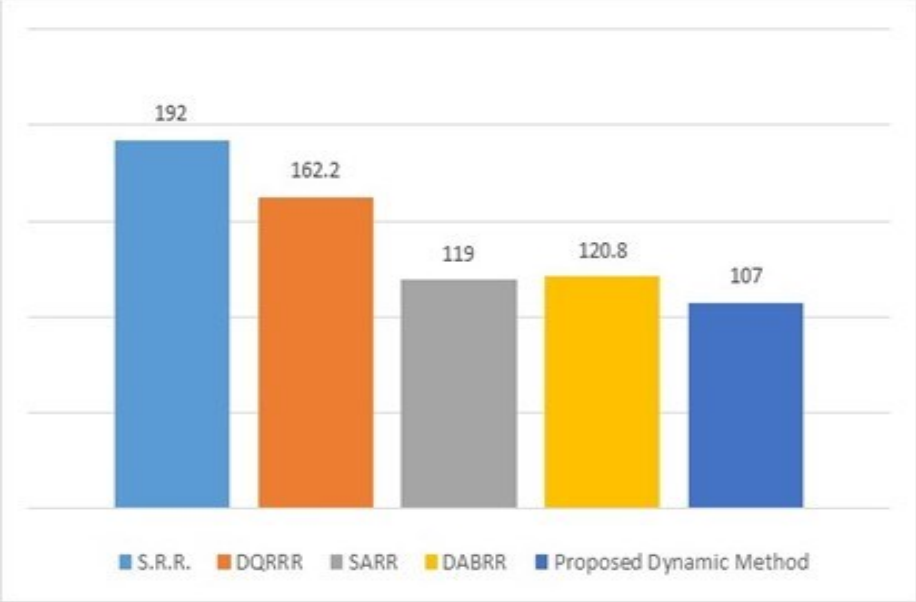


Figure 94: Results of the average waiting time in example 1

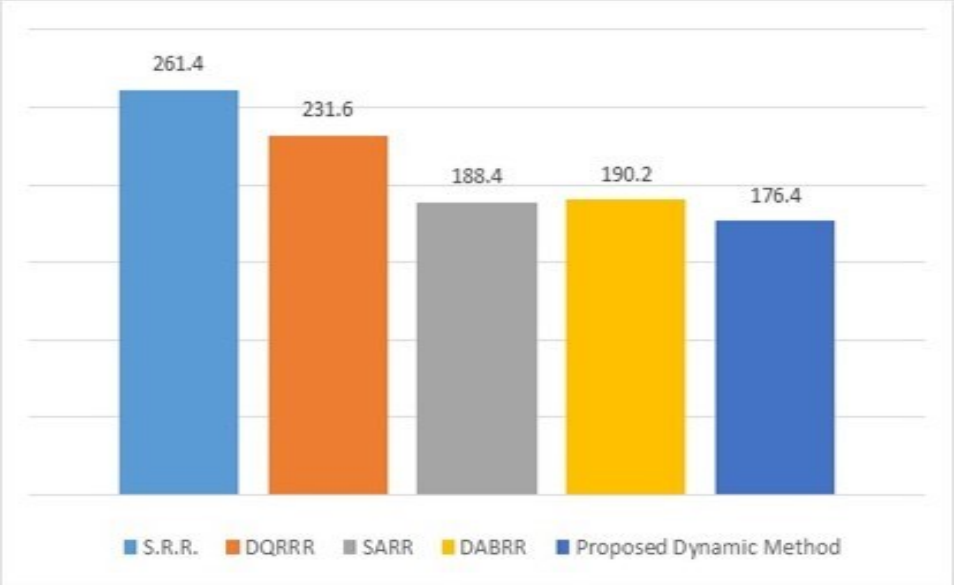


Figure 95: Results of the average turn-around time in example 1

The developed scheme yielded a better result for both the average waiting time and the average turn-around time as shown in figures 94 and 95 respectively [78]. By improving all three parameters, the throughput is improved too which implies that the response time is also minimized.

Example 2:

Table 52: List of processes in example 2

Tasks	Arrival Time (Ar_t)	Burst Time (B_t)
T_1	0	105
T_2	0	85
T_3	0	55
T_4	0	43
T_5	0	35

Figures 96, 97 and 98 show the comparison analysis results for AWT, ATT and NCS respectively.

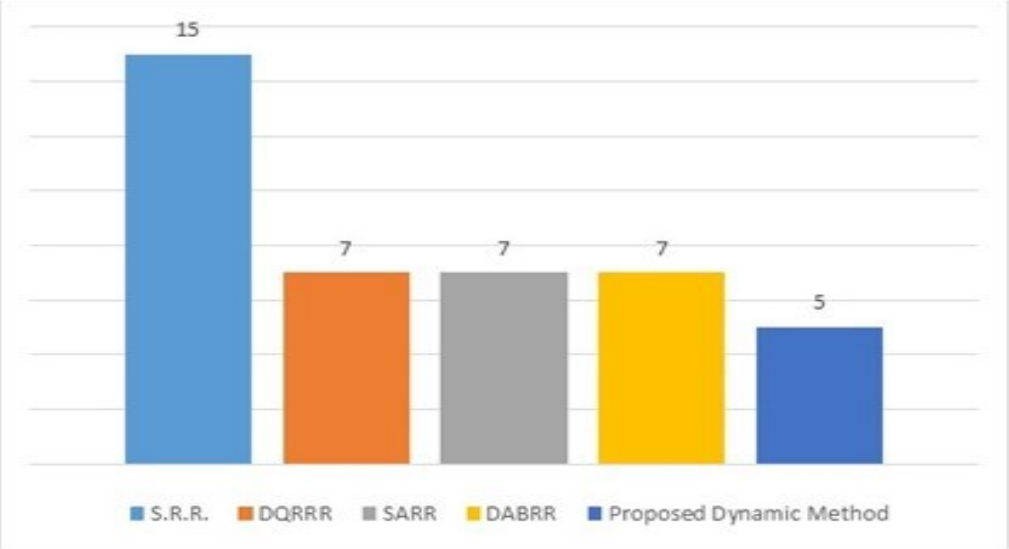


Figure 96: Results of the number of context switches in example 2

The minimum number of context switches was achieved by the developed algorithm.

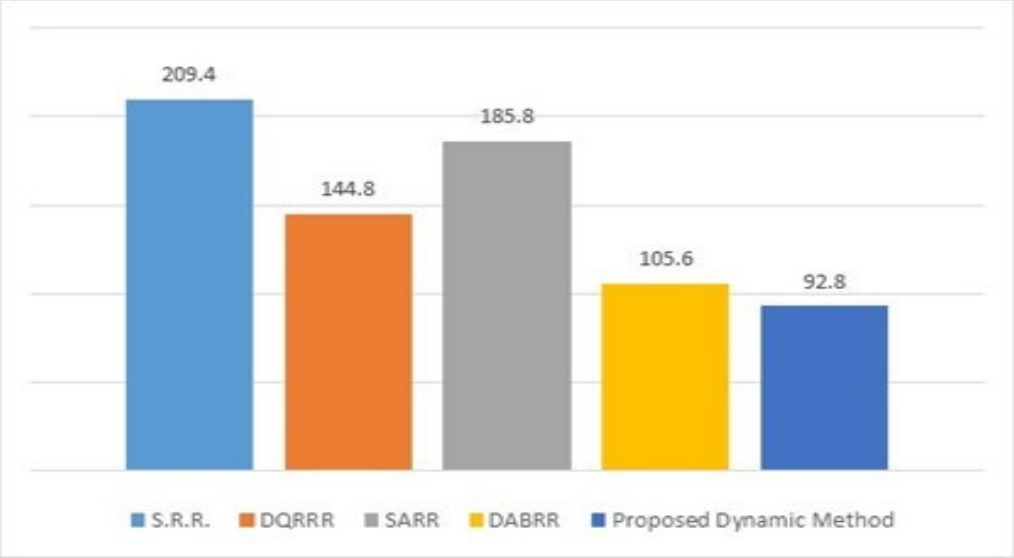


Figure 97: Results of the average waiting time in example 2

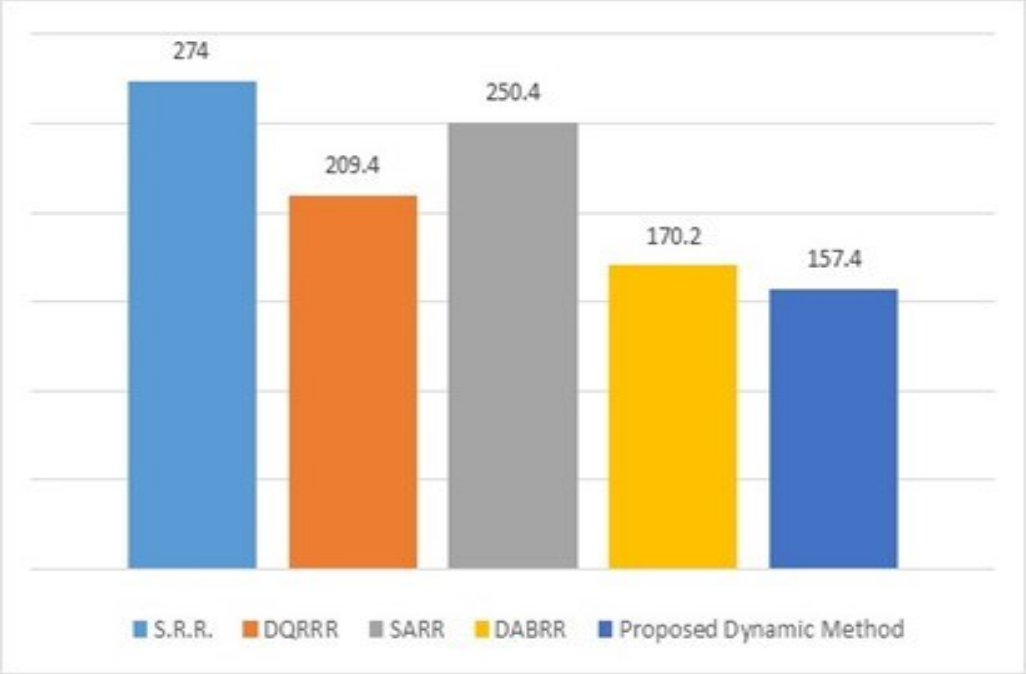


Figure 98: Results of the average turn-around time in example 2

The developed algorithm produced the minimum average waiting and turn-around times.

Example 3:

Table 53: List of processes in example 3

Tasks	Arrival Time (Ar_t)	Burst Time (B_t)
T_1	0	45
T_2	5	90
T_3	8	70
T_4	15	38
T_5	20	55

The results of all three performance metrics are shown in the following figures.

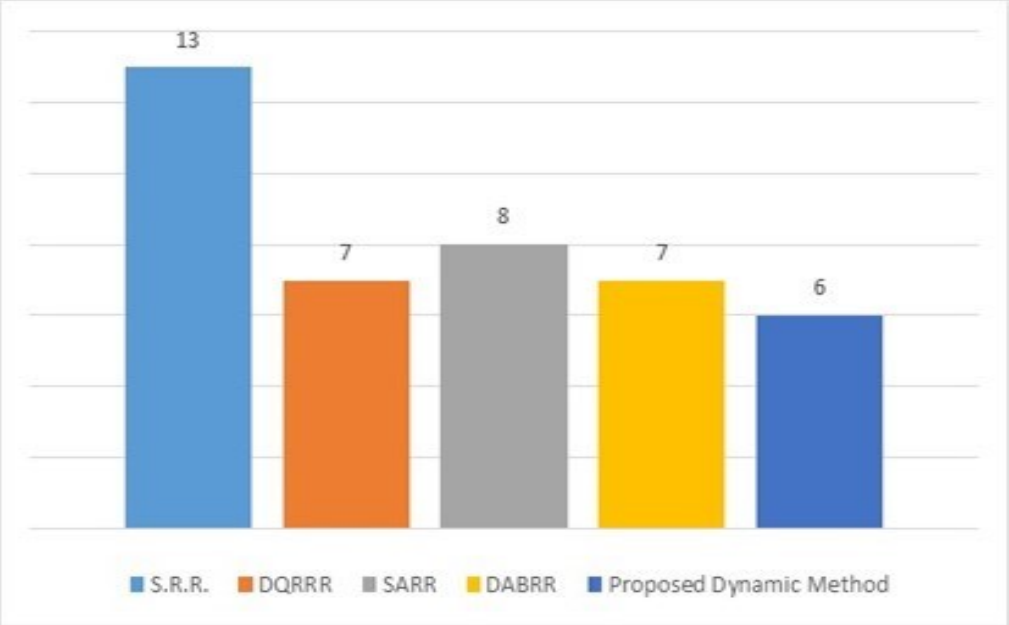


Figure 99: Results of the number of context switches in example 3

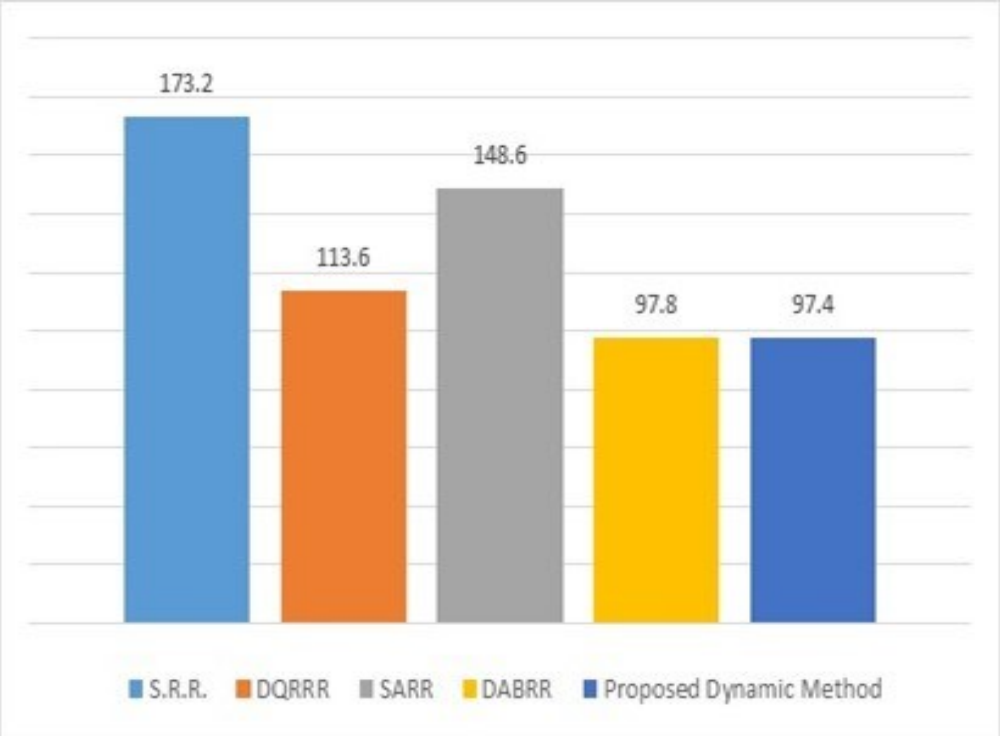


Figure 100: Results of the average waiting time in example 3

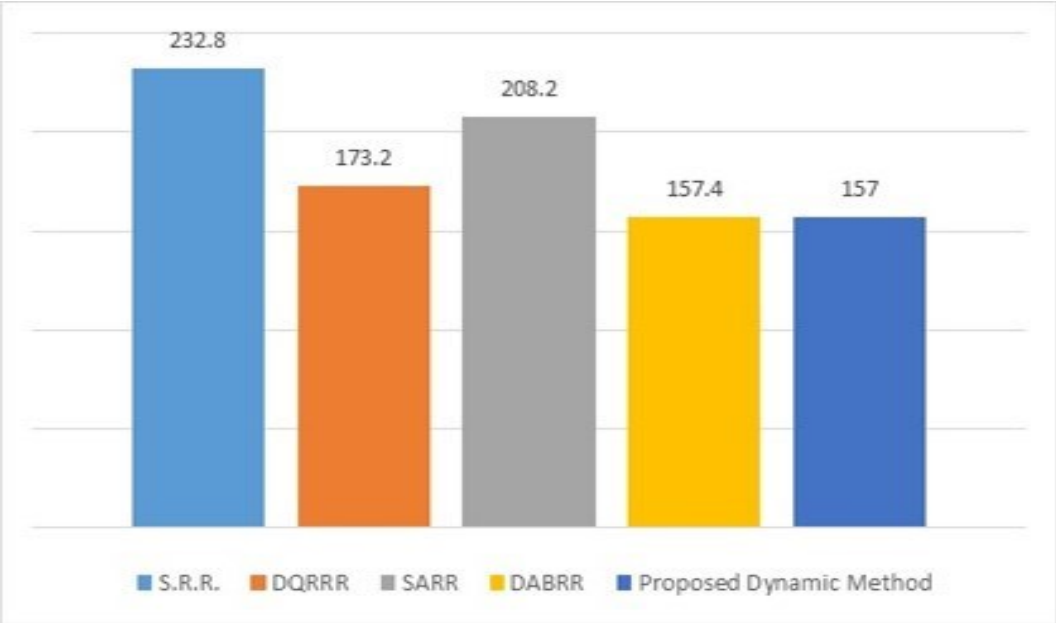


Figure 101: Results of the average turn-around time in example 3

The previous three figures illustrate that the developed algorithm produced the minimum number of context switches and average value for both waiting and turn-around times.

The developed algorithm yields better results for all three performance parameters of the Round Robin as shown in the previous comparison analysis results. The number of context switches (NCS) is dramatically decreased in the developed scheme. All previous cases were given to demonstrate the usefulness of the developed method; however, we performed more experiments using a developed simulation system. More than 100 tasks were tested more than 7000 times with random execution and arriving times. The maximum number of processes generated by the simulation was around 200 which took around an hour to complete. It is obvious that the elapsed time would be small if all processes had the same arrival times. Fig. 102 illustrates the result of several tasks in

Android between static Round Robin and the developed approach for the AWT and the ATT.

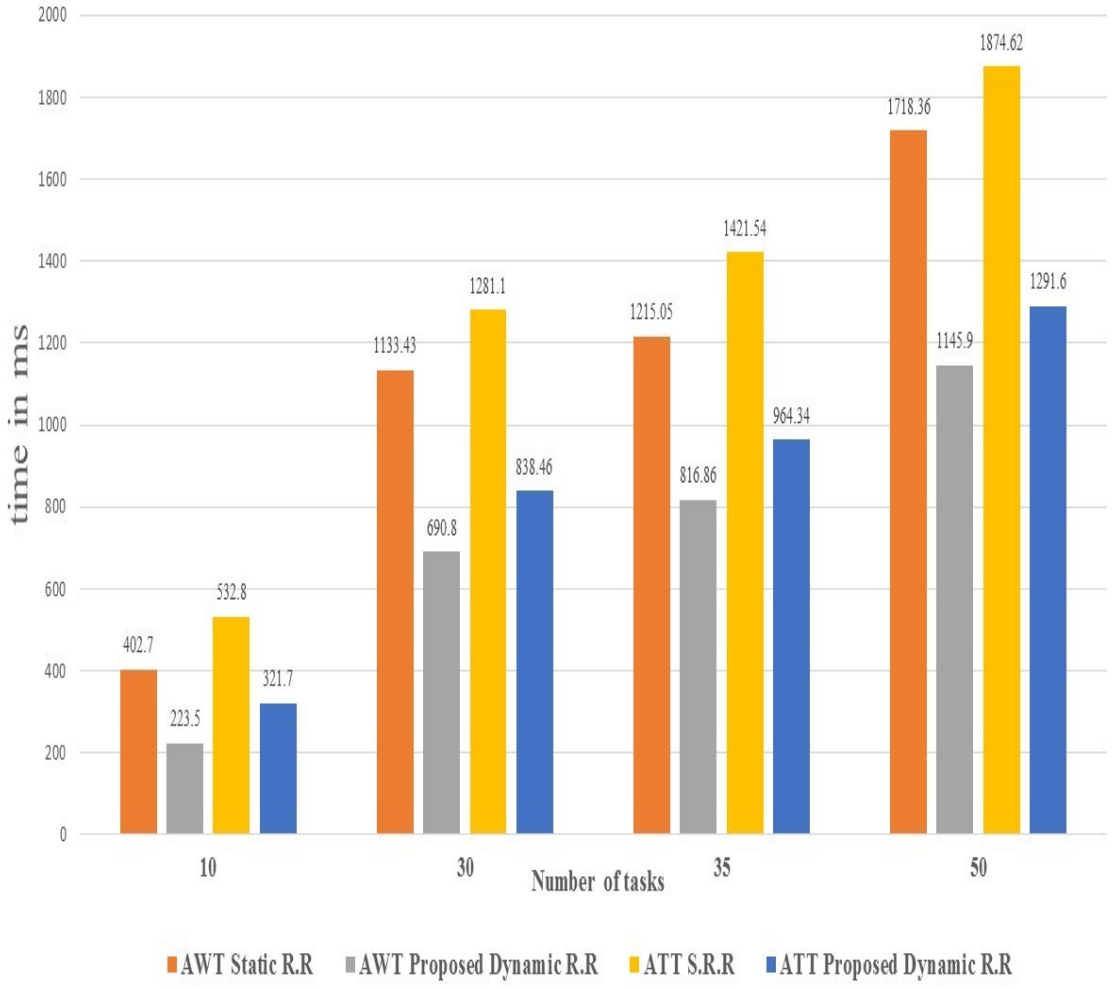


Figure 102: Results of the several tasks in Android

The developed dynamic approach minimized both metrics (AWT & ATT). However, the average improvement was about 36% while the maximum reduction was nearly 43%.

5.5 Developed Scheduling Algorithms for Periodic Tasks

Most of the real-time systems applications such as monitoring, control loop and action planning have periodic activities and they represent the major computational aspects in the systems [81,82]. Those activities need to be scheduled correctly in order to be executed at a specific rate. This rate can be derived from application environments; such activities have to be executed before or at their deadlines [79,80,81]. Choosing which job(s) must be selected first, its or their parameters play a significant role on system performance. The main objective of scheduling is to decide which job is selected and run from the ready queue and assigned to the CPU [81]. Scheduling method affects CPU performance since it determines the CPU and resource utilizations [79,80,81,82]. Two types of real-time systems exist nowadays and can be summarized as follows [79, 80]:

I. Hard systems in which deadline miss means fail and could lead to a disaster result [79].

II. Soft systems where a deadline miss is tolerated and they still perform their functions [79,80].

Scheduling can be defined as a method that specifies which task or a set of tasks is assigned to resources in order to complete a desired job [79,80,81].

A scheduler is responsible for scheduling an activity; it is implemented to ensure that all resources are kept busy and to give users availability to share different resources effectively [79,80]. In real-time systems, schedulers are developed to make sure all

processes meet their deadlines for stability sake/severity. Figure 103 illustrates the time constraints of periodic tasks.

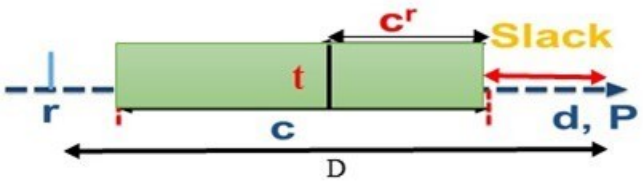


Figure 103: Time constraints of periodic tasks

Figure 103 shows time constraints of periodic tasks which are [79,80]:

1. Release time (r): which is the time at which a process becomes available at the ready queue

2. Execution time (c) which is considered to be the Worst-Case Estimated Time "WCET".

3. Period (P): the time when the process repeats its cycle.

4. Absolute deadline (D): which is the time interval between release time and period of the process. In mathematical form, D = d – r.

5. Relative deadline (d): which is an interval time between the first creation of the process and its deadline; mathematically, it is d = D + r. In many cases, P and d are the same. Slack is defined as the difference between the deadline (d) of any task with its remaining execution time (c^r) and current time as depicted in fig. 103.

Our contribution in this area is done by developing new hybrid scheduling algorithms for periodic tasks that work either on a uniprocessor or multiple processors systems; by hybrid we mean it cooperates with the EDF algorithm when needed [79,80]. They work during run-time to decide which task or a set of tasks should be selected first

from the ready queue and gain system resources such as CPU [79,80]. The main objectives of these algorithms are to:

1. Ensure that all processes meet their deadlines.
2. Keep a system stable.
3. Eliminate idle state of all existing CPUs.
4. Provide a good punctual response time.

Two scheduling algorithms have been developed to schedule periodic tasks in real-time embedded systems which are as follows:

- *Using single value "WCET" as factor to determine which task or a set of task must be selected first.*
- *Using dynamic average estimation, also known as dynamic moving average, for several probability distributions "PDFs" since it is impractical to use WCET as a factor when data size varies a lot which makes it hard to predict the value of WCET.*

5.5.1 Developed Scheduling Using Single Value "WCET"

The previous mentioned techniques such as RM, DM, LST and EDF are applicable on a uniprocessor and are not preferred on multiple ones since they leave some CPUs with idle states and some deadlines are missed [79,80]. Even the method described in [84] is unable to provide an optimal solution since it is useless on tasks with different arrival times.

The objectives of the developed algorithm are to 1) minimize response times if possible, 2) make sure all processes meet their deadline times and 3) keep all resources utilized [79,80]. Several assumptions are taken into consideration in order for the method to work in a properly way and give the best results; the assumptions are as follows [79,80]:

A. Preemptive approach which means any task can be blocked by another task with higher priority.

B. Task migration is allowed so the task finishes its execution on any available processor.

C. All tasks in the ready queue are available upon selection and they are independent.

D. Any process is not allowed to appear on multiple processors at the same time.

E. Combines with EDF algorithm when and if needed.

The following steps describe how the developed scheme works [79,80]:

1. All tasks in the ready queue are examined by each time unit to decide which one should be selected first to assign to available resources; we have chosen the time unit to be 1 ms which is the conventional time unit in many applications.
2. A rate or ratio R_i, where i denotes the task index, is computed using the following equation:

$$R_i = \frac{slacki}{(di - t)} \qquad (48)$$

where d_i is the relative deadline and t is the current time as stated earlier. The dominator part represents how much time left until the deadline. Fig. 104 illustrates both quantities of the equation (48). The light blue arrow points to the current time (t); the small black arrow represents the slack which is the nominator in eq. (48) while the dashed bold yellow arrow indicates the time left until deadline (as shown in our research in [79] and [80]).

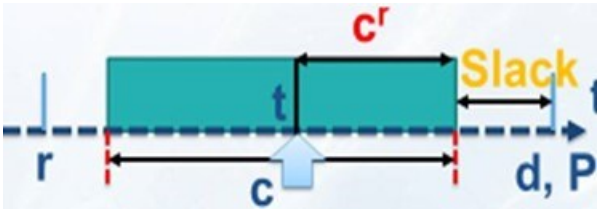


Figure 104: Quantities of eq. (48)

3. The task with smallest rate gets the highest priority and assigned first to the system resources; if more than one task have the same rate; then the task or set of tasks with shortest deadline is selected first.
4. All previous procedures are repeated until the ready queue becomes empty.

The following example "*taken from 85*" illustrates the motivation to come up with the developed approach to schedule periodic tasks.

Example 1: Same arrival time for five processes in the ready queue as shown in table 54 where three CPUs exist and are used.

Table 54: Available processes in the ready queue in example 1

Tasks	Release Time	Deadline Time	Execution Time
T_1	0	2	1
T_2	0	2	1
T_3	0	2	1
T_4	0	8	6
T_5	0	8	6

Using the EDF algorithm, the scheduling result is shown in table 55. P_i indicates the processor ID number.

Table 55: Result of the EDF using 3 CPUs

Time / Processor ID	0	1	2	3	4	5	6	7	8
P_1	T_1	T_4	T_1	T_4	T_1	T_4	T_1	T_4	T_4
P_2	T_2	T_5	T_2	T_5	T_2	T_5	T_2	T_5	T_5
P_3	T_3	NOP	T_3	NOP	T_3	NOP	T_3	NOP	T_1

In processor 3, NOP represents no operation at that time which means it was idle. So T_4 and T_5 missed their deadlines. Both processors 1 and 2 were totally busy while processor 3 was only busy for about 55% of its time.

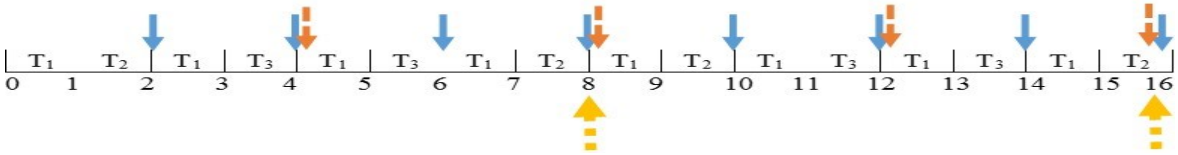
Three examples are presented to demonstrate how the developed scheme performs [79,80]. Both examples can be found in [84,85]. We will apply the developed method on multiple processor environments since it is our concern. First, we will perform the EDF method on one example" as shown in our research in [79] and [80] to show its weakness on multiprocessor and then perform our approach on it to show the difference in the results.

Example 1:

Table 56: List of processes in the ready queue for example 1

Tasks	Release Time	Deadline Time	Execution Time
T_1	0	2	1
T_2	0	4	1
T_3	0	8	2

The Gantt chart for the proposed method is shown as follows:



The blue arrow points to the relative deadline for T_1, the orange dashed arrow points to the relative deadline for T_2 and the yellow dashed arrow points to the relative deadline for T_3 [79,80]. All three processes were scheduled successfully and no deadline miss occurred.

Example 2: Same example we stated earlier for the motivation which is shown in table 54. The results using the developed algorithm is shown in table 57.

Table 57: Results of example 2 using the developed approach

Time / Processor ID	0	1	2	3	4	5	6	7
P_1	T_1	T_2	T_1	T_2	T_1	T_2	T_1	T_2
P_2	T_4	T_3	T_4	T_3	T_4	T_3	T_5	T_3
P_3	T_5	T_4	T_5	T_5	T_5	T_4	T_4	T_5

Example 3: Multiprocessors (4 CPUs) as shown in table 58.

Table 58: 9 processes in the ready queue for example 3

Tasks	Release Time	Deadline Time	Execution Time
T_1	0	15	8
T_2	0	6	5
T_3	0	10	4
T_4	0	4	3
T_5	0	4	2
T_6	0	3	1
T_7	0	3	1
T_8	0	5	1
T_9	0	60	7

Using the developed method (snap shot of the scheduling approach due to space limitation).

Table 59: Result using the proposed method in example 3

Time / Processor ID	0	1	2	3	4	5	6	7	8	9	10	11
P_1	T_1	T_1	T_2	T_1	T_2	T_1	T_2	T_1	T_2	T_1	T_2	T_1
P_2	T_2	T_2	T_3	T_2	T_4	T_2	T_3	T_2	T_3	T_2	T_4	T_3
P_3	T_4	T_4	T_4	T_5	T_6	T_4	T_4	T_6	T_4	T_4	T_5	T_7
P_4	T_5	T_6	T_7	T_8	T_7	T_5	T_5	T_7	T_8	T_5	T_6	T_8

Tables 57 and 59 indicate that all tasks met their deadlines and also all of the three CPUs were fully busy and utilized.

Simulation Experiments

A simulation system using Matlab 2015 was developed to test the developed algorithm under various conditions [79,80]. The simulation proved validation of it and showed it provides the desired results. More than *20000 task sets* were tested with an *average of 30 tasks in each set*; *each set ran for an average of 5000 times*. The simulation system works for uniprocessor and multiprocessor as well; several processors were used and the maximum number was *10 CPUs* [79,80].

The simulation tells how many tasks met their deadlines, how many tasks missed their deadlines and the elapsed time to complete all sets. The execution time (C) and deadline time (d) were randomly generated by the simulation where d is greater than C and several tasks may have the same deadline times; the same applies on the execution time (C). The maximum deadline time was set to 60. The arrival time (r) was also generated randomly by the simulation under a constraint that $r < c$ and d.

Table 60 contains information about a device we used to test the developed algorithm. Several experiments with around 100 tasks in multiple sets were performed to exploit how the simulation behaves and produces results under multiple circumstances [79,80].

Table 60: Characteristics of the used platform

Platform Name	System Type	CPU	Speed	RAM
Windows 10 Pro	64 bit	I5 core 2 Due	2.67 Ghz	4 GB

Example 1: Uniprocessor with the same arrival time (r = 0)

Table 61: Results of using uniprocessor with the same arrival time in example 1

Number of Iterations	Number of sets and tasks	Number of completed tasks	Number of deadline miss	time
5000	5/24	17	0	245s
3000	5/30	8	0	224s
7000	5/30	11	0	429s
10000	5/20	25	0	419s

Table 61 shows that the developed algorithm was successfully scheduled for all tasks and that no task missed its deadline time. GUI in the simulation system took an average of *24*s in each run which is included in the results in time column.

Example 2: Uniprocessor with different arrival times

Table 62: Results of using uniprocessor with the different arrival times in example 2

Number of Iterations	Number of sets and tasks	Number of completed tasks	Number of deadline miss	time
8500	5/23	19	0	480s
9300	5/15	166	0	241s
10000	5/30	6	0	599s
10000	5/20	29	0	375s

There was no deadline miss as shown in table 62 and the arrival time values (r) influenced the number of completed tasks which met their deadlines.

Example 3: Multiple processors with the same arrival time. M represents number of processors; M = 3.

Table 63: Results of 3 CPUs with the same arrival time for all processes

Number of Iterations	Number of sets and tasks	Number of completed tasks	Number of deadline miss	time
5000	5/24	757	0	189s
3000	5/30	314	0	180s
7000	5/30	1299	0	365s
10000	5/20	2338	0	379s

Table 63 shows the results of using 3 processors under the same conditions we used for uniprocessor in example 1 [79]; no deadline miss occurred and the number of completed tasks was doubled several times. The elapsed time reduced significantly which improved the response times. The same test was repeated with different arrival times for each process; the number of processes which met their deadline times varied from run to run due to the fact that different arrival time values were randomly generated.

Example 4: Same arrive time with M = 5 processors; same conditions in example 3 were applied.

Table 64: Results of 5 CPUs with the same arrival time for all processes in example 4

Number of Iterations	Number of sets and tasks	Number of completed tasks	Number of deadline miss	time
5000	5/24	1639	0	173s
3000	5/30	695	0	165s
7000	5/30	1636	0	382s
10000	5/20	3653	0	359s

The developed approach scheduled all tasks or sets of tasks successfully without any deadline miss. The elapsed time was significantly reduced as shown in table 64.

Example 5: different arrival times with M = 7 and the number of sets varied in each run

Table 65: Results of 7 CPUs with the same arrival time for all processes in example 5

Number of Iterations	Number of sets and tasks	Number of completed tasks	Number of deadline miss	time
8500	6/23	4372	0	300s
9300	7/15	6190	0	364s
10000	5/30	6694	0	479s
10000	9/20	4792	0	402s

Comparison Analysis

A comparison analysis between the developed approach and algorithm mentioned earlier in [85] on multiprocessor environments was performed in order to show that the

developed approach outperforms the algorithm in [85] based on number of completed tasks and number of deadline miss occurrences.

In [85], the algorithm uses the following equation to decide which task must be selected first: $R_i = C^r_i / (d_i - t)$ [79]. The higher the rate is the higher the priority is. The previous equation is quite similar to the developed approach except that the task with the smallest rate is selected first to allow CPUs to be busy at all times to avoid any deadline miss. In addition, the slack quantity was chosen instead of the remaining time.

The algorithm in [85] can be used only with all processes having the same arrival times which are assumed to be "0" while the developed algorithm can be used either with the different arrival times or the same arrival times [79,80]. The comparison includes the number of tasks that completed their execution time without any deadline miss and the number of tasks that missed their deadlines. #1 refers to the developed algorithm while #2 represents the algorithm in [85]. The comparison was done under several conditions with the same number of sets, tasks and the same arrive time which was r = 0. Table 66 illustrates results of the comparison analysis.

Table 66: Results of the comparison analysis

Number of Iterations and Processors	Number of sets and tasks	Number of completed tasks		Number of deadline miss	
		#1	#2	#1	#2
3000/4	5/23	780	779	0	0
4000/5	5/15	1281	1281	0	0
5000/6	5/30	1003	998	0	0
7000/7	5/20	2978	2977	0	0
10000/7	5/200	3495	3489	0	0

Table 66 shows clearly that both approaches performed very well in scheduling all tasks without any deadline miss. Nevertheless, the developed algorithm provided more completed tasks. We applied many experiments by increasing number of tasks and sets while maintaining the same number of processors M which was 10, the developed algorithm produced a greater number of tasks which met their deadlines.

A method to schedule periodic tasks to meet their deadlines without allowing any deadline miss to occur was presented. Also the developed algorithm keeps all available CPUs in the system busy at all times to schedule more tasks. It keeps systems stable and provides a good punctual response time as observed in the experiments we performed. Several examples were given to demonstrate how the scheme works. Furthermore, we conducted comparative analysis between the developed algorithm and the algorithm in [85]; our scheme gave the best results in terms of number of completed tasks which met their deadlines under several conditions. Furthermore, both methods yielded no deadline miss in all experiments.

5.5.2 Developed Scheduling Using Dynamic Average Estimation

This section presents an efficient dynamic scheduling algorithm during run-time to schedule periodic tasks in multiprocessor environments and uniprocessor as well using a dynamic average estimation. Dynamic average estimation refers to the changing in different probability distributions when a task is added or removed from them. A value of Worst-Case Execution Time (WCET) is not always available in many real-time applications such as multimedia where data has a great variation. The developed approach

selects which task or a set of tasks must be picked up for execution. A simulation system was developed to show validation of the developed approach.

In multiprocessor environments, when multiple applications run and compete for resources, providing an efficient CPU time for each task is not easy [96,97,98]. They require a priori known of advanced execution time which is impractical in many situations [97,99]. We assume that all processes run-time probability density function (pdf) distributions are well known or can be evaluated [96]. It is required to schedule different tasks on different processors which is influenced by the remaining execution time [96]. Using remaining execution time to develop a method to schedule periodic tasks on multiprocessor environments based on different probability distributions (pdf) is the motivation in this research.

In real-time systems, many tasks can be considered as stochastic ones, which are defined as collection of random variables representing the evaluation of a system of random values over time with large variability [96,99,100]; it is impractical to use WCET in scheduling periodic tasks with the high variation in the coefficient of variance C^2; which can be seen as the relative standard deviation (RSD) [103].

Our contribution in this field is done by developing a new efficient hybrid scheduling algorithm for periodic tasks that works either on a uniprocessor or multiple processor systems based on different probability distributions. By hybrid we mean it cooperates with the EDF algorithm when needed [80]. It works during run-time to decide which task or a set of tasks should be selected first from the ready queue and then it gains system resources such as the CPU. The main objectives of this algorithm are to ensure that

all processes meet their deadlines, keep a system stable, eliminate the idle state of all existing CPUs and provide a good punctual response time if needed [80].

In the developed algorithm, the motivations for it can be summarized as follows: 1. An efficient method on multiple processors and uniprocessor as well; working correctly on multiprocessor environments implies that it also works on uniprocessor without any issue, 2. Gives maximum CPUs utilization since it keeps all of them occupied which delivers all tasks and no deadline miss occurs, 3. It is a feasible approach which means it does what it is supposed to do by ensuring stability under various circumstances and 4. An ability to develop an on-line dynamic scheduling technique which aims to prevent deadline miss at all times under several conditions or circumstances [80].

The objectives of the developed algorithm are to minimize response times, make sure all processes meet their deadline times and keep all resources utilized [80]. Several assumptions are taken into consideration in order for the method to work in the right way and give the best results; the assumptions are as follows:

A. Preemptive approach which means any task can be blocked by another task with higher priority.

B. Task migration is allowed so the task finishes its execution on any available processor.

C. All tasks in the ready queue are available upon selection and they are independent.

D. Any process is not allowed to appear on multiple processors at the same time.

E. Combines with the EDF algorithm when and if needed.

The following steps describe how the developed scheme works:

1. All tasks in the ready queue are examined by each time unit to decide which one should be selected first to assign to available resources; we have chosen the time unit to be 1 ms which is the conventional time unit in many applications.
2. A rate or ratio R_i, where i denotes the task index, is computed using the equation (48), Slack is computed as follows:

$$slack_i = d_i - t - rt_i \qquad (49)$$

   rt represents the remaining execution time and i represents a process index. The dominator part in equation (48) represents how much time is left until deadline as stated earlier in section 5.5.1.
3. A task with smallest rate gets the highest priority and is assigned first to the system resources; if more than one task have the same rate; then the task or set of tasks with shortest deadline is selected first.
4. All previous procedures are repeated until the ready queue becomes empty.

To estimate the remaining time (rt) in discrete distribution since it is most widely used in many applications. let x be a random number representing the execution time of any task; the execution time C and probability P vectors can be represented as follows:

$C = [c_1, c_2, \ldots\ldots, c_n]$ and $P = [p_1, p_2, \ldots\ldots, p_n]$; where $\sum_{1=i}^{n} Pi = 1$;

n denotes number of processes in the ready queue. The execution time vector "C" contains values for all processes available in the ready queue while probability vector includes the associated probability. The discrete distribution has the following characteristics:

A. The expected average execution time $= E[c] = \sum_{i=1}^{n} ci * pi$ (50)

B. $Variance = \sigma^2 = \sum_{i=1}^{n} pi * (ci - E[c])^{2}$ (51)

C. $C^2 = \frac{\sigma 2}{E[c]2}$ (52)

D. At time t = 0, which is considered as the starting time, the remaining time rt is estimated as follows:

$rt0 = E[c] = \sum_{i=1}^{n} ci * pi$ (53)

when any task is executed for time t, where t > 0, then the execution time vector can be written as: $C' = [c_1 - c_t, c_2 - c_t, \ldots\ldots, c_n - c_t]$; where c_t indicates value of execution time at the current time t. any task may have execution time $c_i - c_t \geq 0$, if it is 0, then all entries with that value are removed and the distribution normalized to let the summation of the remaining probability equal 1. So $p_c = \sum_{ci-ct=0} pi$; in other words, p_c contains the probability values of all processes with the execution time value "0". The remaining execution time "rt" is computed as follows:

if current (t) $\leq c_i$, then

$$rt_i = rt_{i-1} - 1 \quad (54)$$

else

$$rt_i = \frac{rt0 - \sum_{j=1}^{i} pj * cj}{1 - pc} - 1 \quad (55)$$

For more information about obtaining equations (53) and (55), the readers are referred to [77]. Both equations represent the estimation of dynamic average value for any task or set of tasks in the ready queue. The complexity of computing both equations is O(1) as shown in [77]. Keep in mind that the probability vector P is normalized each time a process is removed or added to it. The remaining execution time "rt" can take a positive or negative

value according to the changing in a distribution being used whereas the rate "R" can take value ≥ 0. The rate "R" becomes bigger as the process approaches its deadline time. The developed approach can be applied on any real-time system where processing time varies from time to time such as multimedia systems where processing depends on amount of data which has great variations in voice and video [80].

Simulation Experiments

We developed a simulation system to test the developed algorithm under various conditions to prove its validation and show it provides the desired results. More than *5000 tasks* were tested with *an average between 5000 to 7000 times [80]*. The simulation system works for uniprocessor and multiple processors as well. Several processors were used and the maximum number was *10*. The simulation tells how many tasks met their deadlines and how many ones missed their deadlines.

The execution time (C), deadline time (d) and probability vectors were randomly generated by the simulation where d is greater than C and several tasks may have the same deadline times; the same applies on the execution time (C). The maximum deadline time was set to 100 time units. The arrival times (r) were also generated randomly by the simulation under a constraint that r < c and d. Table 60 illustrates the platform being used to perform several experiments for the proposed approach under several circumstances and conditions.

We will test the developed scheme on uniprocessor and multiprocessor environments since they are our concerns. The following tables show the number of how many tasks successfully met their deadlines and how many ones missed using the

developed algorithm. Several number of CPUs “M” were used under multiple circumstances and conditions. In the following tables, number of sets indicates how many categories of tasks are available. It could also represent the number of sources that generate the tasks. In each set, several tasks exist.

Case 1: Uniprocessor with the same arrival time (r = 0)

Table 67: Results of uniprocessor in case 1

Number of Iterations	Number of sets and tasks	Number of completed tasks	Number of deadline miss
1500	3/15	20	0
3000	5/28	70	0
7000	5/35	75	0
10000	5/50	230	0

Table 68 shows that the developed algorithm successfully scheduled several tasks and that no task missed its deadline time. Increasing the number of iteration increases the number of completed tasks as shown in table 68.

Case 2: Uniprocessor with different arrival time

Table 68: Results of uniprocessor in case 2

Number of Iterations	Number of sets and tasks	Number of completed tasks	Number of deadline miss
8500	5/23	89	0
9300	5/25	180	0
10000	5/30	197	0
10000	5/20	158	0

There was no deadline miss as shown in table 3 and the arrival time values (r) influenced number of completed tasks which met their deadlines.

Case 3: Multiple processors with the same arrive time; M = 3.

Table 69: Results of 3 processors with the same arrive time in case 3

Number of Iterations	Number of sets and tasks	Number of completed tasks	Number of deadline miss
5000	5/17	215	0
3000	5/30	147	0
7000	5/44	376	0
10000	5/60	456	0

Table 69 shows the results of using 3 processors under the same conditions we used for uniprocessor in case 1; no deadline miss occurred and number of completed tasks was doubled several times. The elapsed time reduced significantly which improved the response times.

Case 4: different arrive time with M = 5 processors.

Table 70: Results of 5 processors in case 4

Number of Iterations	Number of sets and tasks	Number of completed tasks	Number of deadline miss
5000	5/18	348	0
3000	5/33	190	0
7000	5/40	430	0
10000	5/70	651	0

In the following case, the number of sets varied in each run and M was 7.

Case 5: different arrive time.

Table 71: Results of 7 processors with different arrival times in case 5

Number of Iterations	Number of tasks	Number of completed tasks	Number of deadline miss
8500	40	547	0
9300	15	722	0
10000	55	602	0
10000	14	864	0

The developed algorithm was completely able to execute tasks as much as possible without any deadline miss. Furthermore, the elapsed time reduced significantly at an acceptable rate. The developed algorithm produced more overhead as observed in the experiments. The overhead comes from estimating the remaining execution time “rt” when the used probability distribution changes as time moves on.

The developed approach method to schedule periodic tasks in real-time systems to meet their deadlines without allowing any deadline miss to occur using dynamic average estimation was presented in this section. Only the discrete distribution is presented within this thesis. However, it can be applied on any probability distribution being used. Also the developed algorithm keeps all available CPUs in the system busy at all times to schedule more tasks. Furthermore, it keeps systems stable and provides a good punctual response time as observed in the experiments we performed.

CHAPTER 6

Evaluation of the Designing Framework

6.1 Introduction

In this chapter, we will apply the developed framework on two embedded system applications. Both applications are real-time applications. The applications are pollution detection and fire detection systems. The developed framework is used to estimate the average response time in both applications; the power consumption estimation is left as future work. Both systems use big data in their algorithms.

For pollution detection systems, we will apply the designing framework on the algorithm developed in [38]. The algorithm was developed in the Computer Science and Engineering (CSE) department at the University of Connecticut in 2014. It identifies harmful chemical materials and provides more than 90% accuracy as stated in [38].

For fire detection systems, the developed framework is used to estimate the average response time based on an algorithm mentioned in [104]. It uses images to detect whether there is a fire or not. The algorithm uses image processing and machine learning techniques to disclose fires.

6.2 Pollution Detection Systems

The algorithm in [38] was developed by P. Periaswamy to classify 5 harmful chemical materials in real-time mode. The data for it was obtained from Owlstone Inc. [38]. The input data for that algorithm as stated in [38] has the following characteristics:

I. 5 analytes were included.

II. Each analyte contains various levels of “SCL”, it ranges from 2 SCL to 20 SCL where SCL refers to *Scaled Concentration Level*.

III. Each type of dataset includes blank sets and all of them are equivalent.

P. Periaswamy in [38] takes 0 SCL at the start and the end to check that the datasets are not contaminated before moving on with the classification procedures.

The datasets to classify them provided by the Owlstone Inc. can be seen as big data since around 1000 sets were included for each analyte with positive and negative modes for each type. They are about 80% of the total datasets available at the provider.

The algorithm to detect harmful chemical materials as stated in [38] for both modes is as follows:

1. For each chemical (across concentration) and blank, consolidate the peak location values and the DF “Deflecting Voltage” across all the test runs
2. For each chemical and blank do the following:
    i) For each DF, group the peak location values. The number of groups is determined based on the maximum number of peaks that occurs for a particular voltage (across all runs /concentration)
    ii) For each group, find all the points (peak locations) that are x sigma away from the center. (x is set to 3 when the algorithm is started. If all the chemicals are not differentiated, the value is varied to 2.5, 2, 1.5 and 1 until all the chemicals are classified)
    iii) If any group has less than 80% of points ignore the group.

iv) For each group, find the minimum and maximum range of the peak location values

The output of this step is the voltage, min and max range of each group for each chemical and blank.

3. For each DF voltage, spread all the groups (which we have as range) of each chemical and blank obtained above and find all those groups which do not overlap with each other.
4. For each such non-overlapping group find the gap between them. Consider only those groups which are 0.1V (Compensation Voltage CV) away from each other
5. *Finding the voltage which classifies the maximum number of the chemicals:*

The result from step 4 is examined to find a DF voltage which has the maximum number of chemicals whose ranges do not overlap

i) Sort the result by the maximum number of chemicals the DF voltage classifies.

ii) Record the DF voltage, chemical that is classified, corresponding ranges of the chemicals and the mode (positive / negative) in which these are found.

Until all the chemicals are classified (the sigma values is varied between 3 to 1 in the decrement of 0.5 until all the chemicals are classified). Interested readers are referred to [38] for more information about the developed algorithm and its results and accuracy.

The expected average response time from the developed framework is computed using the following equation:

$$C = ((1 + e_{11})*(C_{check} + C_{initial})) + ((1 + e_{11} + e_{12})*C_{test}) + (e_8 * (C_{wait} + C_{test})) + ((e_{11} + 1 + e_{12}) * C_{processing} + C_{test})) + (e_{11} * C_{failed}) \quad (56)$$

e_8: number of times a branch in the waiting state is not taken.

e_{12}: number of time a branch in the waiting state is taken.

e_{11}: number of times tasks failed and go back to the initial state.

We assume no operation takes place in the failed state, only the process of forwarding tasks to the initial state happens there and takes no time so $C_{failed} = 0$.

The performance equation tells what happens in each state as follows:

$C_{initial}$: clustering the peak values with their associated DF using K-means algorithm. Ignore any cluster with hit rate < 80%.

C_{check}: Finding groups with no intersections to classify and detect. For non-overlapping groups, consider only those groups with gap about 0.1 (CV).

$C_{processing}$: Finding maximum and minimum number of range in each group. Then, find the DF which classifies the maximum number of chemicals. In addition, it sends a notification alert such as alarm sound or blinking light.

C_{test}: it can be seen as if statement to check whether the deadline can be met or not and also if the P.U. is free or not. This value is considered to be too small and can be neglected.

Actual average response time: 57.746 s

<u>Using the developed framework</u>

Matlab 2015 platform on Windows 7 “64 bit” was used to perform the experiments. The algorithm was tested around 5000 times and the average value was recorded each time.

It took about 15 hrs to complete since the datasets were very big, each type of chemical material has around 1000 sets.

Performance parameters:

Bandwidth = BUS Speed * Bus Width (Number of bits) = 64 (bits) * 1969.2MB/s = 126.1 MB/s = 126100000 B/s

Message size = 2000 B.

Arrival rate = λ = 5000 "as input data"

Estimated average response time:

$C_{initial}$ = 29.821 s

C_{check} = 7.949 s

$C_{processing}$ = 23.845 s

C_{test} = 0.465 s

By substituting into eq. (56) the average estimated response time = 63.011 s

The following table illustrates the average actual and estimated response time during design level without any minimization approaches.

Table 72: Results of the average response time in the design level without any minimization

Average Response Time in s	
Actual	Estimated
57.746	63.011

Average Response Time Using Minimization Approaches

For minimization approaches, parallelization and optimizations methods are used during the design level in order to minimize the average response time for actual and estimated values while the scheduling algorithm methods are used during the run-time mode to ensure that all tasks meet their deadlines and to minimize the response time as well if possible. In each state, several CPUs are used. Determining the number of parallel branches is performed based on the algorithm developed in [60]. Due to limited number of CPUs, memory capacity and size, only 3 CPUs were used within this research.

1. **Minimization during design level**

Using 2 CPUs: in each state in the eq. (56), 2 CPUs were used to speed up the response time. The results indicate that the average speed up was nearly 16% for the actual value and about 9.30% for the estimated one as shown in table 73.

Improved average actual response time: 48.392 s.

Improved estimated average response time:

C_{fork} = 0.364 s

C_{Joint} = 0. 275 s

The performance eq. (56) becomes as follows:

(0.275) + (0.364) + Max{27.933, 28.647} + (0.275) + (0.364) + Max{5.549, 7.823} + (0.275) + (0.364) + Max{17.556, 18.769} = 57.157 s.

Table 73: Results of the average response time for the design level after minimization using 2 CPUs

Average Response Time in s		Speed up	
Actual	Estimated	Actual	Estimated
48.392	57.157	16.20%	9.30%

Using 3 CPUs: in each state in the eq. (56), 3 CPUs were used to speed up the response time. The results indicate that the average speed up was nearly 13.52% for the actual value and about 12.74% for the estimated one as shown in table 74.

Table 74: Results of the average response time for the design level after minimization using 3 CPUs

Average Response Time in s		Speed up	
Actual	Estimated	Actual	Estimated
41.849	49.878	13.52%	12.74%

Table 74 shows that the minimization reduction after increasing the number of CPUs raised by nearly 13% while it was about 9% for using only 2 CPUs. Table 75 illustrates the total speed up for both average values (actual and estimated) after using 3 CPUs.

Table 75: Total average minimization "speed up"

Speed up	
Actual	Estimated
29.72%	22.04%

2. Minimization during Run-Time level

The developed Scheduling policy algorithm for aperiodic tasks is used to get further minimization for response time during run-time stage. The following table illustrates the results of the AWT and the ATT for several parameters: the number of tasks (n) with the number of iterations, between static R.R. ***(S.R.R.)*** and the developed dynamic R.R. (***P.D.R.R.***) in [78]. For the static R.R. version, the WCET was assumed to be 60

Table 76: Comparison analysis between the S.R.R. and the developed dynamic R.R. algorithms

Number of tasks (n)	Number of Iterations	AWT		ATT		Response time		Speed up	
		S.R.R.	P.D.R.R.	S.R.R.	P.D.R.R.	S.R.R.	P.D.R.R.	S.R.R.	P.D.R.R.
6	100	60	105.63	210	174.21	60	36.179	0%	27.46%
20	100	570	463.79	630	512.43	60	40.35	0%	32.75%
30	1000	870	742.79	930	782.11	60	46.213s	0%	22.97%
50	1000	1470	1134.78	1530	1216.69	60	42.592	0%	29.01%
100	1000	1940	1789.65	2232	1983.92	60	45.936	0%	23.44%

The average minimization from table 76 is nearly 27.13%. The overall average minimization from design and run-time levels is nearly about 31.89%.

6.3 Fire Detection Systems

In this section, we will estimate the average response time of fire detection systems using the algorithm developed by H. Tian et al in [104]. It depends on an image separation technique to detect fire regions in video files. The algorithm works using machine learning and image processing techniques to differentiate between fire regions and non-fire regions based on pixels processing and analyzing. B. Redakbar and D. Wilson in [105] implemented two methods mentioned in [104] on datasets of images from Tahoe Lake and Southern California regions. A matting technique was used in [105] to distinguish salient regions in large images with small pixel regions corresponding to smoke from other regions in the large images. This method allows us to filter unimportant regions when passing through a classifier [105]. The matting technique developed in [105] depends on either a *local smoothness or principle component analysis "PCA"* to detect fire regions and alert a facility's occupants and send a notification signal to a central station "fire station".

The matting method focuses on splitting any image into regions and trying to predict whether or not a region contains smoke. The designing framework shown in fig. 1 divides the algorithm mentioned in [104] and implemented in [105] into 3 components. Each component is mapped with a state of the HGFSM as shown in table 77. More information can be found in [104,105] for the developed fire detection approach. The experiments were run about 1000 times using Matlab 2015.

Table 77: Mapping fire detection algorithm components with the developed HGFSM

Fire Detection Algorithm Components	HGFSM States
Background modeling and image separation	Initial
Feature extraction	Checking
Classification	Processing and Waiting

Using Local Smoothness Approach

Actual average response time = 217 s

Arrival rate = $\lambda = 2$ images.

The following tables illustrate the results for average actual and estimated response time during design and run-time stages using the developed framework shown in fig. 1. Eq. (56) will be used to determine the average estimated response time.

Estimated average response time:

$C_{initial} = 143.78$ s

$C_{check} = 26.02$ s

$C_{processing} = 69.837$ s

$C_{test} = 0.465$ s

$C_{failed} = 0$ s

e_8: number of times a branch in the waiting state is not taken, = 0

e_{12}: number of time a branch in the waiting state is taken, = 0

e_{11}: number of times tasks failed and go back to the initial state, = 0

C_{wait} = 0 s (assuming identifying fire incident is critical and no wait time is essential).

By substituting into eq. (56) we get that:

$C = ((1 + e_{11})*(C_{check} + C_{initial})) + ((1 + e_{11} + e_{12})*C_{test}) + (e_8 * (C_{wait} + C_{test})) + ((e_{11} + 1 + e_{12}) * C_{processing} + C_{test})) + (e_{11} * C_{failed}) = (1 *(26.02 + 143.78)) + (1* 0.465) + (1 * (69.837 + 0.465))$ = 240.567 s.

Table 78: Results of the fire detection algorithm in the design level without using any minimization method in the Local smoothness approach

Average Response Time in s	
Actual	Estimated
217	240.567

Average Response Time Using Minimization Approaches

For minimization approaches, parallelization and optimizations methods are used during the design level in order to minimize the average response time for actual and estimated values while the scheduling algorithm methods are used during the run-time mode. The same procedures applied on pollution detection systems will be applied here too. Determining number of parallel branches is performed based on the algorithm developed in [60]. Due to memory capacity and size, only 3 CPUs were used within this research.

1. **Minimization during design level Using 2 CPUs:**

In each state in the eq. (56), 2 CPUs were used to speed up the response time. The results indicate that the average speed up was nearly 8.382% for the actual value and about 9% for the estimated one as shown in table 79.

Improved average actual response time: 198.81 s.

Improved estimated average response time:

C_{fork} = 0.364 s

C_{Joint} = 0. 275 s

The performance eq. (56) becomes as follows:

(0.275) + (0.364) + Max{139.948, 125.315} + (0.275) + (0.364) + Max{16.06, 10.75} + (0.275) + (0.364) + Max{60.721, 52.918} = 218.646 s.

Table 79: Results of the average response time for the design level in the fire detection algorithm after minimization using 2 CPUs in the Local smoothness approach

Average Response Time in s		Speed up	
Actual	Estimated	Actual	Estimated
198.81	218.646	8.382%	9.11%

Using 3 CPUs: in each state in the eq. (56), 3 CPUs were used to speed up the response time. The results indicate that the average speed up was nearly 17.46% for the actual value and about 13.92% for the estimated one as shown in table 80.

Table 80: Results of the average response time for the design level in the fire detection algorithm after minimization using 3 CPUs in the Local smoothness approach

Average Response Time in s		Speed up	
Actual	Estimated	Actual	Estimated
187.396	203.894	5.38%	6.77%

Table 80 shows that the minimization reduction after increasing the number of CPUs raised by nearly 13% while it was about 8% for using only 2 CPUs. Table 81 illustrates the total speed up for both average values (actual and estimated) after using 3 CPUs.

Table 81: Total average minimization "speed up" in the fire detection algorithm in the Local smoothness approach

Speed up	
Actual	Estimated
13.52%	15.88%

2. Minimization during Run-Time level

The developed Scheduling policy algorithm for aperiodic tasks is used to get further minimization for the response time during the run-time stage. Table 82 illustrates the results of the AWT and the ATT for several parameters, the number of tasks (n) with the number of iterations, between static R.R. ***(S.R.R.)*** and the developed dynamic R.R. (***P.D.R.R.***) in [78]. For the static R.R. version, the WCET was assumed to be 220 s. One

run takes about an average of 16 hours to complete for 1000 iterations since the algorithm's parameters are optimized by looping over 200 times to ensure that each pixel is tested and its value is the correct one. Those parameters can be loop only once for a faster response time. However, this minimization affects the final results negatively "poorly" since some pixels are not covered well as observed in the conducted experiments. We decided to leave the loop iteration to be 200 as implemented and used parallelization method within it to speed it up. For that purpose, the ***parfor*** command inside Matlab is used which maintains the quality of the algorithm and gives further reduction in the response time. Only 10 tasks (jobs) and 20 tasks are shown in table 82.

Table 82: Results of the fire detection algorithm after using minimization methods in the Local smoothness approach

Number of tasks (n)	Number of Iterations	AWT		ATT		Response time		Speed up	
		S.R.R.	P.D.R.R.	S.R.R.	P.D.R.R.	S.R.R.	P.D.R.R	S.R.R.	P.D.R.R.
10	1000	220	193.57	284.29	237.98	220	190.14	0%	13.57%
20	1000	371.4	288.36	593.06	442.79	256.83	202.41	0%	21.18%

The average minimization from table 82 is nearly 17.375%. The overall average minimization from the design and the run-time levels is nearly about 20.14% after applying several experiments. The maximum number of tasks were 35 and the simulation system suffered from memory heap which affected the resultant performance.

Using Principle Component Analysis Approach

Actual average response time = 20.64 s

Arrival rate = $\lambda = 2$ images.

Estimated average response time:

$C_{initial}$ = 9.68 s

C_{check} = 5.07 s

$C_{processing}$ = *8.47* s

C_{test} = 0.465 s

C_{failed} = 0 s

e_8: number of times a branch in the waiting state is not taken, = 0

e_{12}: number of time a branch in the waiting state is taken, = 0

e_{11}: number of times tasks failed and go back to the initial state, = 0

C_{wait} = 0 s (assuming identifying a fire incident is critical and no wait time is essential).

By substituting into eq. (56) we get that:

$C = ((1 + e_{11})*(C_{check} + C_{initial})) + ((1 + e_{11} + e_{12})*C_{test}) + (e_8 * (C_{wait} + C_{test})) + ((e_{11} + 1 + e_{12}) * C_{processing} + C_{test})) + (e_{11} * C_{failed}) = (1 *(5.07 + 9.68)) + (1* 0.465) + (1 * (8.47 + 0.465)) = 24.15$ s.

Table 83: Results of the fire detection algorithm in the design level without using any minimization method in the PCA approach

Average Response Time in s	
Actual	Estimated
20.64	24.15

Average Response Time Using Minimization Approaches

For minimization approaches, parallelization and optimization methods are used during the design level in order to minimize the average response time for actual and estimated values while the scheduling algorithm methods are used during the run-time mode. The same procedures applied on pollution detection systems will be applied here too. Determining the number of parallel branches is performed based on the algorithm developed in [60]. Due to memory capacity and size, only 3 CPUs were used within this research.

1. **Minimization during design level Using 2 CPUs:**

In each state in the eq. (56), 2 CPUs were used to speed up the response time. The results indicate that the average speed up was nearly 25.43% for the actual value and about 11.06% for the estimated one as shown in table 84.

Improved average actual response time: 15.39 s.

Improved estimated average response time:

C_{fork} = 0.364 s

C_{Joint} = 0. 275 s

The performance eq. (56) becomes as follows:

(0.275) + (0.364) + Max{7.41, 8.96} + (0.275) + (0.364) + Max{3.32, 3.06} + (0.275) + (0.364) + Max{6.51, 7.28} = 21.477 s.

Table 84: Results of the average response time for design level in the fire detection algorithm after minimization using 2 CPUs in the PCA approach

Average Response Time in s		Speed up	
Actual	Estimated	Actual	Estimated
15.39	21.477	25.43%	11.06%

Using 3 CPUs: in each state in the eq. (56), 3 CPUs were used to speed up the response time. The results indicate that the average speed up was nearly 17.46% for the actual value and about 13.92% for the estimated one as shown in table 85.

Table 85: Results of the average response time for design level in the fire detection algorithm after minimization using 3 CPUs in the PCA approach

Average Response Time in s		Speed up	
Actual	Estimated	Actual	Estimated
12.41	16.88	19.34%	21.40%

Table 85 shows that the minimization reduction after increasing the number of CPUs was raised by nearly 19% when 3 CPUs were used. Table 86 illustrates the total speed up for both average values (actual and estimated) after using 3 CPUs.

Table 86: Total average minimization "speed up" in the fire detection algorithm in PCA approach

Speed up	
Actual	Estimated
39.87%	30.10%

2. Minimization during Run-Time level

The developed Scheduling policy algorithm for aperiodic tasks is used to get further minimization for the response time during the run-time stage. Table 87 illustrates the results of the AWT and the ATT for several parameters, the number of tasks (n) with the number of iterations, between the static R.R. ***(S.R.R.)*** and the developed dynamic R.R. (***P.D.R.R.***) in [78]. For the static R.R. version, the WCET was assumed to be 21 s. One run takes about an average of 63 minutes to complete for 1000 iterations since the algorithm's parameters are optimized by looping over 200 times to ensure that each pixel is tested and its value is the correct one. Those parameters can be loop only once for faster response time. However, this minimization affects the final results negatively "poorly" since some pixels are not covered well as observed in the conducted experiments. We decided to leave the loop iteration to be 200 as implemented and used in the parallelization method to speed it up. For that purpose, the ***parfor*** command inside Matlab is used which maintains the quality of the algorithm and gives further reduction in the response time.

Table 87: Results of the fire detection algorithm after using minimization methods in the PCA approach

Number of tasks (n)	Number of Iterations	AWT		ATT		Response time		Speed up	
		S.R.R.	P.D.R.R.	S.R.R.	P.D.R.R.	S.R.R.	P.D.R.R.	S.R.R.	P.D.R.R.
10	1000	21	15.89	28.29	22.03	21	13.49	0%	35.76%
20	1000	31.38	24.32	45.16	34.55	21	15.02	0%	28.47%
45	1000	64.89	53.44	80.74	67.87	21	14.37	0%	31.57%
60	1000	80.12	69.07	99.92	82.14	21	14.75	0%	29.76%
100	1000	134.6	113.96	169.1	139.56	21	12.31	0%	41.38%

The average minimization from table 87 is nearly 33.388%. The overall average minimization from the design and the run-time levels is nearly 32.86% after applying several experiments. The maximum number of tasks were 100 and the PCA approach for the fire detection systems yields better results and acts faster than the Local smoothness technique.

CHAPTER 7

Conclusion and Future Work

7.1 Conclusion

In this work, we developed ***multidimensional framework to develop high performance embedded systems***. By being multidimensional, the designing framework is capable of estimating different performance metrics such as response time, power consumption, reliability, availability and security. However, in this research, only response time and power consumption are considered in this work. The primary objectives of the developed framework are to:

1. Have the ability to estimate the average response time or power consumption as a desired performance metric.
2. Detect or spot bottleneck(s) in a system under investigation or consideration. In addition, the ability to estimate an enhanced performance metric, either response time or power consumption.

The developed multidimensional framework is composed of two stages "levels" which are I. Design level and II. Run-time level and it has three components as shown in fig. 1.

In the design level, the developed framework constructs the performance equations which are considered to be the objective functions. The objective functions are used to derived the estimate either response time or power consumption. Furthermore, the objective functions can tell a designer or designers which part or component of the tested system has bottleneck(s). After finding the bottleneck(s), the developed framework is used

again to derive equations to estimate the average enhanced or minimized performance metric using available resources. Only response time is considered within this work.

In the run-time level, the scheduling policy technique is used to minimize response time if possible and to ensure that all processes “tasks” complete their execution cycles “times” before or at their deadlines. Then, the developed framework was applied on two real-time applications which are fire detection and pollution detection systems to estimate their average response time and compare it with the actual average value. Moreover, we emphasized the tradeoff or the consequences between the reduced response time and unreduced one in terms of code size and power consumption. The code size increased about 65% to over 120% when using GPUs to minimize the response time as shown in our work in [46]. Also the power consumption raised up by less than 20% due to the fact that only 28% of the GPU is being used. Utilizing more GPU processing capability will increase the power consumption since the GPU and CPUs work simultaneously to minimize the performance metric.

Our results show that we have a higher performance gain, which refers to the speed up, when the reduction approach is used. In Android platform, the performance gain was nearly 64% while it was between 30% and 45% in fire detection and pollution detection systems respectively. Finally, we assess the validity of the developed framework using two real case studies.

7.2 Future Work

In the future, we plan to do the following:

A. Develop a simulation system to support the developed framework in order to estimate the average power consumption in many real-time applications since they suffer from a limited number of simulations which are capable of determining the average power consumption.

B. The scheduling algorithm for periodic tasks using the dynamic average estimation suffers from high overhead when we compare it with a single value such as the WCET. So we will consider an approach to minimize that overhead.

C. Use resource allocation methods to get further minimization for response time and power consumption.

D. Investigate the performance metrics estimation using the scheduling policy technique in physical cyber systems which are composed of two or more embedded systems.

REFERENCES

[1] D. Smarkusky, R. Ammar, I. Antonios and H. Sholl, "Hierarchical Performance Modeling for Distributed System Architectures," Computer and Communications, 2000. Proceedings. ISCC 2000. 5th IEEE Symposium, pp. 659–664, July 2000.

[2] C. P. Rosiene and R. Ammar, "A Data Modeling Framework for The Performance Analysis of Sequential and Parallel Software," Proceedings of the ACM Conference on Computer Science, pp. 310-317, 1993.

[3] R. Ammar, "Software Performance Analysis," lecture notes, University of Connecticut, 1991.

[4] T. Bruni, "Heterogeneous Use of Models of Computation," Seminar System/Architecture and Design Methods of Embedded Systems, Summer Semester 2006.

[5] S. Singh, P. Mor and G. Singh, "Application of Embedded Systems in Modern Society," VSRD-IJEECE, Vol. 2. NO. 6, pp. 373-384, 2012.

[6] J. Happe, "Predicting Mean Service Execution Times of Software Components Based on Markov Models," Software Engineering group, University of Oldenburg, Germany.

[7] B. Lee and E. A. Lee, "Interaction of Finite State Machines and Concurrency Models," Proceeding of Thirty Second Annual Asilomar Conference on Signals, Systems and Computers, Pacific Grove, California, November 1998.

[8] A. Stan, N. Botezatu, L. Panduru and R. G. Lupu, "A Finite State Machine Model Used in Embedded Systems Software Development,", pp. 51-63, 2009.

[9] A. Nandi, "System-Level Power/Performance Analysis for Embedded Systems Design," Master Thesis, Carnegie Mellon, May 2002.

[10] B. Lee and E. A. Lee, "Hierarchical Concurrent Finite State Machines in Ptolemy," Proceeding of International Conference on Application of Concurrency to System Design, pp. 30-40, Fukushima, Japan, March, 1998.

[11] G. K. Reddy, S. Baragada, D. S. Kumar and B. P. Rani, "Software Performance Evaluation of a Polar Satellite Antenna Control Embedded System," International Journal of Application or Innovation in Engineering and Management (IJAIEM), Vol. 2. NO. 1, pp. 166-173, January, 2013.

[12] C. U. Smith, Performance Engineering of Software System. Addison-Wesley, 1990.

[13] C. U. Smith and L. G. Williams, "Software Performance Engineering: A Case Study Including Performance Comparison with Design Alternatives", IEEE Transactions on Software Engineering, Vol. 19, No. 7, pp. 720-741, July 1993.

[14] C. U. Smith and L. G. Williams, "Performance Evaluation of a Distributed Software Architecture", Proceedings of Computer Management Group, Anaheim, December 1998.

[15] R. Ammar and T. L. Booth, “Software Optimization Using User Models,” IEEE transactions on systems, MAN, and cybernetics, vol. 18. NO. 4, pp. 552-560, July/August 1988.
[16] J. Viskari, R. Jokinen and K. Hakkarainen, “A Generic FSM Interpreter for Embedded Systems,” proceedings of IEEE EURWRTS, pp. 284-289, 1996.
[17] L. Carmichael, A. Warner, FNAL and Batavia, “A Generic Finite State Machine Framework for the ACNET Control System,” proceedings of ICALEPCS, pp. 28-30, Kobe, Japan, 2009.
[18] B. R. Haverkort, “challenges for modeling and analysis in embedded systems and systems-of-systems design,” 1st Workshop in Advances Systems of Systems (AiSoS 2013), EPTCS 133, pp. 40–46, 2013.
[19] R. Hedge, G. Mishra, and K. S. Gurumurthy, “Software and Hardware Design Challenges in Automotive Embedded Systems,” International Journal of VLSI design and Communication Systems (VLSICS), vol. 2, No. 3, pp. 165-174, September 2011.
[20] T. A. Henzinger and J. Sifakis, “The Embedded Systems Design Challenge,” Proceedings of the 14th International Conference on Formal Methods (FM), pp. 1–15, 2006.
[21] R. Zurawski, “Embedded Systems in Industrial Applications: Trends and Challenges", IEEE Industrial Electronic Society (SIES), 2007.
[22] C. Almeida and J. Rufino, “Interconnected Embedded Systems: Challenges and Main Problems to Solve”, International Workshop on Factory Communication Systems (WFCS 2006), pp. 87-90, 2006.
[23] R. B. Nimmatoori, B. A. Vinay and C. Srilatha, “A Methodology for Estimation of Performance in Real Time Distributed Embedded Systems", Journal of Theoretical and Applied Information Technology, vol. 15, No. 2, pp. 144-150, May 2010.
[24] L. Yuan, G. Qu, T. Villa, and A. S. Vincentelli, “An FSM Re-Engineering Approach to Sequintial Circuit Synthesis by State Splitting,” IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 27, no. 6, pp. 1159–1164, June 2008.
[25] N. P. Dash, R. Dasgupta, J. Chepada and A. Halder, “Event Driven Programming for Embedded Systems - A Finite State Machine Based Approach,” the 6th International Conference on Systems "ICONS 2011", pp. 19–23, 2011.
[26] M. Choy and M. N. Laik, “A Markov Chain Approach to Determine the Optimal Performance Period and Bad Definition for Credit ScoreCard,” The International Journal of Social Science and Management, vol. 1, No. 6, pp. 227–234, October 2011.
[27] W. Lu, “Average System Evaluation Using Markov Chain", summer semester, Stuttgart University, 2006.
[28] S. L. Tsao and S. Y. Lee, "Performance Evaluation of Inter-Processor Communication for an Embedded Heterogeneous Multi-Core Processor", Journal of Information Science and Engineering, Vol. 28, pp. 537-554, 2012.

[29] L. Chen, S. Wei and G. Yu, “Performance Evaluation of Embedded System Based on Behavior Expressions,” 2nd International Conference on Mechanical and Electronics Engineering (ICMEE), Vol. 1, pp. 253-256, 2010.
[30] Y. Y. Cho, J. B. Moon and Y. C. Kim, “A System for Performance Evaluation of Embedded Software,” International Journal of Computer, Information, Systems and Control Engineering, Vol. 1, No. 1, pp. 153-156, 2007.
[31] A. D. Pimentel, “The Artemis Workbench for System-Level Performance Evaluation of Embedded Systems,” International Journal of Embedded Systems, Vol. 3, No. 3, pp. 181-196, 2008.
[32] G. Madi, N. Dutt and S. Abdelwahed, “Performance Estimation of Distributed Real-Time Embedded Systems by Discrete Event Simulations,” Proceedings of the 7th ACM & IEEE international conference on Embedded software " (EMSOFT), pp. 183-192, New York, NY, USA, 2007.
[33] A. Abdel-raouf, T. A. Fergany, R. A. Ammar and H. Sholl, “Performance-Based Modeling for Distributed Object-Oriented Software,” Proceedings of the 3rd IEEE International Conference of Signal Processing and Information Technology (ISSPIT), pp. 769-773, 2003.
[34] Z. Wang and A. Stavrou, “Google Android Platform: Introduction to The Android API, HAL and SDK,” lecture notes, George Mason University.
[35] V. Matos, “Android Multi-Threading,” Notes on Android, Chapter 13, Cleveland State University.
[36] X. Ma, “Android OS,” lecture notes, CSE 120, Fall 2010.
[37] S. Brahler, “Analysis of The Android Architecture,” master thesis, Karlsruher Institute for Technology, October 2010.
[38] P. Periaswamy, “A Novel Clustering Technique to Identify Chemicals," Master thesis, University of Connecticut, 2014.
[39] L. Qiang, “Estimation of Fire Detection Time,” The 5th Conference on Performance-based Fire and Fire Protection Engineering, pp. 233-241, 2011.
[40] M. James, “Flame and Smoke Estimation for Fire Detection in Videos Based on Optical Flow and Neural Networks,” International Journal of Research in Engineering and Technology "IJRET", vol. 3, Issue 8, pp. 324–328, August 2014.
[41] C. L. Mealy, A. Wolfe and D. T. Gottuk, “Smoke Alarm Response: Estimation Guidelines and Tenability Issues-Part 2,” 14th International Conference on Automatic Fire Detection, pp. 1–16, 2009.
[42] Y. Jiang and P. M. Munday, "Smoke Detection Performance and FDS Modeling for Full-Scale Fire Tests," International Conference on Fire Safety Engineering "FSE", pp. 1-9, 2009.
[43] B. J. Meacham, "Concepts of A Performance-Based Building Regulatory System for the United States," Proceedings of the 5th International Symposium on Fire Safety Science, pp. 701-712, March 1997.
[44] J. A. Geiman and D. T. Gottuk, "Alarm Threshold for Smoke Detector Modeling," Proceedings of the 7th International Symposium of Fire Safety Science, pp. 197-208, 2003.

[45] A. Alsheikhy, S. Han and R. Ammar, "Hierarchical Performance Modeling of Embedded Systems“, Computers and Communication (ISCC), 2015 20th IEEE Symposium on Computers and Communications, pp. 936-942, July 2015.
[46] A. Alsheikhy, S. Han and R. Ammar, "Delay and Power Consumption Estimation in Embedded Systems Using Hierarchical Performance Modeling”, 15th IEEE International Symposium on Signal Processing and Information Technology (ISSPIT), pp. 34-39, December 2015, Abu Dhabi, UAE.
[47] Y. H. Jung and L. P. Carloni, “Host-GPU Multiplexing for Efficient Simulation of Multiple Embedded GPUs on Virtual Platforms”, The Proceedings of the Design Automation Conference (DAC), 2015.
[48] S. Nomura, T. Mitsuishi, J. Suzuki, Y. Hayashi, M. Kan and H. Amano, “Performance Analysis of The Multi-GPU System with ExpEther”, ACM SIGARCH Computer Architecture News- Heart 14, Vol. 42, No. 4, pp. 9-14, New York, NY, 2014.
[49] I. Grasso, P. Radojkovic, N. Rajovic, I. Gelado, and A. Ramirez, “Energy Efficient HPC on Embedded SoCs: Optimization Techniques for Mali GPU”, IEEE 28th International Parallel and Distributed Processing Symposium, 2014.
[50] A. Glenis, and S. Petridis, “Performance and Energy Characterization of High-Performance Low-Cost Cornerness Detection on GPUs and Multicores”, IEEE 5th International Conference on Information, Intelligence, Systems and Applications, pp. 181-186, July, 2014.
[51] M. Huang, and C. Lai, “Accelerating Applications Using GPUs on Embedded Systems and Mobile Devices”, IEEE International Conference on High Performance Computing and Communication (HPCC) and IEEE International Conference on Embedded and Ubiquitous Computing, pp. 1031-1038, November, 2013.
[52] S. Kim, H. E. Kim, H. Kim, and J. A. Lee, “Computing Energy-Efficiency in The Mobile GPU”, IEEE International Conference on System on Chip (Soc) Design (ISOCC), pp. 219-221, Busan, South Korea, November, 2013.
[53] L. Polok, and P. Smrz, “Fast Linear Algebra on GPU”, IEEE 14th International Conference on High Performance Computing and Communications (HPCC), pp. 439-444, Liverpool, UK, June, 2012.
[54] G. Calandrini, A. Gardel, P. Revenga, and J. L. Lazaro, “CGPU Acceleration on Embedded Devices, A Power Consumption Approach”, IEEE 14th International Conference on High Performance Computing and Communications (HPCC), pp. 1806-1812, Liverpool, June, 2012.
[55] L. Diaz, E. Gonzalez, E. Villar, and P. Sanchez, “VIPPE, Parallel Simulation and Performance Analysis of Multi-Core Embedded Systems on Multi-Core Platforms”, IEEE International Conference on Design of Circuits and Integrated Circuits (DCIC), pp. 1-7, Madrid, Spain, November, 2014.
[56] A. Kejariwal, A. V. Veidenbaum, M. Girkar, and X. Tian, “Challenges in Exploitation of Loop Parallelism in Embedded Applications”, IEEE 4th International Conference on Hardware/Software Codesign and System Synthesis (CODES +ISSS), pp. 173-180, Seoul, South Korea, October, 2006.

[57] H. Blumi, J. V. Livonius, L. Rotenberg, T. G. Noll, H. Bothe, and J. Brakensiek, "Performance and Power Analysis of Parallelized Implementations on an MPCore Multiprocessor Platform", IEEE International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (IC-SAMOS), pp. 74-81, July, 2007.
[58] N. Baek, and H. Lee, "OpenGL SC Implementation Over an OpenGL ES 1.1 Graphics Board", IEEE International Conference on Multimedia and Expo Workshops (ICMEW), pp. 671, Melbourne, VIC, July, 2012.
[59] C. H. Teng, and J. Y. Chen, "An Augmented Reality Environment for Learning OpenGL Programming", IEEE 9th International Conference on Ubiquitous Intelligence and Computing and 9th International Conference on Autonomic and Trusted Computing (UIC/ATC), pp. 996-1001, Fukuoka, September, 2012.
[60] C. L. Rathbone, "Processing Time Analysis in a Distributed Parallel Environment with Interrupts," Ph.D. thesis, University of Connecticut, 1988.
[61] A. R. Dash, S. K. Sahu and S. K. Samantra, "An Optimized Round Robin CPU Scheduling Algorithm with Dynamic Time Quantum", International Journal of Computer Science, Engineering and Information Technology (IJCSEIT), Vol. 5, No. 1, February 2015.
[62] M. S. Iraji, "Time Sharing Algorithm with Dynamic Weighted Harmonic Round Robin", Journal of Asian Scientific Research, Vol. 5, No. 3, pp. 131-142, 2015.
[63] D. Maste, L. Ragha and N. Marathe, "Intelligent Dynamic Time Quantum Allocation in MLFQ Scheduling", International Journal of Information and Computation Technology, ISSN 0974-2239, Vol. 3, No. 4, pp. 311-322, 2013. International Research Publications House.
[64] A. Noon, A. Kalakech and S. Kadry, "A New Round Robin Based Scheduling Algorithm for Operating Systems: Dynamic Quantum Using the Mean Average", International Journal of Computer Science Issues (IJCSI), Vol. 3, Issue 3, No. 1, pp. 224-229, May 2011.
[65] I. S. Rajput and D. Gupta, "A Priority Based Round Robin CPU Scheduling Algorithm for Real Time Systems", International Journal of Innovations in Engineering and Technology (IJIET), Vol. 1, Issue 3, pp. 1-11, October 2012.
[66] H. S. Behera, R. Mohanty and D. Nayek, "A New Proposed Dynamic Quantum with Re-Adjusted Round Robin Scheduling Algorithm and Its Performance Analysis", International Journal of Computer Applications (0097 – 8887), Vol. 5, No. 5, pp. 10-15, August 2010.
[67] D. Nayek, S. K. Malla and D. Debadarshini, "Improved Round Robin Scheduling using Dynamic Time Quantum", International Journal of Computer Applications (0097 – 8887), Vol. 38, No. 5, pp. 34-38, January 2012.
[68] M. K. Mishra and F. Rashid, "An Improved Round Robin CPU Scheduling Algorithm with Varying Time Quantum", International Journal of Computer Science, Engineering and Applications (IJCSEA), Vol. 4, No. 4, August 2014.
[69] A. Singh, P. Goyal and S. Batra, "An Optimized Round Robin Scheduling Algorithm for CPU Scheduling", International Journal on Computer Science and Engineering, pp. 2383-2385, 2010.

[70] C. Yaashuwanth and R. Ramesh, “A New Scheduling Algorithms for Real-Time Tasks”, International Journal of Computer Science and Information Security (IJCSIS), Vol. 6, No. 2, 2009.
[71] R. Matarneh, “Self-Adjusted Time Quantum in Round Robin Algorithm Depending on Burst Time of the Now Running Processes”, American Journal of Applied Sciences, Vol. 6, No. 10, pp. 1831-1837, 2009.
[72] S. Kuankid, A. Aurasopon and W. Sa-Ngiamvibool, “Effective Scheduling Algorithm and Scheduler Implementation for Use with Time-Triggered Co-operative Architecture”, Elektronika IR Elektrotechnika, ISSN 1392-1215, Vol. 20, No. 6, pp. 122-127, 2014.
[73] B. Miller, F. Vahid and T. Givargis, “RIOS: A Lightweight Task Scheduler for Embedded Systems”, WESE 12th Proceedings of the workshop on Embedded and Cyber-Physical Systems Education, ACM, New York, NY, USA, 2013.
[74] S. M. Mostafa, S. Z. Rida and S. H. Hamad, “Finding Time Quantum of Round Robin CPU Scheduling Algorithm in General Computing Systems Using Integer Programming”, International Journal of Research and Reviews in Applied Science (IJRRAS), Vol. 5, No. 1, pp. 64-71, October, 2010.
[75] C. J. Hwang, W. Mu and A. Sampat, “Effective Scheduling on Mobile Embedded System”.
[76] R. M. Das, M. L. Prasanna and Sudhashree, “Design and Performance Evaluation of A New Proposed Fittest Job First Dynamic Round Robin (FJFDRR) Scheduling Algorithm”, International Journal of Computer Information Systems (IJCIS), Vol. 2, No. 2, 2007.
[77] S. Tasneem, “Application and Effects of Process Residual Time Information in Time-Sensitive Dynamic Scheduling," Ph.D. thesis, University of Connecticut, 2006.
[78] A. Alsheikhy, R. Elfouly and R. Ammar, “An Improved Dynamic Round Robin Scheduling Algorithm Based on a Variant Quantum Time”, 2015 11th International Computer Engineering Conference “ICENCO”, pp. 98-104, Cairo, December 2015.
[79] A. Alsheikhy, R. Elfouly, M. Alharthi, R. Ammar and A. Alshegaifi, “An Effective Real-Time Dynamic Scheduling Approach for Periodic Tasks”, 5th International Conference on Advances in Engineering Sciences and Applied Mathematics (ICAESAM'2016), pp. 78-82, Hong Kong, May, 2016.
[80] A. Alsheikhy, R. Ammar, R. Elfouly, M. Alharthi and A. Alshegaifi, "An Efficient Dynamic Scheduling Algorithm for Periodic Tasks in Real-Time Systems Using Dynamic Average Estimation“,21st IEEE Symposium on Computers and Communication (ISCC 2016) – 20th IEEE Symposium on Computers and Communications, pp. 820-824, Messina, Italy, June, 2016.
[81] G. Muller, “Scheduling Techniques and Analysis”, Buskerud University College, March 2013.
[82] J. J. Chen, C. Y. Yang, H. I. Lu and T. W. Kuo, ”Approximation Algorithm for Multiprocessor Energy-Efficient Scheduling of Periodic Real-Time Tasks with

Uncertain Task Execution Time", Real-Time and Embedded Technology and Applications Symposium, 2008. RTAS '08. IEEE, pp. 13-23, April 2008.
[83] L. Niu, "System-Level Energy-Efficient Scheduling for Hard Real-Time Embedded Systems", International conference on Design, Automation and Test in Europe Conference and Exhibition, pp. 1-4, March 2011.
[84] M. Hwang, P. Kim and D. Choi, "Least Slack Time Rate first: an Efficient Scheduling Algorithm for Pervasive Computing Environment", Journal of Universal Computer Science, Vol. 17, No. 6, pp. 912-925, 2011.
[85] M. Hwang, P. Kim and D. Choi, "Least Slack Time Rate first: New Scheduling Algorithm for Multi-Processor Environment", International conference on Complex, Intelligent and Software Intensive Systems, pp. 806-811, 2010.
[86] M. Kaladevi, M.C. A., M. Phil and Dr. S. Sathiyabama, "A Comparative Study of Scheduling Algorithms for Real-Time Task", International Journal of Advances in Science and Technology, Vol. 1, No. 4, pp. 9-14, 2010.
[87] L. Niu, "Energy Efficient Scheduling Techniques for Real-Time Embedded Systems with QOS Guarantee", 16th International conference of Embedded and Real-Time Computing Systems and Applications (RTCSA), pp. 163-172, August 2010.
[88] L. Almeida, S. Fischmeister, M. Anad and I. Lee, "A Dynamic Scheduling Approach to Designing Flexible Safety-Critical Systems", Proceedings of the 7th ACM and IEEE International conference on Embedded Software (EMSOFT), pp. 67-74, New York, NY, USA, 2007.
[89] J. H. Anderson, V. Bud and U. C. Devi, "An EDF-based Scheduling Algorithm for Multiprocessor Soft Real-Time Systems", Proceedings of the 17th Euromicro Conference on Real-Time Systems (ECRTS), pp. 199-208, 2005.
[90] L. A. Cortes, "Verification and Scheduling Techniques for Real-Time Embedded Systems", Department of Computer and Information Science, Dissertation No. 920, Linkoping University, Sweden, 2005.
[91] S. Baruah and J. Goossens, "Scheduling Real-Time Tasks: Algorithms and Complexity", pp. 1-35, 2003.
[92] A. Dudani, F. Mueller and Y. Zhu, "Energy-Conserving Feedback EDF Scheduling for Embedded Systems with Real-Time Constraints", Proceedings of the Joint conference on Languages, Compilers and Tools for Embedded Systems: Software and Compilers for Embedded Systems, Vol. 37, No. 7, pp. 213-222, New York, NY, USA, July 2002.
[93] N. C. Audsley, A. Burns, M. F. Richardson and A. J. Wellings, "Hard Real-Time Scheduling: The Deadline-Monotonic Approach", Proceedings of IEEE Workshop on Real-Time Operating Systems and Software, pp. 133-137, 1991.
[94] K. Li, X. Tang and B. Veeravalli, "Scheduling Precedence Constrained Stochastic Tasks on Heterogeneous Cluster Systems", IEEE Transactions on computers, Vol. 64, No. 1, pp. 191-204, January 2015.
[95] K. Li, X. Tang and Q. Yin, "Energy-Aware Scheduling Algorithm for Task Execution Cycles with Normal Distribution on Heterogeneous Computing Systems", 41st International conference on parallel processing, pp. 40-47, 2012.

[96] S. Tasneem, F. Zhang, L. Lipsky and S. Thompson, “Comparing Different Scheduling Schemes for M/G/1 Queue”, 6th International Journal on Electrical and Computer Engineering (ICECE), pp. 746-749, Dhaka, Bangladesh, 2010.
[97] J. Cong and K. Gururaj, “Energy Efficient Multiprocessor Task Scheduling Under Input-Dependent Variation”, Proceedings of IEEE C on Design, Automation and Test in Europe conference Exhibition, pp. 411-416, April 2009.
[98] N. Satish, K. Ravindran and K. Keutzer, “Scheduling Task Dependence Graphs with Variable Task Execution Times onto Heterogeneous Multiprocessors”, Electrical Engineering and Computer Sciences, University of California at Berkley April 2008.
[99] J. C. Beck and N. Wilson, “Proactive Algorithms for Job Shop Scheduling with Probabilistic Durations”, Journal of Artificial Intelligence Research 28, pp. 183-232, 2007.
[100] S. Tasneem, R. Ammar and H. Sholl, “A Methodology to Compute Task Remaining Execution Time”, Proceedings of the International Symposium on Computers and Communications (ISCC), pp. 74-79, 2004.
[101] S. Tasneem, L. Lipsky, R. Ammar and H. Sholl, “Using Residual Times to Meet Deadlines in M/G/C Queues”, NCA, pp. 128-138, 2005.
[102] S. Tasneem, H. Sholl and R. Ammar, “Practical Methods for Deadline Scheduling using Process Residual Time”, CAINE, pp. 175-180, 2005.
[103] S. Tasneem, R. Ammar, L. Lipsky and H. Sholl, ”Improvement of Real-Time Job Completion Using Residual Time-Based)RTB) Scheduling”, International Journal of Computers and their Applications, Vol. 17, No. 3, pp. 117-132, March 2010.
[104] H. Tian, W. Li, L. Wang and P. Ogunbona, "Smoke Detection in Video: an Image Separation Approach," International Journal of Computer Vision, Vol. 106, No. 2, pp. 192-209, 2014.
[105] B. Rekabdar and D. Wilson, "Smoke Detection via Linear Separation," University of Nevada, Reno, Department of Mathematics and Statistics, May 2015.
[106] A. Alsheikhy, R. Elfouly, M. Alharthi, R. Ammar and A. Alshegaifi, ” Hybrid Scheduling Algorithm for Periodic Tasks in Real-Time Systems”, Journal of King Abdulaziz University (JKAU), 2016, “submitted and under review”.
[107] Controller Area Network Abstract Version, “pdf file”.
[108] G. I. Mary, Z. C. Alex and L. Jenkins, “Modeling and Analysis of Wireless Controller Area Network: A Review”, Communication and Network, pp. 126-133, May 2013.
[109] N. Nasiriani, R. Ramachandran, K. Rahimi, Y. P. Fallah, S. Bossart and K. Dodrill, “An Embedded Communication Network Simulator for Power Systems Simulations in PSCAD”, 2013 IEEE Power and Energy Society General Meeting, pp. 1-5, July 2013.
[110] W. L. Ng, C. K. Ng, N. K. Noordin and B. M. Ali, “Performance Analysis of Wireless Control Area Network (WCAN) Using Token Frame Scheme”, 2012 3rd International Conference on Intelligent Systems Modelling and Simulation (ISMS), pp. 695-699, February 2012.

[111] K. H. Johansson, M. Torngren and L. Nielsen, “Vehicle Applications of Controller Area Network”, Chapter of handbook of Network and Embedded Control Systems, part of the series Control Engineering, pp. 741-765, 2005.