THE IMPACT OF SOFTWARE ARCHITECTURE ON THE COST OF DESIGN, IMPLEMENTATION AND VERIFICATION OF RELIABLE EMBEDDED SYSTEMS

Thesis submitted for the degree of

Doctor of Philosophy

at the University of Leicester

by

Noor Azurati Ahmad

Embedded Systems Laboratory

Department of Engineering

University of Leicester

June 2013

The impact of software architecture on the cost of design, implementation and verification of reliable embedded systems

Noor Azurati Ahmad

Abstract

The concern of this thesis is the development of software for systems utilising embedded processors. In many cases, the safety of users of "embedded systems" (and other people in the immediate vicinity) depends on the correct operation of this software.

This project explores the ways in which the cost of designing, implementing and verifying the behaviour of systems that include embedded software can be reduced. More specifically, the goal is to determine the extent to which the use of a time-triggered (TT) architecture - as opposed to an equivalent "event triggered" (ET) architecture - could offer benefits to the developers of reliable embedded systems. To evaluate this, a method of software architecture evaluation was developed and is described.

The work detailed in this thesis involved an extensive empirical study of the costs involved in testing TT systems, with and without task pre-emption. Factors considered in this comparison included: [i] implementation costs, including code size, overhead, memory and CPU utilisation of a scheduler; [ii] testing costs, including the ease of obtaining timing data for isolated and in-situ tasks; and [iii] design costs, including execution time, lines of code and number of inputs required to perform a test of schedulability on the task set.

The results from empirical studies suggested the use of TT architectures (compared with equivalent designs based on ET architectures) would require greater efforts at the design phase, but lower efforts during the testing phases. The results also suggested systems based on TT designs are likely to have lower implementation costs than equivalent systems based on ET designs. Taken together, the results point to a lower overall cost for TT systems.

Execution of the method is described through the presentation of experimental case studies. Throughout these activities, the method has been shown to be a capable tool for software architecture evaluation.

[302 words]

Al-hamdu Lillahi rabbil 'alamin.

This thesis is dedicated to my husband, Zaime Kharis, my daughters, Noorhanis Zaiyana and Noorhani Zaiyani, my parents, Hjh Khadijah, Hj Zainuddin and Hjh Khatijah, and my family for their unfailing love, belief, understanding and encouragement

Acknowledgements

The work presented in this thesis was supported by the Government of Malaysia (SLAI Award) and Universiti Teknologi Malaysia (UTM). I would like to express much appreciation to Professor Michael J. Pont for his supervision, guidance and enthusiasm throughout the duration of this project.

Also, my warmest thanks go to colleagues past and present at the Engineering department and Embedded Systems Lab, for the friendly atmosphere and long and pointless coffee break discussions: Ayman, Ricardo, Adi, Mouaz, Nazri, Hui Yan, Imran, Keith, Daniel, Kam, Dong, Farah, Dev, Zemian, Musharraf, Fayyaz, Ridzuan, Susan, Alex, Douglas, Saad, Pao, Irfan, Anjali, Adam, Mohammad, Imran, Amir, Pete, Ioannis and Aley. I am also grateful to the people I have collaborated with over the years.

Thanks also to Duncan from Student Development and Fernando, the Postgraduate Tutor in the Engineering Department for their support.

For comments and discussions and for assistance proof reading various parts of the thesis, well deserved credit goes to Caroline, Zemian, Saad and Nor Mazuita.

Finally, I am truly indebted to my family for their love, encouragement and support: Hjh Khadijah, Hj Zainuddin, Hjh Khatijah, brothers and sisters, and my beloved daughters, Noorhanis Zaiyana *(Yana)* and Noorhani Zaiyani *(Hani)*. And, of course, also my dearest husband, Zaime Kharis, you have been there for me through thick and thin.

Table of Contents

List of figures

List of tables

List of related publications

Ahmad, N. and Pont, M.J. (2010) "Debugging remote embedded systems: The impact of system software architecture", *Proceedings of the 2010 UK Electronics Forum,* Newcastle, UK, 30 June-1 July, 2010, pp.17-23. Published by Newcastle University. ISBN 978-0-7017-0232-8.

Ahmad, N and Pont, M.J. (2009) “Remote debugging of embedded systems which employ a time-triggered architecture”. *Proceedings of the Fifth UK Embedded Forum*. Leicester, UK, 23-24 September 2009, pp. 97. Published by Newcastle University ISBN 978-0-7017-0222-9.

Ahmad, N and Pont, M.J. “How do you debug a spacecraft when it is a million miles from home?” Poster, Postgraduate Festival 2010, University of Leicester.

List of abbreviations

AI	Artificial Intelligent
ALMA	Architecture Level Modifiability Analysis
ATAM	Architecture Trade-off Analysis Method
ARM	Advanced RISC Microcontroller
BCET	Best-Case Execution Time
CP	Control Path
CSP	Communicating Sequential Processes
CPU	Central Processing Unit
DARTS	Design Approach for Real-Time Systems
DM	Deadline Monotonic
DS	Deferrable Server
DCPIG	Dedicated Coloured Process Interaction Graph
EMI	Electromagnetic Interference
EMC	Electromagnetic Compatibility
ECU	Electronic Control Units
EDD	Earliest Due Date
EDF	Earliest Deadline First
EOG	Execution Order Graph
ESL	Embedded Systems Lab
EST	Estimate
ET	Event-Triggered
ETC	Event-Triggered Co-operative
ETP	Event-Triggered Pre-emptive
FIQ	Fast Interrupt Request
FFT	Fast Fourier Transform
FMEA	Fault Model Effects Analysis
FPP	fixed pre-emption point
FTA	Fault Tree Analysis
GBRAM	Goal-Based Requirement Analysis Method
GDB	GNU Debugging
GSN	Goal Structuring Notation
GSM	Global System for Mobile communications

JPL	Jet Propulsion Laboratory
HARTS	Hexagonal Architecture for Real-Time Systems
HIL	Hardware In Loop
HMI	Human Machine Interface
HMON	HARTS Monitor System
I/O	Input/Output
IDE	Integrated Development Environment
IRQ	Interrupt Request
KB	Knowledge-Based
LCD	Liquid-crystal display
LOC	Lines of Code
LR	Link Register
MDE	Model Driven Engineering
NASA	National Aeronautics and Space Administration
NIMSAD	Normative Information Model-based Systems Analysis
NPR	floating non-pre-emptive region
npEDF	Non pre-emptive EDF
PASA	Performance Assessment of Software Architecture
PC	Program Counter
PE	Priority Exchange
PS	Polling Server
PSR	Program Status Register
RAM	Random Access Memory
RTA	Response Time Analysis
RM	Rate Monotonic
RTOS	Real-time Operating Systems
SA	Software Architecture
SAAM	Scenario-based Architecture Analysis Method
SPN	Shortest Process Next
SJF	Shortest Job First
TCB	Task Control Block
TEFSM	Timed Extended Finite State Machines
TORSCHE	Time Optimisation of Resources, Scheduling
TT	Time-Triggered
TTC	Time-Triggered Co-operative

TTH	Time-Triggered Hybrid
TTSA	Time-Triggered Schedulability Analysis
TTP	Time-Triggered Pre-emptive
TTH	Time-Triggered Hybrid
TTRM	Time-Triggered Rate Monotonic
UK	United Kingdom
V&V	Validation and Verification
WCET	Worst-Case Execution Time

Chapter 1
Introduction

1.1 Introduction

This chapter presents an overview of the research area addressed in this thesis and introduces the main goals of the research. It includes a review highlighting a case stressing the importance of testing and debugging, offers contextual information exploring methods for evaluating software architecture based on scheduling strategy cost of design, implementation and the verification of embedded real-time systems. The research hypothesis and the contributions of the research are also described in this chapter.

1.2 Motivation

On July 4, 1997, Pathfinder (see Figure 1.1) landed successfully in the Ares Vallis region of Mars. The spacecraft was a robotic embedded system, designed to collect samples, capture video images and transmit meteorological readings back to Earth. Unfortunately, in its third week of operation, the Pathfinder encountered problems that required a total system reset (Cook and Spear, 1998), resulting in long data acquisition delays (Durkin, 1998).

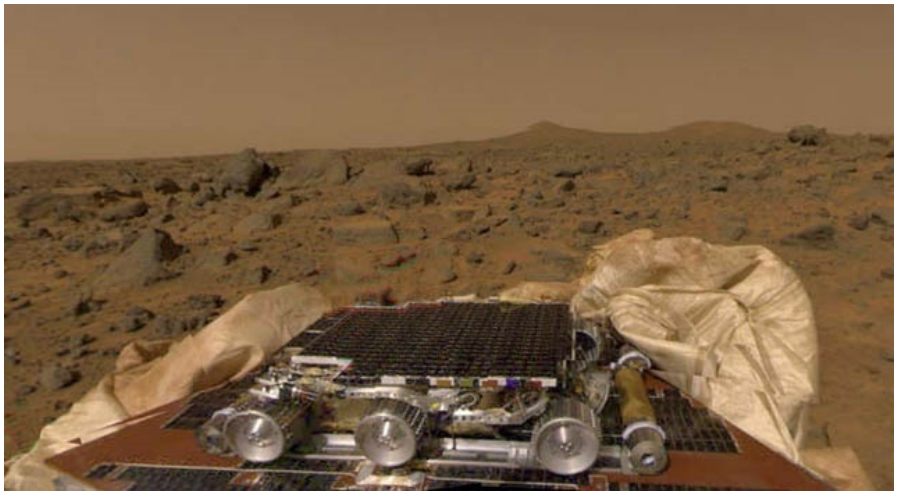


Figure 1.1 Pathfinder on Mars (source: jpl.nasa.gov)

It took three weeks (Durkin, 1998) for the engineers at the Jet Propulsion Laboratory (JPL), to remotely diagnose the problem. The software bug was eventually fixed following extensive simulations lasting eighteen hours (Durkin, 1998). Once the software in the spacecraft was updated, it again became fully operational.

In many ways, the successful debugging of the Pathfinder can be seen as a significant achievement. Despite the device being millions of miles away on another planet, programmers were able to restore it to normal operation. However, if we factor in that it took, what is probably one of the most advanced engineering teams on the planet, around three weeks to address the problem the success is not quite so impressive. For instance; had this been a manned mission, and the failure involved a critical function, the astronauts may not have survived for three weeks.

The importance of rapid testing and verification has already been acknowledged in aerospace, automobile and military domains. For example, NASA and ESA (Sha et al., 2004) have sponsored extensive studies into the impact of costs incurred during development phases and assessment prior to design. In addition, the need to use appropriate software architecture to ensure the testability of complex real-time systems

has been noted by scholars (e.g. Kopetz (1991), Scheler, (2006), Thane (2000) and Xu, (2003)). However, there is not yet broad acceptance over the best way to design such systems in order to reduce testing costs.

There are two main ways to design an embedded software system, namely using event-triggered (ET) or time-triggered (TT) architecture. TT architecture is a subset of ET architecture, as shown in Figure 1.2. However, in the case of TT, a recurring clock tick controls the only event that can trigger action.

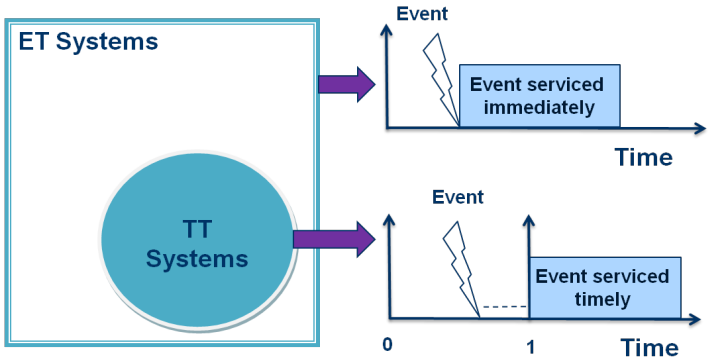


Figure 1.2 Illustration of ET and TT systems

In the first approach, the so-called event-triggered (ET) approach, processing activities are initiated in response to specific external events. In the second approach, the time-triggered (TT) approach, processing activities are initiated at predetermined points in time. This fundamental difference has a huge impact on the entire development process of embedded systems (as shown in Figure 1.3), including design, implementation, testing and validation for real-time critical applications.

In the Pathfinder system, an ET architecture with pre-emptive scheduling was used. Although it is clear that the software architecture used in the Pathfinder could control a complex real-time system, the need for high testing and debugging to detect and fix the problem, which occurred, at great cost and effort is disconcerting.

Figure 1.3 Typical software development lifecycle

Due to the additional software architecture option when designing embedded systems, it is essential to compare whether the use of a TT architecture software superior to that of ET (Allworth, 1981; Pont, 2001). It is also important to examine the impact of limited pre-emptive scheduling and co-operative scheduling, to evaluate if they can provide any benefits when testing reliable embedded systems.

1.3 Research objectives and hypotheses

The aim of this project is to explore the benefits provided by the use of a time-triggered software architecture with co-operative scheduling in the development of reliable embedded systems. More specifically, the following hypotheses will be tested:

H1. Use of limited pre-emptive scheduling in a design results in lower testing costs than the use of fully pre-emptive scheduling and co-operative scheduling in the implementation phase.

H2. Testing a system with a TT architecture incurs less cost than testing an equivalent system with an ET architecture, when experimental-based methods and comparative analysis are used.

H3. The cost of verifying a system with a TT design is always higher than that required to verify an equivalent system with various types of ET design.

1.4 Thesis contribution

This thesis makes the following contributions to the area of research:

- It offers a novel integrated software architecture evaluation approach, based on experimental work carried out to analyse the impact of a TT architecture vs. an ET architecture on the cost of design, implementation and testing real-time embedded systems.
- Assessing and analysing the effects of test running times and other related costs from the TT architecture; using a limited pre-emptive and co-operative schedulability test (represented by TTSA) and ET architecture with a pre-emptive (represented by RTA) and co-operative (represented by heuristic search) schedulability test when the number of tasks increases.
- Assessing the impact of the implementation costs involved in creating TTC, TTH and TTP scheduling for a small and a large system, including lines of code, CPU and memory utilisation.
- Introducing a new experimental-based evaluation approach to compare ease of testing for systems under the TT and ET architecture; based on the underlying hypothesis that testing will be easier for systems in which the timing data obtained for isolated and in-situ tasks is very similar.
- Bridging the gap between testing fully pre-emptive, limited pre-emptive and co-operative scheduling by utilising the effects of task synchronisation methods to provide similar timing behaviour for isolated tasks and the task runs in the completed system.

1.5 Thesis structure

The remainder of this thesis is organised as follows:

- Chapter 2 details the basic concept of software architecture and scheduling strategy in real-time embedded systems. It also explains the evolution of pre-emptive scheduling and co-operative scheduling.
- Chapter 3 examines the motivation that has driven work on the reliability and predictability aspects of real-time software architecture to reduce efforts and costs for testing embedded real-time applications. It also covers related work on real-time system software architecture that might influence timeliness verification at the design stage, implementation stage and testing stage.
- A novel integrated software architecture evaluation method for analysing the impact of a TT and ET architecture on the cost of design, implementation and testing of real-time embedded systems is described in Chapter 4, leading to three experimental-based assessments as discussed in Chapters 5 to 7.
- Chapter 5 covers the first experimental work, focusing on the cost analysis for pre-emptive and co-operative scheduler implementation. The problem, method and results are presented.
- Chapter 6 covers the analysis for the costs of testing assessment, with details of the problem, method and results. It describes the design and implementation of the task in isolation for ET and TT designs. This chapter also provides detailed analyses and case studies to measure the impact of task synchronisation and inter-task communication when testing real-time systems.

- Chapter 7 presents an evaluative study to assess the cost of design, for ET and TT-based architectures using a schedulability test, complete with the discussion of problems presented, methods involved and the results obtained.
- Finally, the contributions of the research are highlighted and limitations and future work for this study are discussed in Chapter 8.

Chapter 2
Embedded Software Architecture and Real-time Task Scheduling

2.1 Introduction

This chapter introduces the concept of embedded real-time systems, and explains their relevance to this work. It contains explanations about tasks, software architectures, scheduling strategy and the operating systems used when implementing such systems. The chapter also covers the evolutionary trends for scheduling algorithms, to understand the broader implications of real-time software architecture.

2.2 Real-time software architecture

In ANSI/IEEE standard 1471-2000, architecture *(Garlan, 2000)* is defined as:

"[Software architecture goes] beyond the algorithms and data structures of the computation; designing and specifying the overall system structure emerges as a new kind of problem. Structural issues include gross organisation and global control structure; protocols for communication, synchronisation, and data access; assignment of functionality to design elements; physical distribution of design elements; scaling and performance; and selection among design alternative".

Real-time systems are used to control physical processes with a diverse array of complexities; ranging from automobile ignition systems, to controllers for inflight systems and nuclear power plants. A real-time system is one in which the correctness of the system is based on the correctness of the logical results obtained, or outputs and their timeliness.

The principal component of real-time systems is embedded real-time software, which can observe and respond to triggering events from the environment, albeit periodically or immediately. The frequency depends on which software architecture the real-time systems utilises. Some architectures are preferred, due to testability, resource utilisation, predictability, extensibility and fault tolerance (Scheler and Schroieder-Preikschat, 2006).

2.2.1 Tasks

The process or task is most important entity handled by embedded real-time software (Buttazzo, 2005a). A task encapsulates all the information involved in the execution of a program, for instance stack, program counter (PC), source code and data (Labrosse, 2008). However, some developers have provided different viewpoints in reference to the processes and the tasks involved. For example, Butazzo defines a task as "*a sequential execution of code that does not suspend itself during execution while a process is more complex computational activity and may contain more than one task. Computational activities in real-time systems are called real-time tasks."* (Buttazzo, 2005a).

A task can be suspended by an internal or external interrupt, as is the case in pre-empted architectures. Moreover, a task can also be defined as an independent thread of execution comprising a sequence of independently schedulable instructions, which competes independently for CPU execution.

2.2.2 Task properties

The categories into which real time tasks can be divided are: periodic, aperiodic and sporadic.

- Periodic tasks are activated at fixed time intervals or periods (Schneider, 2003). Thus, all the points in time, at which such tasks will be activated are known in advance. This type is typically used to monitor sensor data and provide updates on the current state of internal variables and outputs.
- Aperiodic tasks are activated by events that occur in the environment (internal or external events) at unpredictable points. For example, an aperiodic task might be activated when a switch is pressed (Laplante and John Wiley & Sons., 2004).
- Sporadic tasks are a form of aperiodic task, in which consecutive tasks are separated by a guaranteed minimum inter-arrival time (Schneider, 2003).

Each real-time task must meet a set of time constraints imposed in the form of a response time for the task (Buttazzo, 2005a). Due to their nature, the timing constraints for aperiodic tasks can be less critical than those of periodic and sporadic tasks. In many ways, a sporadic task is similar to an aperiodic task, although the minimum separation between two consecutive instances of sporadic tasks restricts the rate at which they can arise. In the case of an aperiodic task, the minimum separation can be 0. For example, in the area of robotics, a task that is generated for handling an obstacle that has suddenly appeared is a sporadic task. The time of occurrence of the task cannot be predicted and a system in which all timing constraints are met is well-timed.

In general, each real-time task, τ_i is characterised by the following parameters (Buttazzo, 2005a):

- Task release time r_i : is the time at which a task becomes ready for execution.
- Task computation time c_i: is the time allocated to the processor for executing the task without interruption.

- Task deadline d_i : is the time before which a task should be completed to avoid damage or performance degradation.
- Task period, P_i : is the minimum length of intervals between the release times of consecutive tasks.
- Task worst-case execution time ($WCET_i$): is the longest allowed computation time for tasks.
- Task best-case execution time ($BCET_i$): is the shortest computation time for tasks.

Some of the parameters defined above are illustrated in Figure 2.1.

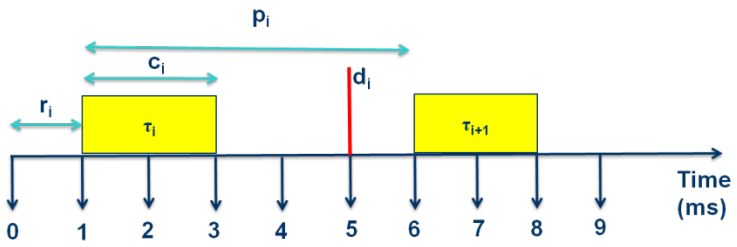


Figure 2.1 Typical parameters of a real-time task (Buttazzo, 2005a)

Real-time tasks must meet the deadline (Buttazzo, 2005a). A task is said to be hard if completion after its deadline can result in catastrophic consequences for the system. In this case, any instance of such a task should be guaranteed a priori in the worst-case scenario (Buttazzo, 2005a). On the other hand, if the effect of missing a task deadline is to decrease system performance, the task is classified as a soft real-time task (Laplante and John Wiley & Sons., 2004).

In addition, one or more of the task constraints described below can be used to define the relationship between tasks (Buttazzo, 2005a, Baruah et. al., 1999):

- Distance: is defined as the minimum time interval between the completion of one task and the start of another;

- Precedence: is used to specify the execution order of two tasks;
- Exclusion: is used to maintain data consistency and control access to shared resources;
- Latency: can be defined as the maximum duration of time between the start of one task and the completion of another;
- Jitter: refers to the variation between the inter-completion [or activation] times for successive jobs of the same task"; and
- Task Offset: is the time between system power on and commencement of the first period of the task.

2.2.3 Shared resources

In every real-world scheduling problem, it is necessary for some (or all) tasks to share some kind of resource; for example a data structure, a set of variables, a main memory area, a file, a piece of program or a peripheral device (as shown in Figure 2.2) (Schneider, 2003). A resource that can be used in more than one task is known as a shared resource. Some shared resources require mutual exclusion such as when they are engaged in competing tasks. Thus, a software system must provide a synchronisation mechanism, such as a semaphore, to allow tasks to access mutually exclusive resources sequentially.

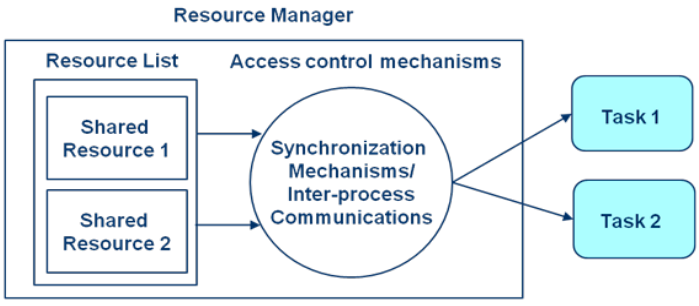


Figure 2.2 Illustration of Resource Management (modified from Pont (2008))

2.2.4 Functional blocks of architecture

Real-time software architecture consists of well-defined fundamental functional blocks, which are referred to in this thesis as the scheduler, the resource manager and the task dispatcher. The architecture is designed to ensure that the processor services the tasks requested, as shown in Figure 2.3. The resource manager allocates memory and a processor to the task, which is then placed on the 'ready list' (a list of tasks that are ready for execution). The dispatcher then scans the ready list to identify a task that can be executed on the available processor, and starts its execution. Specifically in real-time systems, the task must be completed within a specified time, called the deadline.

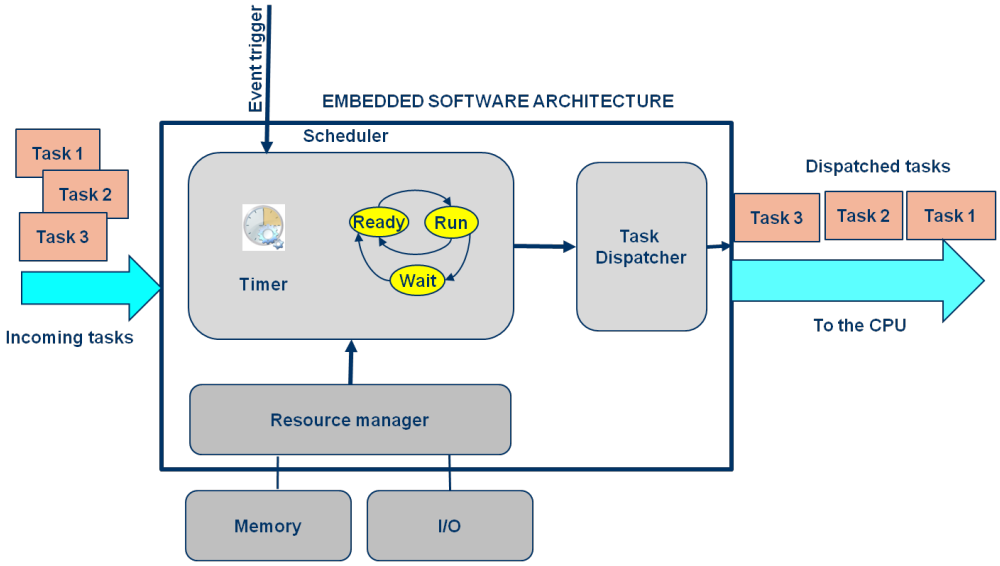


Figure 2.3 Functional Block diagram of a typical Real-time Software Architecture, with the Scheduler, Resource Manager and the Task Dispatcher (modified from Kopetz, 1997).

The set of rules which define these timing constraints are defined by the design engineer, and are assigned to a software algorithm, which produces a schedule for each task, based on which tasks are dispatched to the processor. However, there is a dilemma concerning how the tasks should be prioritised. The processor can only execute a single task at any one time. There are a further two options for the designer; either to allow an existing task to complete, prior to the start of the next task, or stop the existing task

midway, allowing the higher priority task to execute. Thus arises, the concepts of co-operation and pre-emption in task execution, as will be discussed later.

The scheduler is a fundamental block of the software architecture, and the majority of the literature has focused on its design, implementation and evaluation methods. The scheduler is discussed in detail subsequently.

2.3 Software architecture categories

A brief introduction to ET and TT architecture is given in Chapter 1. A detailed explanation of these architectures and their classification will be given in this section.

2.3.1 Time-triggered architecture

TT architectures are widely used by the aerospace and medical systems industries (Schild and Würtz, 2000); they are less familiar to developers of mainstream embedded systems. The main concept behind the architecture is describe by Pont (2001b):

When implementing TT systems, the key thing we need to remember is the "one interrupt per CPU" rule. That is, TT designs only have one interrupt enabled. This single interrupt is usually linked to a timer "tick", which might occur (for example) every millisecond.

The system tick is used to trigger an interrupt (the tasks) service routine. When an event occurs in-between tick interrupts, then the event handler is delayed until the next scheduler invocation. Figure 2.4 illustrates the architecture for a TT system approach.

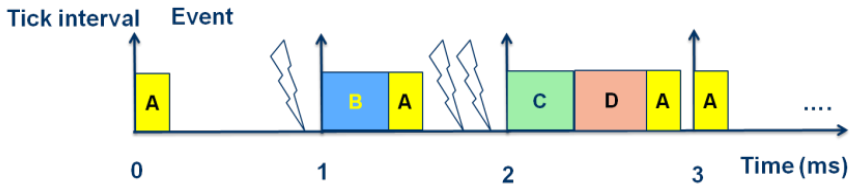


Figure 2.4 Illustration of TT systems

In TT software architecture, scheduling algorithms are developed based on a set of static predetermined schedules (Katcher et al., 1993; Liu, 2000). These schedules must consider all task dependencies and provide for implicit synchronisation of the tasks run times (Xu, 2000; 2003). All task activations can be pre-calculated offline and the entire schedule then stored in a table. At run time, the operating system executes all the tasks based on a lookup table which contains all the guaranteed tasks arranged in the proper order (Buttazzo, 2005a).

2.3.2 Event-triggered architecture

Event-triggered systems are designed to immediately react to a significant change of state or event in the environment, by reconsidering the current schedule. The detection of events is via the interrupt mechanism. For example, an event can cause an interrupt in the current execution. Figure 2.5 illustrates the architecture for the ET approach.

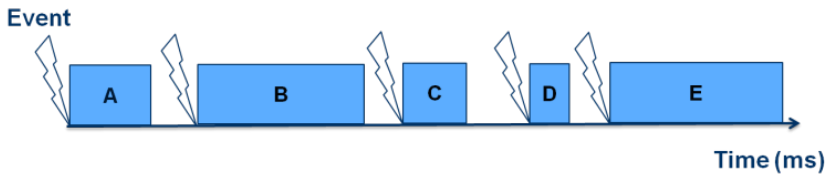


Figure 2.5 Illustration of ET systems

The design involves the handling of multiple interrupts. For example, interrupts may arise as a consequence of periodic timer overflows, the arrival of messages on a CAN bus, the pressing of a switch, the completion of an analogue-to-digital conversion, and so on. In order to create such systems, the developer may write codes designed to manage the various interrupts directly: this will typically involve creating a so-called “interrupt service routine” (ISR) to deal with each event, as ISRs take the shortest time to handle the interrupt (Brunl, 2006; Laplante, 2004). Moreover, ET systems require a

dynamic scheduling strategy to activate the appropriate software tasks to service the event (Schild and Würtz, 2000).

There is a major difference between interrupt handling in ET and TT systems. In an ET design, interrupts can cause the system to respond immediately. For example, pressing a switch for an external interrupt may cause execution of a certain task on an ET system. However, with a TT design, all other interrupts are polled (Pont, 2003), and the system may detect repeat events by periodically executing a task that monitors the switch for changes and then acts appropriately when they are detected.

2.4 Design of the scheduler

As mentioned by Butazzo (2005), task scheduling can be grouped according to type of task: periodic and aperiodic tasks, as depicted in Figure 2.6, are used as the fundamental principles to categorise the schedulers for ET and TT architectures. Each principle is then classified into pre-emptive and co-operative scheduling, at which point, four principles of schedule designs have been derived; namely ET pre-emptive (ETP), ET co-operative (ETC), TT pre-emptive (TTP) and TT co-operative (TTC). The operation of each scheduler is discussed in the following sections.

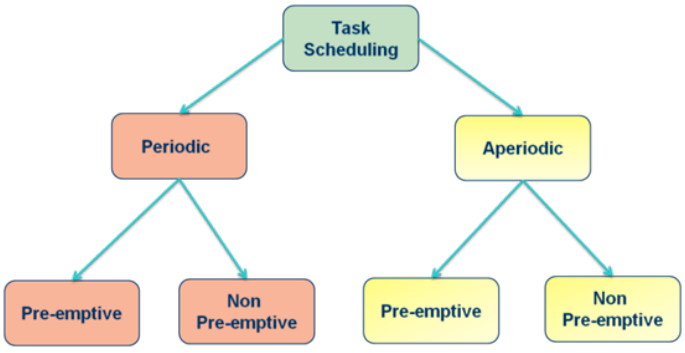


Figure 2.6 Classification of real-time scheduling (Buttazzo, 2005)

2.4.1 ET pre-emptive scheduling

In an ET architecture, events can occur to activate tasks at any time, either periodically or dynamically during execution; subsequently pre-emption becomes an important factor. Numerous papers have discussed pre-emptive scheduling and designed it by taking different approaches to scheduling algorithms, such as fixed priority and dynamic priority. However, in this chapter, the operations of the scheduler and the manner in which these are verified will be scrutinised further.

The operations of the ETP scheduler can be illustrated using Figure 2.7 and Figure 2.8.

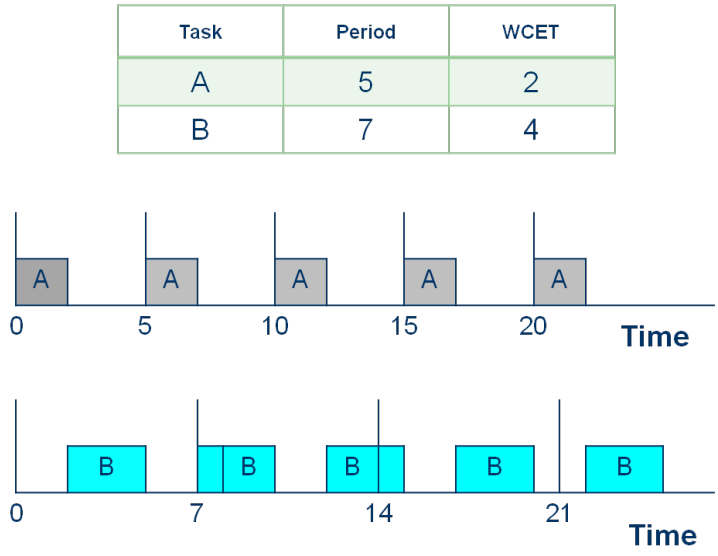


Figure 2.7 ETP with periodic tasks

Figure 2.7 provides an example of two tasks running on an ETP scheduler, which is also referred to as rate monotonic (RM) scheduling. It can be seen, at time 5 that task B was pre-empted, because task A has a shorter duration than task B.

Task	Arrival	WCET	Deadline
A	0	13	22
B	5	9	14

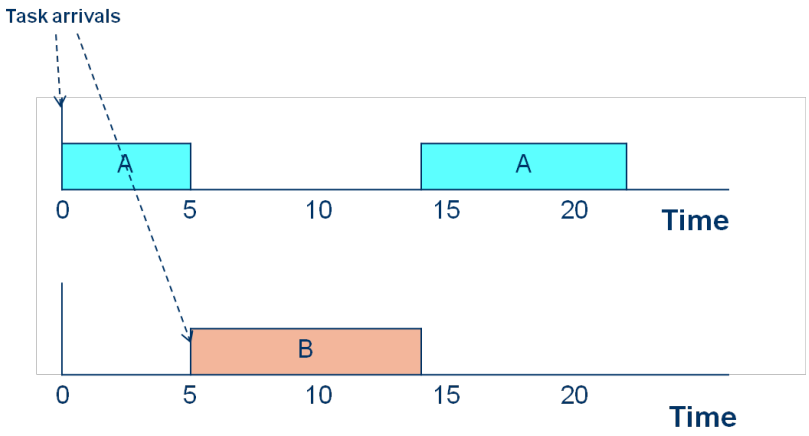


Figure 2.8 ETP with aperiodic task

In contrast, Figure 2.8 illustrates the ETP scheduler for aperiodic tasks, with the EDF scheduling policy applied. Task B arrives at time 5 and has an earlier deadline than task A, thus, in this example it pre-empts task A.

2.4.2 ET co-operative scheduling

The simplest approach to multitasking is to use a “co-operative” or “non pre-emptive” scheme; in other words, tasks should run to completion once they have started (Br©Þunl, 2006; Bertogna et al., 2011). However, when pre-emption is not permitted and tasks have dynamic activations, finding a feasible schedule becomes NP-hard problem. In addition, response times are usually longer (Jeffay et al., 1991b; Short, 2011). Nevertheless, there are a number of scheduling algorithms which are based on a co-operative approach to handle aperiodic tasks including: The First-Come-First-Serve (FCFS) algorithm, Shortest Process Next (SPN) algorithm (Labrosse, 2002) and Spring scheduling (Bletsas, 2007). The listed algorithms are examples of heuristic scheduling approaches, which aim to find a feasible schedule for those aperiodic tasks that contain

resource constraints or precedence constraints and co-operative properties (Buttazzo, 2005a).

Task	Arrival	WCET
A	0	14
B	5	6

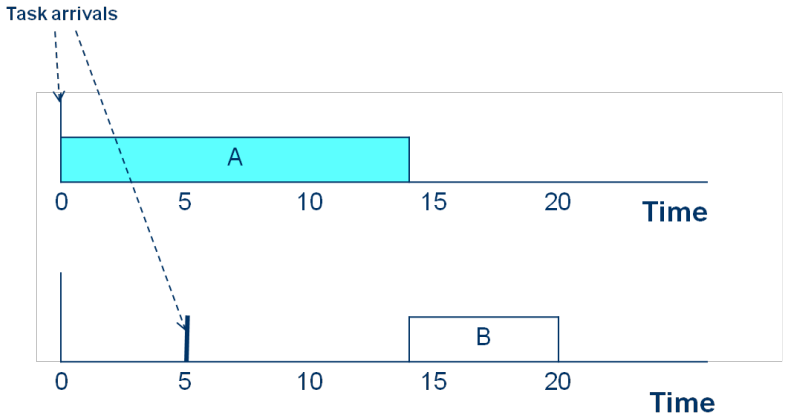


Figure 2.9 ETC with aperiodic task

Figure 2.9 exemplifies an ETC scheduling design. Task A begins execution and runs to completion. Although task B arrives at time 5 it does not pre-empt task A, since this design does not allow for other tasks to pre-empt the task that is currently running, until it completes.

2.4.3 TT co-operative scheduling

A TTC scheduler is characterised by a minor cycle (represented by the scheduler tick), and a major cycle (consisting of the amount of time required for the sequence in all the periodic tasks to be repeated). The minor cycle is typically implemented using a periodic timer interrupt, that is produced by an external timer overflow.

Task	Period	Execution time
A	14	2
B	7	3

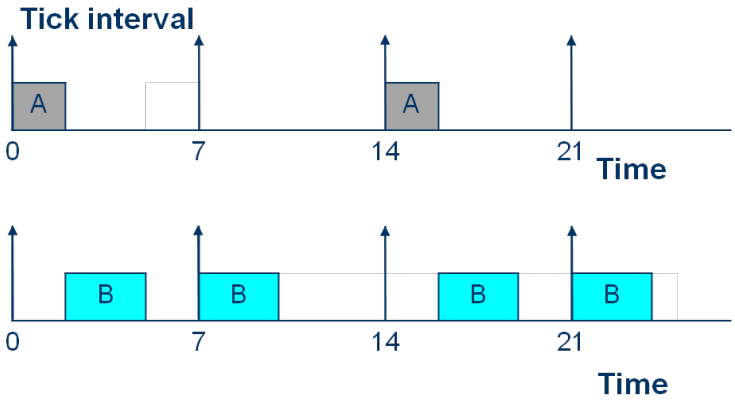


Figure 2.10 TTC with periodic tasks

The operation of the TTC architecture is illustrated schematically in Figure 2.10, which shows Tasks A and B run co-operatively. In a TTC design, all tasks must be completed within the system's tick interval. Therefore, it is crucial to determine the tasks' worst case execution time (WCET) and estimate this before employing them in a TTC scheduler. The worst case execution time (WCET) for every task must be known at the time of design. However, determining the WCET of a task is a problem which faces the developer of embedded real-time systems (Gendy, 2008).

A TTC scheduler potentially offers a highly predictable platform for embedded real-time systems. However, if a system has one or more tasks of WCET, e and also responds within an interval t to an external interrupt, on a situation where $t < (e +$ execution time of the task that handles the event), this means purely co-operative scheduling is not ideal. This scenario is known as task overrun, and will cause unpredictable behaviour in a system, making it unsafe for hard real-time systems.

2.4.4 TT pre-emptive scheduling

Other designs employing pre-emptive scheduling mechanisms for periodic tasks are TT pre-emptive architecture (TTP) (Short et al., 2008) and TT hybrid scheduling (TTH). In the case of a TTP design, fixed priority scheduling, which enforces a "one interrupt per CPU" rule is employed to provide information about the system's predictable behaviour. Task priorities are statically set to RM, DM or EDF scheduling algorithms. In fixed priority scheduling, priorities are assigned to tasks that are statically offline before they are placed in a priority queue (Locke, 1992). A task dispatcher is responsible for assigning the processor to the highest priority task. Any lower priority task executing at this time will be pre-empted and returned to the queue. An example of this scheduler is the rate monotonic type (RM). In contrast, in dynamic priority scheduling or dynamic scheduling, the priority of each task is dynamically assigned, and this can be changed at runtime. An example of this is the earliest deadline first (EDF) scheduler, in which the priority of each task depends on its absolute deadline (Buttazo, 2005a). The main advantage of this form of scheduling is its flexibility when adding a new task or modifying task characteristics. Nevertheless, without careful analysis of the scheduler and resource allocation techniques, there is the possibility that it can become exposed to priority inversion problems (Sha et al, 1990).

The TTH scheduler is also categorised as a pre-emptive scheduler; it allows the system designer to create a static schedule with a single pre-emptive task and a collection of co-operative tasks (as illustrated in Figure 2.12). A system employing a TTH scheduler can be as reliable as a TTC scheduler (Pont, 2001).

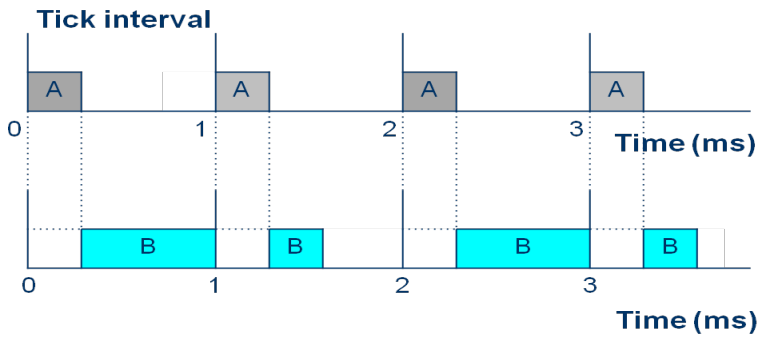


Figure 2.11 A simple TTH architecture

The TTH scheduler provides an attractive proposition as it allows for the creation of a scheduler with minimal resources (Maaita, 2008) when compared to other pre-emptive schedulers.

In Figure 2.12, the time-triggered pre-emptive operation of the rate-monotonic (RM) is described.

Task	Period	Execution time
A	7	2
B	14	7
C	21	5

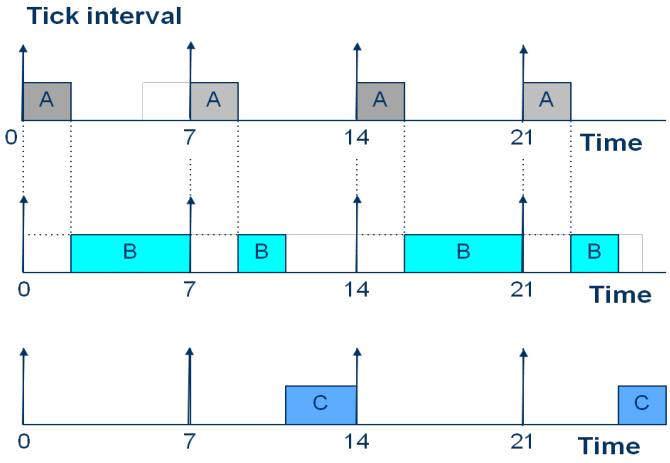


Figure 2.12 TTP with periodic task

In this example, the scheduler manages three tasks: Task C, which is the low priority task; Task B which is a medium priority task; and Task A which is a high priority task.

These tasks are assigned to periods and execution times as shown in Figure 6.6. Tasks A and C are ready for execution at time 7, and as task A has a higher priority than task B, it pre-empts task B and runs to completion, while task C is directed back to the task queue until the point that task B finishes its execution. It is then allowed to continue running from the interrupt point. At time 14, tasks A and B are ready for execution, thus, they will pre-empt task C. At this point, task A will pre-empt task C and run to completion, while task B is placed in the task queue. This process runs continuously throughout the lifetime of the system.

Many real-time applications require more flexible schedulers (with more than one pre-emptive task) than is provided by TTH schedulers; in this case it is tempting to use a fully pre-emptive scheduler. A TTP scheduler is designed as a fixed priority scheduler and can support multiple pre-emptive tasks. In a TTP scheduler implementation, priorities are assigned to tasks statically offline and then placed in a priority queue. A task dispatcher is responsible for assigning the processor to the task with the highest priority. Any lower priority task executing at the time will be pre-empted and returned to the queue. TTP schedulers provide the required flexibility and responsiveness, which is lacking with TTC schedulers and TTH schedulers.

There are many scheduling algorithms implemented in embedded real-time systems, each of which has its own strength and trade-offs. The main aspects that impact on the effectiveness and performance of scheduling algorithms are complexity, response time, feasibility and overheads (Labrosse, 2002). In this study, the research focused on comparing the effects of using pre-emptive and co-operative scheduling in a system. One way to observe this is by reviewing the scheduler's trends and related issues.

Alternatively, a developer may employ a conventional real-time operating system (RTOS) to support a event handling (Schneider, 2003, Labrosse, 2008, Stankovic and

Rajkumar, 2004). Whether an RTOS is used or not, the end result is the same: the system must be designed in such a way that events, which may occur at "random" points in time, and in various combinations, can always be handled correctly.

2.5 Real-time operating system

The use of a Real-Time Operating System (RTOS) assists the developers of real-time applications by providing basic system routines that inherently keep to timing requirements (Labrosse, 2008). Each RTOS has a real-time kernel that contains the main functionality of the RTOS, including task management, memory management and I/O management. To handle multi-interrupts in event-triggered real-time systems, schemes such as the round-robin, pre-emptive priority, or a combination of both, are utilised. These types of operating systems are readily available as commercial products and known as real-time operating systems (RTOS) (Stankovic and Rajkumar, 2004). Figure 2.13 is a block diagram of a kernel-based RTOS.

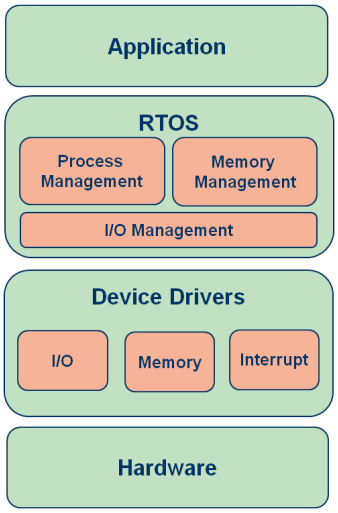


Figure 2.13 RTOS block diagram (Labrosse, 2004)

RTOS provides a number of mechanisms for communication and synchronisation amongst those tasks running on the same processor and using shared resources, including messaging, semaphores, mutexes, message queues and event flags. In order to

synchronise tasks for accessing shared data, mutual exclusion (mutex) techniques can be employed. Mutex allows shared memory to be locked up by the task accessing it, so that no other tasks can pre-empt it. Alternatively, semaphores can be used for task synchronisation. In a VxWorks operating systems, for example, three different semaphore mechanisms are provided: binary semaphores, mutual exclusion semaphores and counting semaphores (Labrosse, 2002). For inter-task communication, message passing algorithms are usually employed. Basically, messages can be sent to and from tasks using message queues.

There are various popular commercial RTOS, as well as open source RTOS in the market place, such as VxWorks, uC/OS II, Jbed and Linux. Some of these come with packages include a real-time kernel, an input/output manager, a file system, debuggers and cross-platform compilers. The main disadvantage when using a commercial RTOSs is that development and maintenance costs are very expensive (Labrosse, 2002).

At present, certified RTOS has become well-known in the embedded software world, as the requirements for software safety and reliability increase in embedded products, such as cars, trains, medical devices and airplanes. For example, the RTOS for IEC 61508 certifications is produced as an international standard for the functional safety of electrical/electronic/programmable electronic safety-related systems. Thus, every RTOS used in safety-critical applications must conform to specific standards, such as EN50128 (Railway), DO-178B (Aerospace) and IEC 601-1-4 (Medical Equipment). Based on the limited information available, such RTOSs are likely to cost more than 60,000 US dollars (Clegg, 2008).

This explains why some people prefer to use their own operating systems (rather than using certain real-time scheduling paradigms) or off-the-shelf commercial RTOS. Although there are a number of non-commercial RTOS in the market, these are

generally complicated and have a higher runtime overhead, both as regards memory usage and execution speed (Labrosse, 2002; Cooling, 2001).

Since highly-predictable system behaviour is a primary design requirement for many embedded systems, TT software architectures have become the focus of considerable attention (e.g. see Kopetz, 1997). In particular, it has been widely accepted that TT architectures are a good match for many safety-critical applications, since these can help to improve overall safety and reliability (Allworth, 1981; Storey, 1996; Bate and Burns, 2003; Obermaisser et al., 2005). For example, the Time-Triggered Group (TTG) established by Airbus, Audi, Delphi, Honeywell, PSA Peugeot Citroën, Renault and TTTech companies promotes cross-industry technologies for a TT solution in many safety-critical industries, including the aerospace, railway and automotive, where safety requirements must be satisfied at low cost (TTA-Group, 2007). In the automotive industry, for example, TT architectures have recently been accepted as a generic solution for highly dependable systems such as the X-by-Wire systems (see Ayavoo et al. (2007). The main reason why TT approaches are preferred in such applications is that they result in systems with very predictable and deterministic behaviour.

2.6 Use of WCET

Another aspect influenced by pre-emptive scheduling is the huge gap in terms of the WCET of a task. The concept of worst-case execution time for a program has for a long time been an element to consider in regards to real-time, especially in terms of the execution of schedulability analysis. Many scheduling algorithms and all schedulability analyses assume some form of prior knowledge about the worst-case timing of a task.

Figure 2.14 depicts several relevant properties of a real-time task. A task typically shows a certain variation in execution times, which is dependent on the input data or

different behaviour in the environment. The set of all execution times is shown in the upper curve. The shortest execution time is referred to as the best-case execution time (BCET); the longest execution time is referred to as the worst-case execution time (WCET). In most cases the state space is too large to exhaustively explore all possible executions and thereby determine the exact worst-case and best-case execution times.

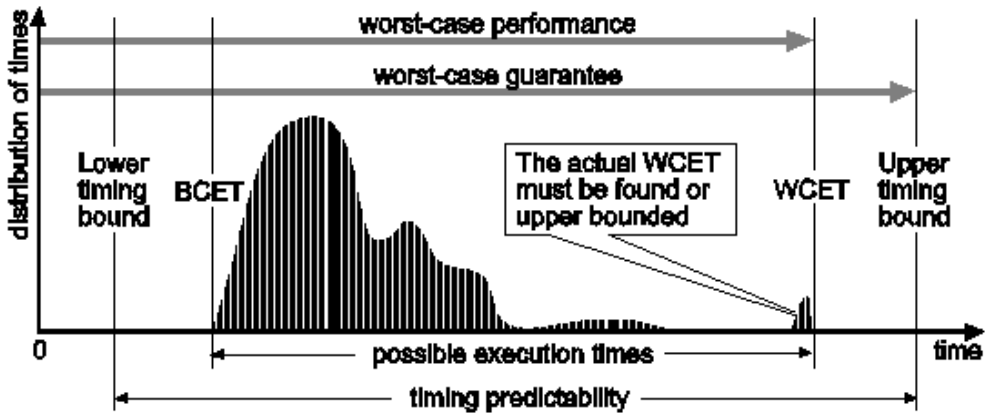


Figure 2.14 Basic notions concerning the timing analysis of systems (Wilhelm et al., 2008)

The determination of the upper bounds for execution times, commonly called Worst-Case Execution Times (WCETs), is a necessary step in the development and validation process when designing hard real-time systems. Knowing worst-case execution times is critical to the success of schedulability analysis in hard real-time systems.

In hard real-time systems, the WCET is estimated and analysed to ensure that the system will not miss its deadline. WCET estimates can be used to verify that the response time for a critical piece of code is short enough that the interrupt handlers finish quickly, or that the sample rate of a control loop can be retained (Wilhelm et al., 2008). Two main criteria for producing WCET estimates are required:

- actual WCET = < $WCET_{EST}$
- $WCET_{EST}$ – actual WCET → Minimal

Firstly, the WCET estimates must be equal to, or slightly greater than, the actual WCET of tasks. Secondly, the bounds or estimates of WCET should be close to exact values. Underestimating the WCET is not safe when using hard real-time systems, as this may cause deadlines to be missed in practice. On the other hand, pessimistic estimates of execution times may lower the utilisation of resources. However, advanced features in modern processors, such as caching and pipelining, complicate timing analysis. Much work has been performed to analyse cache behaviour in a single-task system, in order to predict the timing properties of the system. Although single task-based timing analysis can assist in the acquisition of useful insights about the timing properties of tasks, many of the factors in a multitasking system are not taken into consideration, which definitely affects the accuracy of timing estimates. In a pre-emptive multitasking system, timing analysis becomes even more difficult because of the unpredictability of pre-emption, the interaction among tasks, such as intertask cache evictions, and the underlying scheduling algorithms (Schneider, 2003).

A number of methods for computing WCETs have been published (Kim et al., 1996; Burns and Wellings, 1995; Bernat et al., 2002). To predict WCETs accurately, WCET analysis should not only consider the schedulability analysis for tasks like interrupts, context switch times, sporadic tasks, system clock and release jitter; microarchitecture-related pre-emption costs resulting from pre-emption also need to be considered (Schneider, 2003). Moreover, the majority of these problems are related to dynamic runtime behaviours. For example, most loop bounds were determined by system parameters during the runtime of the scheduler, and dynamic function calls like function pointers are also called at runtime. The presence of unpredictable issues has resulted in poor WCET estimation, making WCET analysis much more difficult. In more recent studies, they noted that the WCET of a task can be affected by fully pre-

emptive scheduling up to 40% (Yao et al., 2011) at runtime, This may cause poor WCET estimation and result in unsafe real-time systems. Therefore, in order to reduce the runtime overheads, current researchers (Bertogna et al., 2010; Buttazzo and Kuo, 2009; Min-Allah et al., 2007) have aimed to reduce the number of pre-emptive tasks in the system, as will be discussed in Section 2.8.

2.7 Evolution of pre-emptive scheduling on uniprocessor systems

Real-time schedulers have been studied since the late 1960s and early 1970s. Initial designs were based on pre-emptive scheduling; these evolved over time, with added functional and performance requirements added to improve the quality of Liu and Layland's low scheduling bound. Updates improved average response times for soft deadlines aperiodic tasks and also improved the NP-hard solution for finding the schedulability of a set of periodic tasks involving resource sharing and improved scheduling overhead for dynamic pre-emptive scheduling (Spuri and Buttazzo, 1996). The current research trends for pre-emptive scheduling are illustrated in Figure 2.15 and will be deliberated upon in this section.

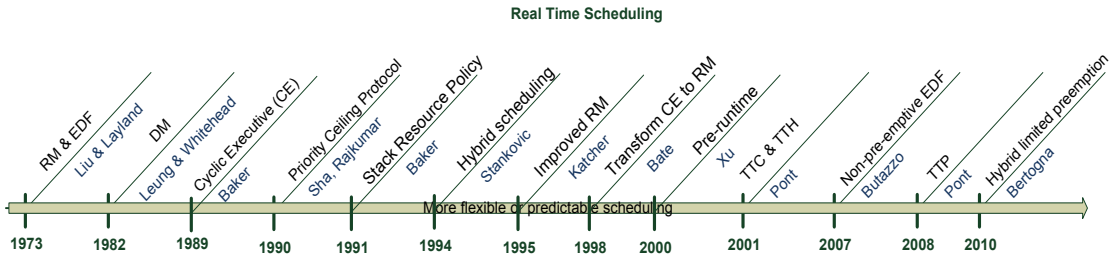


Figure 2.15 Trend of real-time scheduling

One of the earliest and most comprehensively studied schedulers, is the Rate Monotonic approach, which was proposed, discussed and evaluated by Liu and Layland (1973). This tool was designed to handle cases of fixed priority and pre-emptive scheduling, for periodic independent tasks. In the RM scheduling algorithm, task

priorities were set according to request rates i.e. tasks with higher request rates were assigned higher priorities. It was claimed to be an optimal[1] approach amongst those proposed for fixed priority assignments, as stated in theorem 2 by Liu and Layland (1973) *"If a feasible priority assignment exists for some task set, the RM priority assignment is feasible for that task set."*

However, in certain circumstances, the RM approach may lead to a time-overflow affecting lower priority tasks, as indicated by Locke (1992), thus: *"A task with the higher priority (has the shortest periods) could pre-empt the lower priority task and cause the start and completion time of a task maybe delayed arbitrarily. Consequently, the schedulability of the tasks set cannot be guaranteed under RM algorithm."*

An extension to the RM was made with the Deadline Monotonic (DM) approach introduced by Leung and Whitehead in 1982; wherein periodic tasks can have a relative deadline that is less than the duration of their period. DM takes static priority assignments and assumes pre-emptive task scheduling. Tasks with shorter deadlines will then be assigned higher priorities. A running task could be pre-empted by a newly arrived task with a shorter relative deadline. This approach is optimal for tasks with deadlines that are unequal to their periods.

A further development on the basis of the deadline based approach, which can be used when scheduling a set of aperiodic tasks on a single processor, is known as the Earliest Due Date (EDD) or Jackson's rule. The rule states (Buttazzo, 2005a) that;

"Given a set of n independent task, any algorithm that executes the tasks on order of non-decreasing deadlines is optimal with respect to minimising the maximum lateness."

[1] The optimal schedule is the scheduling algorithm that can minimise some given cost function

However, this algorithm fails to encompass pre-emptive issue, since all the tasks arrive simultaneously. In contrast, Horn's algorithm considers pre-emptive issues; he points out that in practice all tasks are not synchronous but rather have arbitrary arrival times; therefore, he recommended employing an Earliest Deadlines First (EDF) approach to establish priorities. He was accustomed to setting the priorities of tasks prior to running them, but retained the provision to reschedule priorities during the runtime. According to this algorithm, tasks with earlier deadlines can be executed with higher priorities. Buttazzo (2005) comments on EDF that;

"The EDF algorithm is a dynamic scheduling rule that selects tasks according to their absolute deadlines. It is intrinsically pre-emptive: the currently executing task is pre-empted whenever another periodic instance with earlier deadline becomes active. EDF is the optimal algorithm for dynamic pre-emptive scheduling for scheduling periodic as well as aperiodic tasks."

In real-time system applications, the system always requires a combination of periodic and aperiodic tasks. For instance, in A-7e aircraft, the operational flight program comprises 75 periodic and 172 aperiodic tasks (Stankovic et al, 1994). One of the simplest methods for implementing mixed task scheduling is to apply standard scheduling algorithms. For example, RM is used for handling periodic tasks, whereas First Come First Serve (FCFS) can be used for aperiodic tasks. This approach has been expanded through the use of a server; as with the Polling Server (PS) algorithm (1989), the Deferrable Server (DS) algorithm (Lechozky et al, 1987) and the Priority Exchange (PE) algorithm (Lechozky, 1987). Butazzo compared these algorithms in terms of their performance, computational complexity, memory requirement and the complexity of implementation (Butazzo, 2008). This comparison led to the conclusion that PS and DS algorithms have an excellent performance, although a greater computational

complexity, memory requirement and implementation complexity when compared with the PE algorithm.

Although the pre-emptive approach have been used to address tasks that demand varying workloads, it suffers from issues that arise, such as those which ensure higher priority tasks cannot be blocked when waiting for lower priority tasks to execute, a problem referred to as the ‘priority inversion phenomenon’. This relates to the problem of scheduling tasks when accessing shared resources. Several approaches have been proposed to address this problem, including avoiding pre-emption during the execution of all critical sections (Butazzo, 2005) and using a priority inheritance protocol (Sha et al., 1990), a priority ceiling protocol (Sha et al, 1990) or a stack resource policy (Baker, 1991). For example, in reference to a priority ceiling protocol, Sha suggested that resources should be protected using semaphore in order to schedule periodic tasks to deliver exclusive access to a common resource (Sha, 1990). However, the majority of these protocols incur a considerable overheads and are difficult to implement. For instance, each time a shared resource is acquired, the acquiring task must be hosted to the resource's priority ceiling. Conversely, every time a shared resource is released, the hosted task's priority must be lowered and returned to its original level. All this extra coding takes time. In more recent work by Short and Pont (2008), a simpler form of pre-emptive scheduling was proposed to support priority inheritance protocols, as well as a variety of fixed pre-emptive tasks and co-operative tasks; this is known as the time-triggered protocol (TTP).

Methods proposed by (Baruah and Chakraborty, 2006), Butazzo (2009), Bertogna (2010), Min-alah (2011) and Yao and Butazzo (2011) are focused on dynamic priority scheduling, and these all have limited pre-emption, in order to reduce runtime overhead. The floating non-pre-emptive region (NPR) and the fixed pre-emption point

(FPP) models are considered when designing limited pre-emptive scheduling (Yao, 2011). In FPP, each task is divided into subtasks. A higher priority task can be permitted to pre-empt lower priority tasks, according to predefined pre-emption points. However, each task must consider a number of NPRs of maximum length. NPRs are floated in the task code. Baruah (2005) computed the longest NPR for each task under EDF. Yao considered both FPP and NPR in their analysis, and Yao (2011) noted that scheduling with fully pre-emptive scheduling may affect the WCET of a task by up to 40%. To address this problem, Yao proposed the feasibility test of a task set with limited pre-emption under fixed priority scheduling. Bertogna (2010) introduced a hybrid limited-pre-emption real-time scheduling algorithm designed to result in a low runtime overhead, while scheduling all systems that can be scheduled using fully pre-emptive algorithms. A method that Bertogna (2010) proposed involves selecting pre-emption points, under the assumption of fixed pre-emption costs at each pre-emption point. Since research in this area is relatively new and has been analysed only using experimental simulation-based approach, the results may not necessarily be accurate. Thus, more analysis and work on the relevant hardware is required.

Most of the researchers in this area are working in a fully pre-emptive environment. As the researchers aim to reduce runtime overheads, limiting pre-emption real-time scheduling becomes the point of innovation in the scheduling field. However, this is not new. In 2001, Pont introduced a time-triggered hybrid (TTH) scheduler, which supports a limited degree of pre-emption in a fully co-operative environment. This became one of the research interests of this study, focusing on analysing its effects on overheads, implementation costs and effort, as compared to pre-emptive scheduling and co-operative scheduling.

2.8 Evolution of co-operative scheduling on uniprocessor systems

One of the most popular co-operative scheduling approaches to handling periodic tasks is cyclic executive scheduling or timeline scheduling (Locke, 1992, Laplante and John Wiley & Sons., 2004). This is simple and provides a highly predictable schedule, which means that a task is guaranteed to run to completion once it is started (Br©Þunl, 2006, Bertogna et al., 2011). A very simple cyclic executive implementation uses the Super Loop and delay functions (Kurian and Pont, 2007). The main advantage of this scheduler is its simple implementation and small requirements in terms of resources. Its main drawback is that the period length of tasks is not fixed, which can increase task jitter (Bate, 1997). Alternatively, TTC addressed these issues using a hardware timer set to generate interrupts on a periodic basis (known as 'tick interval') (Pont, 2001). Tasks will be invoked following every scheduler tick. This provides more a predictable system with very low levels of jitter (Locke, 1992),

Under a static and co-operative scheduling scheme, all periodic tasks are scheduled offline and will be executed at a fixed time, known as the time interval. The scheduler requires a timer to synchronise the activation of the tasks at the beginning of each time interval (Buttazzo, 2005a, Buttazzo, 2005b). Nevertheless, within an online scheduling scheme, there are a number of scheduling algorithms based on a co-operative approach that are used to handle aperiodic tasks, as discussed by Butazzo (2005), including: First-Come-First-Serve (FCFS) algorithm, Shortest Process Next (SPN) algorithm (Labrosse, 2002) and Spring scheduling (Stankovic and Ramamritham, 1989). However, assessing the feasibility of a set of tasks for scheduling becomes a NP-hard problem when task arrival times are unknown.

In 1990, Jeffay proved that the co-operative scheduling of concrete periodic tasks (where release times of the tasks are known) is NP-Hard in the strong sense when an

inserted idle time is disallowed. With this restriction, EDF is still optimal in co-operative scenarios (Jeffay, 1991). However, if the restriction is released, EDF is then not optimal for co-operative scheduling (Buttazzo, 2005a). In more recent work, Short (2012) provides a comprehensive study and comparison of co-operative scheduling, following an earlier suggestion (Short, 2011) that co-operative scheduling of the EDF algorithm with idle time implies polynomial complexity. However, the response jitter of a task is potentially high. Alternatively, the time-triggered co-operative scheduling algorithm (TTC), which is of interest in this study (Pont, 2001, 2007, 2008, 2009), can be characterised as having highly predictable timing behaviour, with a very low level of task jitter (Gendy, 2008, Short, 2012).

2.9 Conclusion

This chapter has provided a comparative study of existing work on pre-emptive and co-operative schedulers; as well as ET and TT design for uniprocessor systems. Schedulers have been reviewed as these have evolved over time. The drawbacks of pre-emptive scheduling have been discussed in light of the literature reviewed. It has been observed that current and future trends are shifting towards a TT architecture, specifically to one incorporating limited pre-emptive scheduling such as TTH scheduling. Thus, cross-scheduler evaluation is required to compare the costs involved in designing, implementing and testing fully pre-emptive, limited pre-emptive and co-operative scheduling. Furthermore, it was found, that there is a need to conduct more cross-architecture evaluations to highlight the different implications arising from ET and TT architecture. In this study, the focus is on the costs involved in testing and verifying systems running such architecture. The remainder of the thesis will review literature on how the cost of testing can be evaluated in ET and TT architecture at the design, implementation and testing phases.

Chapter 3
Testing TT and ET Software Architecture

3.1 Introduction

Having discussed the theory of embedded software architecture and the evolution of pre-emptive and co-operative schedulers, this chapter explores the implications of different software architecture when testing real-time systems. This is because software architecture is the most significant activity, which affects software development costs tremendously. This chapter also describes a review of the relevant literature for verifying and testing ET and TT systems.

3.2 Validation, verification and testing terminology

Prior to reviewing and analysing the results from previous work in this area, it is essential to mention the confusion that has surrounded the use of the terms “testing”, “validation” and “verification” amongst those working on the evaluation of software systems (Thane, 2000).

Validation: is the process of evaluating the correctness of a final program with respect to its specifications.

Verification: is confirmation by examination and provision of objective evidence that a system meets the set requirements, according to IEC61508 (the international standard for electrical, electronic and programmable electronic safety-related systems). This means that verification is performed at the end of each development phase. In terms of cost and time, the verification phase is the most expensive when systems are being developed to safety critical standards.

Testing: requires execution of the system (dynamic verification) while supplying it with predicted and observed inputs and outputs for the purpose of fault finding or deviation from requirements.

According to ISO 26262 - an automotive industry specific functional safety standard - verification implies the following requirement:

In the test phases, verification is the evaluation of the work products within a test environment to ensure that they comply with their requirements. The tests are planned, specified, executed, evaluated and documented in a systematic manner.

The goal of testing is to verify whether a specific input will yield a specific output, as defined by the specification. The alternative is to enhance trust in the system. However, testing is only capable of detecting the presence of errors not confirming their absence. Thus, it cannot ever be conclusively established that all errors have been detected.

3.3 Overview of testing

Verification of embedded real-time systems is one of the most complex and time-consuming activities that takes place during the development of real-time systems. For example, for the Boeing 777 aircraft, 50% of overall software-development efforts and budgeting are spent on the areas of analysis and testing (Burns and McDermid, 1994). Expensive software engineering techniques that are not cost-effective when applied to non-critical systems may on occasion be used for critical systems development.

For small applications, embedded system design generally emphasises implementing the desired functionality in the least costly way possible in order to achieve a “good enough” design within the shortest possible timeframe and using minimal resources (Bletsas, 2007). In addition, because testing represents a major proportion of the effort involved in the development of real-time systems (see Figure 3.1), it is sensible to

adopt design techniques that can simplify the testing process. Choice of architecture is one of the aspects of the design that has an impact on the satisfaction of both real-time constraints and constraints imposed by the embedded nature of the application. However, individual designers can further adapt any such system configuration to meet their own timing constraints, through appropriate scheduling of computation.

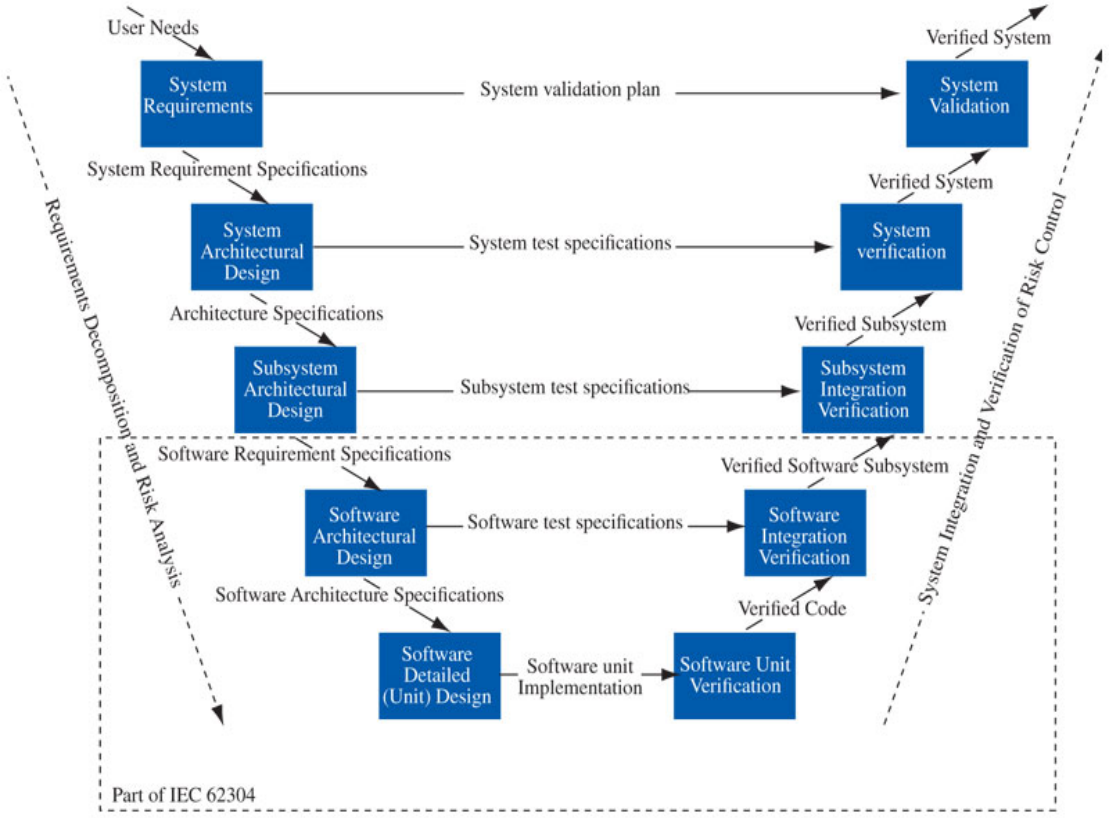


Figure 3.1 'V' lifecycle model for safety-related systems (Storey, 1996)

Both validation and verification (which is commonly referred to as the V&V process) are required in the evaluation of any software system, to ensure that the whole software product fulfils the system requirements and operates according to the user's specifications.

The development of real-time and safety-related systems usually begins with establishing functional requirements, supplemented by temporal requirements, often also called "timing constraints". For example, the requirements of a system can take on

the form of set protocols, such as “the aircraft must process accelerometer data for every 10 milliseconds” or “the gate needs at least 15 seconds to lower itself to the down position when the train is crossing”. The rationale for setting requirements is to convey to the design team, function and implementation for both design and verification purposes.

Hazard and risk analysis will then be produced to provide information about potential dangers in the embedded system as related to safety-critical systems. If a result is incorrect or arrives too late, then the real-time system has failed. The potential consequences of such a failure depend upon the characteristics of the real-time system being developed.

With respect to the functional and timing correctness of real-time systems, Schütz (1993) identified six basic requirements for testing such systems, which are: Organisations, Observability, Reproducibility, Host/Target Approach (Controllability), Environment Simulation and Representativity (Test Coverage). Many general software development methodologies are also combined with suitable test methodologies. These test methodologies should be organised into separate test phases, so that the design phases are compatible with the system structure.

Observability represents the ease of determining whether specified inputs affect outputs, while controllability represents the ease of producing a specified output from a specified input. For its part, reproducibility holds a two-fold functionality; first, to ensure that any errors have been accurately corrected and secondly, to guarantee that any modification will not introduce unnecessary errors. Nevertheless, there are two main factors that rule out the reproduction of an error; tight timing restrictions and the non-deterministic behaviour of real-time software systems. Both factors make the

testing of real-time systems more difficult than that of non real-time systems (Tsai, 1990).

Furthermore, due to the lack of resources, such as power consumption and memory size, typically most embedded real-time systems utilise separate computer systems to run their real-time software and testing systems. In contrast, in a desktop system environment, a similar machine is used for testing and running the program itself.

Despite testing the target system in its actual environment (which sometimes is not permissible due to safety and cost issues), a highly interactive software target simulator with simulated environment could feasibly assist this testing strategy. However, modelling a real environment into a simulation-based system is more complicated when testing complex real-time systems. Designers need to design the system carefully, so that the results produced by the simulator are precisely similar to those in the real environment, particularly the system's temporal behaviour. In reality, simulations are typically slower than in the actual environment (Marwedel, 2003). Therefore, simulation is not an ideal technique for accurately detecting timing problems. However, it can still be used as one of the test phases in the development methodology (Schütz, 1993).

Finally, testing approaches are also determined by the properties of the underlying software architecture (Kopetz, 1998). The software architecture and scheduling models chosen for a system may reflect written requirements. For example, Katcher (1993) described the impact on the requirements of event-driven and time-driven fixed priority-based scheduling. In time-driven scheduling, designers have to deal with a scheduler with periodic timer interrupts. Thus, the design of the tasks will be based on a periodic timer that interrupts execution and invokes the scheduler. On the contrary, tasks in event-driven scheduling will be initiated by external interrupts associated with

the software task priorities. Thus, designers need to identify what scheduling policy they have to use to handle the pre-emption of a higher priority task over an active task; for instance, by using RM or EDF scheduling approaches.

3.4 The influence of software architecture on testing

Various studies, such as those of Kopetz (1991), Schütz (1993), Thane and Hansson (2001) and Lindström et al. (2008) provide a comparative foundation for testing using TT and ET architecture.

The main issue raised was the need to consider a higher number of possible execution scenarios when testing an ET design. Schütz (1993) analysed both architectures by comparing the upper boundary for the number of possible control paths (CP) executed in response to the observation points of the same time interval, using TT systems Equation (3-1) and (3-2) ET systems.

$$\boldsymbol{CP_T(n) = \sum_{k=0}^{n} \binom{n}{k}} \tag{3-1}$$

$$\boldsymbol{CP_E(n) = \sum_{k=0}^{n} \binom{n}{k} k!} \tag{3-2}$$

Where *n* is an independent input and *k* is the number of observation points. In a TT system, the value of *k* is two, since it can only detect the presence or absence of the input states at the end of the tick interval. Thus, the formula for Equation (**3-1**) can be simplified to $CP_T(n) = 2^n$.

The results also suggest that testing should cover a much larger number of input spaces in the system environment when using an ET system rather than a TT system. In fact, the test effort for ET systems grows exponentially with the number of observation

points, which impacts on the number of observation points, and consequently the selection of test cases becomes more complicated, since the same test may involve different behaviours in different executions.

Although the model seems to represent a comparison of the testing effort between the ET and TT architecture perfectly, there are some important aspects that have been overlooked, i.e. the effects of pre-emption costs and the information of order of events. These elements are important as they affect testing effort. For example, if the information on order of events is needed by an application which running on a TT architecture, this may increase the number of observational states (Schütz, 1993).

Thus, Schütz (1993) re-modelled Equation 3-1 by adding subinterval elements, with respect to the arrival of events between two uninterrupted points. The results show that test effort for TT architecture is actually greater than for the previous model. This implies that the simpler model is not precise enough to be used to assess the test effort for TT architectures.

On the other hand, the model (Equation 3-2) only represents the lower boundaries of CP_E. In fact, ET systems have arbitrary points that can be observed in reference to newly arrived input, leading to an explosion in the combinations and consequently the numbers of possible CPs.

It is therefore deduced, that to cover all the aspects that might affect the testing of an ET and TT architecture, using the above testing model, is relatively difficult. As a result, a different form of comparison is needed to distinguish the testing effort required for testing an ET and TT architecture. This should include as many of the aspects as possible that might affect the testing effort. This approach should also provide more accurate analysis when evaluating the testing effort for both software architectures.

In more recent work, Lindstrom (2008) considered pre-emption points and a maximum number of concurrently executing tasks, to observe their effects on testability. In view of the Lindstrom's findings, the number of execution orders increases exponentially when the maximum number of pre-emptions increases. This means that the designated pre-emption points also have a great impact on testability in ET systems (Lindström et al., 2008). This work represents the impact of software architecture on the cost of testing. In reality, there are many aspects that need to be considered, such as implementation costs, as well as the effects of synchronisation mechanisms, as described in the previous chapter.

In spite of the disadvantages of an ET design, for low and average load conditions, the resource utilisation in an ET design is much better than with a TT design. However, this is not the case in peak load scenarios, where the time available for execution is reduced by increasing the processing time for interrupt handling, synchronisation and scheduling algorithms (Kopetz, 1997). In this case, a TT design may perform better than an ET design.

As noted previously, Schütz (1993) discussed the influence of software architecture in testing real-time systems based on six basic elements: Observability, Reproducibility, Representativity (Test Coverage), Host/Target Approach (Controllability), Organisations and Environment Simulation, as mentioned in section 4.3. He used a rolling ball experiment to present his work and demonstrated that TT architecture outperforms ET architecture according to the first three testing requirements.

Recently, Lindstrom (2008) reconsidered these issues (with respect to pre-emption, observations and process instances) for testing ET systems as well as in TT systems. A metric of timeliness testability was used in the study of execution environment constraints and their impact on testability. Due to elements that are uncontrollable in ET

systems, behaviour with respect to the timing and execution order is less predictable in a dynamic real-time system (ET systems) than in a corresponding static real-time system (TT systems). This partly explains why it is difficult to test ET systems. The number of potential execution orders were previously proposed as a testability metric by Thane (2000). The number of execution orders was chosen as a testability metric in Lindstrom's study (Lindström et al., 2008). This metric is in line with previous work on the testability of real-time systems and assigns the highest level of testability to the TT design (Schütz, 1993; Thane and Hansson, 2001; Lindström et al., 2008).

Moreover, based on Thane and Tsai's studies (Thane and Hansson, 2001; Tsai et al., 1990b), deterministic replay and reproducibility are crucial elements in assisting the debugging and testing of real-time systems. However, it is complicated to reproduce an identical behaviour when stimulated with the same test case in an ET design, since the event involves non-deterministic behaviour. This also implies that it is necessary to employ a software architecture with natural partitions in the temporal domain, which can provide deterministic behaviour in the system.

Testability for dependable and predictable real-time systems is determined by the properties of fundamental system architecture (Kopetz, 1991, Schütz, 1994, Linstrom, 2008). Therefore, a test methodology must take advantage of these properties to produce a system that is as easy to test as possible. In addition, Kopetz (1995) and Xu (2003) also claimed that TT software architecture can make timing verification for large real-time systems easier. Xu (2003) noted that designers have to provide an a priori guarantee that all timing constraints will be satisfied for a TT architecture based on the static scheduling method. This can be achieved by identifying all the critical sections of those programs that access shared resources, then computing an offline schedule for all instances of the entire set of periodic tasks (Xu, 2007, Xu and Parnas, 1993, Xu, 2003).

Although, more effort is required to ensure that all different possible overload scenarios and the worst-case overheads are covered in the schedule design (Xu, 2007), testing will become easier for such systems, as noted:

The pre-runtime scheduling approach effectively reduces the number of the possible cases of the actual code's timing behaviours by structuring real-time software as a set of cooperating sequential processes and imposing strong restrictions on the interactions between the tasks. This makes it easier to inspect and verify all the timing behaviours of the software.

Unlike those preparing TT designs, most ET system designers do not need to emphasise the verification of timeliness at the design stage; the main challenge they face is to ensure that the assignment of task attributes and timing requirements can be imposed on the scheduler during the run time (Liu and Layland, 1973; Sha et al., 1990), (Tsai and Bi, 1991).

Despite providing the advantages of TT architecture for testing real-time systems, Bate (1998) and Scheler and Schroeder-Preikschat (2006) revealed contradictory views on this issue. Bate (1998) pointed out in his thesis that a pre-emptive fixed priority structure makes a system easier to verify in comparison to TT architecture, which is based on cyclic executive scheduling. This is because a fixed priority scheduler can be verified using analysis, whereas a cyclic scheduler is verified through testing. Analysis can be performed over a short time with minimum effort. On the other hand, testing is considered as a much more expensive activity than analysis. Furthermore, Scheler claims that neither TT systems nor ET systems are to be preferred with respect to testability (Scheler and Schroeder-Preikschat, 2006). According to Scheler, as the timing constraints of TT and ET systems are both verified using formal techniques, such as a constructive schedulability test and response-time analysis, testing with

typical load scenarios is not sufficient, when hard deadlines have to be kept. Thus, both architectures do not provide any testing benefits.

Since there is still a lack of confidence over determining which software architecture can provide less testing efforts, an assessment to observe the impact of software architecture on cost of testing is vital.

3.5 The influence of scheduling policy on testing

The principal difference between pre-emptive and co-operative scheduling is that task execution can be pre-empted at any time by the release of a higher priority task. This leads to a greatly increased number of possible program paths, making functional testing is more difficult to achieve. More importantly, data flows and updates could be interrupted, causing a task to be pre-empted when a data calculation is only partly finished. If data in a transient state is used then the effect could be difficult to determine.

Furthermore, Pont (2001, 2008) noted that one of the advantages of employing co-operative approaches is that a system can then be tested simply. For instance, isolation of the tasks (for testing purposes) is difficult to achieve in pre-emptive scheduling due to impacts of scheduling overheads (Katcher, 1993). Unlike pre-emptive scheduling, tasks running in co-operative scheduling can be tested completely in isolation (Pont, 2001). This simplifies testing by allowing various tasks to be investigated separately.

Moreover, testing represents a major proportion of the effort involved in the development of a safety critical system, it is wise to adopt structured and design techniques that simplify the testing process. For example, Storey (1996) mentions that:

Simple systems are easier to test. Thus, every attempt should be made to reduce the complexity of the hardware, software and data structures. A reduction in complexity

also offers advantages in other areas such as reliability and the cost of implementations.

Verification and testing are necessary for each phase in the software development processes to ensure all requirements are met. Much literature exists describing studies investigating the cost of testing and verification on the software engineering life-cycle in detail. Lifecycle stages relevant to evaluating the cost on embedded software architecture and scheduling of real-time systems are considered as testing and verification: at the design phase; implementation phase; and testing phase.

3.6 Schedulability test

Many different types of scheduler design were discussed in Chapter 2. From the real-time scheduling design perspective, justifying and demonstrating how requirements are met can also affect the cost of design. Schedulability testing, or schedulability analysis is usually employed to ensure whether all tasks are schedulable or not for particular schedulers. This section explains the evolution of those schedulability testing techniques used with ET and TT designs.

Schedulability testing is one of the well-known verification forms needed to increase the degree of confidence in a system's timing properties at the design stage. Although, over 100 papers have discussed how to improve on accuracy and performance (Zhang et al., 2010, Zhang and Burns, 2009, Bini et al., 2003, Bini and Buttazzo, 2004, Davis et al., 2008, Tindell et al., 1994), there remain no accurate mathematical models for schedulability testing.

Several schedulability test performances have been applied in order to evaluate scheduling mechanisms; for instance utilisation-based on Liu and Layland's analysis (Liu and Layland, 1973) and the Hyperbolic Bound (Bini et al, 2003) analysis for Rate

Monotonic (RM) scheduling, and Response Time analysis (RTA) (Audsley et al., 1993) for Fixed Priority Pre-emptive Scheduling.

The problems with studying the efficiency of multi-task scheduling on single processors started in 1967, when Fineberg and Serlin (Fineberg and Serlin, 1967) studied the problem relative to two tasks. As the complexity of the systems evolved, more tasks were serviced by the systems. Comprehensive work introducing schedulability test techniques was carried out and published by (Liu and Layland, 1973).

Liu and Layland (Liu and Layland, 1973) studied the problem of scheduling periodic tasks under the Rate Monotonic (RM) and Earliest Deadline First (EDF) scheduling policy, and developed corresponding schedulability tests for single-processors with priority driven pre-emptive schedulers. In the case of fixed priority sets of periodic tasks using an RM scheduler, a sufficient but not necessary condition for schedulability testing was introduced. In addition, Liu and Layland (1973) expanded their work to incorporate the dynamic assignment of priorities to a set of pre-emptive periodic tasks. Liu and Layland (1973) also proposed sufficient and necessary condition for dynamic scheduler EDF.

Liu and Layland (1973) developed their analysis from small to arbitrarily large set of inputs, making the following assumptions about the environment:

- *A 1. "The requests for all tasks for which hard deadlines exist are periodic, with constant interval between requests."*
- *A 2. "Deadlines consist of run-ability constraints only, i.e. each task must be completed before the next request for it occurs."*
- *A 3. "The tasks are independent in that requests for a certain task do not depend on the initiation or the completion of requests for other tasks."*

- *A 4. "Run-time for each task is constant for that task and does not vary with time. Run-time here refers to the time which is taken by a processor to execute the task without interruption."*
- *A 5. "Any non periodic tasks in the system are special; they are initialisation or failure-recovery routines; they displace periodic tasks while they themselves are being run, and do not themselves have hard, critical deadlines."*

Further multiple improvements have been carried out by researchers, utilising certain conditions about assumptions. For example, Leung (1982) relaxed assumption A3 whereas Kather (1993) relaxed assumption A4 while conducting their analysis.

Liu and Layland (1973) defined the (processor) utilisation factor, as the fraction of the processor time used, for n tasks, with worst case execution times C_i, and period of tasks T_i, given in equation (3-3):

$$\boldsymbol{U = \sum_{i=1}^{n} (C_i/T_i)} \tag{3-3}$$

For the case of RM, a task set will not miss any deadline if it meets the following sufficient conditions, but this is not necessary, (least upper bound) (Buttazzo, 2005a, Liu and Layland, 1973):

$$\boldsymbol{U \leq n(2^{1/n} - 1)} \tag{3-4}$$

For high values of *n*, the least upper bound converges to (Buttazzo, 2005b),

$$\boldsymbol{\lim_{n\to\infty} U = ln2 \approx 0.69} \tag{3-5}$$

However, in the case of EDF, the least upper bound is given as $U \leq 1$ (Liu and Layland, 1973), therefore, tasks may utilise the processor up to 100% and remain schedulable.

In 1986, Joseph and Pandya introduced an exact schedulability condition for fixed priority scheduling: Response Time Analysis (RTA). The utilisation-based tests introduced by Liu and Layland (1973), have two significant drawbacks identified by (Joseph and Pandya, 1986); i.e. they are (a) not exact, and (b) not precisely applicable to more general process models. Unlike the utilisation based tests for fixed priority scheduling, RTA has the advantage of not only being sufficient, but also necessary; i.e. if the process set passes the test they will meet all their deadlines; if they fail the test, then, at run-time, a process will miss its deadline, unless the computation time estimates, C , prove to be pessimistic.

Leung (1982) relaxed the A3 assumption in order to provide a more flexible process model, which could be adopted to handle tasks with jitter constraints or activities with short response times comparative to their periods. Hence, Leung introduced an algorithm as an extension of RM, where tasks can have relative deadlines shorter than their period. Specifically, each periodic task is characterised by 4 parameters: Phase Φ_i, WCET C_i, deadline, D_i and period, T_i. Leung and Whitehead then generalised the results of Liu and Layland, proving that the Deadline Monotonic (DM) algorithm is optimal for a fixed priority scheduling model.

In order to find a sufficient and necessary schedulability test for DM, the exact interleaving of higher priority tasks must be evaluated for each process. In general, this procedure is costly, since for each task, it requires the construction of the schedule until Di.

$$W_i(t) = \sum_{j=1}^{i} C_j . \left\lceil \frac{t}{\boldsymbol{T_j}} \right\rceil \tag{3.6}$$

Hence, a breakdown utilisation analysis is introduced to show that the average scheduling bound is usually much better than the least bound – as proposed by Liu and Layland (1973).

In 1989, an exact Boolean schedulability test was introduced by (Lehoczky et al., 1989). Exact schedulability conditions were based on the Processor's demand, computed according to the total demand for processor time in a critical instant by a job, combined with the total demand of processor time for all the higher priority tasks. Then the test check if this demand can be met prior to the job's deadline.

Audsley (Audsley et al., 1993) proposed an efficient method for evaluating the exact interference on periodic tasks and derived a sufficient and necessary schedulability test for DM. According to the method proposed by Audsley, the longest response time (WCRT) *Ri* for a periodic task is computed at the critical instant (Joseph and Pandya, 1986), as the sum of its computation time and the interference due to pre-emption by a higher priority task:

$$\boldsymbol{R_i = C_i + I_i} \tag{3-7}$$

Where,

$$\boldsymbol{I_i = \sum_{j=1}^{i-1} \left\lceil \frac{R_i}{T_j} \right\rceil C_j} \tag{3-8}$$

Hence:

$$\boldsymbol{R_i = C_i + \sum_{j=1}^{i-1} \left\lceil \frac{R_i}{T_j} \right\rceil C_j} \tag{3-9}$$

The schedulability test was improved by Sha (see Sha et al., 1990), who claimed that it is difficult to extend schedulability analysis for priority scheduling, taking into account the application constraints that frequently exist in real-time applications, such as precedence constraints, release times that are unequal to the beginning of their periods and low jitter requirements.

Xu (1993) then suggested that it is possible to avoid the application of sophisticated runtime synchronisation mechanisms, by directly defining precedence relations and exclusion relations, on pairs of task segments to achieve process synchronisation and prevent simultaneous access to shared resources (Xu and Parnas, 1993) if pre-runtime scheduling or TT scheduling is being used.

In 1990, Jeffay, Stanat and Martel (see Jeffay et al., 1991a) proved that non-pre-emptive scheduling of concrete periodic tasks is NP-Hard in the strong sense. They presented a necessary and sufficient condition for the schedulability of co-operative periodic tasks under no idling EDF.

In order to improve the accuracy of schedulability, Katcher (1993) relaxed Liu and Layland's (1973) assumption A4 (Katcher et al., 1993). Katcher described scheduler implementation costs and produced new schedulability analysis to account for the timing behaviour of a system when using the scheduler. Unfortunately, the application of their simple theory yielded pessimistic predictions.

In 1995, a more precise model of the activities taking place in a real-time scheduler were made in comparison with Katcher's model (Burns et al., 1995), in particular the influences causing delays in task processing in the scheduler were taken into consideration. Also, it is possible to show that sets of periodic and sporadic tasks are

feasibly executed when a simple theory rejects such sets. Accurately predicting the scheduler's overheads proved to be a complex task.

Over the past 10 years, there have been many improvements which have been carried out in ET architecture to improve schedulability and to test accuracy and boundaries. The details of this evolutionary work are given in Appendix -B. Butazzo (2005) provides comprehensive studies for hard-real-time scheduling theories and schedulability analysis. In a more recent publication, Davis (2008) introduced an approach using the response time upper bound, to determine the right time to compute exact schedulability and new initial values as an advanced starting point; thereby significantly reducing the execution time of exact schedulability tests based on RTA.

In contrast, in light of the TT architecture, researchers placed much effort on increasing flexibility and resolving design fragility issues (Xu, 2000, 2007). For example, the constructive schedulability analysis of TT architecture can be done automatically rather than manually, in order to improve the cost of design to the system.

Later, Yao et al. (2009) introduced an approach using response time upper bound to determine when to compute exact schedulability and new initial values as an advanced starting point, significantly reducing the execution time of exact schedulability tests based on RTA.

In 2010, Bertogna (2010) derived a new hybrid limited-pre-emption real-time scheduling algorithm, which aimed to achieve low runtime overheads, while scheduling all the systems with fully pre-emptive algorithms.

Min-Alah (2011) proposed that a faster schedulability test becomes more prominent when it is applied to online systems. Under fixed priority co-operative real-time

systems, current schedulability tests (in exact form) can be divided into response-time based tests and scheduling points tests.

In 2011, Short (2011) claimed that, in most cases npEDF (non-pre-emptive earliest deadline first) outperforms many other co-operative software architectures. Short also provided a new schedulability test for npEDF based on Gendy's schedulability test. However, the analysis is still immature and may cause high jitter.

In more recent studies, Yao and Butazzo (2011) introduced a feasibility analysis under fixed priority scheduling with limited pre-emptions. This paper presents the schedulability analysis of real-time tasks with co-operative regions, under fixed priority assignments. In particular, two different pre-emption models are considered: the floating and the fixed pre-emption point model.

3.7 The influence of ET and TT architecture on the schedulability test

Each of the software architectures has demonstrated different approaches to verify the timing characteristics of such systems. In light of the TT architecture, based on the static scheduling method, designers have to provide a priori guarantee that all timing constraints will be satisfied, for example, by identifying all the critical sections of those programs that access shared resources and compute a schedule for all instances of the entire set of periodic tasks offline (Xu, 2007, Xu and Parnas, 1993, Xu, 2003). The construction of the schedule is considered to be a constructive sufficient schedulability test. Thus, more effort is required to ensure that all the different possible overload scenarios and worst-case overheads are considered in the schedule design. The principal drawback of offline scheduling is its inflexibility, since it can only handle periodic tasks. However, there are several methods that can be employed to increase its

flexibility ,such as transforming the sporadic requests into periodic requests or introducing a sporadic server task (Kopetz, 1997).

Unlike a TT architecture, an ET architecture need not have an offline computation in order to verify its timing properties. Nevertheless, the main challenge faced by designers of ET systems is to ensure that the assignment of task attributes and timing requirements can be imposed on the scheduler (Bate, 1998) during the run time. This can be achieved by applying the task schedulability test (Liu and Layland, 1973, Sha et al., 1990) or timing analysis (Tsai and Bi, 1991).

3.7.1 Number of schedulable tasks

As mentioned at the outset of the chapter, one of the main differences with TT and ET architectures is the implementation of a schedulability test. Although the aim of the schedulability test for both architectures is similar, to ensure all task sets must meet their timing requirements, the results of the test can differ for the same number of tasks running in TT and ET architectures.

Liu and Layland produced their schedulability test model based on the utilisation of the upper bound concept, under the assumptions that all tasks do not have resource constraints or precedence relations. Liu and Layland (1973) stated that all the task sets with a total utilisation smaller than the utilisation upper bound are schedulable. However, Xu (Xu and Parnas, 2000) proved this analysis to invariably be pessimistic, since the above condition is not always accurate. In cases where there are 20 tasks to run on RM scheduling, each of has an execution time of 1 milliseconds and a period of 28 milliseconds. Using the analysis with the total utilisation is 0.71 (20*(1/28)) and the processor utilisation is 0.705. This shows that the sets of tasks are not schedulable. Similarly, if a worst-case response time schedulability test is applied in the same

example; those sets of tasks also cannot be scheduled. However, it is argued that those tasks can be scheduled when a pre-runtime scheduling approach is used (Xu, 2000). In addition, a feasible schedule for the tasks can be computed offline; thereby the run-time scheduler can use this knowledge to achieve higher schedulability by scheduling tasks more efficiently (Buttazzo, 2005a).

3.7.2 Scheduler fragility

From the point of view of flexibility, an ET design is easy to modify and making it possible to add a dynamic task to an existing node; however, this is not easily implemented in a TT design (Burns, 1995).

One of the challenges facing the developers of TT designs is the scheduler fragility at design time. This means that if developers need to make small changes to the timing of particular tasks, they need to make substantial alterations to the whole schedule. However, this issue refers to an earlier form of TT schedules known as cyclic executives. In contrast, for more advance forms of TT scheduling (Xu, 2007), the schedule can be constructed using algorithms to automate the task schedules. If modification is required, system designers do not need to construct the whole schedule by hand, which would be very strenuous. Instead, they have to modify only the logical structures of the tasks and their time constraints. In order to obtain a new schedule, an automatic scheduling algorithm can be used to reschedule the modified processes and segments. In more recent work, Gendy (2008) provided a constructive schedulability analysis for the TTC and TTH schedulers known as the TTSA (Time Triggered Schedulability Analysis). The TTC and TTH schedule can then be constructed automatically using TTSA algorithms.

Unlike an ET architecture, some people claims that a TT architecture is not flexible because it does not allow for the dynamic admission of tasks. However, Xu (2007) argued that neither ET nor TT architecture can guarantee that the timing constraints associated with a new task with an unknown arrival time can be satisfied, unless both software architectures have information about the tasks' timing properties in advance, such as WCET and task period, in order to determine before runtime whether those timing constraints can be satisfied or not. In TT architecture, more information about the task, such as release times and precedence constraints are required. Although some people claim that ET architecture requires a small fraction of information be added to a new task, the results show that the system under ET architecture has a higher system overhead, resulting in lower processor utilisation, and most importantly, making the system's runtime behaviour more difficult to test and predict (Xu, 2003).

3.7.3 Complexity of scheduling test algorithms

One of the most important tools in scheduling research is complexity theory (Leung, 2004). It identifies the efficiency of an algorithm based on its run time, or computational time, which is measured by the number of basic steps it takes to perform an operation (Leung, 2004). In scheduling studies, the complexity of algorithms is represented by measuring the running time of algorithms as a function of the number of tasks, *n*. This is reasonable since the algorithm would be expected to take more time as the number of tasks increases. If the growth rate is an exponential function of *n*, this means the algorithm is not practical, except for small numbers of Task Scheduling problems.

Each schedulability test algorithm has different complexity functions, which are represented by the big-oh notation and n as in the number of tasks. Liu and Layland's

analysis has *O(n)* complexity, since its schedulability test condition depends only on the number of tasks.

The commonly used scheduling algorithm for offline scheduling is the Brute-force (Burns et al., 1995), branch-and-bound (BnB) (Xu and Parnas, 1990), and heuristic search. The performance of algorithms is usually measured as the time the algorithm takes to find a feasible schedule. For example, Brute-force and BnB strategies take longer to find a feasible solution for large numbers of tasks than heuristic search algorithms. This is because Brute-force and BnB algorithms search and test all possible combinations of settings, whereas heuristic algorithms search only to test which offers a solution.

In TTSA (time-triggered schedulability analysis) testing only, some combinations of test paths are applied in order to find a feasible schedule for a set of tasks (Gendy, 2008); causing the complexity to be *O(n.t)*.

Furthermore, the exact schedulability test, such as the response time analysis (RTA) (Audsley, 1995) and processor demand analysis (Lehoczky et al., 1989) have pseudo-polynomial complexity; this means the schedulability test using those algorithms is time consuming, due to its high computational complexity (Bini, 2003). Ramirez (2009) provided a complete complexity comparison for schedulability test algorithms for RM scheduling.

In addition, a system which employs an ET architecture with co-operative scheduling and the arrival time of tasks is unknown; causing a problem when minimising the maximum lateness and when finding a feasible schedule, as this can become NP-hard (Butazzo, 2005).

3.8 Current state of the art

In 2009, Lindstrom conducted a comparison analysis for the testing of ET and TT systems inspired by Schütz (1993) works. Pre-emptions, observations and test inputs were highlighted as the main components when comparing TT and ET architecture. The experiment was conducted in simulations, as the number of test inputs for ET architecture is usually large and difficult to analyse using a hardware implementation.

In Chapter 2, some of the current trends for scheduling were discussed. Since schedulability testing is very important in scheduling design, researchers always need to provide algorithms for this. Thus, the trends for schedulability tests are similar to the trends for scheduling (as shown in Figure 3.2).

Current scheduling research focuses on improving the schedulability test's accuracy and complexity. Recently, Butazzo (2010) proposed an approach using the response time upper bound to determine when to compute exact schedulability and new initial values as an advanced starting point. This is something that significantly reduced the execution time for the exact schedulability tests based on RTA.

Min-Alah (2011) suggested that the speed of the schedulability test becomes a more prominent feature when applied to online systems. In the light of fixed priority co-operative real-time systems, current schedulability tests (in exact form) can be divided into: response time based tests, and scheduling point based tests.

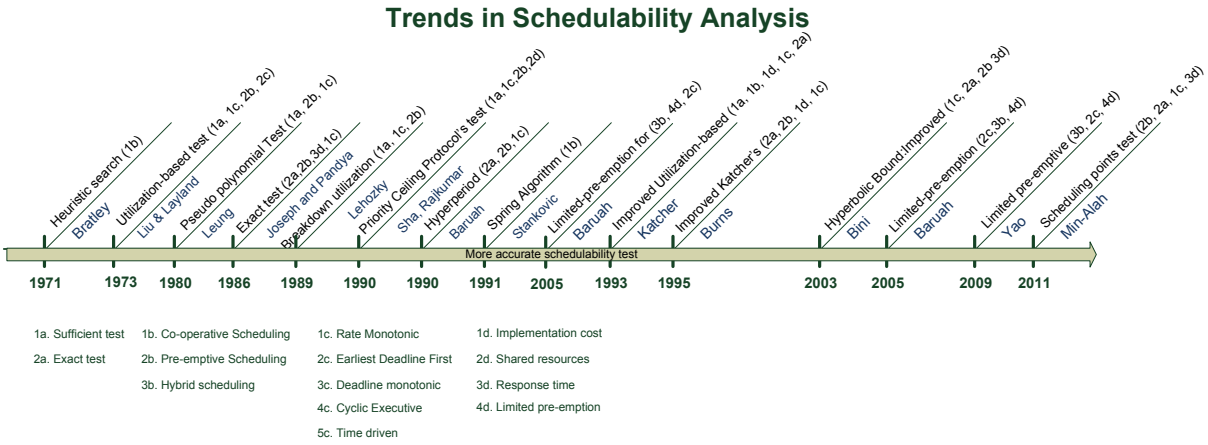


Figure 3.2 Trends in Schedulability test.

Yao and Butazzo (2011) proposed a feasibility analysis under fixed priority scheduling with limited pre-emption. This paper presents the schedulability for of real-time tasks with co-operative regions, under fixed priority assignments. In particular, two different pre-emption models were discussed; the floating and the fixed pre-emption point models.

In 2011, Short proposed a schedulability test for npEDF scheduling, which was adapted from the TTSA schedulability algorithm (Gendy, 2008). Although the complexity and number of tasks that can be scheduled has improved when compared to the TTC, the jitter is high. This is unacceptable for real-time systems, because they require high responsiveness.

3.9 Cross-architecture evaluation on cost of design

In order to assess the cost of scheduler's design performance, cross-scheduler architecture evaluation is usually employed. Cross-architecture evaluation of cost of design is defined as a comparative study between two real-time software architectures.

One of the earliest cross-architecture evaluations was conducted by Kopetz (1991, 1997), who discussed the theory and practical work of TT and ET architecture focusing

more on distributed systems. Extensive work was conducted by Kopetz and his research group comparing TT and ET architecture for scalability, and the testability of distributed real-time systems (Fohler et al., 2001, Puschner and Nossal, 1998).

Katcher (1993) provides a wider cross-architectural evaluation, covering ET and TT within fixed priority domains with pre-emptive and co-operative scheduling. Therefore, in total, Katcher developed and compared four different architectures in terms of their schedulability testing, which also included the implementation costs. Unfortunately, the application of this simple theory yields pessimistic predictions.

One of the most recent cross-architecture evaluations was that performed by Lindstrom (2009). The work involved testing real-time systems in ET and TT designs. The details of his work were discussed in the previous sections.

Lastly, Xu's papers provide a detail explanation of pre-runtime scheduling (a type of TT scheduling). Xu also clarified some of the misconceptions included in Burns' (1995) and Tindell's (1994) papers, which claimed that TT designs endured with design fragility and inflexible behaviour. For example, in the latest TT schedulability test, an automatic schedulability analysis can be employed. If a new task needs to added to this system, then the designers are not required to reconstruct the whole system (Xu, 2003). Xu's works is theoretically biased, therefore it offers insufficient means of evaluation. The actual system implemented must be tested so that performance can be effectively evaluated.

3.10 Scheduler implementation issues

The use of embedded software within real-time applications has increased dramatically across almost all industrial fields. According to Potocki De Montalk (1991), the number of words of executable code in civil aircraft doubled every couple of years between

1965 and 1995. Furthermore, the code size has expanded further in recent years, as real-time systems have become more advanced (Abella et al., 2011). For example, the lines of code for space missions increase consistently over time from 1990 to 2010. In 2010, the LOC of the MER mission consisted of 600, 000 lines. Based on this trend, it is predicted that software will become more complex and increasingly costly to maintain as a consequence. As is becomes more complex the need for continuous measurement, monitoring and control increases.

At this point, we have merely discussed verification issues associated with ET and TT architecture at design time. In practice, many issues need to be considered with respect to hardware implementation. For example, we know that a processor is the most important shared resource for tasks. Processor attributes contribute to the implementation cost in the form of overheads, blocking, dispatch latency and worst case interrupt response time, as will appear in a kernel scheduler implementation (Labrose, 2002). Cho et al. (2007) clarified that the term scheduler implementation also refers to the process of implementing a physical (software or hardware) scheduler that enforces the task sequencing determined by the schedule.

This section presents an overview of the literature concerning scheduling implementation costs; in particular scheduling overheads, context switch overheads and blocking, which are all influenced by pre-emptive and co-operative scheduler's implementations.

This section also discusses other issues that are required to implement these schedulers, most of which researchers do not consider in their discussions. For example, Katcher (1993), Burns (1995) and Arakawa et al. (1993) provided only a comparative study of implementation costs for pre-emptive versus co-operative scheduling in theory and

practice, without any explanation of the implementation cost of the code in reference to schedulers.

3.10.1 The impact of lines of code on cost of implementation

Given that a scheduling "algorithm" is a set of rules that, at every moment in the system's run-time, determine which task must be allocated resources to execute, the scheduler "implementation" is the process of transforming these rules into an executable source code (Sommerville, 2007). Therefore, the source code is interpreted as the lower-level software representation of the system, which practically dictates its functional and timing behaviour. Thus, the scheduler's source code should be observed in order to determine the complexity of the scheduler's implementation.

The importance of code size analysis is described in this section. Size measures can be measured using lines of code (LOC), function points and feature points. This is not only a key indicator of software cost but also a base unit from which to derive other metrics to describe project status and software quality measurement. LOC is one of the oldest and simplest ways to predict programming effort (Shen et al., 1983). It measures any line of program text that is not a comment or a blank line, regardless of the number of statements or fragments of statements on the line. Although the usage of LOC as an indicator for software cost and effort is of questionable validity, Rosenberg (1997) claimed it remains an important metric in software engineering, enabling it to become a uniform basis for evaluation in almost all empirical studies metrics, including function points (Albrecht and Gaffney, 1983). In addition, it is always used to predict software development and maintenance efforts, as in the COCOMO model. Moreover, there are many empirical studies demonstrating the efficacy of LOC for planning, monitoring the progress of projects and comparing the effectiveness of several software metrics (Basili,

1980). This shows that the LOC technique is a valid indicator of how to evaluate software costs.

In more recent study, Kitchenham (2010) provides a comprehensive review of software metrics evaluation research between the years 2000 and 2005. One of the outcomes mentioned is that at any specific time, it is inappropriate to use code metrics to predict fault rates in a largely evolving system, due to the lack of correlation between pre-release and post-release faults (Fenton and Pfleeger, 1997). They claimed that LOC does scale linearly with fault counts for the pre-release of a system's elements, which has modules that were developed in the same way and which have the same potential to detect faults (Kitchenham, 2010).

3.10.2 CPU utilisation and memory requirements

The most important parameters required to evaluate system performance relate to the use of CPU and memory utilisation. A CPU utilisation or time-loading factor, *U*, is a relative measure of any non-idle processing that is taking place. Systems with a high CPU utilisation value may cause problems, since the system is risky as regards time-overloading; whereas, systems with low CPU utilisation are not cost-effective. The term time-overloading relates to when the percentage of time the application spends executing or operating a system kernel code and task is very high (Hunt and John, 2012).

Memory is an important aspect of any embedded system design and is greatly influenced by the software design; it in turn may dictate how the software is designed, written and developed. Katcher (1993) measured CPU utilisation to evaluate performance of pre-emptive and co-operative schedulers. Chu et al. (2007) used memory requirements to compare the performance of scheduler implementations

running in centralised and distributed systems. Moreover, Anderson et al. (1997b), measured CPU utilisation to observe the performance of a system with and without locking synchronisation methods. Nahas et al. (2009) produced a fair comparison of various TTC scheduler implementations and used CPU and memory requirements as evaluative parameters. To produce a more accurate analysis, CPU and memory requirements for scheduler implementation should be considered.

3.10.3 Real-time systems overheads

Real-time system overhead is the time spent in the kernel performing a service for a specific task, such as invoking or terminating it (Katcher, 1993). Several types of interrupt mechanisms, such as timer interrupts and event interrupts, can invoke task processing. Interrupts are used for various reasons, firstly, to enable the processor to deal with external aperiodic events and secondly, to provide accurate timing of system operations.

The first interrupt is associated with ET scheduling and the interrupt handling operation. For example, if a switch is pressed, an external device signals a hardware-generated interrupt. The corresponding interrupt handler is then dispatched to the processor and the interrupt handler starts executing. The task is inserted into the ready queue immediately after it becomes ready.

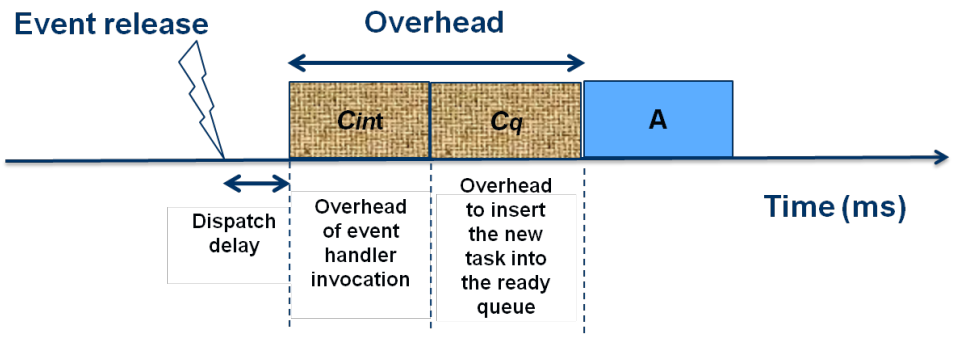


Figure 3.3 Scheduling overheads of event-triggered scheduling

Figure 3.3 illustrates an interrupt-driven operation. The total scheduling overhead of ET scheduling is *Cint* + *Cq*, where *Cint* is the fixed overhead of every interrupt invocation and *Cq* is the time required by the interrupt handler to insert the new task into the ready queue. Since the scheduler is invoked on task arrival, this may lead to a large overhead. In other words, to handle more interrupts, the processing overhead costs will be increased.

The second interrupt is associated with tick scheduling or TT scheduling. Unlike ET scheduling, the scheduler is usually associated with a timer handler that is invoked periodically. The interrupt occurs at regular intervals (also referred to as clock interrupts), affecting the scheduling overhead. A drawback of this approach is that when an event release occurs between clock interrupts, then the event handler is delayed until the next scheduler invocation.

In addition, real-time scheduling overheads are also caused by context switches, which are required when maintaining the context of the tasks involved in a pre-emption operation. The context-switch overhead is the time spent by the scheduler to service the event interrupt that triggers the context switch, and to perform the scheduling action at the context switch. For instance, when the processor acknowledges the interrupt (for pre-empting current task), it begins to complete current instructional activities, and then save all the contents of current task register information to the stack. Then, it branches (‘vector’s) to the related interrupt service routine (ISR) and starts executing ISR codes. After the body of the ISR is executed, the processor restores all the register information from the stack.

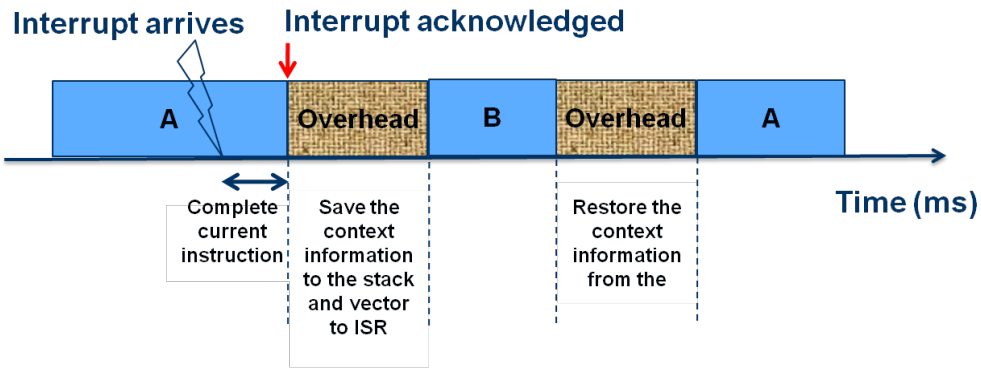


Figure 3.4 Context switching overheads

Figure 3.4 illustrates the context switch overhead associated with pre-empting task A, saving task A's context, loading task B's context and resuming task B. The functionality depends on how many registers have to be saved and restored by the processor. The more registers a CPU has, the higher the overhead is (Labrose, 2002).

3.10.4 Blocking

Blocking is time spent either in the kernel or in an application task, when a higher priority task is prevented from running; this is also referred to as a priority inversion problem (Sha et al, 1990). This is a problem unique to systems with co-operative scheduling. They can also avoid pre-emptive context switches due to blocking. Thus, blocking can help to reduce the cost of a context switch overhead. However, from a scheduling perspective, this can introduce long blocking segments, which can affect task response time.

3.10.5 Other implementation costs

In 2007, Gebhard and Altmeyer mentioned two other costs that need to be considered for each pre-emption. These are not only the scheduling and context switch costs identified by Katcher (1993) and Burns (1995), but the real costs involved in scheduler implementation. Those identified are as follows:

- a scheduling cost: arising due to the time taken by the scheduling algorithm to suspend the running task, insert it into the ready queue, switch the context, and dispatch the new incoming task;
- a pipeline cost: arising due to the time taken to flush the processor pipeline when the task is interrupted and the time taken to refill the pipeline when the task is resumed;
- a cache-related cost: arising due to the time taken to reload the cache lines evicted by the preempting task. The length of time depends on the specific point at which preemption occurs and on the number of preemptions experienced during the task

These implementation costs result in the degradation of task WCET by up to 40% when runtime overhead is included (Yao and Buttazo, 2011). Furthermore, the fact that the schedulability test accuracy of real-time tasks was found to be degraded by 20% from the ideal, when the implementation cost is considered in pre-emptive scheduling this is an important aspect to be explored (Katcher 1993). This can be one of the factors that can affect cost and the effort of verification and testing of real-time systems.

3.11 Evaluation on cost of implementation in TT architecture

Over a period of 10 years, the implementation process for TTC schedulers on a broad range of low-cost embedded microcontroller platforms has been one of the prominent areas of interests for ESL researchers. An early work in this area was carried out by Pont (2001) which described techniques for implementing TTC architectures using a comprehensive set of “software design patterns” written in C programming language. The resulting “pattern language” was referred to as the “PTTES Collection” which contained more than seventy different patterns.

Moreover, Professor Michael Pont and his PhD students also considered the design and implementation of a time-triggered hybrid (TTH) scheduler, which allows a single, time-triggered, pre-emptive task to be scheduled in the TTC scheduling framework. This architecture can wither be considered as an extended version, or as a modified implementation of the original TTC scheduler. Various comparisons of a TTH scheduler and a TTC scheduler implementation have been described by researchers (Pont, 2001; Maaita and Pont, 2005; Hughes and Pont, 2009; Phatrapornnant, 2007, Short, 2012).

Nahas (2008) compared the cost of a set of representative implementation classes for a TTC scheduling algorithm such as TTC-ISR, TTC Dispatch and TTC-Adaptive in hardware-based implementation. The implementation costs (including CPU, memory and power requirements) involved in creating each scheduler are also considered when distinguishing between the different TTC implementations. Note that the source codes in all the outlined scheduler implementations were written in programming language C. This language is used because of its efficiency; it can support functions and modules as well as affording good access to hardware via pointers. It is also available with every embedded processor (8-bit to 32-bit or more) (Pont, 2002) (Lindgren et al., 2008).

In addition, Wang (2008) and Short (2008) compared fully pre-emptive scheduling using RM and static schedule using TTC. They proved that the system still has predictable behaviour with limited resources. For example, TTC needs less than 651.1% of RAM as compared to RM scheduling. Moreover, (Hanif, 2008) implemented a simple but flexible TT architecture for practical embedded applications such as motor applications. However, the design is still immature and the research is ongoing.

Moreover, since 2001, ESL researchers have also been concerned with the implementation of TT architectures on multi-processor embedded platforms (Athaide et

al., 2008, Muhammad and Pont, 2010). Atheide (2007) proposed a novel high determinism multi-core processor with two capable software scheduler implementations that allow for application software to be designed, as for a single-core system by leveraging the TT nature of the underlying system. On the other hand, Amir (2010) compared various TT-based Shared-Clock scheduler implementations to provide high reliability communications at low cost, using a Control Area Network (CAN) protocol. Nevertheless, detailed comparisons relating to cost of scheduling implementation have still not been done in this research group.

3.12 Error detection approaches in ET and TT systems

In order to detect errors that may be affected by the timing fault, transient fault, random fault, systematic fault or permanent fault (Tsai, 1990, Storey, 1996), testing and monitoring processes are required. In real-time systems, the crucial aim of testing is to detect timing errors. However, what actually causes timing errors? Specifically, real-time systems are usually exposed to timing errors (Tsai and Bi, 1991), persistent errors and synchronisation errors (for multi-tasking real-time systems) (Tsai et al., 1990b), since the system has to deal with timing constraints. A typical timing constraint with a task is the deadline, which represents the time before which it should complete its execution to be certain of not causing damage to the system. For hard real-time systems, any instance of the task should be guaranteed a priori in the worst-case scenario, since the consequences of a missed deadline in such systems can be catastrophic.

Tsai (1991) has produced three theorems with regards to the violation of timing constraints in real-time systems. The first theorem is related to computational errors, which may occur at runtime. If the tasks of a timing constraint take more time to execute than expected, the program needs to be re-designed to reduce execution time.

The second theorem relates to scheduling errors. As explained in Chapter 2, a scheduler determines which task will run next in a multitasking system. However, if tasks cannot obtain enough CPU time to execute, this means that timing constraints cannot be guaranteed. Thus, a method for re-scheduling or re-assigning tasks with higher properties is essential. The third theorem relates to resource constraints, in which operating systems usually provide a synchronisation mechanism to ensure sequential accesses to mutually exclusive resources. When a running task occupies mutually exclusive resources, such as a semaphore, all tasks blocked on the same resource are kept in a queue associated with the semaphore that protects the resource. Another task will enter a waiting state whenever the semaphore is unlocked by a running task. However, this may cause a timing error if the task spends too much time in a waiting state.

Since the failure of embedded real-time systems can be dangerous to the environment or human life, timing errors must be detected effectively. Over the past 20 years, a number of testing and monitoring techniques have been extensively developed, based on real-time testing requirements. Nevertheless, it has been discovered that non-deterministic software architecture behaviour used in a system complicates the testing and monitoring processes. In addition, the testing process itself is far from trivial, not least because it is often difficult to determine which task or process is the root cause of an observed problem (Tsai, 1990). In this section, research work on monitoring and testing system faults as applied to existing ET and TT systems is described.

3.12.1 Issues in ET systems

One of the main reasons why it is hard to debug and test an ET system is because the run time behaviour of the scheduler can be unpredictable, making it difficult to analyse (Tsai et al., 1990b). Non-deterministic behaviour can cause a reproduction of execution

times that is difficult to achieve. A deterministic replay of execution behaviour in a way that guarantees the reproduction of program errors is essential in order to detect the source of any faults. If errors are irreproducible, logs and traces of inter task messages should be collected and investigated to assess whether any abnormality can be detected in the task behaviour. The user can check the messages and the time spent in the sequence to identify a failed task message and localise faults within that task. This requires sophisticated monitoring techniques and deterministic replay mechanisms, which are very costly and time-consuming to develop.

There have been a number of previous research studies that have explored the design of monitoring for testing and debugging of single nodes in real-time. For example, Thane et al. (2000) presents a method that calculates all possible execution orders for a system with periodic tasks only and fixed priority scheduling using an execution order graph (EOG). The system was developed according to a real-time kernel that supports pre-emptive scheduling. In addition, the kernel has a recording mechanism that it uses to record significant system events such as task start, pre-emption and access to a real-time clock.

A software-based monitoring technique was developed and managed to avoid the "Probe Effect" and to provide reproducible deterministic observation. Prior to these implementations, a software-based monitoring system that employed a passive testing method was described. Time encapsulation and a schedulability analyser were used by Tokuda (1988) in the ART real-time monitoring system for testing purposes.

Software-based monitoring approaches need to consider the effects of probes caused by software instruments. If probes are not removed, they might slow down a system's performance (Schütz, 1993). As an alternative, hardware-based monitoring approaches are also a possibility. Tsai et al (1995) utilised a hardware instruction counter to enable

deterministic replay. A “Non-interference” monitoring system was also designed by Tsai; this ensures that the system does not affect the execution of the target system during the verification process. However, this approach requires dedicated hardware to monitor the target systems at a very low level, leading to additional problems, such as a lack of scalability and non-portability (Thane, 2000) which may increase testing costs.

Another disadvantage is that the testing of this system requires a number of tasks and system information in order to trace errors. In addition, event histories that are monitored and recorded using a software or hardware monitor can only be replayed on the same system in relation to reproducibility issues. It is also questionable whether it is possible to use the same event history in different programs. These are the main problems associated with reproducibility and observability with the testing of ET systems. The next section will discuss the testing methods applied in TT systems.

3.12.2 Issues in TT systems

Much research involves deciphering TT scheduler errors in software-based, hardware-based and simulation-based solutions. In general, the same issues linger in relation to the testing the TT pre-emptive scheduler with respect to reproducibility and observability. In general, errors are difficult to reproduce in the pre-emptive and ET domain. However, some of these issues can be reduced, since predictability and deterministic behaviour are provided for in TT design (Shutz, 1993).

In order to detect timing-based errors in TT systems, a variety of monitoring approaches can be used, such as the scheduler and watchdog approach (Bate, 1997, 1998, Pont, 2001) and a non-invasive monitoring system (Chan and Pont, 2010). Moreover, an implementation of the task guardian, as recommended by Hughes and Pont (2008) can also help to detect and handle task overruns in TTC scheduling

algorithms. However, all this experimental work has vulnerabilities, as it is reliant on monitoring (software based and hardware-based) techniques to detect errors. In reality, testing is a core means to identify where the sources of faults are located. This area of work is yet to be considered within an ESL research group.

Alternatively, the Hardware-in-the-loop (HIL) technique (Short and Pont, 2005) provides simulation-based testing using TT architecture that allows developers to investigate system safety and reliability efficiently. Typically, simulations can help to find errors in system designs. However, most actual systems are too complex to allow for the simulation of all the possible inputs. In addition, there is no guarantee of the absence of error, since simulations cannot exhaustively be run for all possible combinations of inputs and internal states. For example, a timing constraint violation still occurred on the F/A-18 aircraft system, even though the system had been verified through simulation-based testing (Shepard and Gagne, 1991, Shepard and Gagne, 1990). As a result, it was necessary had to identify the software components causing this issue in order to improve the simulation technique applied. This has further increased the complexity of the software and rendered its maintenance difficult; this a problem many hard real-time systems have encountered (Shepard, 1991).

3.13 Assessing timing behaviour

Previously the issues and current testing approaches available with ET and TT systems have been reviewed and discussed. It is worth noting that, most of the work discussed above relates to dynamic testing, because only testing on running systems can provide obvious implications for testing using ET and TT. In reality, there are various software verification issues including static analysis, formal methods and evolutionary testing. Other timing testing techniques will also be discussed in this section.

3.13.1 Formal verification methods

A range of modelling methods is used in the production of safety-critical systems. Formal methods may be used to model such systems as an aid to implementation, verification and validation (Storey, 1996). The formal methods utilised include extensions of temporal logic to allow for quantitative reasoning about time, analysis for timed Petri nets (Storey, 1996), real-time logic, and timing analysis using graphs (Tsai, 1991). Formal verification methods have been introduced in an attempt to prove the correctness of programs with respect to system specifications and software engineering. One of the formal specification methods is using different languages, such as timed CSP and TRIO formulas (Mandrioli et al., 1995), or techniques such as timed Petri nets, to produce an unambiguous specification of timing requirements.

In more recent work, Nourch et al. (2007) introduced an approach named ArchMDE for ”Architecture-centric Model Driven Engineering” to address the development and the formal verification of real-time software architecture. It uses concepts derived from Model Driven Engineering (MDE) and software architecture for the automatic generation of a network of timed automata, in accordance with a blackboard architectural style. Timed automata were used only by analysers of real-time formal models. Since then, the use of timed automata based tools has spread to almost every aspect of real-time MDD, such as controller synthesis, code synthesis, scheduling and probabilistic analyses. A small number of tools, such as UPPAAL (Behrmann et al., 2004), RED (LAUREN, 2001), and VerICS (Kacprzak et al., 2008), have been actively maintained and so have evolved over an extended period of time. As zone-based techniques are now well-established, there is a bright future awaiting timed automata based tools.

In contrast, modelling languages, like UML, describe concepts, rather than implementations of solutions. Thus, they are useful for organising designs and specifications to address the different perspectives of the system, meeting the needs of developers and customers. In particular, they capture a notion of correctness, in terms of the requirements the system has to meet. Formal methods typically address model correctness, operating on a (possibly very close) mathematical formalisation of the model. This makes it possible to prevent errors inexpensively at the early design stage (David, 1999). Another approach that can be applied is timed process algebra or finite-state machines (Clarke, 1995; Dasarthy, 1985). However, this is extremely time-consuming and costly (Storey, 1996). Thus, only projects which involve the highest level of criticality are reserved for this method of assessment.

3.13.2 Measurement techniques

Measurements can be used in different ways. End-to-end measurements of a subset of all possible executions produce estimates, not bounds. These may give the developer an idea of execution times in common cases and the likelihood of the occurrence of a worst case. Measurements can also be applied to code snippets, after which results are combined to estimate a whole program using similar methods to those used in static methods. Therefore, this make it possible for safe bounds to be stamped on rather simple architectures.

Measuring can be used in the same spirit as testing, in order to identify errors. Testing is the only method that allows a thorough examination of a test object's run-time behaviour in the actual application environment. Only by means of this examination, can dynamic aspects that are especially important for the correct functioning of real-time systems be considered. Examples of these dynamic aspects include the duration of

computations, the actual memory required during program execution and the synchronisation of parallel processes.

The main disadvantage of this method is that it can only detect the presence of errors; not their absence. Instead, the testing carried out must increase confidence in the system, even though it may still contain undetected errors; by ensuring that the software meets these requirements. In addition, when using this method, the real cause of the fault cannot be detected. Therefore, analysis of additional methods may be needed (Tsai, 1993).

3.13.3 Timing analysis

Timing analysis is critical for real-time systems (Tsai, 1991). An analytical method based on system execution traces for real-time systems can be used to find the causes of a violation for timing constraints from the collected data. The timing behaviour of a target system at process level can be observed using a non-interference monitoring system (Tsai, 1993). Timing analysis can also be represented as a graph, which is useful when highlighting the timing errors in real-time systems. For instance, the Dedicated Coloured Process Interaction Graph (DCPIG) can be used to detect deadlocks, distributed terminations, starvation and missed operation errors (Tsai, 1991). However, this form of analysis can only detect the segments which contain sets of tasks immediately responsible for the violation of the timing constraint; it is unable to identify the source of faults.

3.13.4 Evolutionary testing

Alternatively, evolutionary testing has been widely studied in the literature; it has been applied to many test data generation scenarios including temporal testing (Wegener et al., 1999), stress testing (Briand et al., 2005), finite state machine testing (Derderian et

al., 2010) and exception testing (Tracey et al., 2000). Wegener (1998) produced a so called evolutionary testing technique to discover which input situations produce a temporal error. The method begins by identifying the shortest and the longest execution times for test objects. To search for the longest and shortest execution times automatically, a genetic algorithm is used. The evolutionary process runs repeatedly until a temporal error is detected; for example when an execution time is found which is outside the specified timing bounds of the system being tested. The only disadvantage is that it is difficult to find suitable test parameters for genetic algorithms. The work has been extended, to improve the test quality, with a combination of evolutionary testing techniques reliant on systematic testing (Gotchmen, 1998). In addition, (Harman et al., 2012) delivered a comprehensive study on the trend for search-based test generation techniques that offer a suite of adaptive automated and semi-automated solutions in large complex problem spaces with multiple competing objectives. As real-time systems require fully automated and effective systems testing, a combination of search-based and adaptive random testing could be used to automatically generate test cases and test oracles when integrated with an environment simulator, to enable early testing of such system (Iqbal et al., 2012). In 2011, researchers were aiming to improve test coverage in real-time systems. For example, Ha et al. introduced an evolutionary algorithm that generated timed test traces to achieve transition coverage using Timed Extended Finite State Machines (TEFSM) in an abstract time domain (Ha et al., 2011).

3.14 Importance of fault localisation in testing

Part of the testing process is intended to localise system faults within the fault diagnosis process. Even for a simple system, such as a switch system with a time factor, fault analysis requires too much effort. This is due to the requirement for a detailed analysis

of all the possible faults that may occur within the system. The analysis is described in detail in Appendix -F.

In large and complex systems, software not only contains a single unit but also consists of an integration of separate software units or modules, such as functional, object and data flow structuring. By partitioning software systems into smaller chunks, it may be possible to reduce the total number of problems into one software unit (see Figure 3.5).

Furthermore, Tsai et al's (1990a) work mentioned to using fault localisation techniques for testing real-time software systems. These techniques are proven to reduce a large quantity of the time and effort required for diagnosing system faults. Furthermore, 95% of the problems can be represented by the fault localisation approach (Tsai et al., 1989). In order to accomplish this, dynamic testing can be used.

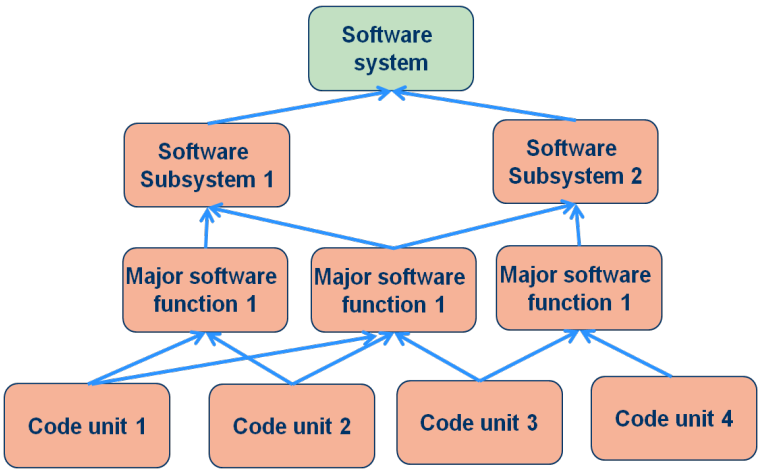


Figure 3.5 Typical software structures (Cooling, 2003)

In order to ensure that software system performs correctly, a program of software testing needs to be completed (Cooling, 2001). In general, each software unit needs to be tested individually or in isolation. This testing technique used facilitates developers to determine whether each code unit or software function behaves exactly as is expected. In addition, it reduces the difficulties resulting from discovering errors

contained in more complex portions of the application; test coverage is often enhanced because attention is given to each unit.

Many general software development methodologies are included with the range of suitable test methodologies. Test methodologies should be organised into separate phases so that the design phase is compatible with the system structure. For example, Gomaa (1986) introduced a strategy describing a Design Approach for Real-Time Systems (DARTS). The strategy begins by identifying the system's main functions and conducting a data flow analysis. Using this information, the system is structured into tasks and interfaces between tasks. In DARTS, the task interfaces are categorised into task synchronisation and task communication modules after developing the task structure. Each module contains more than one task, with similar functions. To verify the modules and tasks design, unit testing strategies can be employed. For example, an isolated task's test phase can be performed for each task and a task integration test can be performed for each module.

3.15 Discussion

In embedded real-time systems, instead of tight timing restriction, the non-deterministic behaviour of real-time software systems can make testing real-time systems more difficult when compared to non real-time systems (Lindstrom, 2008, Tsai, 1990). With deterministic and predictability behaviour imposed on a TT design, there is some reduction in the complexity involved in testing real-time software systems. Additionally, because they have a very predictable patterns of behaviour, testing TT systems can be comparatively straightforward: in fact, one of the factors which motivates organisations to adopt TT architectures is a desire to reduce the time taken to conduct system testing activities. This has become a motivating force for this research,

in order to determine whether TT designs can improve on the current testing techniques.

3.15.1 Verification at design level

Sections 3.6 to 3.9 have presented existing approaches to verify the timing constraints for ET and TT systems at the design stage. Obviously, software architecture and schedulability testing can affect the cost of a system's design. Previous studies have shown that the cost of designing a system with a TT architecture is greater than that with an ET architecture. However, since more advanced approaches have been developed to improve schedulability test techniques, the results may differ. As a result, an investigation into the effects of software architectures and scheduling strategy as linked to cost of design will be carried out.

In addition, in order to evaluate the performance of the scheduler design, schedulability theory provides theoretical modelling to assess and measure of schedulable tasks, (Liu and Layland, 1973, Bini, 2002, Baruah, 1990, Buttazo, 2005, Yao, 2010). However, as is evident from the literature review, the schedulability test becomes more accurate when the complexity and the schedulability's test-run time increases (although the latter is not desirable for ET architecture, it may have an effect on TT).

Moreover, extensive theoretical studies have been conducted to evaluate performance measures, demonstrating that schedulability theory is well established; however, there exists a vacuum in experimental research. In addition, most of the work conducted in the last few years has only focussed on improving schedulability analysis for a single architecture, most commonly for ET. Only a few papers were found to reflect cross-architecture evaluations. Thus, more assessments of the scheduler design need to be performed in order to assist designers in their choice as to the most cost-effective

scheduler design currently. From the literature review it is evident that the current state of the art scheduler design is TT architecture with limited pre-emption.

A misconception that has been observed in the literature is that TT is prone to schedule fragility (Xu, 2003). However, experimental evaluations to counter these misconceptions could not be identified within the literature. Furthermore, the disadvantage of TT architecture is stressed as its fragility; referred to as Cyclic Executive (CE) (Burns, 1995), this relates to TT's need to reschedule when each new task is added, thereby increasing the time-consuming nature of the testing. The current schedulability test mentioned by Xu (2003) is different from CE, and it is easy to add new tasks. In 2008 Gendy proposed a schedulability test for TTC and TTH with *O(n.t)* complexity, in which the running test time is faster than Xu's heuristic search for TT. However, comprehensive cross-architecture evaluations for design costs for the current state of the art software are scarce in the literature.

3.15.2 Assessment of implementation cost

Despite the usefulness of the studies carried out in the area of schedulers, there exists a vacuum in terms of discussion with regards to scheduler implementation and its implications in practical real-time embedded systems. More specifically, while there has been a great deal of interest in the development, assessment and refinement of real-time scheduling algorithms, the process of translating between algorithms and implementations has not been widely considered. This is a claim supported by Cho et al. (2007) who clearly stated that few studies address the architecture and the implementation of schedulers. The great majority of the studies reviewed during the course of this project mainly emphasise design issues and only discuss implementation issues from a removed perspective. The potential impact a particular software

implementation would have, on the actual run-time behaviour of the system when implementing the scheduler has not been considered.

Theories to evaluate pre-emptive based schedulers and schedulability analysis have seen massive development both at the theoretical and implementation level. However, in order to cap the complexity arising from infinite pre-emption, the trend is to limit that pre-emption (Betogna, 2009, Yao 2010).

Moreover, despite the usefulness of the studies carried out in the area of scheduler implementation, the literature survey showed that a comparative study of limited pre-emption scheduler, fully pre-emptive scheduler and fully co-operative scheduler implementation is lacking. Due to limited pre-emption and the scheduler's popularity for real-time embedded systems, as performed by TTH scheduler, this trend is expected to continue or progress over the next few years. Therefore, the need to document and analyse the impact of the TTC, TTH and TTP scheduler implementations in a systematic way is surely of vital importance and benefit to designers. The evaluation conducted in this thesis is illustrated in Figure 3.7.

Based on the study, only a handful of papers have been found that evaluate the cross-scheduling algorithms, performance from an implementation cost point of view (Katcher, 1993, Burns, 1995, Betogna, 2010) in comparison to the architectures, with implementation costs i.e. overheads, included. This emphasises the need to carry out comparative studies by adding more assessment parameters, i.e. CPU, memory utilisation and code size, for all architectures; in order to study the impact on overall implementation cost.

Reference is made to Section 3.6 and Section 3.11, where extensive theoretical studies were described as having been conducted to evaluate the performance of scheduling

theory and implementation costs. It has been discovered that most of the analytical works were used simulation-based approaches (Butazzo, 2005, Bertogna, 2010, Yao, 2011). Therefore, it is necessary to conduct hardware implementation in order to analyse the impact of scheduler implementation.

In resource-constrained and embedded systems (i.e. washing machines, mobile phones, mp3 players), designers are frequently concerned over CPU and memory requirements. These requirements should therefore be considered as a comparative measure for the implementation of the scheduler.

3.15.3 Cross-architecture evaluation on cost of testing

A broader perspective and impact analysis describing cross-architecture studies is very limited in the literature. In addition, scholars have noted the need to use appropriate software architecture to design for the testability of real-time systems (e.g. Kopetz (1991), Schütz (1993), and Xu, (2003)). However, it has been found that there is a vacuum in the following areas:

- Experimental testing performance evaluations of TT and ET architectures are scarce;
- Testing has not been covered extensively in the literature review, for cross-architecture comparative evaluation based on experimentation (Schütz 1993, Lindstorm, 2009); and
- The current trend in the research points towards limited pre-emption (Bertogna, 2010, Yao, 2011); thus, it is important to observe the impact of this on testing (as has been discussed in Section 3.6).

Since ET and TT systems are two completely different entities, the requirements and designs for each system are different. However, in order to develop a comparative

study, it is important to develop an equivalent system with the same functions and a similar number of tasks. This presents a great challenge to test designers. In addition, in order to produce an accurate analysis, hardware implementation is necessary. Both systems need to be programmed and implemented using the hardware, and a testing evaluation can be run in developed systems.

Based on the studies, there is a shortage of assessment and a limited analysis of software cost metrics when testing embedded software in ET and TT designs, and more specifically, in areas touching on co-operative and pre-emptive scheduling techniques. It is also important to observe how TTC, TTH and TTP schedulers perform when assessed at the testing and verification phases. This analysis will provide useful information to embedded systems software researchers who are aiming to achieve a limited pre-emptive scheduling and TT architecture (state-the-of-the-art scheduling and real-time software architecture). The trend for cross architecture comparative studies is described in Figure 3.6.

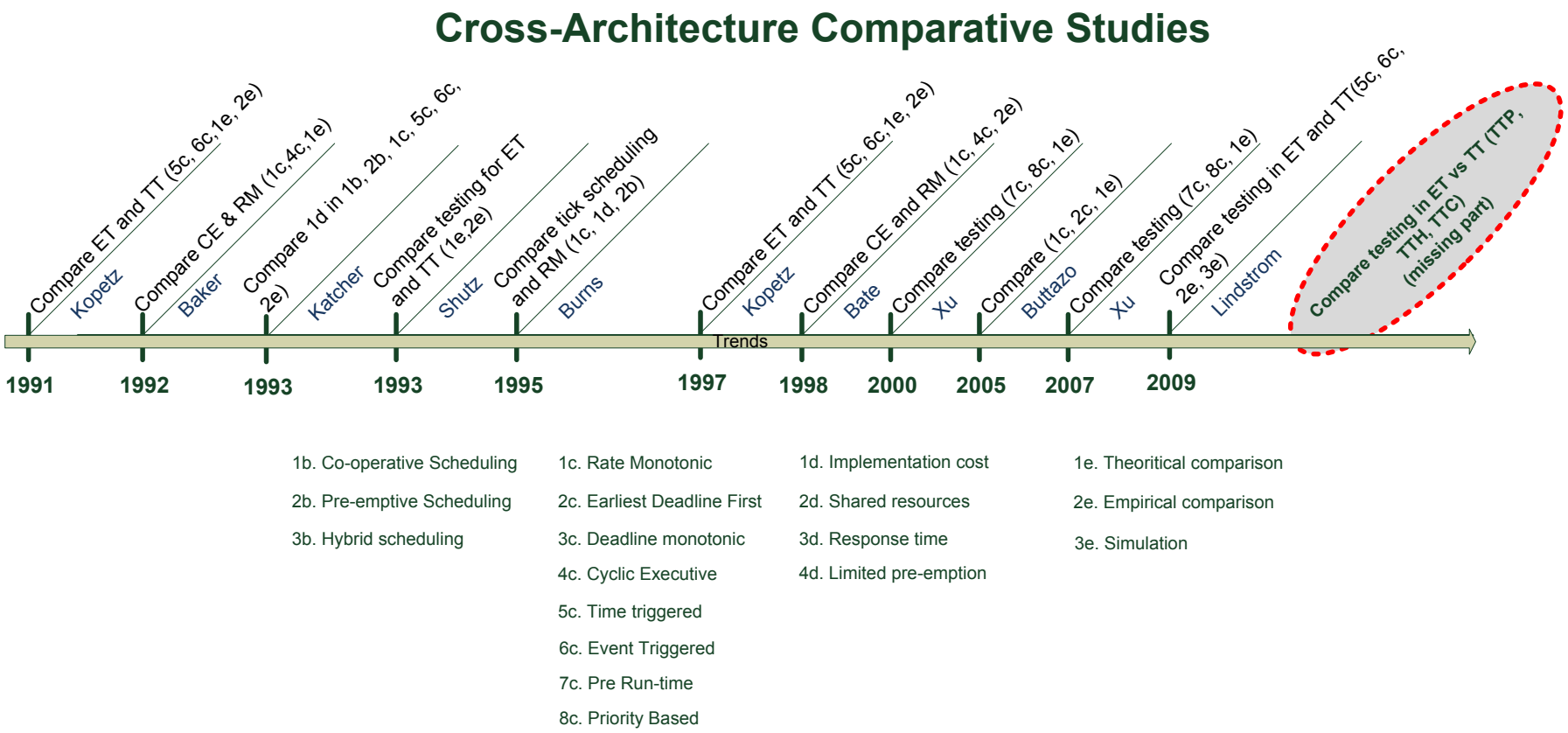


Figure 3.6 A review of cross-architecture comparative studies, highlighting the gap

3.15.4 Task isolation

As discussed above, a considerable volume of work has been produced to demonstrate how TT architecture can improve on the testability of real-time systems (Kopetz, 1991; Lindström et al., 2008; Schütz, 1993) in comparison to an ET architecture. However, to the author's knowledge, to date no empirical research exists addressing the question of how ET and various TT scheduling including TTP, TTH and TTP schedulers effects testing costs and efforts within an experimental-based approach.

In order to address this issue, a testing technique needs was designed for the purpose of conducting this comparative study. According to Tsai (1990), fault localisation is an effective means by which to find and source faults. In a software system, detecting the presence of a fault at the lowest level of the software structure (as shown in Figure 3.5) is essential. Hence, the smallest and most important component needing to be tested is the task. Although the idea of task testing is not new, there is a vacuum of studies revealing whether task isolation can help us to assess the testing effects of various software architectures and scheduling policies.

To our knowledge, there is an absence of empirical studies that show whether ET, TTP, TTH or TTC systems can easily reproduce the identical behaviour of a task running on a complete system with a task in isolation on a uniprocessor system. Most of the work done by Schütz (1993), Kopetz (1997), and Lindstrom (2008) discussed testing on real-time distributed systems. Although many modern systems are equipped with distributed or multiprocessors, some designers continue to prefer uniprocessors, ostensibly to avoid degradation of system performance (Amdahl, 2007).

3.15.5 Impact of shared resources in testing

The usage of semaphore or other synchronisation protection mechanisms in pre-emptive scheduling can cause additional overheads. Anderson (Anderson et al., 1997a) provides a comparative study of the overhead for lock-free and lock-based synchronisation in pre-emptive scheduling. It shows that the processor utilisation achieved is only about 94% when a lock-free approach is used. On the other hand, were lock-based techniques employed, the processor utilisation would be approximately 99.4% (Anderson et al., 1995). Lock-free techniques have three advantages, which allow tasks to work independently: (1) One task can access any shared resource without the need for detailed knowledge about another task's objects; (2) a new task can be added dynamically since operating tables do not have to be recomputed; and (3) this approach provides less overhead and lower task response times when compared to lock-based techniques. Alternatively, co-operative scheduling can be employed. In the TTC architecture (for example), pre-emption is not permitted; thus, there is no issue related to a conflict with shared resources.

As synchronisation mechanisms are very useful for ET systems and pre-emptive scheduling, it is important to observe their impact on testing. Analysis is also aimed at bridging the gap between ET systems - with various synchronisation implementations - and TT systems - with TTP, TTH and TTC implementations - particularly when testing real-time systems.

3.16 Conclusion

An extensive literature review covering the most important aspects related to the second hypothesis, as mentioned in Section 1.3, have been presented in this chapter. The drawbacks of event-triggered architecture have been discussed relative to the reviewed literature. The main concern with ET architecture derives from its non-deterministic

behaviour, which makes it difficult to test. The chapter also examined testing for time-triggered architecture and limited pre-emptive scheduling, which is currently considered to be the "state of the art" approach for real-time systems. As a result, it was found, that more cross-architecture testing evaluations need to be carried out to highlight the implications of different architectures on the cost of testing and verification (see Figure 4.6). To supply this need, comparative studies and evaluations are presented in the following chapter.

Moreover, extensive theoretical studies have been conducted to evaluate performance measures to the extent that schedulability theory is well established; however, there exists a vacuum in terms of experiments in the reviewed literature. In addition, most of the work conducted in recent years have only focussed on improving the schedulability analysis for a single architecture, particularly ET. Few papers were found to reflect cross-architecture evaluations. Thus, more assessment of the scheduler design needs to be performed in order to help designers to choose the most cost-effective design. From the literature review, it became evident that the current state of the art scheduler design is TT architecture with limited pre-emption.

A misconception observed in the literature regarding TT is that it is prone to schedule fragility (Xu, 2003). However, more comprehensive arguments against this misconception, based on experimental evaluations could not be identified during our literature survey. Furthermore, the disadvantages of TT architecture, due to its schedule fragility, referred as Cyclic Executive (CE) (Burns, 1995), meaning the TT's need to reschedule when each new task is added, lengthens the testing time. The current Schedulability Test mentioned by Xu is different from CE, in that it is easy to add to new tasks. Gendy in 2008, proposed a schedulability test for TTC and TTH with $O(n.t)$ complexity, in which the running test time is faster than that of Xu's heuristic search for

TT. However, comprehensive cross architecture evaluations and design costs for the current state of the art are scarce in the literature.

An extensive literature review covering the most important aspects relevant to the third hypothesis mentioned in Section 1.3 were presented in this chapter. The schedulers were reviewed as they evolved over time according to a schedulability analysis as has been summarised above. The drawbacks of the event-triggered architecture were discussed in reference to the reviewed literature. Based on the review, it has been observed that current and future trends are shifting towards TTH as a cost-effective prospective architecture. Furthermore, it has been found, that more cross-architecture evaluations need to be carried out to pinpoint the different implications arising from the use of architectures on the cost of design. Owing to the need for more experimental based evaluation, and to verify the theoretical models, comparative studies and evaluations are presented in the following chapters. Figure 3.7 illustrates the evaluation of software architecture conducted in this study.

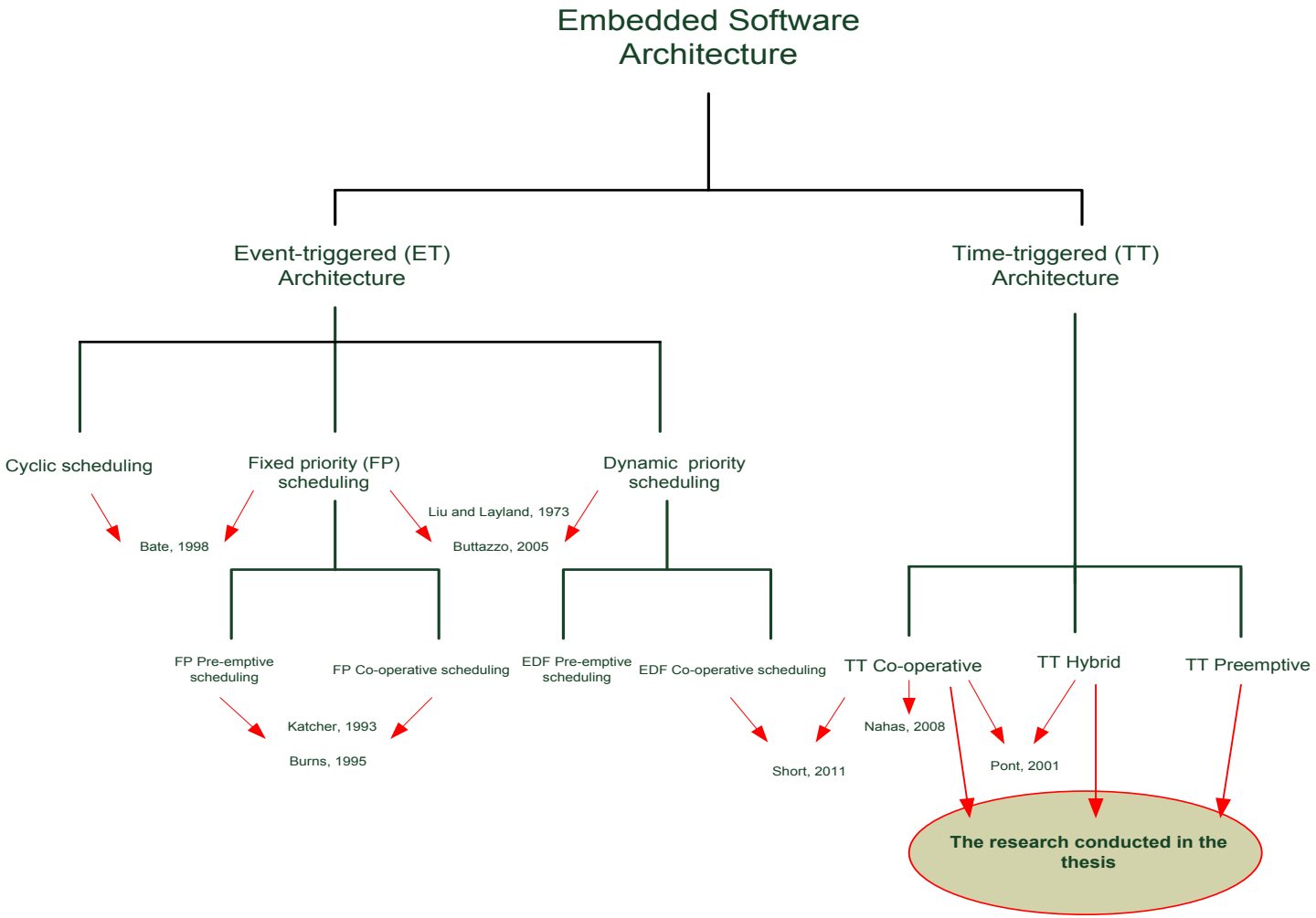


Figure 3.7 Evaluation conducted in this study

Chapter 4
A Novel Software Architecture Evaluation Model

4.1 Introduction

The rationale for this research was presented in Chapters 2 and 3. The main idea of this study is to explore the advantages associated with systems with high predictability. This includes TT systems, which can assist in reducing the costs arising from timing verification for real-time software systems in the design, implementation and testing phases. With this motivation, a method to evaluate the impact of software architecture as well as real-time scheduling was introduced to fulfil the following objectives: (1) identify the cost involved in evaluating the fundamental costs involved in ET and TT software architecture at the design, implementation and verification phases; and (2) assess that cost by using evaluation techniques and experimental approaches, then compare the results in order to show which of the software architectures produces a low-cost system (this will be discussed in Chapters 5, 6 and 7).

4.2 Necessity of software architecture evaluation

Scholars and designers have put forward more than 100 software inspection and verification approaches related to timing requirements in concurrent processes, and some real-time software architecture trade-off evaluations. However, there is a lack of evidence of real-time software behaviour with which to explore the impact of software architecture in reference to its design, implementation and verification costs, as a single goal. Such an evaluation would be useful for practitioners aiming to choose which software architecture is suited to verifying the timeliness characteristics of their systems. The results of this analysis are also of value to the research community; they will facilitate analysis of the cost and effort involved in existing timing verification

methods, for both ET and TT systems (in design, implementation and testing stages). This can be achieved by pointing out to what extent the verification of software with timing requirements will be affected as the system becomes larger and more complex.

The first aim here was to point out existing ET-based and TT-based timing verification approaches that propose concrete means to guarantee if critical timing constraints will be met. This is useful for both designers and scholars wishing to explore the cost and effort involved in various timing verification mechanisms for both software architectures. A key issue is the need for a combination of different sorts of software cost metrics; for instance, involving a running time that consumes a run schedulability test for a large number of tasks, based on overhead costs that need to be taken into account in order to trace errors in TT and ET schedulers by assessing the code size needed.

The second aim of this analysis, therefore, was to shed light on the software architecture role of timeliness testing and cost, evinced by existing timing verification approaches. This can help designers to select suitable software architecture, while avoiding lateness and additional predicaments when testing their systems. For example, systems which consider the ET architecture as a means to reduce the testing effort may not be suited to large complex real-time systems where the number of tasks is enormously high.

The third goal of the analysis was to contribute to an understanding of the ET and TT software architecture behind the timeliness testing and verification. This is useful for both scholars and designers, as a greater understanding of the software architecture increases their knowledge of why a particular architecture has (or is expected to have) the intended impact on system size. In addition, a need to use appropriate software

architecture to design for the testability of large real-time systems was noted by scholars (e.g. Kopetz (1991), Scheler, (2006) and Xu, (2003)). However, there is still a lack of assessment and analysis of software cost metrics when designing, implementing and testing embedded software in ET and TT designs, or more specifically, in co-operative, limited pre-emptive and pre-emptive scheduling techniques. A detailed discussion of the method used is given in the next section.

4.3 Description of method

This study introduces a novel evaluation model for comparing the cost of design, implementation and for testing real-time systems in order to ultimately assist designers to choose a cost-effective software architecture and scheduling policy. This work was conducted by several researchers, such as Kazman (1999) and Bate (2008), to analyse a trade-off in real-time software architecture, but not specifically for ET and TT architecture.

Kazman (1999) introduced a software architecture trade-off analysis and Bate (2008) proposed a graphical notation with cost estimate weighting analysis, to identify all the costs involved in order to guide designers or practitioners when making decisions about which software architecture would be ideal for their systems. In this study, the evaluation of software architecture does not only provide a quantitative or cost weighting analysis. In fact, real data from experimental results is used as evidence to assess the software architecture comparatively. In addition, the software architecture introduced in this study takes into account the cost involved at each phase of the software development cycle. It is widely known that verification is the most expensive activity in the software development process; thus, the main focus of this research was to evaluate the cost involved, in order to verify the timing behaviour for each software architecture. This will include assessing schedulability test running time and input

requirements, overheads and the simplicity of reproducing similar timing data for testing isolated tasks.

The results of the experimental work, strongly support the notion that an overall development cost evaluation is likely to fall below actual cost, as these two costs are incorporated in addition to design costs. The costs and their implications have been explained in detail in their relevant chapters respectively.

The method proposed is compared with Bate’s method, which is given in Figure 4.1.

The first step in Bate's model is to identify design goals. Once the goals are identified, all claims and arguments proceeding from the goals can be derived using GSN principles. Then, qualitative and quantitative assessment criteria can be extracted by expanding the goals for the system into sub-goals, relating to the efficiency of scheduling policies or greater design choices, such as using offline or online mechanisms. Based on a combination of software architecture evaluations results and weighted factors, a quantitative measure for a particular change to the design solution is provided.

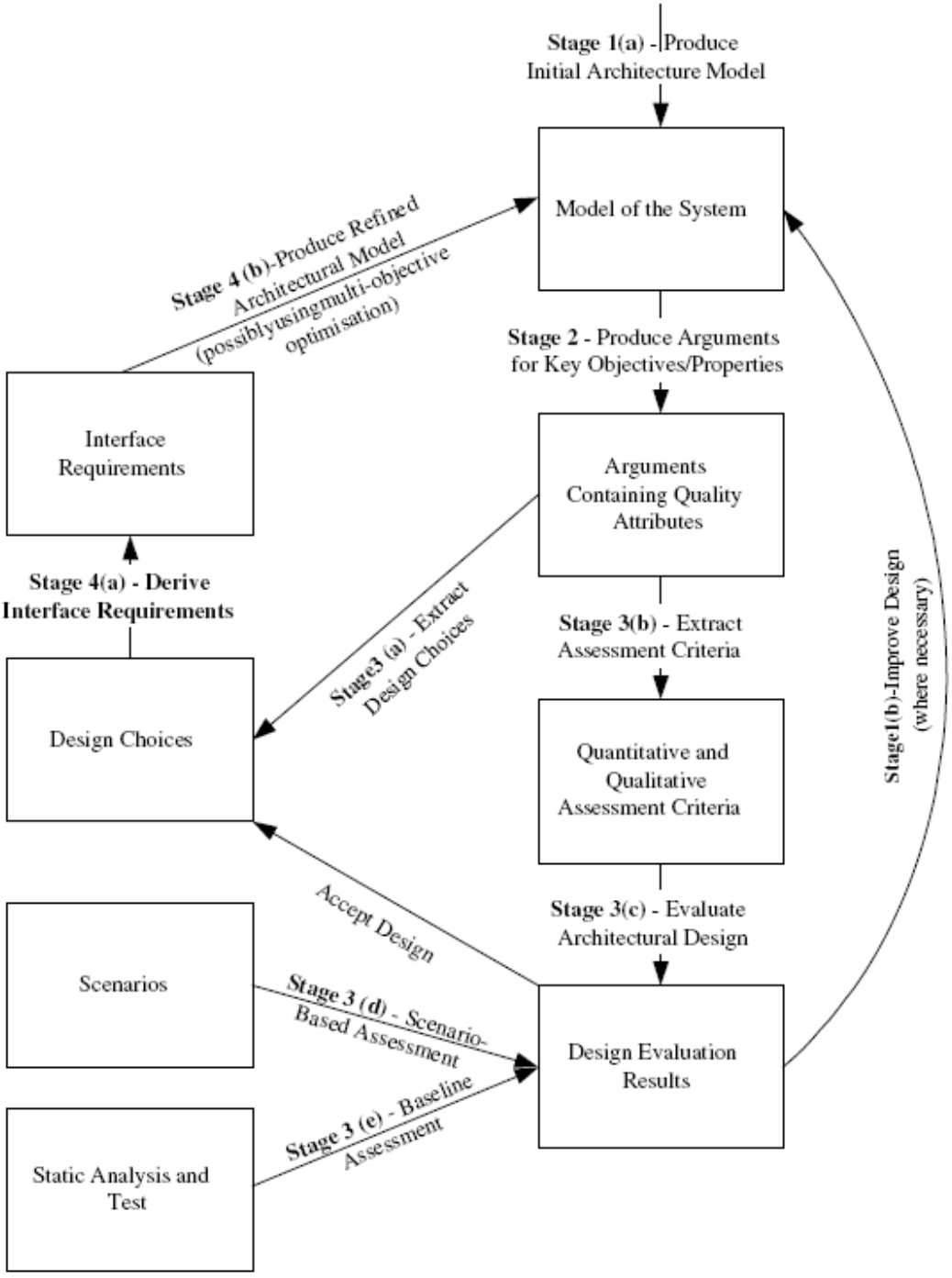


Figure 4.1 Bate's software architecture evaluation method (Bate, 2008)

4.4 A novel method for evaluating embedded real-time software architecture

This section is concerned with presenting the method derived, along with the rationale behind it. This method was inspired by Bate's trade off analysis and followed a step by step approach. However, in this study, costs were not presented using weighting techniques. Rather, costs were observed and compared based on experimental results and the impact of software architecture and scheduling strategies on software and hardware aspects, such as memory utilisation, overhead and lines of code.

4.4.1 Process

Figure 4.2 provides an overview of the method. The individual stages of which are explained in the following sections.

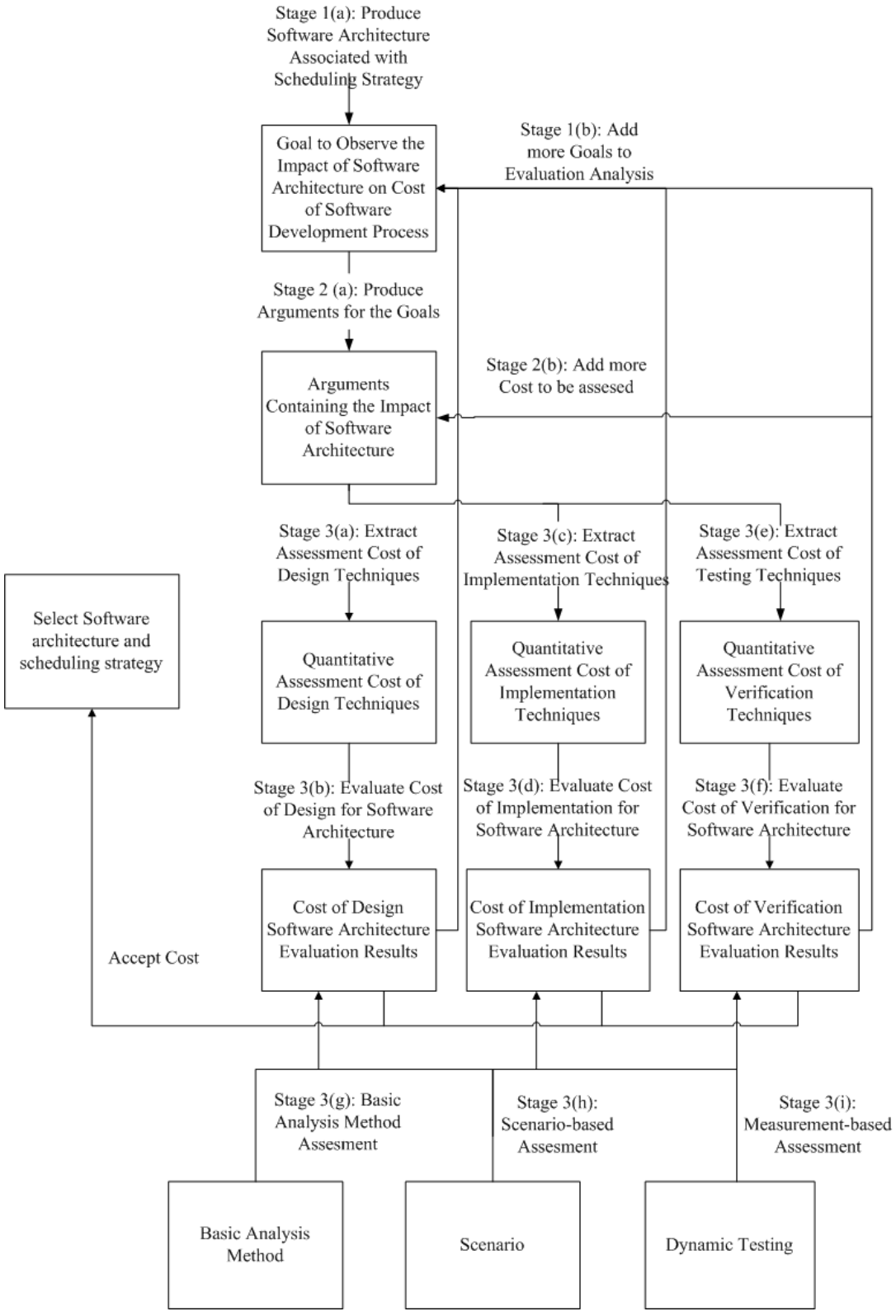


Figure 4.2 Overview of Evaluation Model for Software Architecture.

4.4.2 Stage 1 – Selecting the software architecture associated with the scheduling strategy

The evaluation cost method selects the main software architecture and scheduling strategy for assessment during the first stage. For instance, when carrying out the discussion for the RT embedded system architecture, it emerged that the impact of pre-emptive and co-operative scheduling on the system cost for ET and TT systems must be included as shown in Figure 4.3.

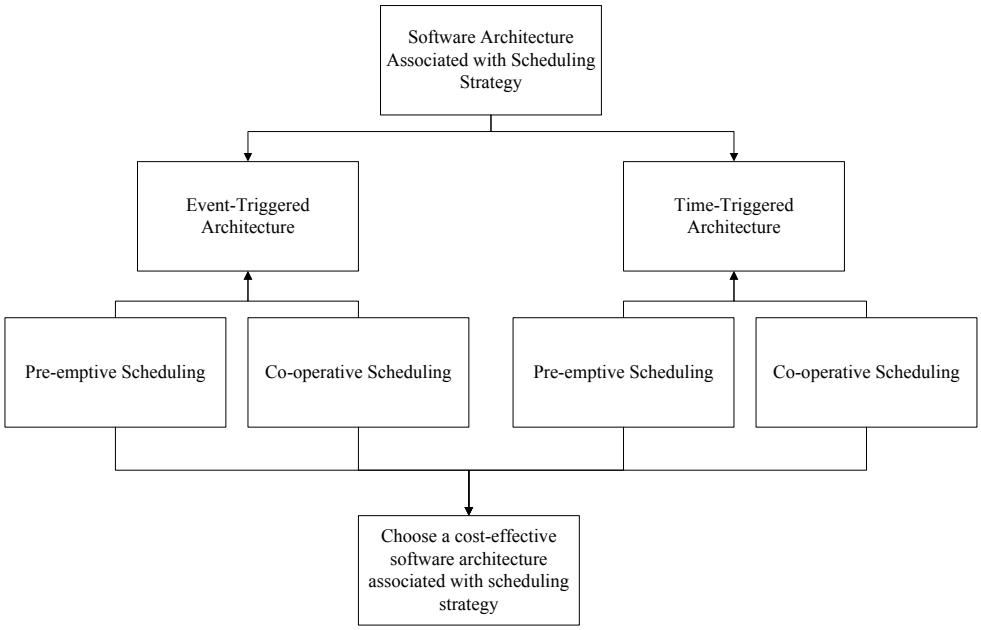


Figure 4.3 Example list of RT embedded systems' software architectures for assessment for RT embedded systems.

4.4.3 Stage 2 – Producing an argument for the goals

In the second stage, the aims of the system assessments were decomposed into more detailed goals, although there are many arguments that can result in a low-cost system. For instance, in this study, the specific goals focused on were low system overheads, ease of testing, simplicity in implementation, and most importantly, ensuring that the

timing requirements for the TT and the ET architecture would be met. The goals of the evaluation are described in Figure 4.4.

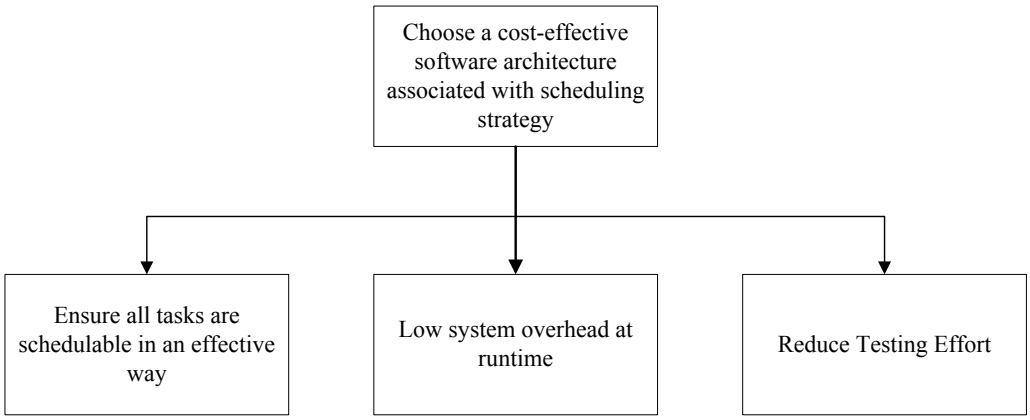


Figure 4.4 Example Evaluation objectives for RT-Architecture

4.4.4 Stage 3 – Extracting the software architecture's cost information from the argument

Stage 3 relied on a structured argument to derive a software architecture and the scheduling of strategic costs. This assessment needs to be done experimentally. Thus, a basic analysis, scenario-based and dynamic testing could be used for the evaluation.

When extracting objectives, it is necessary to consider whether that objective can be met. For example, the objective that the system must have a low design cost could have led to cost-effective software architecture. From a design cost perspective, more evaluation costs can be implemented until the impact of ET and TT software architecture can be seen and then designers can be guided to choose suitable software architecture.

4.4.4.1 Stage 3 (a) – Extracting the evaluation criteria on the cost of design

Once a suitable cost of design has been established, the evaluation criteria for that cost need to be extracted. For example, ET and TT architectures share dissimilarities in

complexity, running time, LOC and test inputs in the light of schedulability analysis. Thus, these can be the relative factors used to evaluate the cost of design across a variety of software architectures.

4.4.4.2 Stage 3 (b) – Evaluating the software architecture on the cost of design

Given the need to assess a set of evaluation criteria, the next stage calls for ways to determine how individual objectives from the design argument can be measured and analysed as different software architecture and scheduling strategies and design costs can be compared. This stage is involved in turning a given objective and argument into the experimental design cost evaluation.

4.4.4.3 Stage 3 (c) – Extracting evaluation criteria on cost of implementation

Once an argument exists about the suitable cost of implementation, the evaluation criteria for the cost needs to be extracted. For example, the accuracy of any schedulability analysis can be degraded when the implementation cost is considered. The pre-emptive scheduling implementation is usually affected by a context switch event that occurs when a higher-priority task interrupts a lower-priority task. It is impossible to take into account all of the effects of the context switch on schedulability analysis. Furthermore, an interrupt can happen at any time an event occurs. This makes evaluation of the implementation cost problematical. On the other hand, such issues do not exist in co-operative scheduling, since all the tasks should run until completion. In fact, a context switch then occurs only at the tick interval. Thus, based on this issue, context switch overheads can be used to understand the comparative and quantitative impact of software architecture on the cost of implementation of embedded systems. Moreover, the impacts of pre-emptive and co-operative scheduling implementations can

also be examined according to the usage of memory and CPU, the LOC required to implement scheduling and the impact of the number of tasks.

4.4.4.4 Stage 3 (d) – Evaluating the software architecture on the cost of implementation

Given a set of evaluation criteria for assessment, the next stage undertaken was to determine individual objectives according to the implementation argument as measured and analysed; thus, different forms of software architecture and scheduling strategy relative to the costs of implementation can be compared. The steps involved in turning a given objective and argument into experimental implementation and cost evaluation. Results can be obtained using measurement-based techniques and scenario-based techniques. For example, context switch overheads can be measured using execution time measurement-based techniques. In order to produce more comparative results, scenario-based techniques such as the impact of the number of tasks can be used.

4.4.4.5 Stage 3 (e) – Extracting the evaluation criteria on the cost of testing

Once a suitable cost for testing and verification has been determined, the evaluation criteria for the costs need to be extracted. There are numerous issues to be discussed relating to the testability of real-time systems, including reproducibility, controllability and observability. One can use one or more of these aspects for the purposes of evaluation. In this study, for instance, the ease of reproducing similar timing data for the isolated task and the in-situ task became the main criteria with which to observe the impact of software architecture on the cost of testing. Testing will be easier for systems in which the timing data obtained for isolated and in-situ tasks is similar. In reality, many issues can be considered when evaluating the cost of testing.

4.4.4.6 Stage 3 (f) – Evaluating the software architecture on cost of testing

Given a set of evaluation criteria to assess, the next step is to determine how individual objectives from a verification argument can be measured and analysed using different software architecture and scheduling strategies to compare the cost of verification. The stage involved here is turning a given objective and argument into experimental verification for cost evaluation. In order to measure the timing data for an isolated task, measurement techniques can be used. The WCET of isolated tasks can then be compared with the WCET of the task running in the complete system. In order to evaluate this according different software architectures, a similar system needed to be developed for two or more of the software architectures under evaluation.

4.4.4.7 Stage 3 (g) – Basic analysis test

This test is one of the three tests considered for assessing the impact of software architecture or scheduling strategies and cost. For most system problems there already exist a wide range of analysis and test methods that can be applied to evaluate the key properties and objectives. For example, with respect to timing, there is schedulability analysis.

4.4.4.8 Stage 3 (h) – Scenario-based assessment

The second assessment technique is scenario-based assessment. This method can be used when there is a lack of precise information that would be suited for evaluation. Scenarios of change can then be applied to a particular evaluation, and the impact of costs evaluated from two perspectives: i.e. whether TT architecture has a higher impact than ET architecture; and if TT architecture has a higher impact, then by what percentage over ET architecture, and why has the difference occurred?

4.4.4.9 Stage 3 (i) – Dynamic test

For real-time systems, correct system functionality depends on logic as well as on timing correctness. Static analysis alone is insufficient to verify the temporal behaviour of real-time systems. Dynamic testing and measurement-based approaches are important for examining runtime behaviour, based on an execution in the application environment as noted by Schütz (1993):

"Any performance tests and tests intended to evaluate the temporal behaviour of the systems yield only meaningful results if they are conducted on the target system".

4.5 Method comparison

To compare the impact of different software architectures, one has to identify matching factors for evaluation purposes carefully. This can be accomplished by identifying common arguments in software architectures for each of these evaluation criteria. Since most of the evaluation techniques in this study are experimental-based, and there is a range of design choices that can be considered in embedded systems development, a comparative framework to justify the selection and formation of its components and elements is required.

The evaluation framework introduced in Table 4.1 was used as the tool for analysis. The framework is based on using NIMSAD as an evaluation tool. The NIMSAD (Normative Information Model-based Systems Analysis and Design) evaluation framework makes it apparent that the assessment of most methodologies will be assisted by subjecting them to a NIMSAD analysis. NIMSAD is also useful for developers of methodologies, as it enables them to identify conceptual gaps that they may wish to fill and provides useful criticism of design factors requiring further thought. The NIMSAD framework uses the entire problem solving process as the basis

of evaluation and for evaluating methods in any category. According to NIMSAD, methodologies are evaluated according to four elements, which are: the Methodology Context, the Methodology User, the Methodology, and finally Evaluation; the way the methodology evaluates the other three elements. Jayaratna (1994) defines an extensive set of questions. Babar and Gorton (2004) summarised those questions according to a software architecture evaluation. In this study, a list of questions were used as a comparative tool for analysis.

The goal of this evaluation was to provide an overview of current software architecture evaluation methods and to expose if the methods differed in any aspect of the embedded software architecture evaluation. Therefore, a neutral, common and quite extensive NIMSAD framework for method evaluation was utilised to derive the fundamental element categories for the framework. The NIMSAD framework was earlier applied to the evaluation of software engineering methods. The application of the framework to software engineering methods provided a basis for a detailed element definition of categories. With regard to various questions this study tries to address maturity, practicality and the scope of the methods used to identify differences. On the other hand, the goal was also to study if the methods really constitute a method. These elements are considered in the following categories:

Table 4.1 The NIMSAD framework and its interpretation in comparing software architecture evaluation methods

Category	Elements	Questions
Context	Software architecture definition	Does the method explicitly consider a particular definition of SA?
	Specific goal	What is the particular goal of the methods?
	Quality attributes	How many and which quality attributes are covered by the method?
	Applicable stage	Which is the most appropriate development phase to apply the method?
	Input & output	What are the inputs required and outputs produced?
	Application domain	What is/are the application domain(s) the method most frequently applied?
User	Target group	Who are the stakeholders addressed by the method?
	Motivation	What are the user's benefits when using the method?
	Benefits	What are the benefits of the method to the stakeholders?
	Process support	How much support is provided by the method to perform various activities?
	Necessary skills	What skills does the user need to accomplish the tasks required by the method?
	Guidance	How does the method guide the user while applying the method?
Contents	Method structure	What are the activities to be performed and in which order to achieve the goals?
	Software architecture description	What form of SA description is recommended (e.g., formal, informal, particular ADL, views etc.)?
	Evaluation approaches	What types of evaluation approaches are used by the method?
	Tool support	Are there tools or experience repository to support the method and its artefacts?
Validation	Maturity of method	What is the level of Reliability maturity (inception, development, refinement or dormant)?
	Method's validation	Has the method been validated? How has it been validated?

4.5.1 Context

Context pertains to the concept of the method itself and the situation in which the methodology is intended to be used, according to what is considered to be important in the situation. Within this context many possible factors contribute to the perception of the attributes and characteristics of the situation under concern and the environment within which it operates. These factors could potentially influence the identification and definition of the problem in question. Software architecture analysis generally has one of three goals; for example prediction of future maintenance costs, identification of system inflexibility and a comparison of two or more alternative architectures (Babar and Gorton, 2004). Depending on the goal, the method employs different techniques for some of its main steps.

A precise and well-documented definition of a software architecture is critical for a successful software architecture evaluation. It is difficult to define metrics to assess the capability of a software architecture with respect to quality attributes, without precisely describing the software architecture according to a particular evaluation method.

4.5.2 User

Users are those using the software architecture evaluation method. The ‘content’ is supported by methodology that, in turn, is used by the methodology user. Thus, the methodology user is the problem solver or decision maker. It is necessary to know what guides his decisions, what kind of abstract thinking is required from him, how well he gets to know the methodology he utilises, and how he can acquire the necessary skills. These things are included in the evaluation of the methodology user element.

A stakeholder is any person or organisational representative who has a vested interest in a system. The methods studied also vary in terms of number and categories of stakeholders involved in evaluation, including architects, designers, and end users.

4.5.3 Content

Content refers to the method or approach adopted by the user in the transformation of the situation, as represented by the 'content' element of the framework. Jayaratna (1994) mentioned that content comprises three essential phases: problem formulation, solution design, and design implementation. These three phases provide a structured approach to the complex activity of problem solving. In a scenario-based method, for example, the activities include scenario development and scenario evaluation activities. In addition, a tool is required to support the evaluation process and to capture the design artefacts together with the decision rationale, measurement and administrative information.

4.5.4 Evaluation

To prove the method, an evaluation was required. Evaluation provides a measure for the effectiveness of the 'content' and the 'user' within the particular 'context', and the degree of success achieved when resolving the perceived problem. In the evaluation, questions are directed according to the application of three elements of the framework, to determine the potential impact of the transformation upon the context of the organisation, the potential impact of the 'user', and upon the outcome of the transformation and the content, rationale and direction of the transformational process.

Software architecture evaluation methods can also be compared from the perspective of maturity as this may foster confidence in method users. Thus, the existing evaluation methods can be classified in any of the four maturity phases of the software evaluation

methods lifecycle, namely requirement phase, the design phase, the implementation phase and the testing phase.

4.6 Overview of software architecture evaluation methods using NIMSAD elements

4.6.1 Scenario-based architecture analysis method

Kazman introduced the software Architecture Analysis Method (SAAM) in 1993 (Kazman et al., 1994). The goals of SAAM were mainly geared to evaluating software architecture against desired quality attributes. SAAM was developed for modifiability and used for various quality attributes.

The most appropriate time to apply SAAM is after the requirement phase and before the implementation phase. Designers, developers, software architecture descriptors, and quality requirements are the main inputs when using this method. The outputs of the method include mapping between scenarios and software architecture components, and the anticipated amount of effort associated with each changing scenario.

SAAM involves different users, such as the architect, developer, maintainer and product manager. SAAM provides a number of techniques to perform various activities associated with this process, such as classifying quality attributes, eliciting scenarios, and scenarios.

SAAM follows six steps: scenario development, architecture description, scenario classification and prioritisation, individual scenario evaluation, scenario interaction, and overall evaluation. In situations involving comparing multiple software architectures, scenarios are assigned a weighting to determine the overall rank of different software architectures. SAAM evaluates each scenario by mapping it onto a software architecture description and investigating whether the software architecture supports it

(direct scenario) or not (indirect scenario). The cost of accommodating each indirect scenario is estimated by counting the number of changes required. Scenario interaction analysis reveals whether the inclusion of multiple indirect scenarios affects the same components, and is a sign of poor separation of concern. SAAM is a mature approach, which has been validated using different case studies, such as user interface development environments, key word in context (KWIC) systems and embedded audio systems (Kazman et al., 1994).

4.6.2 Architecture level modifiability analysis

A unified architecture-level analysis method, which focused on modifiability was introduced in 2003 (Bengtsson et al., 2004). Architecture level modifiability analysis (ALMA) has been developed around a conceptual framework that is known as goal-oriented evaluation. Goal setting is the most important activity associated with this method, as the remainder of activities are performed in the light of the evaluation goals. The specific goal of this method is to address modifiability related issues at the software architecture level, such as the maintenance of cost prediction, risk assessment or software architecture selection.

The main benefits of using ALMA are identification of software architecture risks, and an estimation of the efforts required to accommodate change, or the selection of an optimal SA. Inputs include software architecture specifications and quality requirements. ALMA has successfully been applied in telecommunications, information systems, embedded systems and medical domains. ALMA usually involves only a small set of users, namely the development team and software architect. Five main activities of ALMA include: setting a goal, describing software architecture, eliciting scenarios, evaluating scenarios and interpreting results. ALMA uses impact analysis to

evaluate software architecture against change scenarios. Impact analysis is performed by identifying the components affected by these scenarios, deducting what modifications are required, and determining ripple effects. ALMA provides a framework with which to describe results quantitatively. ALMA has been applied across several industrial cases including software architectures at Ericsson Software Technology, the Dutch Department of Defence and the Dutch Tax and Customs Administration (Bengtsson et al., 2004).

4.6.3 Performance assessment of software architecture

Williams and Smith (1998) presented a scenario-based method with which to assess the performance of software architecture; known as Performance Assessment of Software Architecture (PASA). The aim of this method is to help developers select a suitable architecture. PASA includes performance sensitive software architecture styles and anti-patterns, as analysis tools, formalising the software architecture analysis activity and the performance engineering process.

The specific goal of PASA is to assess the capability of software architectures with respect to the performance and quality objectives of a system. PASA guides the software architecture analysis activity, using performance related scenarios as a source of reasoning, workload specifications, software plans, execution environment, resource requirements and processing overheads. PASA can be applied early in the development cycle, post-deployment, or during an upgrade of a legacy system. Typically, only a development team is involved. PASA has ten steps including the process overview, architecture overview, identification of critical use cases, selection of key performance scenarios, identification of performance objectives, architecture clarification and

discussion, architectural analysis, identification of alternatives, presentation of results and economic analysis (Williams and Smith, 2002).

This method incorporates both qualitative and quantitative techniques, to illustrate the potential risks that may be inherent in software architecture. This method also demonstrates how scenarios can be useful when characterising run-time quality attributes such as performance. PASA itself or techniques related to it have been validated in different case studies. The method has been applied to embedded systems, real-time systems (Moreno et al., 2008), and within the financial domain.

4.6.4 Architecture trade-off analysis method

The Architecture Trade-off Analysis Method (ATAM) was initially positioned as a software architecture design method to support design trade-offs. Later, it was presented as a model for software architecture analysis (Babar and Gorton, 2004).

The specific goal of ATAM is to promote a disciplined reasoning mode, for analysing software architecture's capability with respect to multiple quality attributes. It also helps make trade-offs between competing attributes. ATAM will be applicable to any stage of the software development, however, it is most effective when applied at an early stage of the software development lifecycle. The inputs for ATAM include business goals, software specifications, and software architecture description. The outputs of ATAM are a list of scenarios, defining any risks, sensitivity points and trade-off points (Kazman et al., 1999).

The application domains include combat systems, web-based systems and embedded systems. ATAM claims to provide several technical as well as social benefits. ATAM also involves various users or stakeholders, including an evaluation team, customer representatives and an architecture team (Kazman et al., 2000).

ATAM is a heavy weight process comprising four phases: presentation, investigation and analysis, testing and reporting. There are nine activities in these phases: present the ATAM, present business drivers, present architecture, identify architectural approaches, generate a quality attribute utility tree, analyse architectural approaches, brainstorm and prioritise scenarios, analyse architectural approaches and present results (Kazman et al., 2000).

ATAM does not prescribe any specific evaluative techniques. Rather, it uses various theoretical models to identify the quality attribute communities for quantitative analysis and to apply qualitative reasoning heuristics that are documented according to attribute-based architectural styles, architectural patterns, tactics and quality sensitive scenarios. ATAM is considered to be a mature approach, that has been validated in different domains such as Battlefield Control Systems (BCS) and AGV transportation systems (Boucké et al., 2006).

4.6.5 Goal-based requirement analysis method

In 1996, Anton developed a goal-base method, entitled the Goal-based requirement analysis method (GBRAM) (Anton, 1996). The main aims of this method were to identify, elaborate, refine and organise goals according to requirement specifications. Obviously, this method can be applied at the requirement phase of the development lifecycle. The users involved in this analysis are practitioners and software architects.

There are two process elements that they focused on predominantly, goal analysis and goal evolution. Goal analysis concerns exploration of documentation as according to the organisation and classification of goals, whereas goal evolution concerns the way goals change from the moment they are first identified to the moment they are operationalised in a system specification.

This method has been used in a Career Track Training System (CTTS), which was part of the business re-engineering project for an Air Force Base (AFB).

4.6.6 Bate's software architecture evaluation method

Bate (2008) introduced a systematic approach to evaluate a design trade-off based on the goal structuring notation (GSN). The goals of Bate's method were mainly to present a technique to understand the software architecture trade-off, so that the cost of design changes could be estimated.

This method can be applied in the software requirement phase and design phase. The method used for deriving the trade-off analysis problem is based on a goal structuring notation (GSN). Designers, developers, software architecture description and quality requirements are the main inputs for this method. The outputs of the method include mapping scenarios and software architecture components, and the anticipated amount of effort associated with each change scenario.

Bate's method involves different users, including the software architect, designers, developer, and maintainer. The process starts be employing GSN to decompose the top-level objectives of the system in a hierarchical tree-like fashion. The decomposition is continued until the objectives reach a suitably low-level; they are then employed to measure how well specific individual objectives are met. The detailed process informing the method is presented in Figure 4.1. There are four main activities included here, presenting the current design, producing an argument for the key objectives, extracting information from the argument (i.e. design choices and assessment criteria) and decomposing the design. The assessment criteria can then be converted to a quantifiable measure, and appropriate weighting applied.

Bate's method is not considered to be a mature method, since it has not been applied to a real application. The introductory paper only demonstrated a case study outlining the task allocation problem (Bate, 2008).

4.7 Conclusion

This thesis introduces a method that can be used to evaluate the impact of software architecture on the cost of design, implementation and testing. Before an evaluation can be made, it is necessary to identify the costs that are representative for the software architecture. Once cost is identified, assessment can be done using experimental approaches based on dynamic testing, scenario-based testing and basic analysis testing. This method can also present an unambiguous comparison of a variety of software architectures and scheduling strategies, for use with real-time systems. The experimental work in the research conducted was undertaken to perform evaluation as an integrated entity, based on which a novel evaluation model of real-time software architecture can be proposed. From Chapter 5 onwards, the detailed costs of design, implementation and testing in ET and TT architecture are examined.

Chapter 5
Assessment of Implementation Cost

5.1 Introduction

An extensive literature review, covering the most important aspects related to the first hypothesis tested in this research, has been presented in Chapter 3. The drawbacks of pre-emptive scheduling have been discussed in light of the reviewed literature. In this chapter, the implementation costs for both co-operative and pre-emptive schedulers will be evaluated. The results of this assessment, will then be related to the costs of scheduler implementation, such as overheads, CPU utilisation, memory utilisation and lines of code in reference to the first hypothesis.

5.2 Problem statement

The performance of pre-emptive versus co-operative based schedulers is assessed, in the light of implementation costs, for which the costs need to be defined, measurable parameters, and a suitable method to measure the parameters determined. This is followed by a discussion on the comparative analysis involved.

In doing so, the scheduler needs to be evaluated in line with the following hypothesis:

Use of limited pre-emptive scheduling in a design results in lower testing costs than the use of fully pre-emptive scheduling and co-operative scheduling in the implementation phase.

5.3 Problem description

In order to assess the problem outlined in section 5.2, software implementation of the schedulers, suitable tool-sets to evaluate their performance, and the necessary measurement equipment are required, as shown in Figure 5.1.

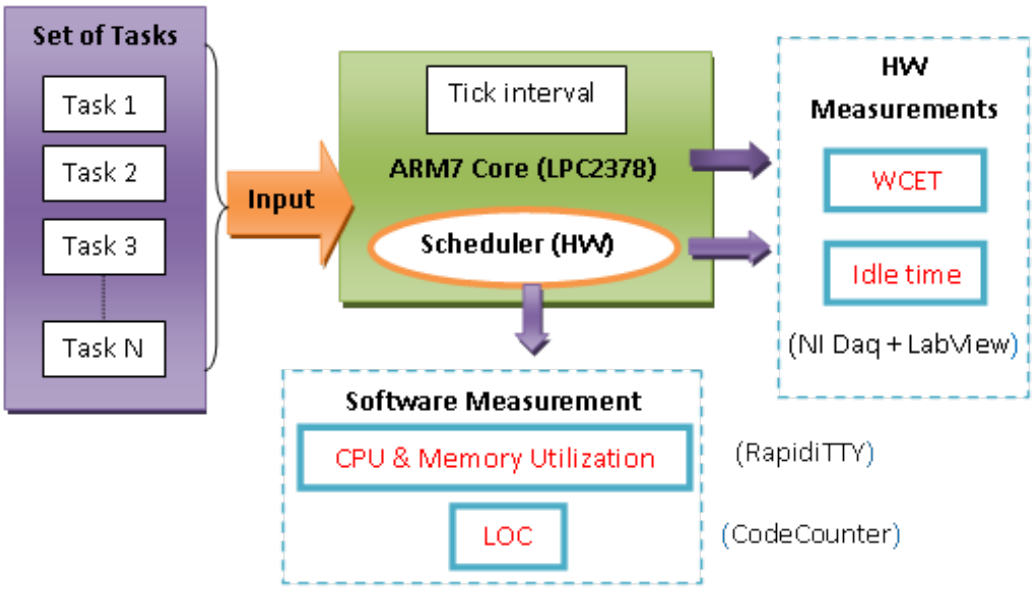


Figure 5.1 Evaluation of implementation cost, using hardware and software based performance measures

In this instance, TT architecture becomes the focal element for this study. To evaluate the implementation cost of co-operative, limited pre-emptive and fully pre-emptive scheduling, it has been feasible to consider TTC, TTH and TTP, as the schedulers for this assessment. The schedulers will be stored in the ARM7 core board. Some examples of tasks are presented in order to conduct this experiment. To measure the implementation cost, tools such as RapidiTTy (RapidiTTy, 2010), Labview (National, 2010) and CodeCounter (Code, 2006) are required to measure CPU and memory utilisation, lines of code, worst case execution time (WCET) and idle time. Complete descriptions of these tools are provided in Appendix A.

The available software tool RapidiTTy (RapidiTTy, 2010), which is based on TT technology, allows developers to create, test, and maintain reliable and resource efficient embedded systems, as well as to perform timing analysis, does not provide any measurement for idle-time. Thus, suitable hardware for the purpose of measurement

5.4 Adopted methodology

5.4.1 TTC, TTH and TTP schedulers implementation

Schedulers handle the interrupts in Time Triggered Co-operative (TTC), Time Triggered Hybrid (TTH) and Time Triggered Pre-emptive (TTP) architectures. Codes for handling priority allocation are not required in TTC. However, they are required for TTP in order to obtain a decision; whereas, TTH runs several tasks in a co-operative manner, but also allows one pre-emptive task allocation. It is important to understand how each real-time scheduler is programmed and executed within the processor system. Most real-time kernel operations are based on subroutines that use parameter-passing techniques, known as context switching. These can be useful for handling external aperiodic or sporadic events, and to provide accurate timing for system operations. The schedulers, along with their associated overheads are depicted in Figure 5.2.

TTC architecture provides the simplest way to handle multi-tasking, as shown in Figure 5.2(a). The architecture does not allow pre-emption. When the processor receives a tick interrupt signal (which is triggered by the overflow of a hardware timer), it saves essential registers to the stack, obtains information identifying the interrupt type, and then branches out to the interrupt service routine (ISR). This process causes latency when handling interrupts, varying in a range between 1-100 microseconds (Cooling, 2003). The ISR will execute the scheduler `Update( )` function, to update the tick count, and determine which tasks are due to run and sets the corresponding flags. After this, the `Dispatch( )` function will prompt all the tasks in the task array to execute.

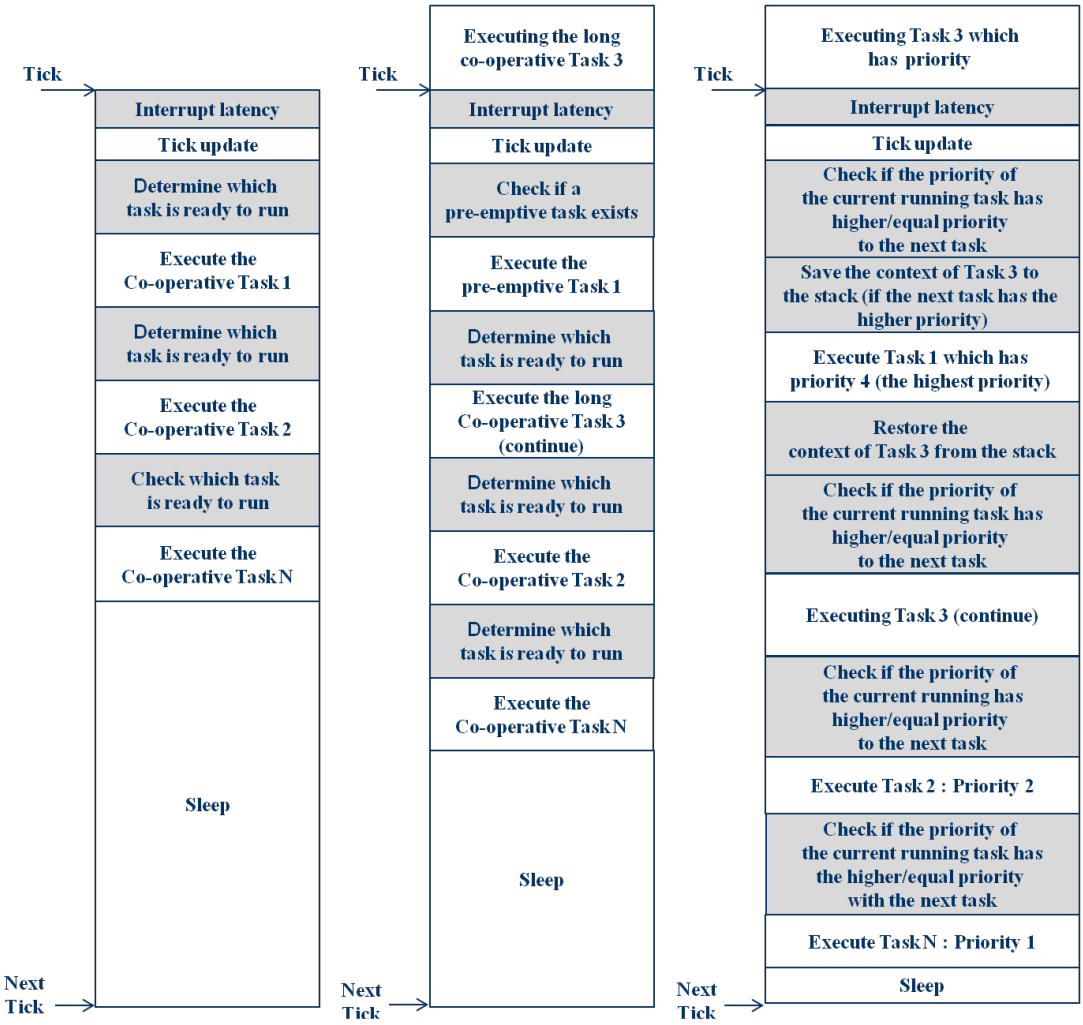


Figure 5.2 TTC, TTH and TTP scheduling operations with associated overheads

Such arrangements produce an overhead to the scheduler and this can be increased based on the number of tasks that are to be implemented in a given tick interval. When the ISR or scheduler functions remain inactive, the system is usually placed in a low-power sleep or idle mode. Once it enters the idle mode, the system only wakes up upon the occurrence of the next tick interrupt.

TTH and TTC are very similar in approach, except for at the start of the scheduler, as shown in Figure 5.2(b). Unlike TTC, TTH supports single task pre-emption activity in its implementation. Note that the scheduler only supports a sporadic type of task, that once pre-empted, will run periodically. Assume that a currently executing long co-

operative task reaches the tick interrupt (see Figure 5.2(b)). The tick is updated on an interrupt from the real-time clock. This results in the current task being replaced by the tick handler. All current contexts must be saved before this handler is loaded up and implemented, which, as a result, incurs extra overheads. After the tick interrupt occurs and all the starting scheduler operations take place, the TTH scheduler directly checks for the existence of a pre-emptive task and executes it to completion (if the pre-emptive task exists). If it does not exist, the scheduler will execute the other tasks co-operatively. The checking process is programmed in the `update()` function in the TTH scheduler, thus causing extra overhead.

Many real-time applications require more flexibility and responsiveness than can be provided by TTC and TTH schedulers. TTP schedulers are fixed priority pre-emptive schedulers in which all the task priorities are assigned to tasks statically offline. It also supports mutexes that implement priority ceiling protocols (Sha et al., 1990, Sha et al., 1991), which can be used when various tasks seek to use the common resources simultaneously. When all task priorities are the same, the TTP scheduler will behave similarly to the TTC scheduler. The tick is updated on an interrupt handler, and then a scheduler `update()` function is used to assign the processor to the task with the highest priority. Any lower-priority task running at this time will be pre-empted and placed back in the queue. This process is then repeated until the lowest priority task completes its execution. Complete TTP operations are described in Figure 5.2(c). Obviously, depending on which context-restore action is taken, extra time is needed to prioritise tasks.

The main element used for constructing a context switch in TTP is a ‘task control block’ (TCB). It holds the task state information or context, such as registers, program status register (PSR), link register (LR) and program counter (PC) when the task is pre-

empted. When the task regains control of the processor, the TCB allows the task to resume execution exactly where it was left. The structure of the TCB is shown in Listing 5.1, and all the TCBs contain dynamic information. Thus, they have to be located in RAM.

```
/ Task Control Block structure
typedef struct {
      uint32_t PSR;
      uint32_t R0;
      uint32_t R1;
      uint32_t R2;
      uint32_t R3;
      uint32_t R4;
      uint32_t R5;
      uint32_t R6;
      uint32_t R7;
  uint32_t R8;
  uint32_t R9;
  uint32_t R10;
  uint32_t R11;
  uint32_t R12;
  uint32_t LR;
  uint32_t PC;
} tcb_t;
```

Listing 5.1 Task control block in TTP architecture

Before a higher-priority task can be executed, all the current contexts of the lower-priority task must be saved on the stack, and the context information for the higher-priority task must be loaded and executed. All the task control information is stored in a TCB when the task is not running on the processor. When the execution of higher-priority task completes, the content of the TCB for the lower-priority task is loaded back into the processor. In addition, due to the change in the PC, code execution resumes from the point at which the PC is pointing. The context switch code is written in assembly language because most C compilers cannot manipulate processor registers directly from C (Labrose, 2006). The process is illustrated in Figure 5.3.

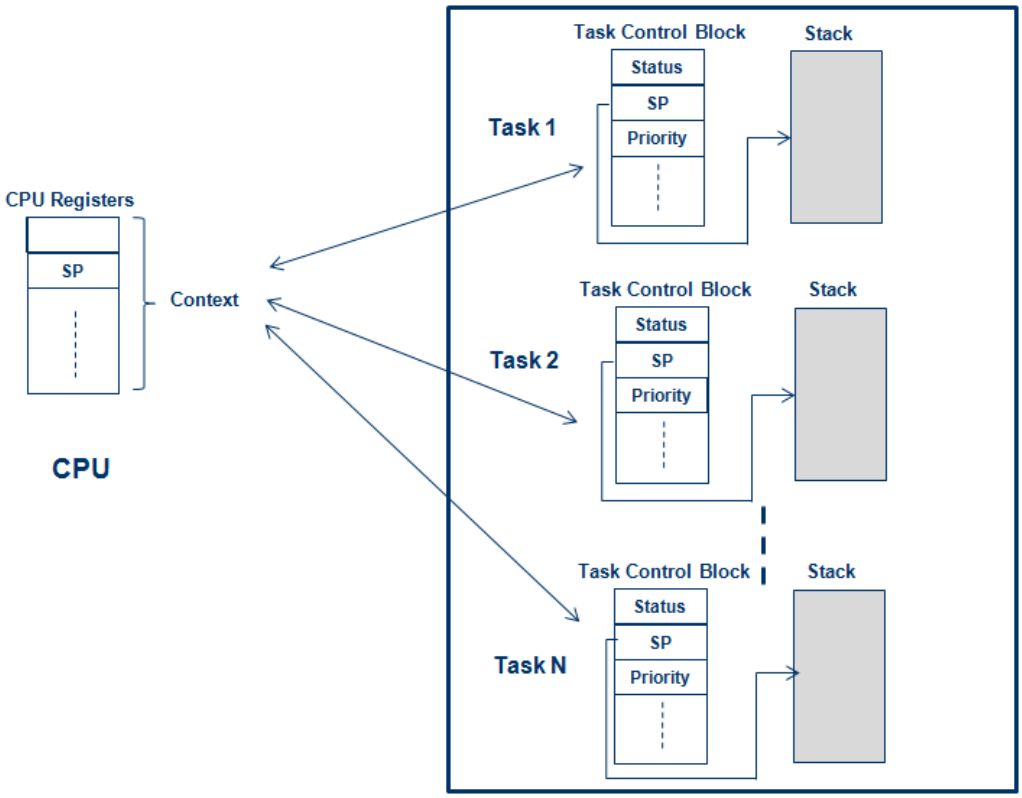


Figure 5.3 Context switch operation (Labrosse, 2006)

5.4.2 Implementation costs definition

An implementation cost is defined as the time it takes for the scheduler to run a particular number of tasks, and according to the ease of implementing a scheduler. In order to deduce the cost, the parameters, i.e. overheads, LOC, memory and CPU utilisation of the TTC, TTH and TTC schedulers, need to be measured. Any of the parameters showing high values indicate that the scheduler consumes more cost or effort. The parameters are measured against increments in the number of tasks from 1 to 100 tasks. According to previous experimental work, this number of tasks is sufficient for observing and analysing the impact on system behaviour (Short, 2010).

5.4.3 Overhead measurements

There are three important parameters required to measure scheduler overhead:

- Worst case execution time of tasks (WCET [i]): obtained by setting a pin high at the beginning of the task and low at the end of the task. The widths are measured using a data acquisition NI card with LabVIEW software (as shown in Appendix E). In each study, 10000 consecutive pulse widths are measured and recorded. This is found to be sufficient for the purpose of this study.

- Total worst case execution time of tasks: obtained by combining the worst-case execution times for all tasks.

- Idle time: obtained by making direct measurements from the hardware. A pin was set to high at the beginning of the interrupt service routine (ISR) of the tick interrupt and to low before the scheduler enters 'idle mode'. The measurement of the widths used a similar technique to (i).

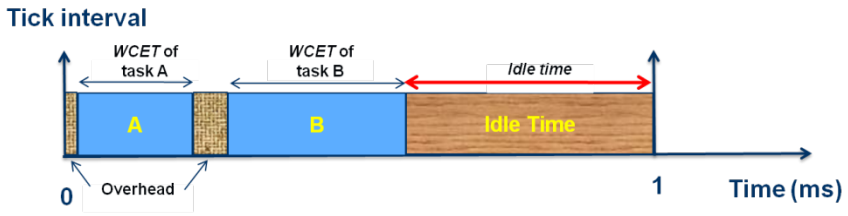


Figure 5.4 Overhead measurement for time-triggered co-operative scheduler

In order to reduce operating power when a system is not using ISR or scheduler functions, the scheduler is usually placed in a low-power sleep mode (Pont, 2001). Most processors have an idle mode so that battery life can be increased. Once in the idle mode, the system will only wake up when the next tick interrupt takes place. The overhead value for TTC is straightforward to determine, since all the tasks run co-operatively and must complete execution within a tick interval. The formula is given as follows:

$$\text{Overhead} = \text{Tick\_interval} - (\text{Idle\_time} + \sum_{i=1}^{n} \text{WCET}[i]) \quad (5\text{-}1)$$

As shown in Figure 5.4 the overhead can be measured as:

$$Overhead = (1 - (idle\_time + (WCET\ of\ task\ A\ and\ WCET\ of\ task\ B))) ms \quad (5\text{-}2)$$

Figure 5.5 depicts overheads in time-triggered pre-emptive scheduling. In this scheduling scheme, high priority tasks are allowed to pre-empt the current run task, which has lower priority. For example, the figure shows the situation in which a short periodic pre-emptive task interrupts a long periodic task.

Equation (5-1) can be used as a basic formula to calculate the overhead of pre-emptive scheduling. The scheduling scheme allows pre-emption over low priority tasks and also supports long execution time tasks, which may exceed the tick interval; therefore this calls for some modifications to be applied.

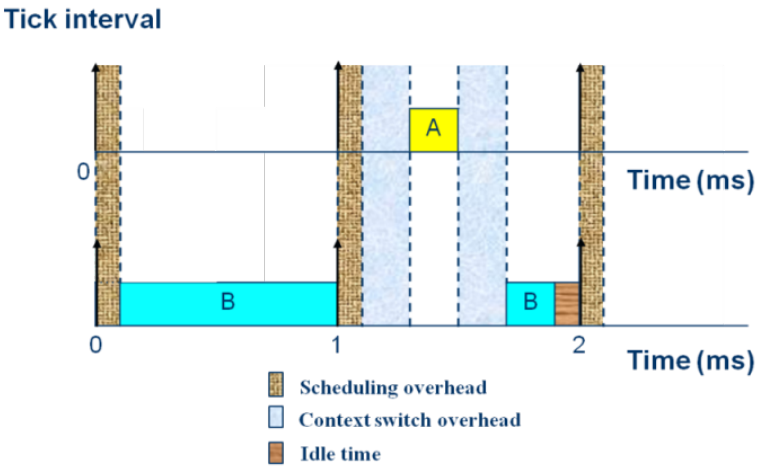


Figure 5.5 Scheduling and context switch overheads in time-triggered pre-emptive systems

As a long task execution time could exceed the tick interval, or be pre-empted by a high priority task, the overhead calculation needs to consider the duration of a tick interval in which all the tasks completely finish their execution. Hence, the overhead of the pre-emptive scheduler is measured as in Equation (5-3):

$$\text{Overhead} = \text{Duration of tick interval to complete execution of all tasks -}$$
$$(Idle_{time} + \sum_{i=1}^{n} WCET\ [i]) \tag{5-3}$$

Note that in many designs, a pre-emptive task will be used for periodic data acquisition, typically through an analogue-to-digital converter or similar device. The task may execute every tick interval. Therefore, all such task execution time need to be taken into account when measuring the real-time overhead of TTH and TTP scheduling.

$$\text{Overhead} = \text{Duration of tick interval to complete execution of all tasks} -$$
$$(Idle\_time \sum_{i=1}^{n} WCET\ [i] + \text{WCET of other tasks in the duration}) \tag{5-4}$$

An example will be given in the analysis results section.

5.4.4 Measuring CPU and memory utilisation using the simulation tool

Processor and memory utilisation are taken from the RapidiTTy simulation tools (Rapiditty, 2010). Visualisations of CPU and memory requirements are shown in Figure 5.6 and Figure 5.7.

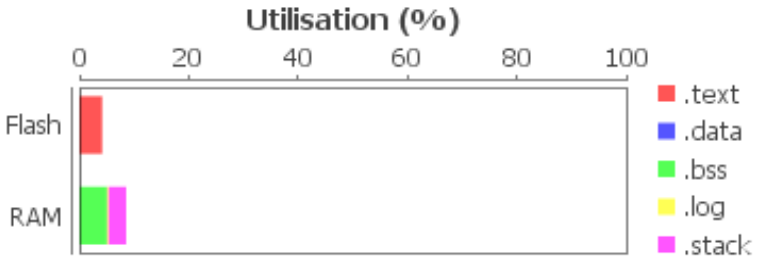


Figure 5.6 Sample of visualisation of memory utilisation

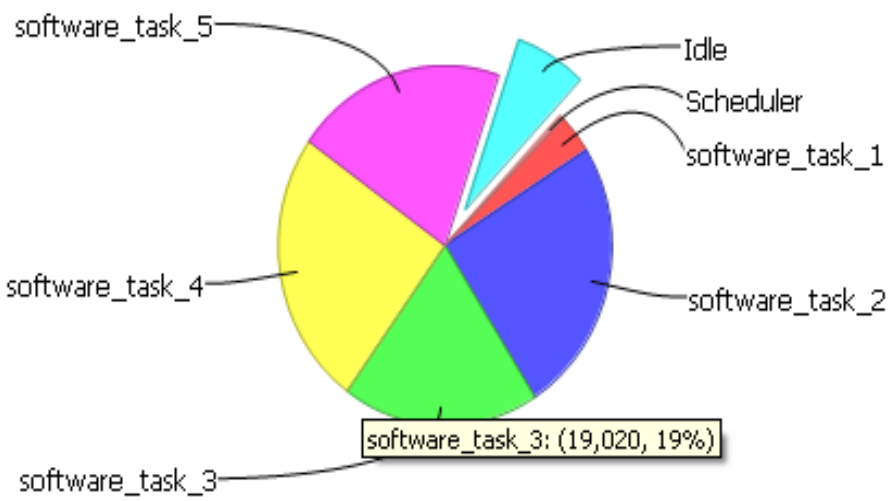


Figure 5.7 Sample of CPU Utilisation for 5 tasks

5.4.5 Scalability analysis

The idea of this study is to identify the suitable number of tasks required for analysis in order to investigate the effect on the performance in pre-emptive and co-operative schedulers. A few research papers and journals, related to performance measurement for uniprocessor systems, have been reviewed in order to determine a reasonable number of tasks for this analysis (Liu and Layland, 1973, Locke et al., 1991, Audsley et al., 1993, Arakawa et al., 1993, Katcher, 1993, Buttazzo, 2005, Burns and Wellings, 1995, Bini, 2003, Xu and Parnas, 2000, Devi, 2003, Gendy, 2008, Short, 2010).

On average, most studies focused on analysis for systems with 20 tasks or fewer. For example, Locke et al. (1991), Audsley et al. (1993), Arakawa et al. (1993), Katcher (1993) and Burns and Wellings (1995) applied their analyses to a small avionics case study which consisted of 18 tasks. In addition, from the analysis made by Bini (2003), Xu and Parnas (2000) and Katcher (1993), the performance of schedulability analysis can be evaluated using only 20 tasks. For example, with an analysis of 20 limiting

tasks, Katcher claimed that TT with a co-operative scheduler outperforms ET with pre-emptive scheduler.

The higher numbers of tasks that have been studied in previous research range from 80 tasks (Katcher et al., 1993) up to 1000 tasks (Devi, 2003). These studies aimed to compare the performance of a small and a large system in a uniprocessor. However, in Gendy's (2008) analysis, 50 tasks were used to represent a large system. In a more recent study, Short (2012) used from 4 up to 256 tasks for a performance analysis (Short, 2012). Devi classified the pseudo-polynomial-time test into two groups of tasks in order to analyse their accuracy and efficiency in the determination of schedulability. The first group was equipped with a thousand tasks per task set, while the other groups were allocated a hundred tasks per task set.

Based on the previous work (Katcher, 1993, Devi, 2003), the performance of a scheduler can be sufficiently observed when a system has hundreds of tasks. For example, Katcher's analysis has shown that the breakdown utilisation decreases exponentially as the number of tasks grows (Katcher, 1993). Therefore, a comparison of scheduler implementation for time-triggered co-operative and pre-emptive scheduling algorithms was made by generating a number of tasks in a task set which varied between two and a hundred.

5.4.6 Measuring LOC using the Code Counter Software Tool

LOC is calculated by measuring all the lines containing program headers, declarations, and executable and non-executable statements. Comments and blank lines are excluded from the calculation (Conte, 1986). Table 5.1 shows an example of LOC measurement.

Table 5.1 LOC measurement

Source code line	LOC	Comment	Blank
// Change the LED_pin from OFF to ON (or vice versa)		*	
if (LED_state == 1)	*		
{	*		
			*
LED_state = 0;	*		
PORT_Pin_Write(LED_pin1, 1); // Set to off	*	*	
}	*		

In this example, the total LOC is 5, the number of comments is 2 and the number of blanks is 1. All the scheduler's program has been developed using the C programming language. Thus, the implementation files (*.h* to *.c*) in the scheduler, system and task folders have also been measured.

The LOC of the scheduler implementation has been counted using Code Counter Software Tools. The sample results of the code counter are shown in Figure 5.8.

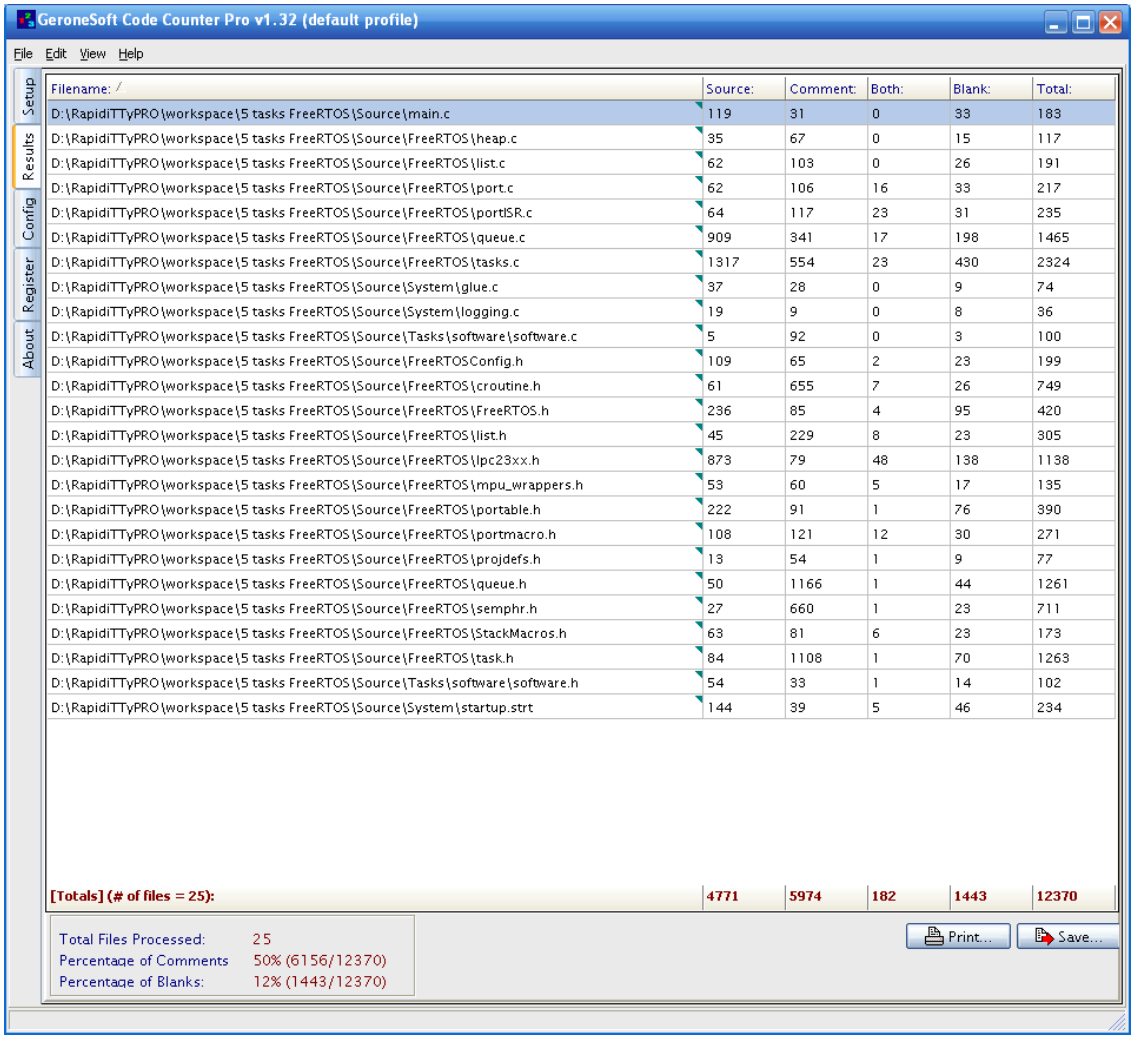


Figure 5.8 Code Counter (Code, 2011)

5.5 Experimental setup

5.5.1 Hardware platform

It is assumed in this project that the target platform for the embedded systems is a small microcontroller, which will be programmed using C language. In particular, the empirical studies reported in this thesis for the single-processor systems have been conducted using an LPC-2378STK development board supporting an NXP LPC2378 (NXP, 2011) processor from Olimex (Olimex, 2009). The LPC2378 is a modern 32-bit microcontroller with an ARM7 core. The processor was used as an oscillator with a frequency of 12 MHz, and a CPU with a frequency of 60 MHz. The oscilloscope has

been deployed to visualise the I/O pin voltage, which was required during overhead measurement. Figure 5.9 shows the hardware used in this experiment.

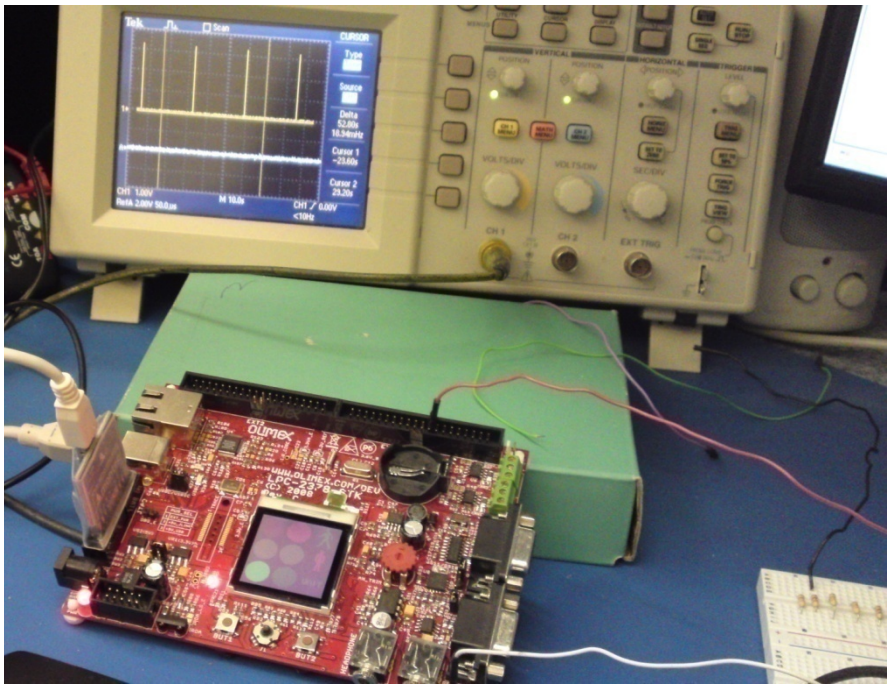


Figure 5.9 Measurement using a hardware based setup

The GNU C compiler for ARM7 operating in Windows has been used. Meanwhile, the TTE Systems' RapidiTTy (v2.0) has been used as the IDE and simulator. This tool provides a graphical presentation of timing analysis, hence enabling designers to visualise the behaviour of the tasks running on the TTC, TTH and TTP schedulers. Figure 5.10 shows a sample of timing analysis in the RapidiTTy tool (RapidiTTy, 2010).

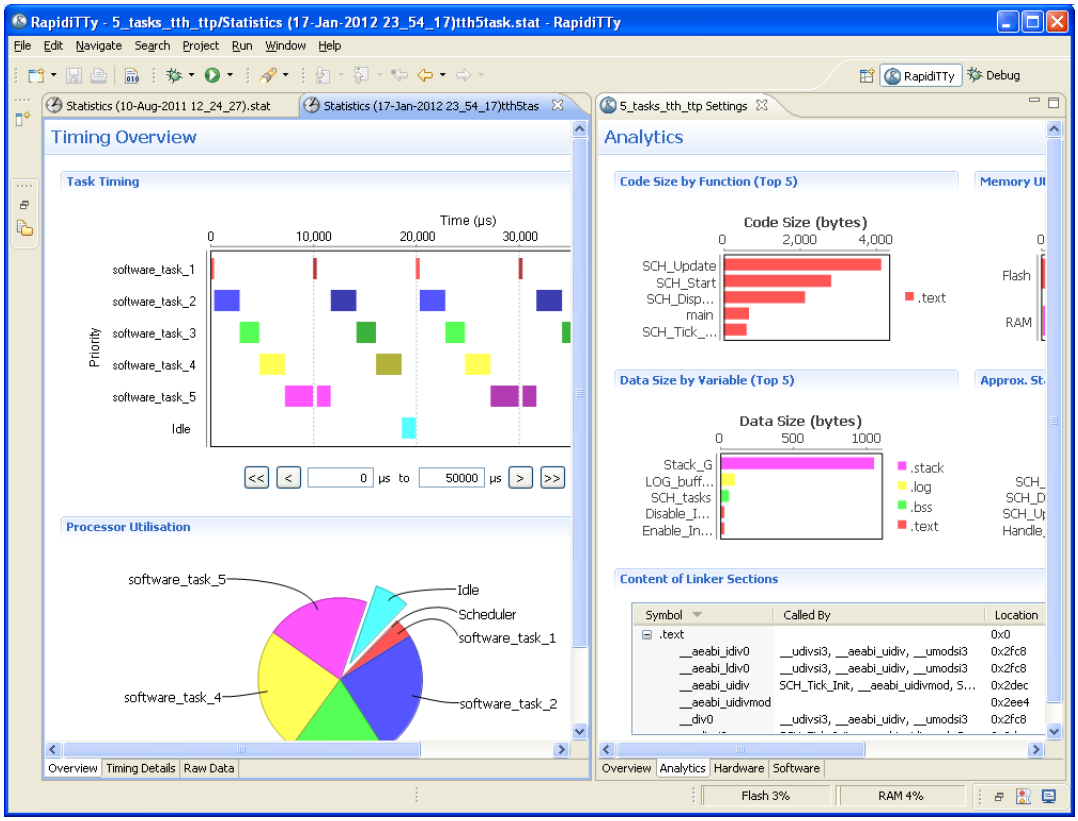


Figure 5.10 RapidiTTy timing analysis (RapidiTTy, 2010)

5.5.2 Timing analysis tools

Although the RapidiTTy v2.0 IDE tool supports the timing analysis for TT schedulers, by acquiring detailed information about timing behaviour for each individual task using logging techniques, the results produced can be affected by the overhead of logging timing data mechanisms. In fact, some timing data cannot be acquired due to hardware limitations. For example, the JTAG debugging connection, which is used to acquire timing data, is established when the scheduler is in active mode. Hence, if the scheduler enters idle mode, the connection will be lost. However, the tool provides graphical representations of timing statistics for each task so that designers can visualise the timing behaviour of the tasks easily.

Note that the calculations of overhead require the measurement of idle time. Obviously, the timing analysis of this tool is unable to provide such timing information. Therefore,

in order to address these issues, the timing data measured using a National Instruments data acquisition card 'NI PCI-6035E' has been used in conjunction with LabVIEW 2009 software (as shown in Figure 5.11and Figure 5.12). An oscilloscope has been used to visualise the data.

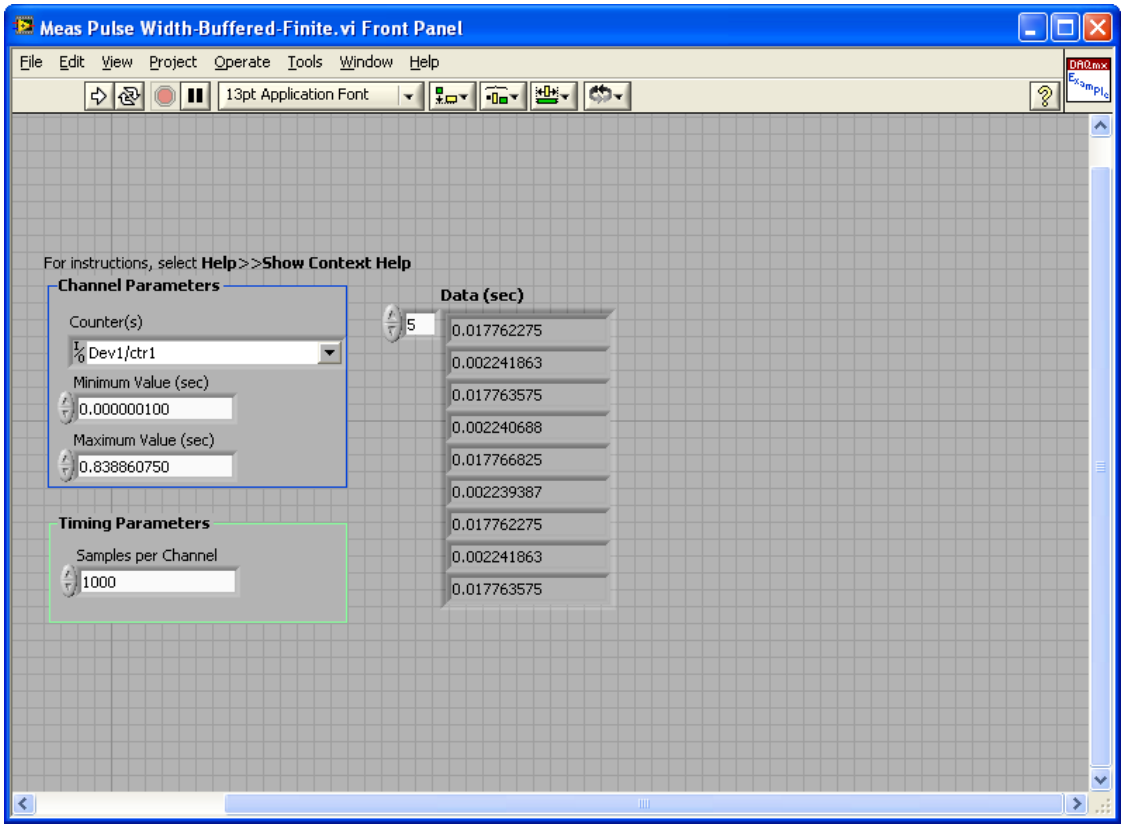


Figure 5.11 Front panel for measuring Pulse Width-Buffered-Finite (National, 2010)

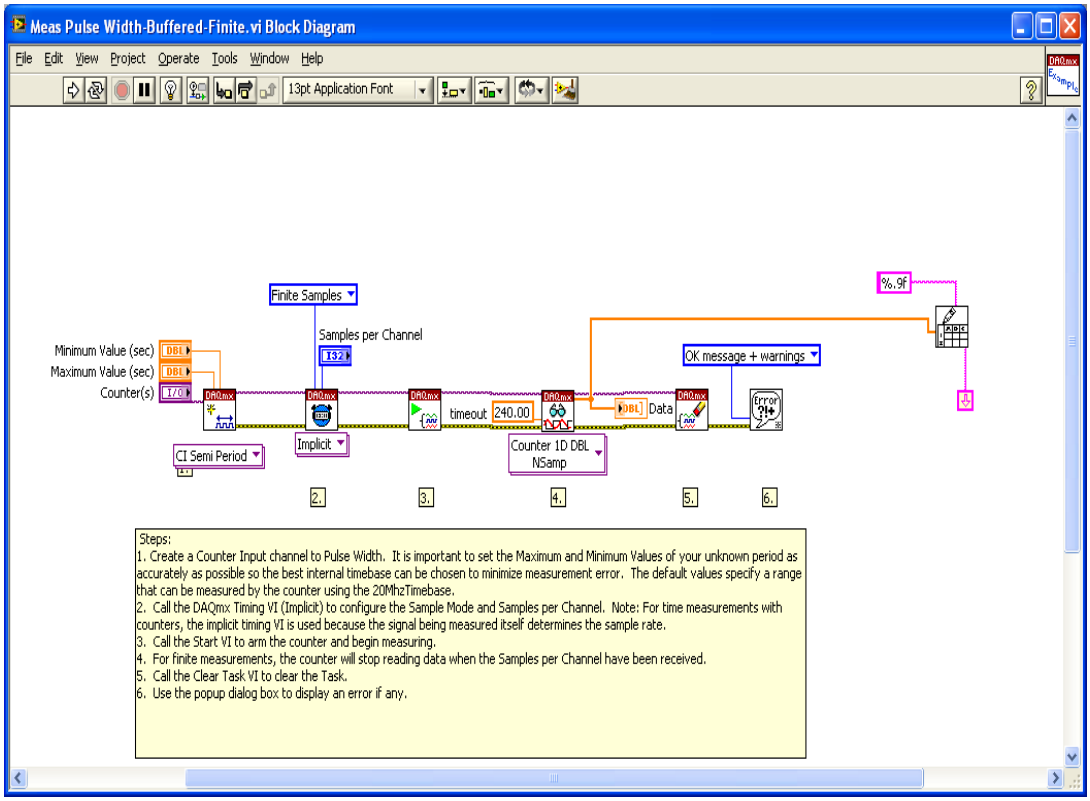


Figure 5.12 Block diagram for measuring Pulse Width-Buffered-Finite (National, 2010)

5.5.3 Generation of task set

In order to explore the effect of real-time overhead in TTC, TTH and TTP scheduling, firstly a small number of tasks has been generated to run in each scheduler. Table 5.2 shows the sample of task specifications for one task to be tested in TTC, TTH and TTP scheduling. The reason for starting the test with one task is to observe the minimal differences in WCET for each task when running in different scheduling algorithms. It has been assumed that the delay of all tasks is 0 and that all the tasks which start their execution in a given tick interval will complete their execution before the next tick occurs. Such a restriction is not an essential requirement in the TTH and TTP designs as the schedulers have the ability to handle “long tasks”.

Table 5.2 Task specifications for 1 task

Software architecture	Task number	Period (Tick)	WCET (us)
TTC	1	1	814.5
TTH	1	1	814.5
TTP	1	1	814.5

Table 5.3 presents a task set consisting of 5 tasks. In the TTC design, the WCET of tasks has been designed to have duration less than the tick interval. On the other hand, the tasks in the TTH and TTP schedulers have been designed to have longer WCET than the tick interval. Furthermore, the priorities of the tasks need to be defined in the TTP scheduling. Task 1 had the highest priority, followed by Task 2, Task 3, Task 4 and Task 5. In the TTH scheduler, only one short task is allowed to pre-empt a task. The other tasks run co-operatively and have equal priorities lower than that of the pre-empting task. Thus, Task 1 has been set to pre-empt the task specifications while the other tasks have been set to co-operate. Note that Task 2 has the longest WCET and was pre-empted by Task 1 at runtime.

Table 5.3 Task specifications for 5 tasks

Software architecture	Task number	Priority (5 = High)	Period (Tick)	WCET (us)
TTC	task 1		1	543
	task 2		1	543
	task 3		1	271.5
	task 4		1	814.5
	task 5		1	814.5
TTH	task 1	Pre-emptive	1	271.5
	task 2	Co-operative	1	1900.5
	task 3	Co-operative	1	1900.5
	task 4	Co-operative	1	1900.5
	task 5	Co-operative	2	4072.5
TTP	task 1	5	1	271.5
	task 2	4	1	1900.5
	task 3	3	1	1900.5
	task 4	2	1	1900.5
	task 5	1	2	4072.5

All the tasks have been created as dummy tasks. The WCETs of tasks have been defined as constants by using a "software delay" technique, in order to control the execution of each task. For example, to implement a task function, the code in Listing 5.2 was entered in the function.

```
void Task_N(void)
{       const uint32_t counter = M;
           for (x = 0; x <= counter; x++)
              {
               }

}
```

Listing 5.2 Sample of the task_N function

The duration of Task N was adjusted using the "counter" value. The software delay contains a *for-loop* structure implemented particularly to generate *T* microseconds delay (approximately) based on the value of the counter. The relationship between the counter value and the WCET is shown in Figure 5.13. As the counter value increases, the execution time of the task increases. For example, when the counter is set to 1000, the execution time is equal to 271.5 microseconds. The execution time has been measured using the NI card and Labview 2009.

The test has been conducted to observe the effect on real-time overhead in co-operative and pre-emptive scheduling as the number of tasks increases. As the WCET of each task is known, as well as its idle time and tick interval, the overhead value can be calculated as in Equation (5-1) for TTC and Equation (5-2) for TTH and TTP. The scenarios and measurements were repeated until the number of tasks had been varied from 2 to 100. The execution times of the tasks were in the range from 0.2715 microseconds (counter = 1) to 27150 microseconds (counter = 100000). The tick interval was set between 1000 and 10000 microseconds.

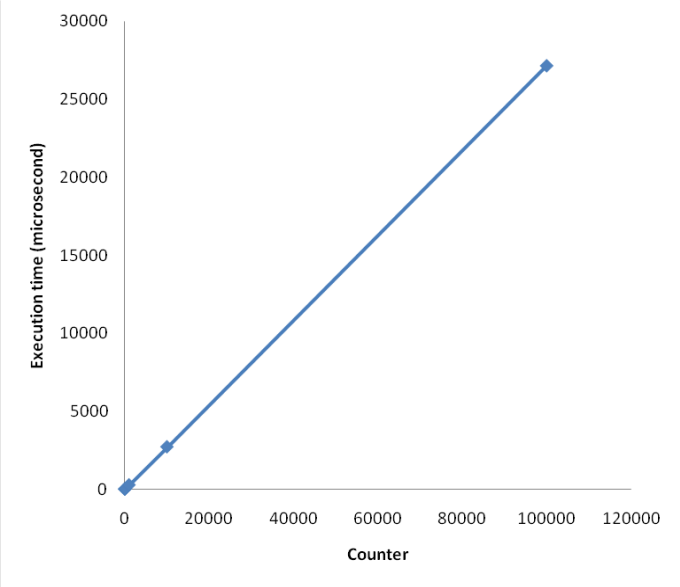


Figure 5.13 Counter value and execution time

Although the results can be expected, the execution time for different counter values was measured using Labview tools to confirm the WCET of the loop function. The results are plotted in a graph as shown in Figure 5.13.

5.6 Results for cost of implementation

This section presents the results of costs for the TTC, TTH and TTP scheduling implementations and the effect on overhead, LOC, memory utilisation, CPU utilisation and number of pre-emptions, evaluated as the number of tasks increases. Over 100 random task sets were generated to test a small number of tasks, and 1000 random task sets were generated for a large number of tasks. The total processor utilisation of each task set has been selected to be between 20% and 80%. Systems that are too highly utilised are undesirable because changes or additions cannot be made in the system without risk of time-overloading. In each study, 10000 consecutive pulse widths were measured to give average results.

5.6.1 Impact on small and large systems

The first purpose of the evaluation is to observe the impact, as the number of tasks increases, when TTC, TTH and TTP scheduling are used in a system. One pre-emptive task is included in the TTH and TTP evaluation so that the impact can be observed comparatively.

5.6.1.1 Small number of tasks

Table 5.4 and Table 5.5 show the percentage overhead for systems with 1 task and 5 tasks.

Table 5.4 Overhead for 1 task

Software architecture	WCET (us)	Tick Interval (us)	Idle time (us)	WCET and Overhead (us)	Overhead (us)	Overhead %
TTC	814.5	10000	9150 ± 2	850 ± 2	35.5 ± 2	0.36%
TTH	814.5	10000	8900 ± 2	1100 ± 2	285.5 ± 2	2.86%
TTP	814.5	10000	8906 ± 2	1094 ± 2	279.50± 2	2.80%

The overhead values presented in the table show that with the TTC implementation, the size of the overhead is low. However, the processor suffers more overhead when TTH and TTP are employed. The high overhead is mainly caused by the scheduling overhead of the interrupt invocation and the interrupt handler when adding a new task into the ready queue. The difference between the TTH and TTP overhead percentages is also very small, about 0.06%. To obtain more unambiguous effects on overhead for co-operative and pre-emptive scheduling algorithms, the system was tested with five tasks. In the TTC scheduling, the total WCET of the tasks has been designed to be less than the tick interval to avoid task overrun. However, for the pre-emptive scheduling, a long task has been created (which exceeds the tick interval) particularly to perform pre-

emption activities in the scheduling. The timing behaviour of the five tasks running in the pre-emptive scheduling is schematically illustrated in Figure 5.14.

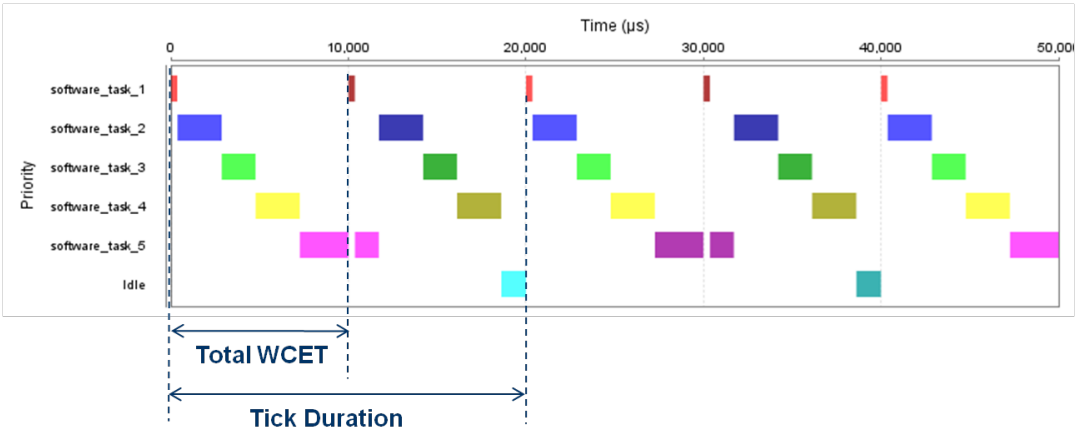


Figure 5.14 Task timing behaviour in the TTH scheduler and the TTP scheduler with same priorities

As can be seen, Task 5 has been pre-empted by Task 1 while it runs. Although, the TTH scheduler is not a fully priority-based scheduler, all the tasks (except Task 1) run co-operatively using earliest active first (EAF) scheduling strategies (Pont, 2001, Short,2012). Similarly, the results of the TTP scheduler can be illustrated as in Figure 5.14 if the priority of Task 1 has been set to the highest priority, while the others have equal priority. Note that if the other tasks have different priority, but the priority of Task 1 remains the highest and the WCETs of the tasks unchanged, the overhead on the system remains similar (as shown in Table 5.5). This is because the number of context switches of the tasks remains the same for both scenarios.

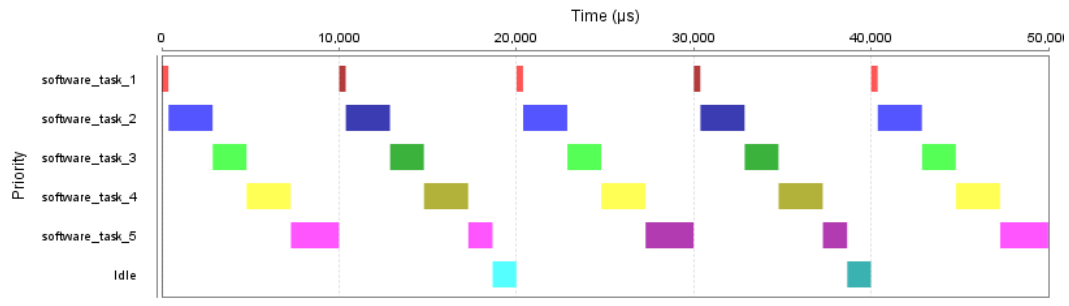


Figure 5.15 Task timing behaviour in TTP with different priorities

The average overhead results (in percentages) for the 5 tasks running in the TTC, TTH and TTP schedulers are presented in Table 5.5. This diagram is generated by the Rapiditty tool.

Table 5.5 Overhead for 5 tasks

Software architecture	Total WCET (us)	Tick Duration (us)	Idle time(us)	WCET and Overhead (us)	Overhead (us)	Overhead %
TTC	2986.5	10000	6519 ± 2	3481 ± 2	494.5 ± 2	4.95%
TTH	16018.5	20000	1880 ± 2	18120 ± 2	2101.5 ± 2	10.5%
TTP	16018.5	20000	1322 ± 2	18678 ± 2	2659.5 ± 2	13.3%

The CPU overhead results show that the TTC and TTH schedulers have lower overheads in comparison to the TTP scheduler.

Note that only one pre-emption occurred in TTH and TTP testing, when the Task 1 interrupt occurred while Task 5 is running. To measure the overheads for TTH and TTP, the duration of the tick interval required to complete all the task execution and all other tasks' WCETs during the tick interval need to be considered. For example, in the TTH scheduler, the overhead for 5 tasks can be calculated as follows, using data from specifications in Table 5.3 and its idle time (in Table 5.5):

Overhead = 20000 – (1880+10045.5+ (271.5+1900.5+1900.5+1900.5)) microseconds

From the above calculation, the overhead of the tasks has been determined to be 2101.5 microseconds. This gives an overhead percentage of 10.5%.

5.6.1.2 Large number of tasks

As shown in Figure 5.16, there appears to be larger context switch overhead for a system with 20 tasks operating under the TTP scheduler. Complete results of the experiments are shown in Appendix -D.

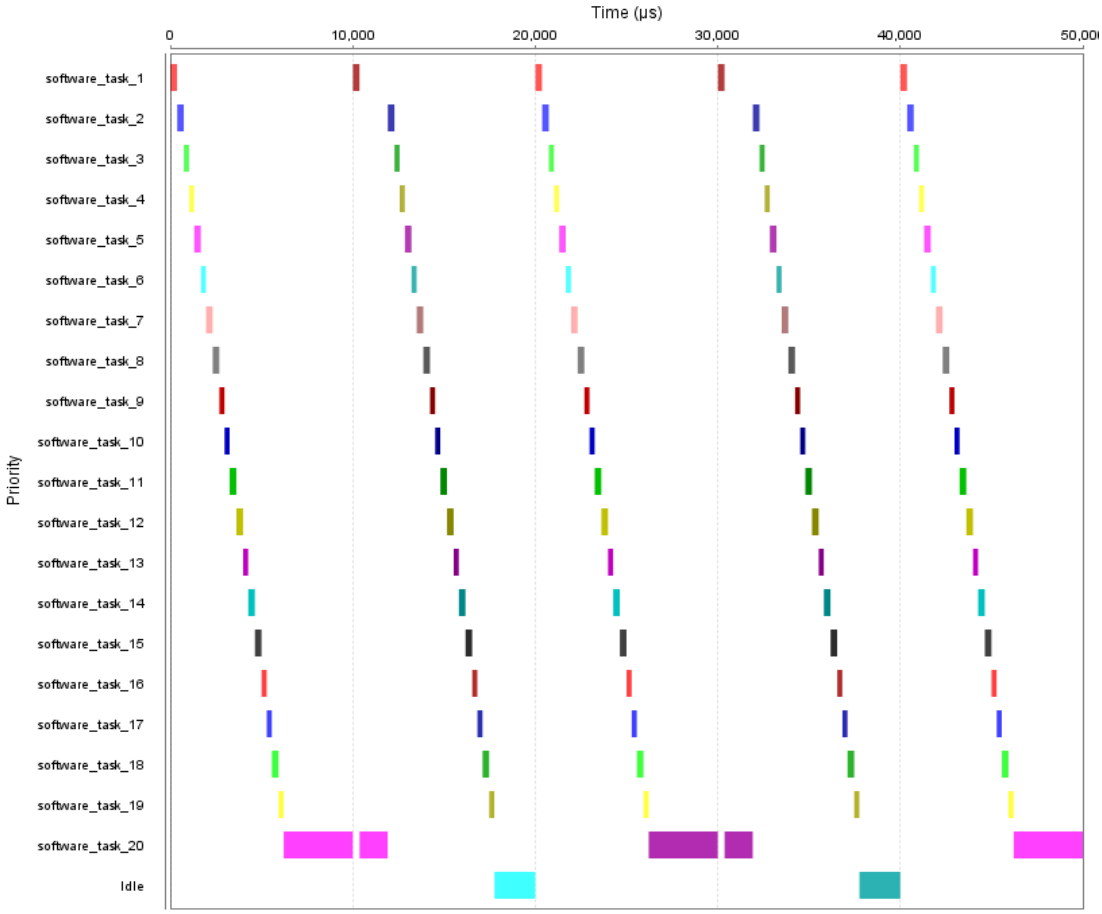


Figure 5.16 Timing behaviour of 20 tasks in the TTP scheduling

In 20000 microseconds tick duration, about 40 context switch processes occurred.

A comparison of the overhead percentages for increasing numbers of tasks, for the TTC software architecture on one processor, is plotted in Figure 5.17. The same experiment is replicated for the TTH and TTP architectures. The aim of the comparison is to explore the impact of the number of scheduled tasks on the overhead behaviour of co-operative scheduling (represented by the TTC scheduler) and pre-emptive scheduling

(represented by the TTP scheduler), as well as hybrid scheduling (represented by the TTH scheduling). This experiment is based on 1000 to 10000 randomly generated task sets which are schedulable. For each task set, the utilisation is between 20% and 80% and the maximum period is 20,000 microseconds.

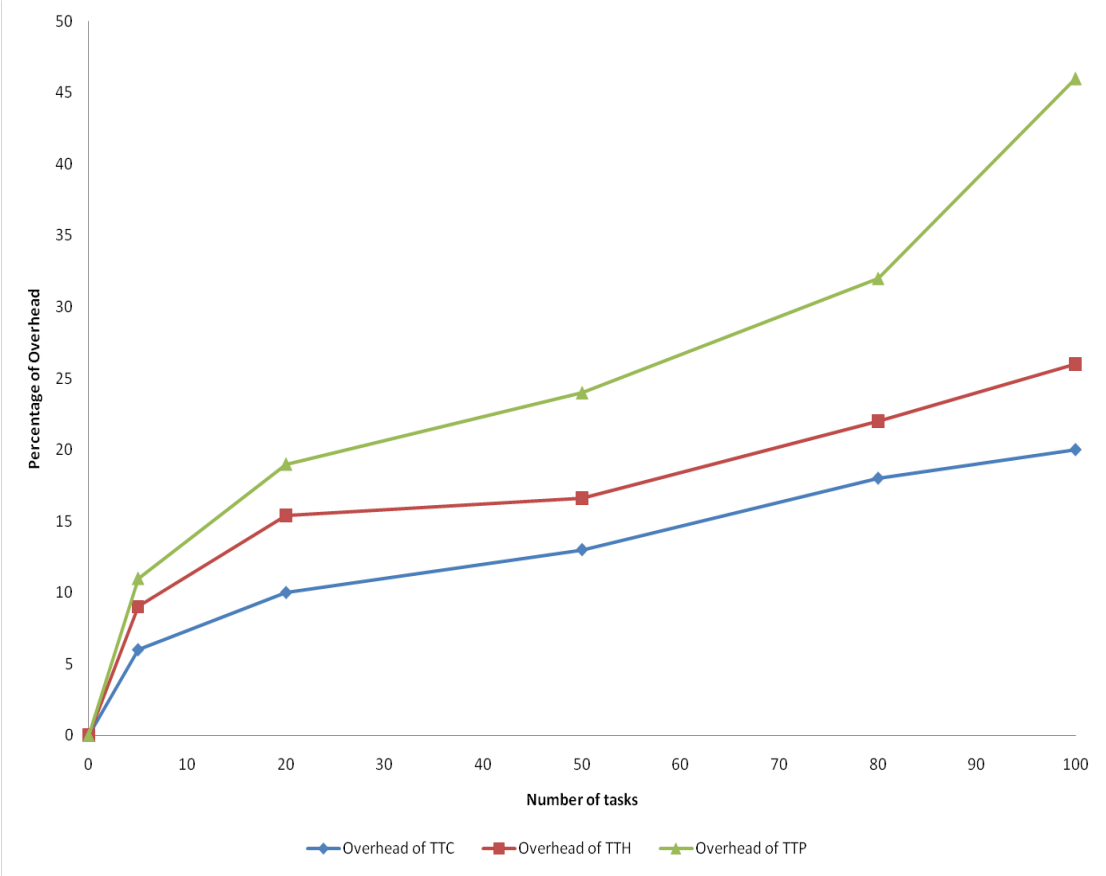


Figure 5.17 Overhead rate for TT software architecture

By analysing the overhead values of a range of numbers of tasks, it is noticeable that in the case of the TTP architecture, the overhead level increases higher than that for the TTC and TTH architectures as further tasks have been scheduled to run in the system. In fact, it has shown an exponential growth for a large number of tasks. In addition, the graph shows that the percentage overhead for the TTP architecture is almost double that of the TTC architecture for every increment of the number of tasks shown. The distinction between the overhead percentages for TTP and TTH is small for a small

number of tasks (about 4%). As the number of tasks becomes larger, the difference in overhead can vary to 20%.

5.6.2 LOC of TTC, TTH and TTP architecture

All the source code for the individual software architectures is stored in the source folder (Figure 5.18). The source code of the architecture is held in the main.c file and the other three folders in the file project.

- Scheduler : It contains C and header files of the scheduling algorithms
- System : It contains the system's startup file (*.strt*) and system and task initialisation
- Tasks: All the task source codes are stored in the folders.

The results of LOC for TTC, TTH and TTP scheduling are depicted in Figure 5.19. In this example, each software architecture consists one task to be executed.

By comparing the implementation source code of the individual main file and source folders for the different software architectures, the results indicate that TTP had higher LOC compared to the other architectures by 35 percent (with TTC) and 38 percent (with TTH). It is worth noting that an assumption that all schedulers run only one task was made. The graph demonstrates that the variations of LOC can be clearly seen in the scheduler and system folders. However, LOC for TTH and TTC in the system folder is the same. The dissimilarity for both schedulers was exposed in the scheduler folder where the C and header files of the scheduler have been allocated.

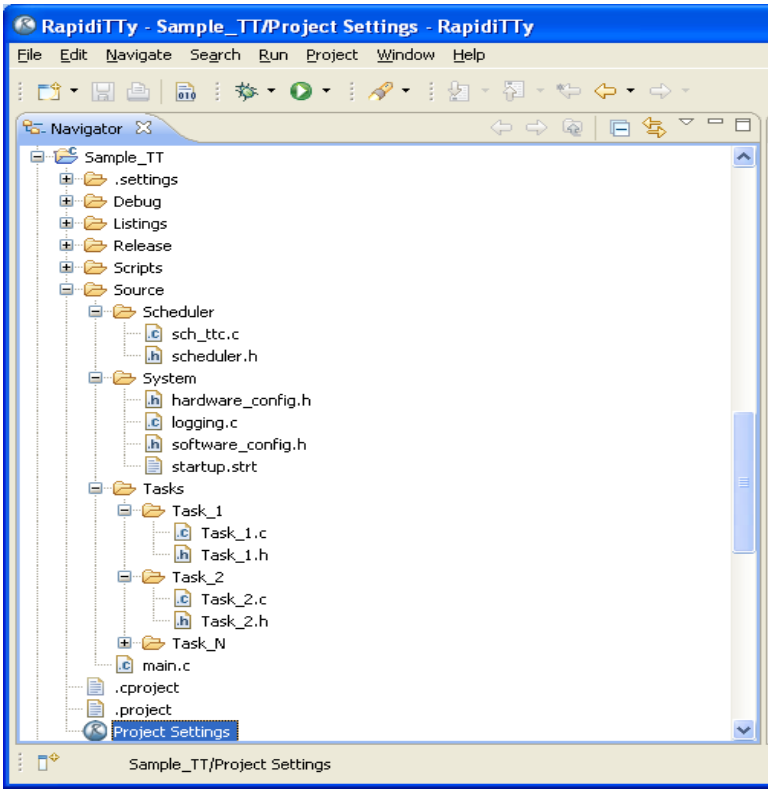


Figure 5.18 Source code files in TT project

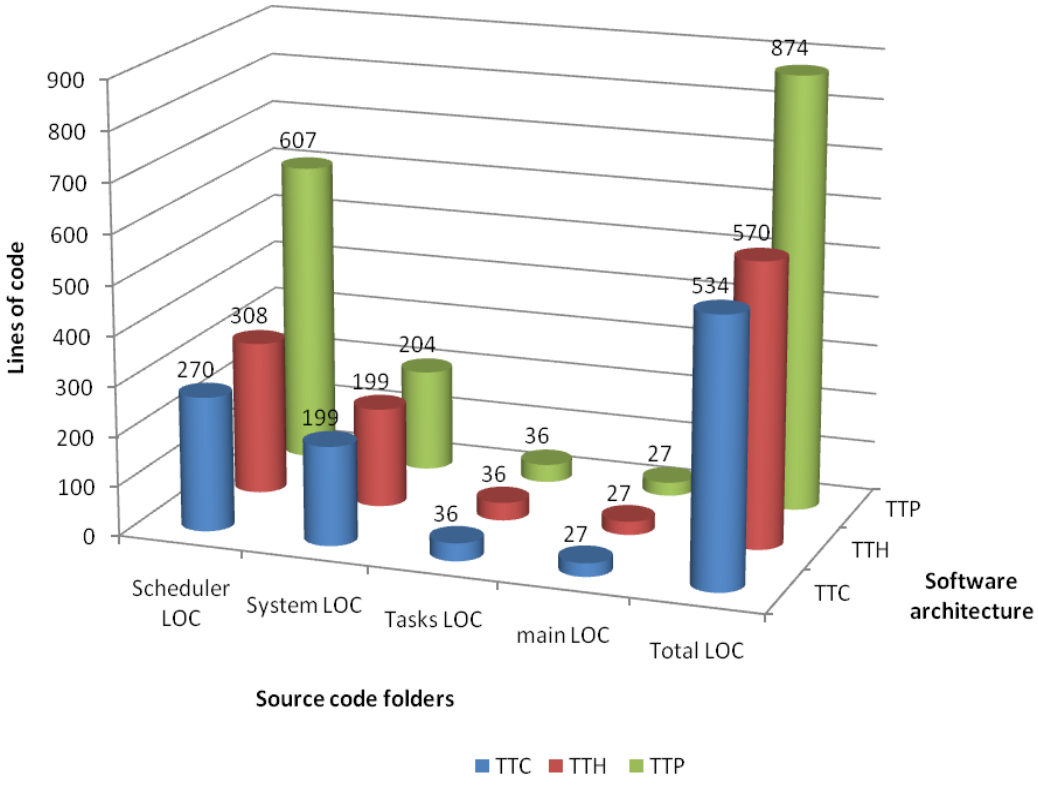


Figure 5.19 Details of LOC for TT software architecture

The major concern of this assessment is to observe to what extent the software architecture implementation can be affected by LOC. The results also suggest that the TTP scheduling had a more complicated design and greater fault rates.

5.6.3 Impact of number of tasks

Each task contains a program that is responsible for a specific function for real-time applications. Therefore, LOC increases when the number of tasks rises. Using the similar project developed for the overhead assessment, a graph of LOC and number of tasks has been plotted in Figure 5.20.

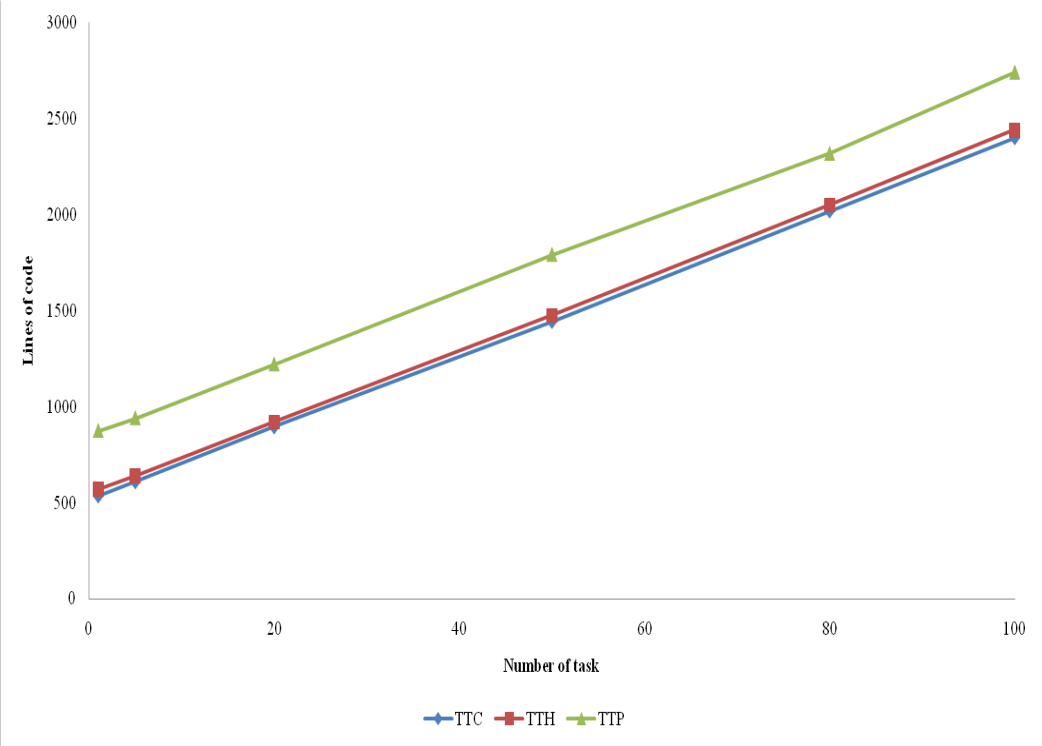


Figure 5.20 Impact of LOC by number of task

By plotting the number of lines of code against the number of tasks, it can be seen that the LOC of TTC, TTH and TTC increased proportionally. Note that the experiment used samples of dummy tasks with similar data sets for the purpose of overhead measurement (see section 3.5.3). By observing the growth of the LOC in such architecture, it is transparent that the TTC architecture has 38 fewer lines of code than

the TTH architecture for all the range of numbers of tasks. Similar effects occur for the TTP architecture. It has been observed that the difference between the TTP and TTH architectures is about 299 lines. The lines of code increase when more tasks are added to the systems (Lindström et al., 2008). For the same number of tasks which have the same LOC, the impact of the number of tasks cannot be seen clearly. However, the LOC varied when different software architectures were employed.

5.6.4 Impact of memory utilisation

Memory utilisation is an important aspect to be assessed when comparing the implementation costs of TTC, TTH and TTP. The code and data of the TTC, TTH and TTP programs are stored in Flash and RAM. Figure 5.21 depicts the impact on memory utilisation as the number of tasks grows.

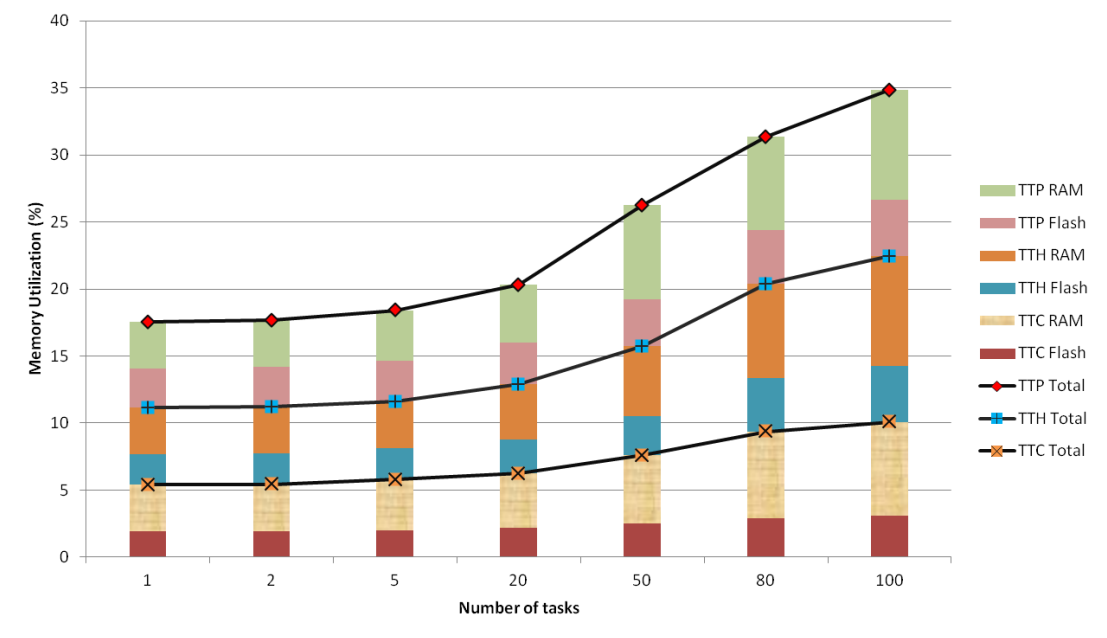


Figure 5.21 Memory utilisation when number of tasks increases

The results show that only one pre-emption task has been included within the system so as to compare fairly the TTH and TTP schedulers. As can be seen, the increase in RAM usage is higher than that for Flash when the number of tasks increases for all the three

schedulers. However, the memory utilisation increases gradually in the TTC scheduler, TTH scheduler and TTP scheduler. For a small number of tasks, the difference in the memory requirements for the TTC scheduler and TTP scheduler is 0.9 percent. However, as the number of tasks rises to more than 20, the memory requirements for the TTH and TTP schedulers rise rapidly. The difference for 100 tasks between TTC scheduler and the TTP scheduler becomes 2.4%.

5.6.5 Impact of processor utilisation

Another important factor to be observed is the impact on processor utilisation as a system becomes bigger. Obviously, the processor utilisation is increased. However the main point of interest is to identify which schedulers require more CPU utilisation when the number of tasks grows. Figure 5.22 presents a comparative view of CPU utilisation for the TTC, TTH and TTP schedulers.

In the TTC scheduler, CPU utilisation increases rapidly as the number of tasks grows. For example, for 50 tasks, the CPU utilisation becomes constant at the 75% level. On the other hand, the CPU utilisation increases higher for the TTP scheduler, and for 50 tasks, the CPU utilisation is almost 85%.

5.6.6 Impact of number of pre-emptions

Finally, the impact of the number of pre-emptions is used to observe the cost of implementation. Every pre-emption can cause context switch activities, which produce system overheads. Figure 5.23 depicts the evaluation results for overheads when the number of pre-emptions increases.

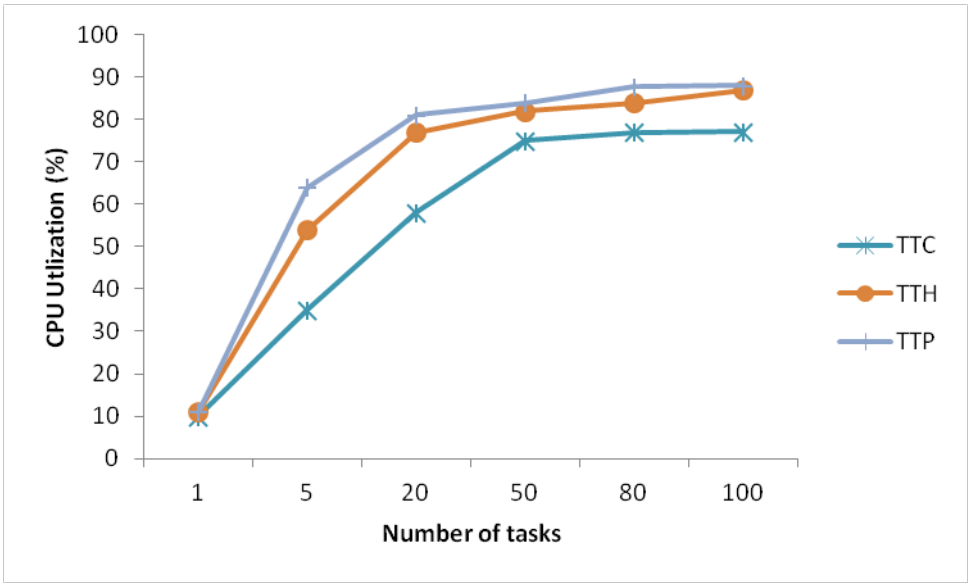


Figure 5.22 CPU utilisation for TTC, TTH and TTP scheduling

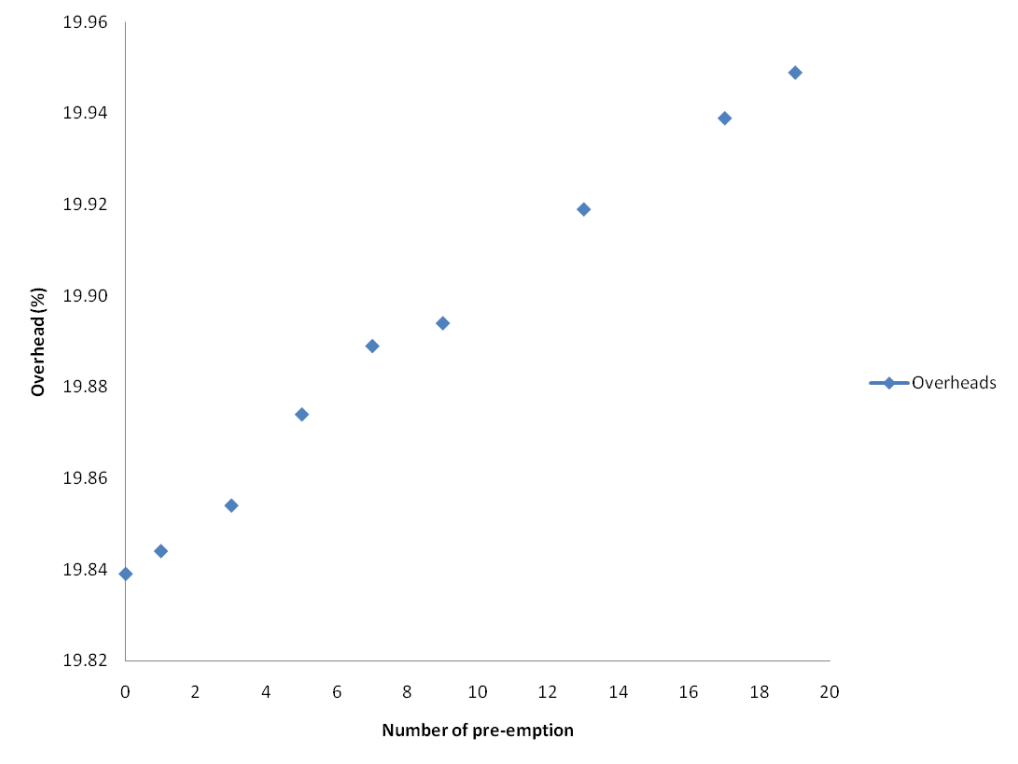


Figure 5.23 Impact of number of pre-emptions in TTP scheduler

The observation of overheads is conducted by increasing the number of pre-emptions from 0 to 20. Visualisation of the timing analysis can be seen in Appendix A 2.2. In the absence of pre-emption, the overhead for 20 tasks amounts to only 19.84%. As the

number of pre-emptions goes higher, the overhead increases rapidly. For pre-emptive systems, when the number of tasks gets larger, the number of pre-emptions increases, which may lead to the increase in system overhead.

5.6.7 Comparison of LOC with other RTOS

In order to determine whether the TT software architecture has indeed reduced the size required in implementing real-time scheduling, the results have been compared with other commercial RTOS such as FreeRTOS and MIcroC/OS as shown in Table 5.6.

Table 5.6 Comparison of LOC for TT scheduling with commercial RTOS

TT scheduling/RTOS	LOC
TTC	469
TTH	507
TTP	811
FreeRTOS	4500
MicroC/OS	5500

The results show that MicroC/OS and FreeRTOS require more than four thousand lines of code in their operating system (OS) implementation. By contrast, in the TT software architecture, less than a thousand of lines of code are used to implement the scheduling system operation. Figure 5.24 expands the results of LOC for varying numbers of tasks.

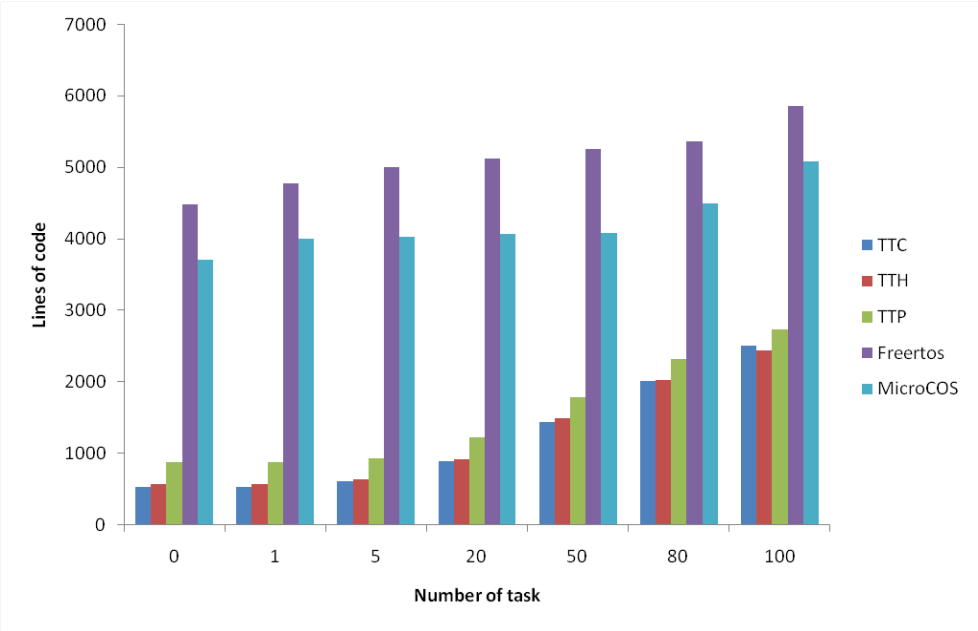


Figure 5.24 Impact of number of tasks on the LOC

Despite presenting the results of LOC of software architecture and RTOS, the impact of memory utilisation is also examined. Figure 5.25 shows the effect on memory utilisation when the number of tasks increases.

As can be seen in Figure 5.25, FreeRTOS requires more memory than TTP. The memory required by FreeRTOS is almost four times higher than that required by TTP scheduling even though both architectures provide similar attributes which can support pre-emptive scheduling.

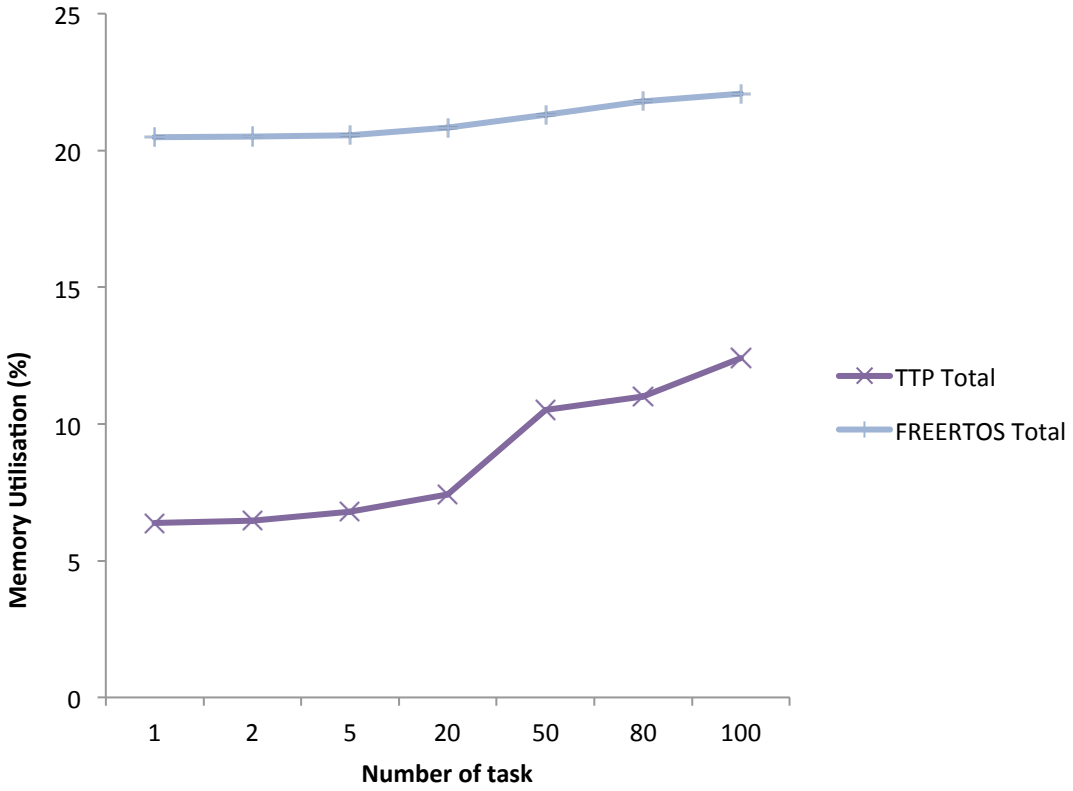


Figure 5.25 Memory utilisation for TTP and FreeRTOS

5.7 Discussion

The real-time overhead does have a considerable impact on the schedulability of the task set. This overhead arises from the time spent in handling the tick interrupt, the time spent in updating and testing the delay of each task in turn (in order to check which task should run next), and the time spent in saving/resuming the state of pre-empted tasks in the TTH and TTP designs. The level of this overhead depends on various factors including the number of tasks in the system, the scheduler type and the speed of the hardware used to implement the system.

The observed patterns are caused by the architecture of the system. In the TTC architecture, the update() function – invoked after the tick interrupt – determines which tasks are due to run and sets the corresponding flags. Then, the Dispatch() function will

execute the flagged tasks. A consequence of this process is the scheduler overhead which will vary depending on the number of tasks that are implemented in a given tick interval (Nahas, 2008). This means that when the number of tasks increases, the overheads also increase.

The effect of overhead in the TTH architecture is rather higher than in the TTC architecture. Although TTH can be implemented in the same way as TTC, running the pre-emptive task has an impact on the overhead. The TTH implementation controls the pre-emptive task by checking for its existence in the Update() function – within the tick ISR – and if it exists, the task will be promptly executed. This is a simple way to allow a pre-emption activity to influence other co-operative tasks. However, in such an implementation, the overhead in TTH will be slightly higher than that for TTC. In general, the overhead percentage for TTH increases in parallel with that for TTC, but is somewhat higher.

Under the TTP architecture, the scheduler always executes the highest priority task that is ready to run. Upon completion of an ISR, the scheduler resumes execution of the highest priority task ready to run. The process of saving the current task's context and restoring the new task's context induces overhead in the system. The overhead can become higher when the CPU has a higher number of registers to be saved and restored. In addition, based on observation, the amount of overhead depends on how often the context switch service is invoked. When the number of tasks is small, the overhead is minimal. However, the percentage of overhead gets bigger as the number of tasks increases. This is because the processor needs to manage more context switching during runtime.

Liu (2000) stated that the overhead cost is high if a pre-emptive scheduling approach is employed by the operating systems. This is due to the fact that each pre-emption incurs

processing overheads. The processor must decide which task to run, and then swap processor states.

Consequently, a major focus in the design of operating systems has been to avoid unnecessary context switching to the greatest possible extent. However, this has not been easy to accomplish in practice. In fact, although the cost of context switching has been declining when measured in terms of the absolute amount of processor time consumed, this appears to be due mainly to increases in processor clock speeds rather than to improvements in the efficiency of context switching itself (Liu, 2000).

By using a co-operative architecture, the context switch overhead can be reduced. The context switch is needed after a task completes its execution, and the next task is ready to run. This operation typically occurs in multi-tasking systems. However, the context switch overhead becomes much more complex because the processor needs to perform a save and restore service during the swap operations.

Upon further investigation, it becomes clear that co-operative scheduling has the simplest architecture in comparison to any versions of pre-emptive scheduling. One of the factors that promotes this result is the context switch implementation. In the TT implementation, the context switch is stored in the system folder. The results in Figure 5.19 revealed that TTP had higher number of LOC compared with TTC and TTH. In addition, TTP design is a priority-based design while TTH and TTC do not employ any priority scheme in their design, except for one pre-emptive task in TTH. This priority-based implementation involves comparing the priority of current running task with the pre-emptive task. This process must be included in the TTP scheduling implementation. Obviously, the total LOC of TTP is the highest amongst the three schedulers; this is also shown in the graph in Figure 5.19. There is a possibility that the LOC of the

scheduling implementation can be reduced so that the complexity and effort are minimised.

As an embedded system has tight memory size and CPU utilisation, it is also necessary to analyse the cost of memory and processor utilisation for each software architecture, as well as the impact of number of pre-emptions. In this study, the impact of the number of tasks on all the above costs was examined. More memory is required by TTP in comparison to TTH and TTC. The TTP scheduler implementation itself needs more memory space than the others. In practice, the TTP scheduler should have higher memory utilisation than the TTH scheduler since it can support more than one pre-emptive task. Thus, the context switch mechanism and activities during pre-emption will increase the memory utilisation. The impact of context switch overhead in the TTP scheduler is shown in the results of the impact of number of pre-emptions (section 5.6.6).

The impact of processor utilisation is also investigated. Clearly, TTP requires more CPU utilisation than TTC. Only one pre-emptive task for each of the TTH and TTP schedulers has been included in this study. The reason behind this is to observe the impact of TTH and TTP schedulers with similar task attributes. Although both schedulers run a single pre-emptive tasks for evaluation purposes, it shows that the TTP scheduler has higher CPU utilisation than the TTH scheduler. If more than one pre-emptive task is included, the CPU utilisation will be increased due to context switch processing activities.

In the previous evaluation, only a single pre-emptive task is considered in order to provide a reasonable comparison with the TTH scheduler in which can support a single pre-emptive task. In section 5.6.6, the analysis of costs examines the effects on overhead when the number of pre-emptions increases. In practice, more than one pre-

emptive task will pre-empt other tasks if pre-emption is allowed, as in the TTP scheduler. The results clearly show that the overhead increases along with the number of pre-emptions. This may result in increased memory utilisation as well as processor utilisation. As mentioned in Chapter 4, the main drawback of pre-emptive scheduling is the context switch overhead when pre-emption occurs. The need to save and restore the current task activities – to allow the higher-priority task to run – may increase the cost of scheduling implementation.

A scheduler can be presented as a small operating system that manages a set of tasks (Pont, 2001). In embedded applications, many developers employ a conventional RTOS to support event handling. The implementation of RTOS is very complex. For example, the MicroC operating system employs about 5500 lines of source code. Unlike conventional RTOS, simple schedulers like TTC and TTH use only 400 to 600 lines of code in their implementation. Finally, the assessment investigates a comparison of memory utilisation for the TTP scheduler and FreeRTOS architecture as the number of tasks increases. Only a single pre-emptive task was included for each evaluation of the number of tasks. The results imply that, even though both architectures provide similar characteristics such as supporting pre-emption activities and having context switch overhead, the TTP scheduler requires less memory than FreeRTOS, which gives many advantages, particularly when aiming for a cost-effective system.

5.8 Conclusion

The main focus within this chapter has been the assessment based on the first hypothesis which relates to the implementation cost of pre-emptive and co-operative scheduling. The study analysed the implementation costs for the time-triggered co-

operative (TTC) scheduler (to represent co-operative scheduling), the time-triggered hybrid (TTH) scheduler (to represent combination of co-operative and pre-emptive scheduling) and the time-triggered pre-emptive (TTP) scheduler (to represent pre-emptive scheduling). The context switch overhead for the TTP scheduler increases exponentially as the number of tasks grows. The increase is also higher in comparison to that for TTH and TTC. Besides the effect on overhead of the number of tasks, the extent to which the software architecture affects the code size, memory and processor utilisation, as well as the number of pre-emptions, has been explored. In addition, the behaviour of co-operative and pre-emptive scheduling algorithms has been compared in order to identify which of these techniques will help to reduce the cost and effort of embedded software implementation. In conclusion, the results have shown that the first hypothesis under test is valid. The TTC scheduler, when compared to the TTP scheduler, is applauded due to its simplicity of implementation that requires small code size and low cost. The discussion and results of the second assessment will be presented in the following chapter.

Chapter 6
Evaluation of the Cost of Testing

6.1 Introduction

As discussed in Chapter 4, assessment to examine the impact of software architecture on the cost of testing real-time embedded systems is required. Therefore, this chapter provides a detailed cost analysis of testing. In addition, the main intention of this chapter is to explore the advantages of the system relative to high predictability, such as that present in TT system, thereby assisting the testing process for finding faults in real-time software systems. With this motivation, a testing strategy to localise errors to find their main source was considered. Further investigation was also undertaken to explore the extent to which inter-task communication and synchronisation mechanisms, as undertaken in embedded systems, can be affected throughout the course of the testing phase.

6.2 Problem statement

The problem involves accurately evaluating the cost of testing complete systems. This requires a discussion of the cost of testing at different phases of the system development lifecycle, requiring selection of a base system to form the basis of a comparison, and to define the testing procedures to be evaluated.

A testing cost analysis for TT and ET software architectures needs to be undertaken in order to assess the following hypothesis, which appears in Chapter one as H2:

Testing a system with a TT architecture incurs less cost than testing an equivalent system with an ET architecture, when experimental-based methods and comparative analysis are used.

6.3 Adopted methodology

6.3.1 Cost of Testing

The cost of testing is defined in terms of complexity, duration of the executed test, and ease of testing. Furthermore, the tests should be repeatable and the results reproducible when applying the same conditions.

In order to assess the said costs, measurable parameters had to be defined. Therefore, a complete system which runs itself into a fault was considered. In doing so, the fault needed to be localised in order to isolate the task that had caused the fault to emerge. The fault was assessed and identified by determining whether the timing requirements had been met or not. It was important to replicate the same timing error, which should be reproducible for a single task, as opposed to the whole set of running tasks.

In order to evaluate the cost of testing, the following hypothesis was used:

Testing will be easier for systems in which the timing data obtained for isolated and in situ tasks is very similar.

This hypothesis was used as a basis for conducting experimental work to evaluate the cost of testing during the testing phase. The ease of isolating a single task is a parameter that impacts upon the cost of testing. In fact, it is one of the most important aspects to assess when evaluating and testing systems. Thus, if we easily managed to isolate the task in a system, the cost of testing that system would be less.

6.3.2 Measuring WCET of tasks

In order to measure the execution time for each task, the code for the task was instrumented for sending signals at the start and end of each task code. The easiest way was to use a digital output port. This port was dedicated to the testing of embedded

applications. One or two bits of an 8-bit digital output port 4 of LPC2378 were used as a gateway to monitor the timing behaviour of the tasks.

During the course of the code measurement, it is important to execute it in a state that produces the WCET. The WCET for the task was measured using a measurement-based method on real hardware and obtained by taking the maximum measured time over a number of trials using a set of selected input data. Note that, the impact of pre-emption is evident in lower priority tasks or pre-empted tasks.

In order to measure the execution time for a task in ET and TT architecture, the task must have an analysable design. This means that it must has a definitive starting and stopping point in each cycle. The execution time of the tasks was obtained by setting the high pin at the beginning of the task and the low pin at the end of the task. The widths were then measured using the NI card with LabVIEW software.

As a result, 10000 consecutive pulse widths were measured and recorded for each study. This was deemed to be sufficient for the purpose of this research. In an ET system, current running tasks can be pre-empted by higher priority tasks. For pre-empted tasks, for example task 2, which is shown in Figure 6.1, the execution time is the difference between t_6-t_1, minus the execution time used by task 1, (t_4-t_3)during that time period.

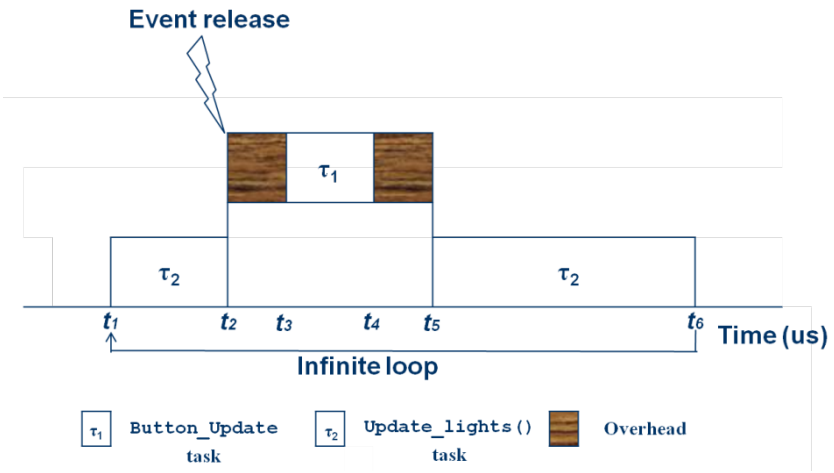


Figure 6.1 Measuring execution times for an ET system

It is important to note that the measured execution times include, if not all, most of the RTOS overheads. In fact, the overheads usually affect the execution time of the lowest priority tasks. Consequently, the WCET of the lowest task can be used to demonstrate the effects of the overheads.

In FreeRTOS implementation, the "Button_Update" task is called periodically using a vTaskDelay() function. Conversely, the "Update_Lights" task is continuously executed in the loop. Judging by the reading measured by the execution time from NI instruments data generated by the pulse width in Figure 6.2, it the RTOS overhead for context switching and scheduling was assumed to be included in the measurements and evenly distributed across each task.

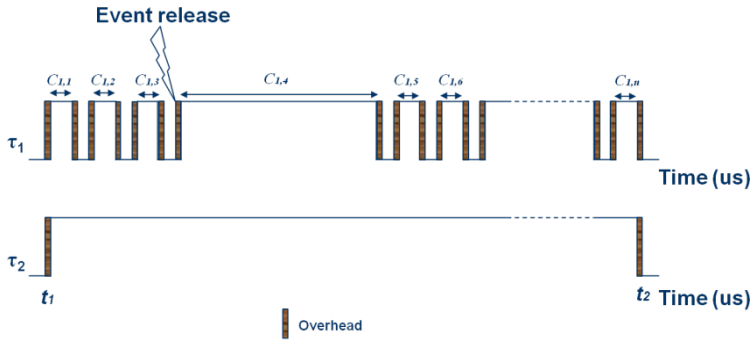


Figure 6.2 Measuring execution times for a FreeRTOS system

The execution time of task 2, *C2* is calculated thus:

$$C_2 = (t_2 - t_1) - (C_{1,1} + C_{1,2} + C_{1,3} + \ldots\ldots + C_{1,n}) \quad (6\text{-}1)$$

Where t_2 is the time the pin low is executed, t_1 is the time the pin high is executed, and $C_{1,1}$, $C_{1,2}$, … $C_{1,n}$ is computed as the amount of time that the task with higher priority is executed. As can be seen, task 1 has two execution times: $C_{1,1}$ if the push button is in the release state, or $C_{1,4}$ if the push button is pressed and then released. In this example, the "button_update" task is a periodic task with a 10 millisecond period time, and thus the "lights_update" task will be pre-empted for every 10 milliseconds. When an event is released in which the button is pressed, the execution time of task 1 becomes $C_{1,4}$. After the button is released, the execution time is back to normal – the execution time of the "button_update" task if the push button is not pressed.

In TT systems, the execution time of tasks in co-operative scheduling can be straightforwardly measured using the relative time. For pre-emptive schedulers, such as the TTH and TTP scheduler, the measurement of execution times has to take into account for the execution times used by higher priority tasks during the time period. Examples of pre-emption scenarios in tick-based scheduling are illustrated in Figure 6.3.

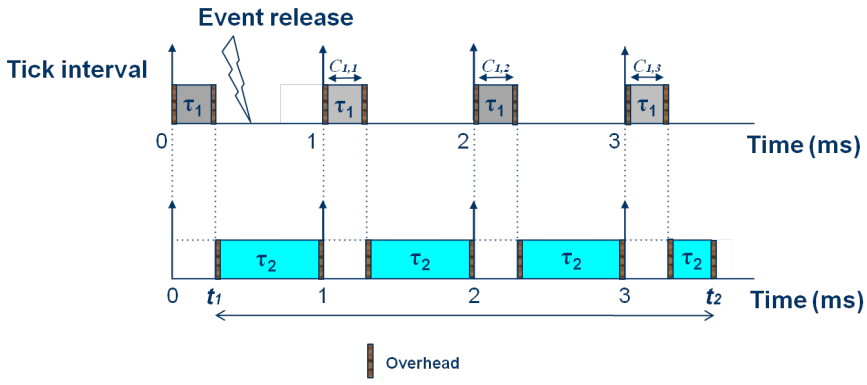


Figure 6.3 Measuring execution times for a TTH and TTP scheduler

In order to calculate the actual execution time of task 2, the duration of t_1 and t_2 which is obtained from the setting of pin high and low in task 2 code, should be subtracted with the total of execution times of task 1 in that duration since the task has higher priority and shorter time period than task 2. Hence, the execution time of task 2 is:

$$C_2 = (t_2 - t_1) - (C_{1,1} + C_{1,2} + C_{1,3}) \quad (6\text{-}2)$$

In general, the concepts of measurement for Equation 6-1 and Equation 6-2 are similar. As the duration of lower priority task is known, as well the execution times of higher priority tasks, the actual execution time of the lower priority task can be measured by subtracting the execution times of the higher priority tasks from the measured duration time of lower priority tasks.

6.4 Case study 1: Assessing the cost involved in task testing

In the case study presented in this section, the aim was to explore the timing behaviour of the tasks during the testing operation. The particular goal was to consider both a simple TT system and an equivalent system implemented using an ET-based architecture. These studies considered the tasks in isolation (as is normal when undergoing detailed examination during a testing process); and also considered the tasks operation when in the complete system. The underlying hypothesis was that testing will be easier for systems in which the timing data obtained for “isolated” and “in situ” tasks is very similar.

Detailed studies have been conducted on the measurements of the worst-case execution times (WCETs) of tasks employing ET and TT designs. In each case, the timing of tasks has been measured (i) when the tasks are isolated; and (ii) when the tasks are executed in the complete system.

Two small case studies were undertaken in order to evaluate which of the software architectures can reduce testing effort. These studies and the results obtained are described in this chapter.

6.5 The traffic lights system

Case study 1 was used to evaluate the extent to which TT architecture can provide effortless task testing using a task in an isolation strategy – when the incurrence of the event is unpredictable. This study involved the design of a traffic lights system in TT and ET architecture. A description of the case study is presented in this section.

6.5.1 Task functions

The study is of a traffic lights system that acts as a controller for the traffic lights and pedestrian crossing lights used at a typical crossroads in the UK. Crossroads can have traffic lights at each of the four possible directions and pedestrian crossing lights on all four sides. A pedestrian crossing lights system consists of two images (a green walking man and a red standing man) which alert pedestrians to when it is safe to cross the road, and when to give way to motor vehicles. In addition, there is a button used to alert the traffic light system that pedestrians are waiting in a queue to cross the road. Once pressed, the button illuminates with a 'wait' sign as an indicator for pedestrians to wait until the light turns green.

The traffic lights system can be simulated graphically using the LCD on the LPC2378 ARM processor board (Olimex, 2009). The hardware setup for implementing this experiment is shown in Figure 6.4. The NI board was connected to one of the ports on the Olimex board for the purpose of measuring the execution time of the task.

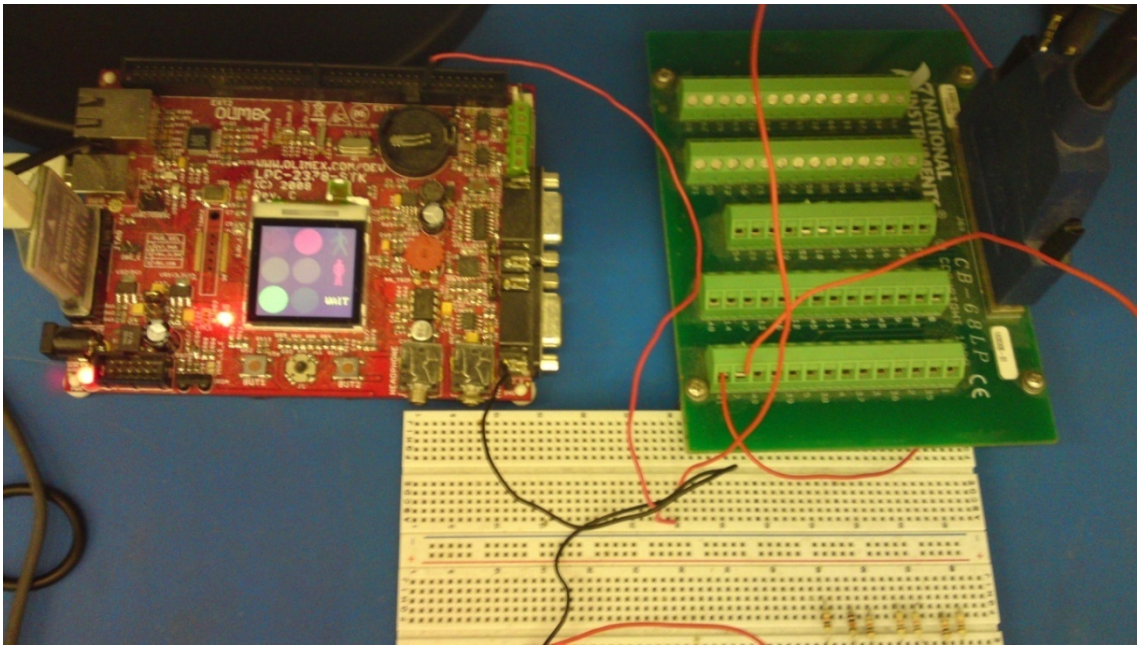


Figure 6.4 Hardware for Traffic Light Systems for testing

The state diagram of the system is shown in Figure 6.5.

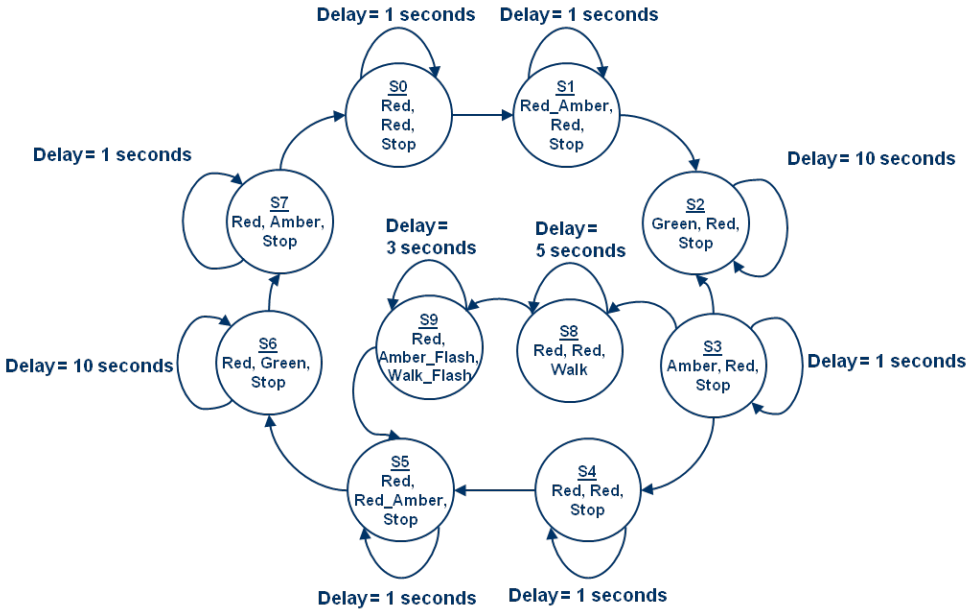


Figure 6.5 State diagram for the traffic light system

Under normal operations, the traffic lights will turn the cycle through the usual states, S0, S1,….S7. When a pedestrian presses the button the ‘wait’ indicator will light up

(before entering S8); and when the system reaches one of the states where both lights have transitioned to red, the pedestrian crossing light will transition to green.

At this point, the wait indicator should turn off and the pedestrians be given time to cross (5 seconds). As a warning to both drivers and pedestrians, returning to normal operation after a crossing will involve state S9.

A complete system of traffic lights consists of two tasks:

- A “Button_Update” task (high priority task) which updates the states of the pedestrian switch button,
- A “Lights Update” task (low priority task) which controls the traffic lights states and the transmitted data to the LCD screen.

6.5.2 Implementation of a system with an ET architecture

In an ET system, the significant external event triggers are often relayed to the computer system by means of an interrupt mechanism. The main program is a polled loop instruction. The various tasks in the system are scheduled via either hardware or software interrupts, whereas dispatching is performed by interrupt-handling routines. Moreover, this system often lacks explicit temporal control; thus, temporal control is programmed using hand-coded delay blocks.

6.5.2.1 The foreground/background system

Foreground/background methods are employed in order to demonstrate an event-triggered architecture. In this architecture, the polled loop instruction is replaced by the code that performs useful processing - a background task.

The background task is fully pre-emptive by any foreground task. The foreground operation for the system is the same as that of the interrupt-only system. If more than

one foreground process exists, round-robin, pre-emptive priority or hybrid methods would be needed to provide scheduling. A typical foreground/background method is given in Figure 6.6.

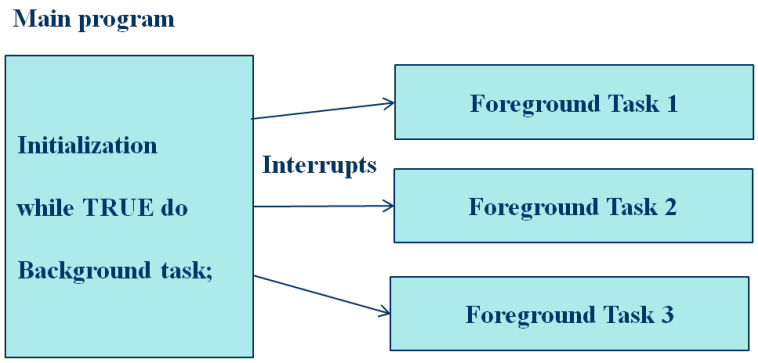


Figure 6.6 A foreground/background system

In a background/foreground system setting, the application consists of exactly one loop without an exit condition. Within this loop, the application calls for subroutines in a sequential order, which implement the application's logic. The loop's execution time essentially determines the application's temporal behaviour. The loop is commonly referred to as the background. If an interrupt occurs, an interrupt service routine (ISR) pre-empts (suspends) the loop and services the interrupt. The ISR is commonly referred to as foreground; hence the name background/foreground system. The application typically spends time in the background part, executing the main loop. When an interrupt occurs, the system switches to the foreground and the ISR services the interrupt. Once the ISR is completed, the system switches back to the background operation and resumes the main loop.

The main application domain of background/foreground systems are small embedded systems such as washers, dryers, microwave ovens, and simple radios. In comparison to multi-threaded systems with explicit temporal control, background/foreground systems require less system overheads and less understanding of concurrency and temporal

control. However, low system overheads have limitations; the application's output is non-deterministic with respect to the timing. The points in time at which the application produces an output changes depending on the application's execution path for each run in the loop and how many and what types of interrupts occur. The application's timing is also sensitive to modifications to the loop. For example, one additional inner loop in the main loop changes the timing behaviour of everything that follows after this inner loop. Such a change can alter the whole system's behaviour. In addition, modifying the ISR changes the timing behaviour depending on how often the ISR pre-empts the main loop.

In the case study, the system included a single interrupt that occurred periodically. It has the highest priority and can be used to handle a task that requires immediate attention.

6.5.2.2 Event-triggered with RTOS support

In order to handle multiple interrupts in event-triggered real-time systems, a complex operating system using round-robin, pre-emptive priority, or a combination of both schemes would be required. These types of operating systems are readily available as commercial products known as real-time operating systems (RTOS). The RTOS's design is based on the foreground/background architectures with added functions such as a device driver, a network interface and complex debugging tools. The operating system represents the highest priority task, kernel or supervisor and has been demonstrated through the use of an off-the-shelf RTOS called FreeRTOS (Barry, 2001).

FreeRTOS is a real-time, pre-emptive operating system targeting embedded devices. This FreeRTOS is a portable, open source, mini real-time kernel which can be used to

develop commercial applications for small embedded systems. In addition, its scheduling algorithm is dynamic and priority-based. Although scheduler decision points occur at regular clock frequency, asynchronous events can also invoke the scheduler's decision points. Therefore for this reason, FreeRTOS is adopted for implementation in this case study in order to compare its implementation with a pure ET architecture and a pure TT architecture.

6.5.3 Implementation of a system with a TT architecture

In TT architecture, numerous scheduling algorithms were developed with different system behaviours. Some commonly using time-triggered scheduling algorithms are co-operative scheduling, pre-emptive scheduling and hybrid scheduling (Pont, 2001). In order to carry out this study, a time-triggered co-operative (TTC) was adopted to represent a pure time-triggered (TT) architecture, a time-triggered hybrid (TTH) and a time-triggered pre-emptive (TTP) scheduler.

Note that, the system was designed and implemented in a different way from an ET and a FreeRTOS architecture. Due to the tick-based scheduling attribute of the TT architecture, the system can be implemented using a multi-state (input/timed) system (Pont, 2002).

Listing 6.1 presents the states in the traffic lights system including the pedestrian traffic lights system states.

```
typedef enum
{
        // ROO == Red, Off, Off; the state of each light
              ROO_ROO_STOP_1,
              RAO_ROO_STOP,
              OOG_ROO_STOP,
              OAO_ROO_STOP,
              ROO_ROO_STOP_2,
              ROO_RAO_STOP,
              ROO_OOG_STOP,
```

```
        ROO_OAO_STOP,
        ROO_ROO_WALK,
        ROO_OFO_FLASH_1, // off
        ROO_OFO_FLASH_2, // on
        ROO_OFO_FLASH_3, // off
        ROO_OFO_FLASH_4, // on
        ROO_OFO_FLASH_5, // off
        ROO_OFO_FLASH_6  // on
} state_t;
```

Listing 6.1 Possible system states

The traffic lights sequencer executes a sequence of pre-determined manoeuvres. Transitions between states are controlled by the passage of time or by system input as presented in Listing 6.2.

```
// Time for every state, in multiples of 500 ms
const uint8_t time[] = {
             2, 2, 20, 2, 2, 2, 20, 2,

             10, 1, 1, 1, 1, 1, 1
};
const state_t next[] = {
             RAO_ROO_STOP, OOG_ROO_STOP, OAO_ROO_STOP, ROO_ROO_STOP_2,
             ROO_RAO_STOP, ROO_OOG_STOP, ROO_OAO_STOP, ROO_ROO_STOP_1,
            // Pedestrian traffic light states
             ROO_OFO_FLASH_2, ROO_OFO_FLASH_2, ROO_OFO_FLASH_3,
             ROO_OFO_FLASH_4, ROO_OFO_FLASH_5, ROO_OFO_FLASH_6,
             ROO_OOG_STOP
};
```

Listing 6.2 Time and state arrays

The time spent in each state is shown in the time array and is associated with states in the next array. These timing values are defined using the periodic execution of the task. For example, say a task has a 500 milliseconds period of time; the system begins at the *RAO_ROO_STOP* state, repeatedly executing the task for two times so that it remains in the state for 1 second. Thus, as can be seen in the time array, time 2 is required for the *RAO_ROO_STOP* state. This then moves into the next state, *OOG_ROO_STOP* and remains there for 1 second, repeatedly executing the task. The process will be continuously performed in an orderly manner.

In alternative scenario, when a pedestrian pushes the button, the traffic lights sequence will be stopped at the *RAO_ROO_STOP_2* state and the sequence of pedestrian traffic

lights states will be executed. A snippet of the “Lights_Update” function is shown in Listing 6.3.

```
void Lights_Update(void)
{
      …..
      if (--current_time == 0)
      {
            if (current_state == ROO_ROO_STOP_2 && pressed)
            {
                  current_state = ROO_ROO_WALK;
            }
            else
            {
                  current_state = next[current_state];
            }
            current_time = time[current_state];
            …..
      }
}
```

Listing 6.3 Lights_Update ( ) function in TT systems

6.5.4 Interrupts

Interrupts which are triggered by external hardware are used to demonstrate an event occurring in the ET system. External interrupts can be used in reaction to the external sensors. In handling the external events for ET systems, two different interrupt handlers can be used: Interrupt Request (IRQ) and Fast Interrupt Request (FIQ) interrupts. External interrupt sources should be initialised before enabling IRQ and FIQ interrupts, to avoid unexpected interrupts occurring before an appropriate handler has been set up. An IRQ and FIQ exception causes processor hardware to go through a common procedure as shown in Figure 6.7.

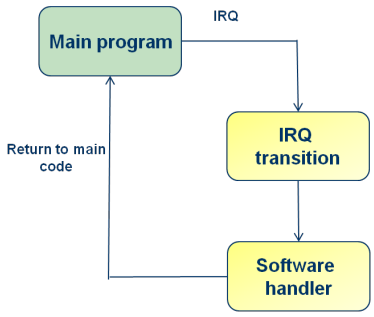


Figure 6.7 IRQ interrupt handler (Labrosse, 2002)

In this experiment, a pedestrian switch button was set as an external source to cause an IRQ exception. When an external interrupt was raised, or the switch button pressed, the normal sequential execution of instructions was be halted to allow the IRQ handler to execute – in this context, an interrupt service routine (ISR), followed by the "Button_Update" task. `Listing` 6.4 shows how the "Button_Update" task was called using the IRQ handler.

```
void Handle_Interrupts_IRQ(void)
{
	Button_Update();
}
```

Listing 6.4 IRQ handler

Upon completion, the processor mode reverted to the original main code. Note that, on entry to the IRQ handler, IRQ exceptions are disabled and should remain disabled until the current interrupt source has been cleared, in order to avoid further triggering of an exception.

Alternatively, an ET architecture could be implemented using FreeRTOS to support dynamic scheduling. FreeRTOS can be configured to operate under a pre-emptive mode. In the scheduler, after the clock is reset, the FreeRTOSConfig.h parameter configUSE_PREEMPTION would be referenced to determine which mode is being used and in this experiment, this is set to 1. This means that whether the pre-emptive kernel is configured or otherwise, the kernel is set to co-operative. The co-operative kernel is not a point of interest for this study.

FreeRTOS was employed due to its supportive function, since it can support event handling. When an event occurs, the scheduler requires an online scheduling decision. If a task is unblocked and it has a higher priority than the current task, then a context

switch is executed. Finally, the context is restored, soft registers are un-stacked, and the scheduler returns from the interrupt.

By contrast, the task activation in a TT system is controlled by tick interrupts. Thus, a task which is ready may not be noticed and acted upon by the scheduler until the next tick interrupt. Note that the tick interrupt is a single interrupt source in a TT system, as these require a fast response time. Thus, normally a FIQ interrupt is set.

In order to implement the experiment under a TT-based architecture, a switch-poll task; a "Button_Update" task is created; this periodically observes whether the push button is pressed or released. If the system detects that the push-button switch is pressed or released, the 'state' variable will react by changing its state. By using the time-triggered cooperative (TTC) scheduler, all the tasks are predetermined before execution. The system runs the switch-poll task, which is invoked every 10 milliseconds. When the switch button has been pressed, the "Button_Update" task is placed in the pending (task) queue. The scheduler holds the tasks that have been released or unblocked since the last tick interrupt. When the scheduler executes, it moves the tasks in the pending queue to the ready queue and places them in the ready queue to be executed.

6.5.5 Task properties

One of the main aims of the experiment was to develop a similar system with two different software architectures: one with a TT architecture and another with an ET architecture. In addition, the study also investigated the implementation of the system using the off-the-shelf RTOS. These systems have two tasks: a "Button_Test" task and an "Update_Lights" task. However, the task properties of those systems are different as shown in the next sub sections.

6.5.5.1 Task properties on an ET architecture

In an ET-based system, an “Update_Lights” task runs as a background task in the main function; while a “Button_Test” task runs as a foreground task.

Table 6.1 Task properties for the traffic light systems on an ET-based system

Name	Delay	Task type
Button_Update	0	Foreground task
Update_Lights	Delay_c1 = 1000 Delay_c2 = 10000 Delay_c3 = 5000 Delay_c4 = 3000/6	Background task

A “hardware delay” function is used to control the timing of the traffic light system (see Listing 6.5) (Pont and Association for Computing Machinery, 2001).

```
void Update_Lights(void)
       {
       ……
       while (TRUE)
              {
              TRAFFIC_Set_Light_State(1, TRAFFIC_RED);
              // Delay to control the state on hardware timer 0
              Hardware_Delay_T0(delay_c1);
              TRAFFIC_Set_Light_State(0, TRAFFIC_RED_AMBER);
              Hardware_Delay_T0(delay_c1);
              TRAFFIC_Set_Light_State(0, TRAFFIC_GREEN);
              ……
              }
```

Listing 6.5 Implementation of the “Update_Lights” task for an ET-based system

Where Hardware_Delay_T0 ( ) is a function implemented particularly to generate N millisecond delay (approximately) based on hardware Timer 0 as shown in Listing 6.6.

```
void Hardware_Delay_T0(const uint32_t Delay)
       {
       // Set up required match register
       T0MR0 = ((TTE_PCLK(TTE_PC_TIMER0) / 1000U) * N) - 1;
       T0MCR = 0x06; // Interrupt on match, and automatically restart
       counter
       // Counter enabled
       T0TCR |= 0x03; // Start timer 1 (Timer Control Register)
```

```
T0TCR |= 0x02;
while ((T0TCR &= 0x01) != 0) ; // Loop until Timer 1 matches
(T1TCR[0] == 0)
T0TCR |= 0; // Stop timer 1 (Timer Control Register)
}
```

Listing 6.6 A hardware delay function

6.5.5.2 Task properties on a FreeRTOS

The FreeRTOS kernel creates a task by instantiating and populating a TCB. Then, the tasks are allocated in the ready list in order of priority. Tasks with similar priority are serviced on a round-robin basis. Moreover, tasks can be synchronised by binary and counting semaphores and mutexes. In this example, mutex was used to synchronise the tasks. The usage of mutex will be discussed in detail in the next chapter. Error! Not a valid bookmark self-reference. depicts the parameters of the tasks to implement the traffic lights system on FreeRTOS.

Table 6.2 Task properties for the traffic lights system on the RTOS

Name	Delay	Period (ms)	Task priority
Button_Update	0	10	2
Update_Lights	Delay_c1 = 1000 Delay_c2 = 10000 Delay_c3 = 5000 Delay_c4 = 3000/6	-	1

The priority of the “Button_Update” task is set to 2 which denote it as the higher priority task than the “Update_Lights” task. To generate a delay for a task, the FreeRTOS Application Programming Interface (API) vTaskDelayUntill() function can be used as shown in Listing 6.7.

```
void Update_Lights(void)
        {
        ……
        while (TRUE)
```

```
{
portENTER_CRITICAL();
TRAFFIC_Set_Light_State(1, TRAFFIC_RED);
portEXIT_CRITICAL();

vTaskDelayUntil(&lastWakeTime, delay_c1);

portENTER_CRITICAL();
TRAFFIC_Set_Light_State(0, TRAFFIC_RED_AMBER);
portEXIT_CRITICAL();

vTaskDelayUntil(&lastWakeTime, delay_c1);

portENTER_CRITICAL();
TRAFFIC_Set_Light_State(0, TRAFFIC_GREEN);
portEXIT_CRITICAL();
……
}
```

Listing 6.7 Implementation of the "Update_Lights" task in FreeRTOS

From Listing 6.7, the two functions named portENTER_CRITICAL() and portEXIT_CRITICAL().These are mechanisms to protect simultaneous accesses of shared resources, such as semaphores and mutex. Note that in this chapter, neither ET, TTC nor TTH systems use the shared resources protection method except in the RTOS and TTP implementation. A complete discussion of this issue is presented later in this chapter.

6.5.5.3 Task properties on a TT architecture

The task's properties relative to the system are based on a TTC architecture, a TTH architecture and a TTP architecture are shown in **Error! Not a valid bookmark self-reference.**, For the TTH scheduler, besides the two basic task properties, each task must be set to either a TRUE or FALSE condition to determine whether it is a pre-emptive or non pre-emptive task. If it is TRUE, it means that it is a pre-emptive task, otherwise, it will behave as a non-pre-emptive task. In this scheduler, only a single task can be assigned as a pre-emptive task.

Table 6.4 and For the TTP scheduler, the task's priority needs to be set beforehand. The task with high priority (for example, the Button_Update task) can pre-empt the lower priority task (for example, the Update_Lights task).

Table 6.5. Because timer interrupts can be used to periodically implement a repeating task with a fixed time frame, the properties of the tasks, such as the delay and period must be determined prior to the runtime. For the TTC scheduler, two basic task properties need to be established before the runtime; which are the task's delay and task's period.

Table 6.3 Task properties for the traffic lights system on a TTC architecture

Name	Delay	Period
Button_Update	0	10
Update_Lights	0	500

For the TTH scheduler, besides the two basic task properties, each task must be set to either a TRUE or FALSE condition to determine whether it is a pre-emptive or non pre-emptive task. If it is TRUE, it means that it is a pre-emptive task, otherwise, it will behave as a non-pre-emptive task. In this scheduler, only a single task can be assigned as a pre-emptive task.

Table 6.4 Task properties for the traffic lights system on a TTH architecture

Name	Delay	Period	Pre-emptive
Button_Update	0	10	True
Update_Lights	0	500	False

For the TTP scheduler, the task's priority needs to be set beforehand. The task with high priority (for example, the Button_Update task) can pre-empt the lower priority task (for example, the Update_Lights task).

Table 6.5 Task properties for the traffic lights system on a TTP architecture

Name	Delay	Period	Priority
Button_Update	0	10	2
Update_Lights	0	500	1

Note that the tick interval is set to 1 millisecond. Thus, the "Button_Update" task will be invoked for every 10 ticks; while the "Update_Lights" task will be activated for every 500 ticks. For the TTH and TTP architecture, another property of a task is added as shown in Table 6.4 and Table 6.5.

6.5.6 Executing test for task in isolation

Previous sections have discussed the implementation of the traffic lights systems on three different platforms. In addition testing in an ET system using FreeRTOS was also included. The techniques to be applied to validate the timing constraints of dynamic systems are very demanding (Liu, 2000). For example, an interrupt which occurs in the complete systems could not be easily and precisely reproduced for the testing and debugging processes (Thane and Hansson, 2001); and the test coverage of an ET system is very high because interrupts can occur at an arbitrary point which leads to an enormous number of test inputs (Schultz, 1993).

In this context, the WCET behaviour of the tasks was observed. The WCET of the background tasks can be measured using observation points or testing points. In the

"Update_lights" code, the observation points of the traffic system needs to be specified. Test 1, Test 2, Test 3, are examples of testing points placed in the codes as shown in Listing 6.8.

```
void Update_Lights(void)
        {
        ……
                while (TRUE)
                {
                        //Test point 1
                        TRAFFIC_Set_Light_State(1, TRAFFIC_RED);
                        Hardware_Delay_T0(delay_c1);

                        // Test point 2
                        TRAFFIC_Set_Light_State(0, TRAFFIC_RED_AMBER);
                        Hardware_Delay_T0(delay_c1);

                        // Test point 3
                        TRAFFIC_Set_Light_State(0, TRAFFIC_GREEN);
                        Hardware_Delay_T0(delay_c2);
                        ……
                }
        }
```

Listing 6.8 The background task in isolation

In normal operation, interrupts (for example, when someone is pressing the button) should be invoked at the test point and the execution time of the update_lights() function is recorded. Eight (8) testing points were identified to find the WCET of the task, as shown in Figure 6.8.

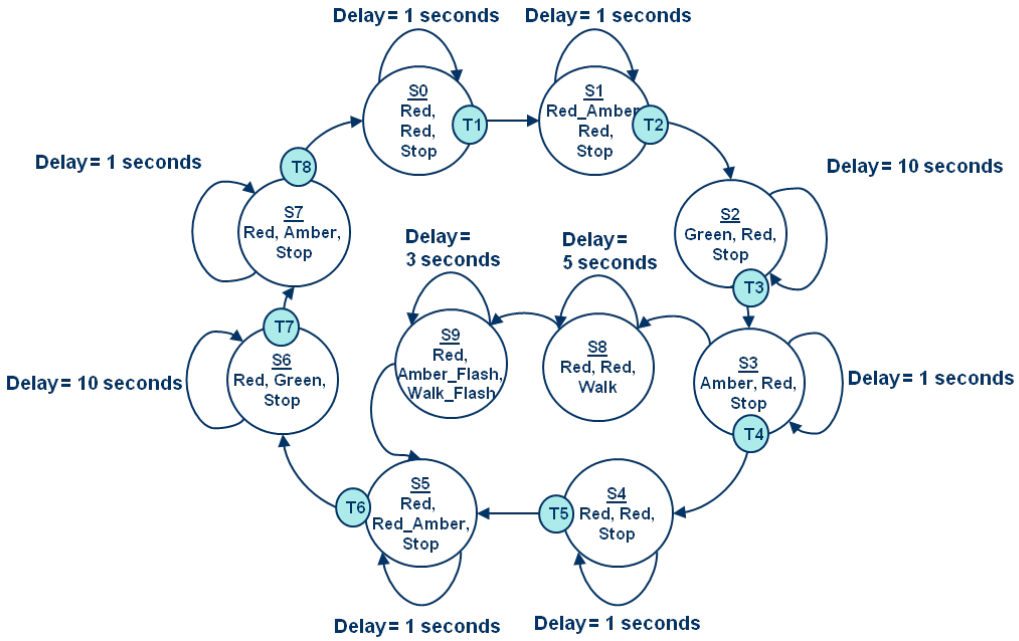


Figure 6.8 Testing and interrupt points

Every test must run independently and one at a time so that the timing behaviour of the task can be monitored when interrupts occur at a specific point. When implementing a task in isolation, the external interrupt is disabled. The variable, which is used to read the push button states is set to TRUE as revealed in Listing 6.9.

Complete System

```
void Button_Update(void)
{
	boolean_t pin_high = !GPIO_Read(BUTTON_PIN);
	if (pin_high == TRUE)
		{
			button_pressed = TRUE;
			TRAFFIC_Set_Wait_Light(TRUE);
		}
	//Clear Interrupt
}
```

Task in isolation

```
void Task_Harness_Button_Update(void)
{
	boolean_t pin_high = TRUE;
	if (pin_high == TRUE)
		{
			button_pressed = TRUE;
			TRAFFIC_Set_Wait_Light(TRUE);
		}

	//Clear Interrupt
}
```

Listing 6.9 Implement the task harness for the ET system

At test points, this Task_Harness_Button_Update () task is called and its functional and temporal behaviour should be similar with the task runs in the complete system. This strategy is applied to analyse impacts of interrupts. Moreover, most importantly, the task can be tested individually by employing as many test cases as possible to trace which of the tasks has the source of timing errors.

6.5.7 Using the task harness for testing in TT systems

Testing for tasks in isolation in a TT system is more straightforward. Figure 6.9 illustrates that the "task harness" is called after the isolated task, during the testing process.

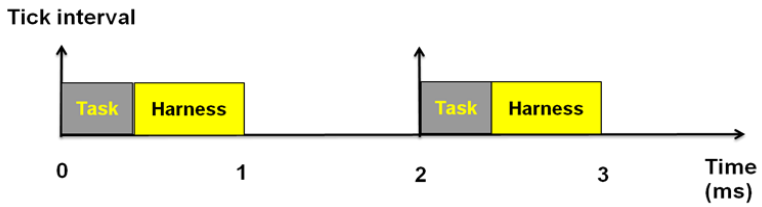


Figure 6.9 The task harness

Note that, the WCET of a task can only be obtained when all possible input values are considered. Thus, the task harness should contain potential input values required by lower priority tasks from higher priority tasks. For example, the states of the traffic light system are controlled by time values and input from the pedestrian push button. Therefore, to test lower priority tasks such as in the update_lights() task, the values of variables that are needed when the button pressed are declared in the task harness. Listing 6.10 provides an example of the task harness in the traffic lights system.

```
void Harness_Button(void)
{
	// Pass the values of the output ports to this function:
	static boolean_t pressed = TRUE;
	static uint32_t pressed_duration = 0;
	static boolean_t toggle_state = FALSE;

	Harness_Button_Output(pressed, toggle_state, pressed_duration);
}
```

Listing 6.10 The task harness function

Table 6.6, Table 6.7 and Table 6.8 presents the implementation of the task harness to test the update_lights() task.

Table 6.6 Task harness for the traffic light systems on a TTC architecture

Name	Delay	Period (ms)
Update_Lights	0	500
Harness_Button	0	5000

Table 6.7 Task harness for the traffic light systems on a TTH architecture

Name	Delay	Period (ms)	Pre-emptive
Update_Lights	0	500	False
Harness_Button	0	5000	False

Table 6.8 Task harness for the traffic light systems on a TTP architecture

Name	Delay	Period (ms)	Priority
Update_Lights	0	500	2
Harness Button	0	5000	1

It has been found that, the task harness runs co-operatively with the update_lights() task with respect to the TTC and TTH schedulers. Every 5000 milliseconds, the "Harness Button" task updates the value of the pressed variable to TRUE which represents the action of the push button being pressed. By using this strategy, the WCET of the task in the complete system can be produced as a task running in isolation.

6.5.8 Results for Case Study 1

The results of the experiments are shown in the following sub-sections. Note that the analysis focussed on lower priority tasks, in this case the update_lights() task. The aim of this was to ensure the impact of ET-based and TT-based architecture was observed. Table 6.9 presents the WCET of the update_lights() task for five different software architectures.

Table 6.9 The comparison of the execution times of the isolated and in-situ tasks

Software architecture	WCET (us) of Update_Lights task		Difference
	Task in the Complete System	Task in Isolation	
Event triggered architecture	34813142.38	33817934.41	995207.97
Full-featured RTOS	32997972.90	32997915.45	57.45
Time triggered architecture:			
TTP	132795	132955	160
TTH	132072	132152	80
TTC	132150	132152	2

The time requirement for implementing this task should be 33 seconds, as designed. The WCET for ET and RTOS architecture is obtained when the pedestrian presses the button. Hence, the duration between the start state from the S0 state to the finish state, the S9 state (Figure 6.5) should be 33 seconds, as designed.

From the measurement, the task running in the complete system in ET architecture had a WCET of approximately 34.8 seconds; whereas the task in isolation had a WCET of 33.81 seconds. There is almost a 1 second difference. By contrast, the difference is about 57.45 microseconds between the system running in RTOS. This means that ET finds it difficult to reproduce the test and detect the error.

For TT architecture, the WCET is different because the system design is dissimilar – using a multi-state (input/timed) strategy. As can be seen, TTP architecture provides a higher difference than the TTH followed by the TTC architecture.

The graph in Figure 6.10 illustrates the maximum difference of the WCET of the update_lights() task running in the complete system and the task in isolation.

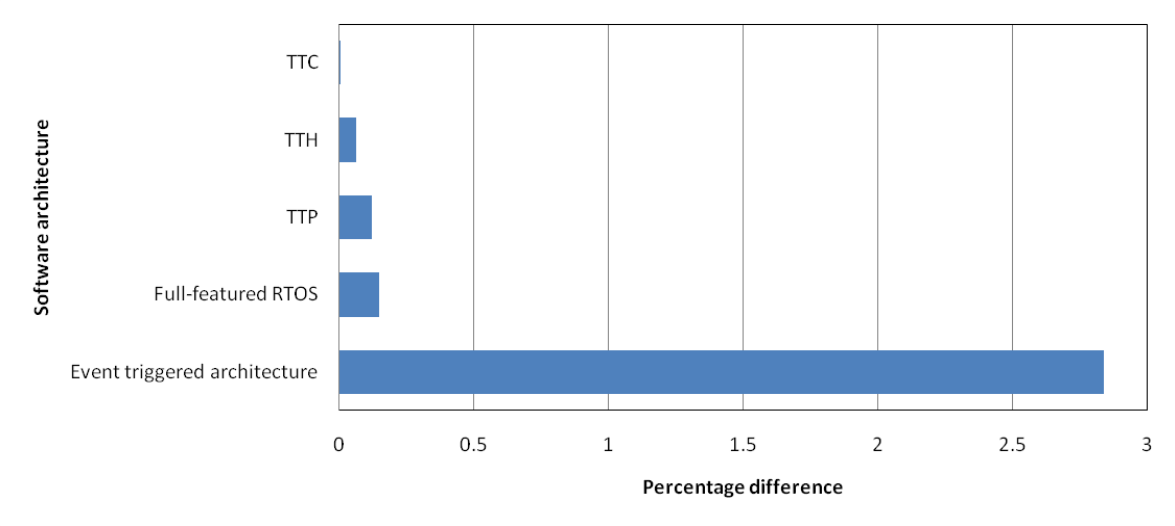


Figure 6.10 The maximum difference of the WCET of the "Update_Lights" task in percentage.

The results of this graph correspond to the data obtained and depicted in Table 6.9. The ET architecture showed the highest percentage of 2.8% to compare the WCET of the update_lights() task in isolation with its WCET in the complete system. Conversely, the other architecture, such as RTOS, TTP, TTH and TTC architecture have differences lower than 0.5%. Clearly, the TTC architecture provides the least difference with a percentage of approximately 0.001%.

6.6 Case study 2: Assessing the effects of shared resources mechanism

It is important to apply an appropriate method in order to analyse the impact of testing for pre-emptive systems which employ synchronisation mechanisms. The objective of the experiment was to compare the methods that could provide a small difference of the WCET of a task when it runs both in the complete system in isolation.

In this experiment, it was assumed that all overheads (as described in Chapter 2) were included in WCET of the lower priority task. Therefore, the analysis focussed on lower

priority tasks, in which the WCET is always affected by interruptions from higher priority tasks. This interruption would be expected to cause overheads. The measurement of the WCET was repeated for lower priority tasks and was executed using the FreeRTOS and TTP; each system employs different mutual exclusion methods, which are described in the following sub-sections.

The case study for the FFT system can be used to evaluate the extent to which the TT architecture can provide effortless task testing, using the task in the isolation strategy – when a long task is involved. This study involved the design of a FFT system in the TTP, TTH and TTC schedulers. A description of the case study is presented in this section.

6.7 The FFT system

The second case study used a Fast Fourier Transform (FFT) system. The system takes a buffer filled with samples and the maximum frequency (half the sampling rate) and returns the first harmonic frequency and displays it on LCD screen. There are three tasks involved; one for the sampling, one for the FFT and one for the output.

A Fast Fourier Transform (FFT) system. The system samples the generated signal (at 1 KHz), carries out a FFT on the sampled data and finally displays the output of the first harmonic frequency to the LCD screen, which consists of three required tasks; one for the sampling, one for the FFT and one for the output.

(i) Signal_Acquisition: This task samples the ADC data and stores them in a adc_buffer.

(ii) Frequency_Calculation: This task performs the FFT and produces the first harmonic frequency

(iii) Output_Value: This task displays the first harmonic frequency onto the LCD screen.

Communication between the tasks takes the form of shared-memory. The shared resources (in this case the buffer) are protected with disabled interrupts, disabled scheduling and semaphores.

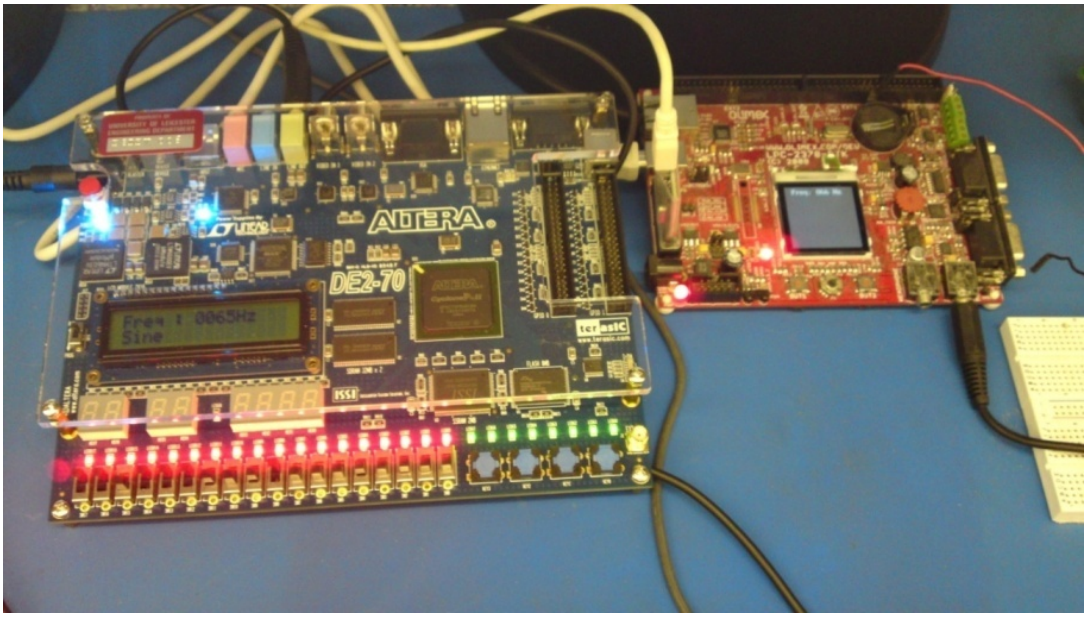


Figure 6.11 Hardware for FFT Systems for testing

The main concern was how to minimise testing efforts, which may arise in a uniprocessor system when concurrent tasks use shared resources, if the WCET of the isolated task and in situ task are very similar. That implies that shared resources mechanisms and inter-task communication techniques could make the testing timing properties simpler.

6.7.1 FFT functions

The system first samples the generated signal (at 1 KHz) and then performs a FFT on the sampled data before finally outputting the first harmonic frequency onto the LCD

screen. This involves three tasks; one for the sampling, one for the FFT and one for the output.

Signal_Acquisition:

Sample the ADC data and store them in an adc_buffer. The function is shown in Listing 6.11.

```
void Signal_Acquisition(TTE_UNUSED void *params)
{
	ADC_Init(SIGNAL_ADC, NULL);
	uint32_t index = 0;

	while (TRUE)
	{
		Enter_Critical();
		adc_buffer[index++] = ADC_Acquire_Raw_Reading(SIGNAL_ADC);
		Exit_Critical();

		if (index == FFT_BUFFER_SIZE)
		{
			index = 0;
		}

		vTaskDelay(1);
	}
}
```

Listing 6.11 **Signal_Acquisition()** function

Frequency_Calculation:

Perform the FFT and produce the first harmonic frequency. The function is shown in Listing 6.12.

```
void Frequency_Calculation(TTE_UNUSED void *params)
{
	portTickType lastWakeTime = xTaskGetTickCount();
	vTaskDelayUntil(&lastWakeTime, FFT_BUFFER_SIZE - 1);

	while (TRUE)
	{
		Enter_Critical();
		harmonic_freq = Perform_FFT(adc_buffer, 500, NULL);
		Exit_Critical();

		vTaskDelayUntil(&lastWakeTime, FFT_BUFFER_SIZE);
	}
}
```

Listing 6.12 **Frequency_Calculation()** function

Output_Value:

Display the first harmonic frequency to the screen. The function is shown in Listing 6.13.

```
void Output_Value(TTE_UNUSED void *params)
{
	portTickType lastWakeTime = xTaskGetTickCount();
	vTaskDelayUntil(&lastWakeTime, FFT_BUFFER_SIZE);

	while (TRUE)
	{
		Enter_Critical();
		uint16_t freq = harmonic_freq;
		Exit_Critical();

		char buff[] = "Freq:     Hz";
		buff[6] = ((freq / 100) % 10) + '0';
		buff[7] = ((freq / 10) % 10) + '0';
		buff[8] = (freq % 10) + '0';

		LCD_Send_String(buff, 5, 12, LARGE_FONT, WHITE, BLACK);

		vTaskDelayUntil(&lastWakeTime, FFT_BUFFER_SIZE);
	}
}
```

Listing 6.13 `Output_Value()` function

6.7.2 Task properties

In order to implement the FFT system in the FreeRTOS, the priorities and period of tasks need to be identified. As the **Signal_Acquisition()** task collects the ADC data without having to miss a single sample, it must be set to the highest priority. In this example, the higher the number of the task priority, the higher the priority becomes. After 256 data samples have been acquired and buffered from the ADC, the **Frequency_Calculation()** task processes the samples.

Table 6.10 Task properties of the FFT systems using FreeRTOS

Name	Period (ms)	Task priority
Signal_Acquisition	1	3
Frequency_Calculation	256	2
Output_Value	256	1

The system was also implemented in TT architecture. As with the TTC-based design, pre-emption was not allowed to occur; in fact the **Frequency_Calculation()** task has very long execution times, which may exceed the tick interval. Therefore, pre-emptive software architectures should be applied. In this case, a hybrid scheduler was considered as an intermediate solution between a TTC-based design and a fully pre-emptive solution. The task properties of a TTH-based design are shown in Table 6.11

Table 6.11 Task properties of the FFT s.ystems using TTH

Name	Delay (ms)	Period (ms)	Pre-emptive
Signal_Acquisition	0	1	True
Frequency_Calculation	256	256	False
Output_Value	256	256	False

Only the `Signal_Acquisition()` task is set to TRUE, which means it is allow to pre-empt other tasks. By contrast, the `Frequency_Calculation()`task and the `Output_Value()` task will behave co-operatively.

Table 6.12 presents properties of task using TTP architecture. This is a fully pre-emptive TT-based design.

Table 6.12 Task properties of the FFT systems using TTP

Name	Delay (ms)	Period (ms)	Task priority
Signal_Acquisition	0	1	3
Frequency_Calculation	256	256	2
Output_Value	256	256	1

6.7.3 Hardware measurements

A similar technique is used to measure the WCET of the tasks in which a pin on the ARM7 microcontroller is set high at the start of each measured task and is then set to a low point before the end of that task. The widths of the resulting pulses are measured using a National Instruments data acquisition card in conjunction with appropriate software. The resolution of the timing measurements is 0.1 microseconds. In each study, 10,000 consecutive pulse widths are measured for the lower priority task in each experiment in order to provide the results presented in this thesis.

6.7.4 Experimental methodology for shared resources

There are several techniques to be employed in order to protect shared resources using FreeRTOS. In this section, some of the methods applied in the experiment are described.

6.7.4.1 Critical sections

In order to protect shared resources which are said to be serially reusable, `Enter_Critical()` and `Exit_Critical()` functions are used. All the shared resources include certain peripherals, shared memory and the CPU should be protected inside these functions as shown in Listing 6.14.

```
Enter_Critical();

// Critical section codes
// This code will be executed only by one thread at a time

Exit_Critical();
```

Listing 6.14 Critical section

These sections of codes are known as critical sections of codes. Once a task entered the sections, it cannot be pre-empted.

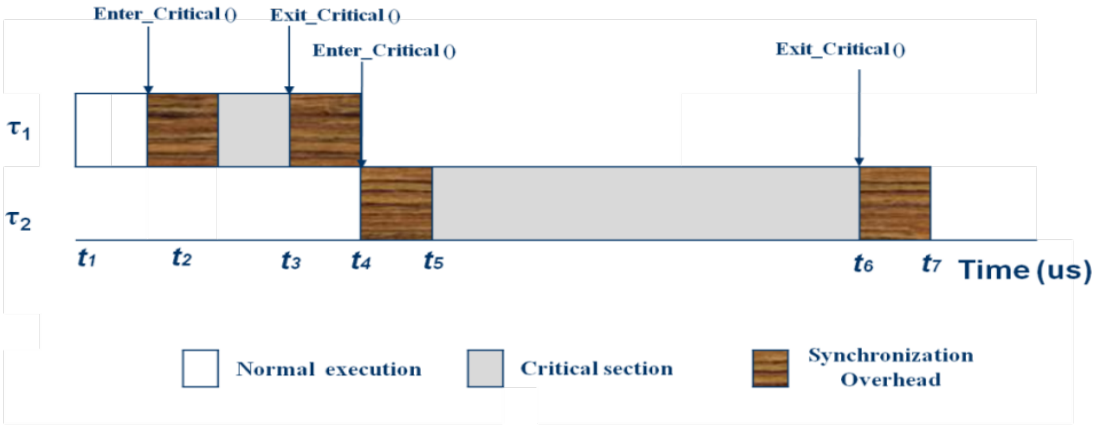


Figure 6.12 Illustration of Synchronization Overhead

6.7.4.2 Disabling and enabling interrupts

One of the simplest ways to gain exclusive access to a shared resource is by disabling and enabling interrupts. In order to implement this, FreeRTOS provides two macros: to disable and then enable interrupts from the C code: `portENTER_CRITICAL()` and `portEXIT_CRITICAL()`, as shown in Listing 6.15.

```
void Enter_Critical(void)
{
portENTER_CRITICAL();
}

void Exit_Critical(void)
{
portEXIT_CRITICAL();
}
```

Listing 6.15 Disabling and enabling interrupts

6.7.4.3 Disabling and enabling scheduling

Alternatively, if two or more tasks can share data without the possibility of conflicts, disable and enable scheduling can be used. While the scheduler is locked, interrupts are enabled, and if an interrupt occurs whilst in the critical section, the ISR is executed immediately. At the end of the ISR, the kernel always returns to the interrupted task, even if the ISR has made a higher priority task ready to run. The scheduler is invoked when xTaskResumeAll() is called to see if a higher priority task has been made ready to run by the task or an ISR. The sample for assessing the shared data by disabling and enabling scheduling is shown in Listing 6.16.

```
void Enter_Critical(void)
{
vTaskSuspendAll();
}

void Exit_Critical(void)
{
xTaskResumeAll();
}
```

Listing 6.16 Disabling and enabling scheduling

6.7.4.4 Semaphores

The most popular synchronisation mechanism offered by most multi-tasking kernels is the semaphore. There are three folds of semaphores usage included to control access to a shared resource (mutual exclusion), to signal the occurrence of an event and to allow two tasks to synchronise their activities.

Tasks designed to synchronise their activities execute *wait* and *signal* operations on shared semaphores. If a task executes a *wait* operation and the value of the semaphore is one or greater, then the task can decrement the semaphore and continue. If the semaphore has the value of zero at the time the process executes the wait operation,

then decrementing the semaphore would result in a negative value. In order to continue execution, the code should acquire a key or a semaphore.

```
// The semaphore has been previously constructed
void Enter_Critical(void)
{
xSemaphoreTake(lock);
}

void Exit_Critical(void)
{
xSemaphoreGive(lock);
}
```

Listing 6.17 Semaphore

6.7.4.5 Disabling and enabling interrupt Mutex

The implementation of mutex is shown in Listing 6.18. In TTP scheduling, a mutex mechanism is used to provide synchronisation of shared resources amongst tasks.

```
// The mutex has been previously constructed
void Enter_Critical(void)
{
Lock_the_mutex();
}
void Exit_Critical(void)
{
Unlock_the_mutex();
}
```

Listing 6.18 Mutex

6.7.4.6 Message queue

The implementation of the system is not similar to previous synchronisation methods, since this form requires a task to send “messages” into a queue as represented by the Signal_Acquisition ( ) task (in `Listing` 6.19); and another task is to receive “messages” from a queue and then perform the process as represented by the Frequency_Calculation ( ) task (in Listing 6.20).

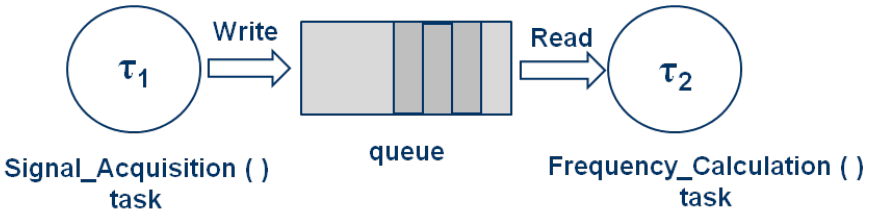


Figure 6.13 Illustration of message queues

Inter-process communication is achieved via the creation of queues. Most information exchanged via queues is passed by value and not by reference which should be a consideration for memory constrained applications. The queue reads or writes from within the interrupt service routines (ISRs) which are non-blocking. The queue reads or writes with zero timeout which are non-blocking. All other queue reads or writes block with configurable timeouts.

```
void Signal_Acquisition(TTE_UNUSED void *params)
{
	ADC_Init(SIGNAL_ADC, NULL);

	while (TRUE)
	{
	uint16_t adc_reading = ADC_Acquire_Raw_Reading(SIGNAL_ADC);

		if (xQueueSend(readings, &adc_reading, 0) != pdPASS)
		{
			Enter_Safe_State();
		}

		vTaskDelay(1);
	}
}
```

Listing 6.19 Send message queue

```
void Frequency_Calculation(TTE_UNUSED void *params)
{
while (TRUE)
	{
	static uint16_t adc_buffer[FFT_BUFFER_SIZE];

	for (uint32_t i = 0; i < FFT_BUFFER_SIZE; i++)
	{
	if (xQueueReceive(readings, &adc_buffer[i], 1) != pdPASS)
		{
			Enter_Safe_State();
		}
	}
```

```
        static uint16_t freqs[FFT_FREQ_COUNT];
        uint16_t harmonic_freq = Perform_FFT(adc_buffer, 500, freqs);

        if (xQueueSend(harmonic, &harmonic_freq, 0) != pdPASS)
                {
                        Enter_Safe_State();
                }

        for (uint32_t i = 0; i < FFT_FREQ_COUNT; i++)
                {
                if (xQueueSend(frequencies, &freqs[i], 0) != pdPASS)
                        {
                                Enter_Safe_State();
                        }
                }
        }
}
```

Listing 6.20 Receive message queue

6.7.5 Executing task in isolation with shared resources protection mechanisms

It is not an easy task to test tasks in isolation, when shared resources protection mechanisms are involved. If a resource is being shared, this means that the values of the shared variables can also be modified by other tasks. Therefore, it is important to understand the possible inputs or data required for testing a task in isolation.

The main concern is how to isolate lower priority tasks with a consideration of synchronisation, inter-task communication and protection mechanism issues. Protection mechanisms can be very complex and may make the task too complicated to test independently. Thus, it is important to identify which mechanisms can provide a small difference in execution times between the task which is executing in the complete system and the task which is running individually. Thus, the study aimed to compare the effects of timing behaviour of the tasks when the synchronisation involves both tasks (which can be observed when tasks are running in a complete system); and when there is no synchronisation involved (which can be observed when the task is running in isolation).

In order to test a task in isolation for the lower priority Frequency_Calculation ( ) task, samples of ADC data are required. Supposing the data gathered by the ADC reading is

a sine wave, then a sample of the data time can be placed in that buffer for testing purposes as shown in Listing 6.21. These data represent sine wave input data which is stored in a shared buffer between the Signal_Acquisition () task and the Frequency_Calculation ( ) task.

```
static uint16_t adc_buffer[FFT_BUFFER_SIZE]=
{30, 143, 46, 308, 355, 18, 563, 0, 461, 195, 157, 490, 0,
550, 46, 320, 348, 21, 427, 90, 256, 404, 0, 560, 0, 406, 263, 93, 526, 0,513,
112, 243, 418, 0, 556, 0, 389, 280, 76, 532, 0, 501, 130, 223, 433, 0,555, 6,
358, 299, 65, 540, 0, 493, 147, 208, 448, 0, 553, 17, 341, 315, 52,542, 0,
481, 161, 190, 473, 0, 549, 32, 325, 338, 35, 547, 0, 469, 181, 172,481, 0,
547, 40, 309, 347, 25, 552, 0, 456, 197, 156, 492, 0, 544, 54, 292,365, 13,
555, 0, 445, 211, 142, 501, 0, 538, 77, 282, 379, 2, 554, 0, 427, 230,124,
512, 0, 529, 97, 251, 396, 0, 559, 0, 414, 248, 109, 521, 0, 523, 113,
249,411, 0, 560, 0, 398, 262, 80, 528, 0, 510, 127, 233, 427, 0, 558, 1, 383,
281,65, 535, 0, 503, 145, 218, 439, 0, 555, 9, 367, 298, 52, 543, 0, 495, 160,
199,456, 0, 552, 21, 348, 314, 40, 547, 0, 482, 168, 172, 467, 0, 548, 34,
334, 329, 26, 552, 0, 471, 185, 164, 479, 0, 543, 48, 317, 361, 20, 553, 0,
459, 202, 151,492, 0, 540, 60, 299, 376, 8, 557, 0, 447, 220, 137, 500, 0,
536, 75, 283, 393,0, 559, 0, 432, 235, 118, 512, 0, 526, 90, 267, 409, 559,
400, 253, 103, 299, 376, 8, 557, 393, 512, 0, 526, 90, 267, 409, 559, 400,
253, 103, 299};
```

Listing 6.21 Sampled data in the shared buffer for testing purposes

Once the data is defined, then the Frequency_Calculation ( ) task is ready to be tested in isolation – without the Signal_Acquisition () task and the Output_Value task as in the complete system. The WCET of the Frequency_Calculation ( ) task is recorded and compared with the WCET of the task while running in the complete system.

6.7.5.1 Shared resources in the FFT system

In order to demonstrate the effectiveness of shared resources of inter-task communication issues in the testing of an embedded system, a FFT system is used. In the FFT system, the tasks need to communicate with each other through the use of global variables as shown in Listing 6.22 and Listing 6.23.

The code shows that the adc_buffer [] array is filled with 256 samples by the Signal_Acquisition () task. Because the Frequency_Calculation () task executes once every 256 milliseconds or 256 of the Signal_Acquisition () task which is called every 1 millisecond, it begins to execute after the Signal_Acquisition () task is released, and the locking mechanism of the adc_buffer [] array.

```
// A shared (global) variables
uint16_t adc_buffer[FFT_BUFFER_SIZE];
void Signal_Acquisition()
{
        …….
        Enter_Critical();
        adc_buffer[index++] = ADC_Acquire_Raw_Reading(SIGNAL_ADC);
        Exit_Critical();
        …….
}
```

Listing 6.22 Shared buffer in the **`Signal_Acquisition()`** task

```
// A shared (global) variables
extern uint16_t adc_buffer[FFT_BUFFER_SIZE];
void Frequency_Calculation()
{
        …….
        Enter_Critical();
        harmonic_freq = Perform_FFT(adc_buffer, 500, NULL);
        Exit_Critical();
        …….
}
```

Listing 6.23 Shared buffer in the **`Frequency_Calculation()`** task

Regardless of any pre-emption, the Frequency_Calculation () task is free to process the last 256 samples. Note that, the pre-emptive task (in this case, the Signal_Acquisition () task) cannot gather and buffer the next 256 samples since it is being blocked by the Frequency_Calculation () task. By using this technique, the shared variable or shared hardware can be prevented from any conflict.

Alternatively, to avoid resource conflicts without using critical section protections, double-buffered data arrays can be used as part of the software architecture. Instead of using a single adc_buffer[]array, a double-buffered adc_buffer[x][y]array can be used where *x* is an index variable to indicate the active array and *y* is the allocation of 256 ADC data samples. The *x* value will be toggled after 256 samples are gathered which means that the next 256 data samples will be buffered in the second 256-size-buffer. Using this technique, the data are sampled without being blocked by other tasks. In fact, this is an easy way to implement this shared array without ever having to drop a single sample.

6.7.5.2 Shared resources in the traffic light system

In the traffic lights system example, the LCD is an example of shared hardware resource hardware. The LCD is used to graphically simulate the traffic lights functions, as well as the pedestrian traffic lights. Each task needs to take its turn in using the LCD. For example, in the Update_Lights () task, functions that will draw the traffic lights' colours on the LCD will be protected as shown in Listing 6.24.

```
void Update_Lights()
{
……
            Enter_Critical();
            TRAFFIC_Set_Light_State(1, TRAFFIC_RED);
            Exit_Critical();
……
}
```

Listing 6.24 An example of shared resources mechanism for the traffic lights system

As can be seen in Listing 6.24 when the TRAFFIC_Set_Light_State(1, *TRAFFIC_RED)* function is executing which means that the LCD is in use, other tasks are not allowed to pre-empt or use the LCD. When an interrupt occurs during this execution (for example, when the button is pressed), and the Button_Update () task

wants to access the LCD (as shown in Listing 6.25), the task has to wait until the Update_Lights () task "unlocks" the usage of the LCD.

```
void Button_Update()
{
.......
                Enter_Critical();
                if (!button_pressed)
                {
                       button_pressed = TRUE;
                       TRAFFIC_Set_Wait_Light(TRUE);
                }
   Exit_Critical();
   ……..
}
```

Listing 6.25 Critical section in higher priority task

Once the shared resource is unlocked, the LCD can now be used by the **Button_Update()** task to display the "WAIT' light onto the LCD screen. The effectiveness of the approach used to protect the shared resources is examined. The main concern is to find the best approach which produces less time difference between the task running in the complete system and the task running in isolation in the light of the protection of shared resources.

6.7.6 Results for Case Study 2

In the second case study, the system required a pre-emption task for the sampling of data in every 1 millisecond and needed to perform the FFT task, which has a very long execution time. Thus, the TTC was excluded when testing. However, the TTH scheduler (which supports a single time-triggered pre-emptive task can be added to a TTC scheduler) and can be employed to test the system.

The double-buffered technique is also employed in the TTH implementation, along with lock-based synchronisation mechanisms.

Table 6.13 shows the synchronisation and shared resource protection strategies (implemented using standard RTOS) which have a different impact on the WCET of the task, ranging approximately from 90 to 130 microseconds.

Table 6.13 Comparison of WCET of the Frequency_Calculation ( ) task execution times of an isolated task and a task in a complete system

Software architecture	WCET of Frequency_Calculation task (us)	
	Task in the complete system	**Task in isolation**
Full-featured RTOS		
Disable Interrupt	6968.95	6875.10
Disabled Scheduling	6989.95	6886.95
Mutex	6995.50	6860.00
Binary Semaphore	7004.95	6905.83
Time-triggered architecture		
TTH	6858.05	6823.78
TTP (Mutex)	6995.24	6921.51

Disable interrupt techniques provide less difference than the other lock-based mechanisms. In contrast, mutex gives the highest difference amongst other techniques. A graph in

Figure 6.14 illustrates the comparison of the difference between the WCET of Frequency_Calculation ( ) task using different shared resources strategies.

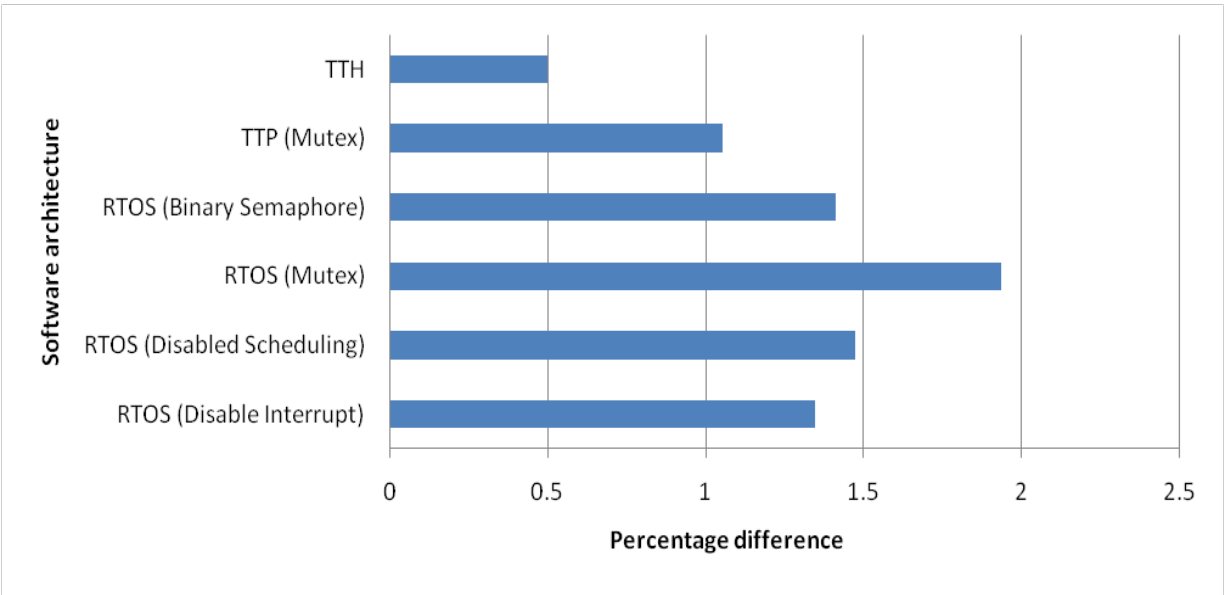


Figure 6.14 The maximum difference of WCET of `Frequency_Calculation` ( ) task

TTH, which uses the lock-free based shared resources technique, showed a small difference of approximately 0.5%. The other software architectures, which support the pre-emptive scheduler, showed a difference between 1.5% and 1.8%. The TTP scheduler uses mutex techniques to synchronise the shared data. As can be seen, the difference of the isolated WCET and complete WCET of the task running in the TTP scheduler is smaller than the mutex implementation in RTOS by 0.7%. This may be due to the fact that the mutex implementation in RTOS is more complicated than its implementation in the TTP scheduler. This also shows that, the disable interrupt results in the lowest difference in comparison to the other FreeRTOS synchronisation strategies.

The Frequency_Calculation ( ) task's function takes a buffer filled with samples and the maximum frequency (half of the sampling rate) and returns the first harmonic frequency and displays it onto the LCD screen. Communication between the tasks will take the form of shared-memory. The shared resources (in this case a shared buffer) are

protected with disabled interrupts, disabled scheduling, semaphores and mutex. In addition, the message queue is also implemented to the system. Tasks with a higher priority sample the data and send it to the message queue. Conversely, the other task will receive the message queue and put it into the buffer. This technique requires 'send and receive' or 'write and read' text on the message queue.

The 'send message queue' task must have higher priority than the 'receive' or 'read' from the message queue task. Thus, it is not possible to isolate the task which utilises the message queue mechanism as performed by the Frequency_Calculation ( ) task and the signal acquisition task in the FFT system.

Table 6.14 presents the WCET of the Update_Lights()task implementation using four different software architectures and four different synchronisation and inter-task communication techniques. The WCET of the Frequency_Calculation ( ) task with the same implementation (excluding the TTC scheduler) as in the first case study is shown in table 6.13

It is evident that a system with the TTC scheduler could easily reproduce the isolated task with similar behaviour as the task which is running in the complete system. A task which runs under the co-operative scheduler has less overheads and no pre-emption related costs in comparison to the pre-emptive schedulers. The results show that:

$$C_{preempt} > C_{co\text{-}opeative}$$

Table 6.14 Comparison of WCET of the Update_Lights()task execution times of an isolated task and a task in a complete system

Software architecture	WCET of Update_Lights()task (us)	
	Task in the Complete System	**Task in Isolation**
Full-featured RTOS		
Disable Interrupt	33798886.33	33792841.94
Disable Scheduling	33798500.33	33792841.94
Mutex	32997972.90	32997915.45
Semaphore	32998018.25	32997915.23

Where $C_{preempt}$ is the execution time of a task in the pre-emptive scheduler and $C_{co\text{-}operative}$ is the execution time of a task in the co-operative scheduler. Overheads in the pre-emptive scheduler are caused by the time needed to save the state of the active task to a task-control block and the time to load the new active task state from the ready state; and also the effects of using synchronisation and inter-task communication strategies.

In the TTC system (for the Update_Lights () task), the difference between the execution times for the isolated task and the same task “in situ” is 0.0015% (see Figure 6.15). Systems using the TTH and TTP schedulers have a difference between 0.06% and 0.15%. By contrast, in the RTOS system, the difference is approximately between 0.12 % and 0.14%.

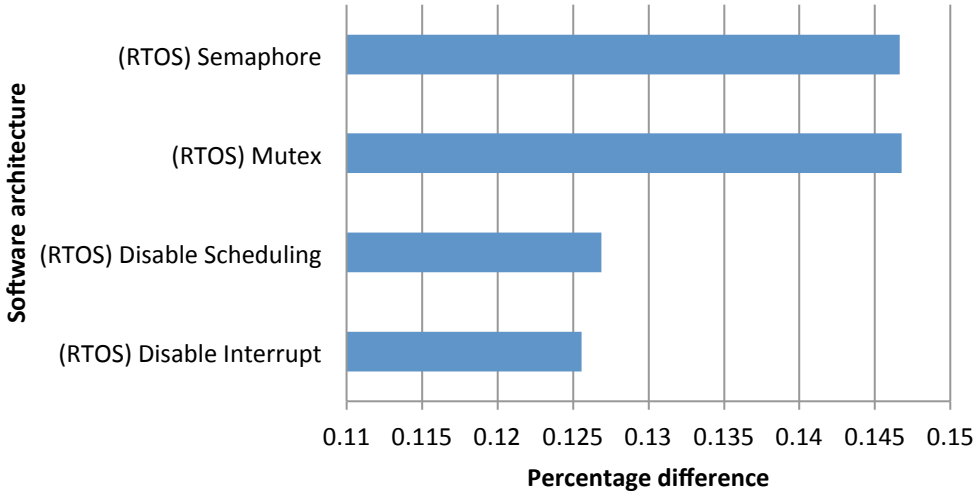


Figure 6.15 The maximum difference of WCET of `Update_Lights()`task

6.8 Discussion

This section discusses the results of Case Study 1 and Case Study 2.

6.8.1 Discussion for Case Study 1

'Interrupt only' systems are a special case of foreground/background systems, which are widely used in embedded systems. The systems are easy to write and typically have fast response times because the scheduling process can be done via hardware. One major drawback of these systems is the time wasted in the jump-to-self loop and the difficulty in providing advanced services such as device drivers and interfaces to multiple-layered networks. This procedure can be tedious and error-prone. Another disadvantage is its vulnerability to malfunctions owing to factors including timing variations, unanticipated race conditions and, hardware failure. Some companies avoid such designs, which are based on interrupts for these reasons.

Conversely, the TT approach provides a more attractive option for real-time systems requiring highly reliable behaviour, due to its predictability and safety benefits on offers (Allworth, 1981; Nissanke, 1997 and Pont, 2001). In the TT architecture,

numerous scheduling algorithms have been developed with different system behaviours. Some commonly used time-triggered scheduling algorithms are co-operative scheduling, pre-emptive scheduling and hybrid scheduling (Pont, 2001).

Systems that utilise the pre-emptive scheduler have more overheads in comparison to the co-operative based scheduler. Thus, it is much easier for the TTC scheduler to produce similar timing data for isolated tasks and in-situ tasks than the TTH and the TTP scheduler. If the TTC scheduler is used, similar timing data of a task can be reproduced as close to 99% to the task which is running in the complete system. Furthermore, some problems like priority inversion and recursive deadlock are observed as occurring during run-time only and are proven to be very difficult to reproduce. Thus, this makes the testing of pre-emptive based systems more difficult. In contrast, tasks run to completion, and the mutual exclusion issues are eliminated within the remit of the co-operative scheduler.

6.8.2 Discussion for Case Study 2

As noted in Section 6.5, the traffic lights system runs two tasks, the Update_Button () task and the Update_Lights () task which share a mutually exclusive resource (in this case, the button state variable), on which both operations are defined. Thus, the code implementing such operations is a critical section which must be executed in mutual exclusion. In these cases (except for the TTC scheduler), pre-emption is allowed and the Update_Button () task which has a higher priority than the Update_Lights () task, then one task can block another from accessing the critical section. For example, the Update_Lights () task is activated first and after a while, it enters the critical section and locks the semaphore. While it is executing its critical section, the Update_Button () task arrives as it has a higher priority, pre-empting the Update_Lights () task and starts

executing. However, when attempting to enter its critical section, it is blocked either using the disabled interrupts technique, the semaphore or mutex and the Update_Lights() task resumed. The Update_Button() task is blocked until the Update_Lights() task releases the critical section by enabling the interrupts or executing the signal(s) primitive, which unlocks the semaphore.

Temporarily masking or disabling interrupts offers the lowest overheads which prevent simultaneous access to a shared resource. However, this method is typically used only when the critical section has a few instructions and contains no loops. When a binary semaphore is used for this purpose, then each critical section must begin with a wait(s) primitive and must end with a signal(s) primitive.

Nevertheless, semaphore implementation may consume the inherent dangers such as recursive deadlock, priority inversion or task-death deadlock. All these problems occur at run-time and can be very difficult to reproduce and make the testing and debugging processes harder. The results also show that the implementation of these techniques can increase the system's overheads by 1% in comparison to the TTH synchronisation approach, and therefore making testing more difficult.

The results of the overhead cost of the traffic lights system and the FFT system are not similar. The difference is mainly caused by the interrupt activities and significantly depends on the system operations.

The concept of ownership in the mutex principles enables problems of semaphore implementation to be addressed. Mutexes guarantee that only one task can lock a given mutex. When that task unlocks the mutex, the other tasks can enter that code region. Semaphore and mutex operations are invoked each time a critical section is accessed and this represents a significant run-time overhead. The WCET of a task using the

semaphore mechanism occurs when the semaphore is already locked and when the system call is made. This type of overhead is associated with locking and unlocking the semaphores.

Finally, the message queue technique was implemented on the FFT system. However, it is not possible to isolate the task as it requires two tasks for 'writing' and 'reading to and from' the message queue.

6.9 Conclusion

The chapter has presented the results from the case studies, which explored the testing of systems with a TT architecture and an ET architecture using pre-emptive scheduling. The results showed that – with the TT architecture – the timing behaviour of "isolated" and "in-situ" tasks is similar whereby the WCET of a task in TTC scheduling was easily reproduced for testing purposes. Conversely, when TTP scheduling was used, the WCET of a task was difficult to produce. Hence, from the results, it is apparent that the second hypothesis "*Testing a system with a TT architecture incurs less cost than testing an equivalent system with an ET architecture, when an experimental-based method and comparative analysis are used"* could be used to evaluate cost of testing for different software architecture.

Systems that utilise a pre-emptive scheduler have more overheads in comparison to the co-operative based scheduler. The impact of additional synchronisation overheads in testing real-time systems has been discussed in this chapter. Some problems like priority inversion and recursive deadlock only occur during run-time and are very difficult to reproduce. Thus, this will make the debugging of pre-emptive based systems more difficult. In contrast, tasks run to completion using the co-operative scheduler, therefore causing and the mutual exclusion issues to be eliminated. In addition, using

lock-free techniques to underlying TT software architecture (i.e. TTH scheduler) could also reduce the testing effort in reproducing similar timing behaviour for any task that is running in isolation and the ones running in a complete system. It is anticipated that the findings of this initial study will lead to development in the field of cost testing assessment.

Chapter 7

Effects of ET and TT architecture on the Cost of Verification at the Design Phase

7.1 Introduction

This chapter studies the cost analysis from the design perspective and provides the results of the final hypothesis under testing. The problem is defined and the parameters are identified which help to assess the cost of verification with the TT and ET design. The results are discussed at the end of this chapter.

7.2 Problem statement

In order to perform a design cost analysis for the software architectures TTP, TTC, ETC, ETP, the following hypothesis needs to be proven:

The cost of verifying a system with TT design is always higher than that required to verify an equivalent system with various types of ET design.

7.3 Problem description

In embedded systems, especially in safety-critical designs, the cost is assessed in terms of intangible properties, such as the complexity and the time taken to design, time taken and verifying/testing the design. In other words, a good design is one which is carried out in the shortest possible time, utilises minimal resources, is easy to test and verify and is easy to implement, whilst meeting all the set requirements. Therefore, a cost-effective design would have all of the above.

In order to assess the cost-effectiveness, the parameters which form the basis of this analysis need to be defined, so as to compare the four architectures in the problem statements. The most critical functional block of a real-time embedded system, i.e. the

task scheduler, was chosen for the studies. Since the scheduler is embedded software, its design depends on the underlying selected architecture, and it must strictly meet given conditions, which is believed to assist in developing the necessary arguments for our said hypothesis. In addition, it is important to note that for each of the underlying architectures, the scheduler has to be designed accordingly, which turns out to be quite different in each case.

A fundamental schedulability test approach in the real-time scheduling is assessed on: (1) comparison of the schedulability test between the ET and TT software architecture on their running time, lines of code and the required test inputs; and (2) examination of the impact of the testing performance with increased number of tasks.

7.4 Adopted methodology

In order to deduce the cost of design, schedulability analysis was utilised, therefore defining the tangible parameters for its analysis. The parameters are described as follows:

- **Number of inputs required:** This consists of a set of task specifications for each architecture which is varied in terms of number and attributes. For instance, the most common and important ones are: Worst case execution times, deadlines and periods. A complete list is stated in Table 7.6.
- **Test running time**: This refers to the time it takes to run a single test for a given set of tasks. It has to be measured, for the implemented schedulability test algorithm, which is not the scheduling algorithm.
- **Lines of code**: This is the number of instructions written in the programming language, which are used to implement the schedulability analysis.

The use of a higher number of inputs has made the analysis more realistic and accurate, i.e. the test results are closer to the realistic implementation. Conversely, much effort is required to identify all task constraints, which indirectly requires more time in development, thus adding complexity to the design's cost.

If the test running time is long, this causes the design to be time-consuming and cumbersome as it is now imperative to carry out tests during the design phase in order to verify the coding.

It has been initially thought that small codes guarantee full functionality, however this is highly unlikely as in real applications lines of code indirectly influence the complexity of the programme. Therefore the larger the code size is, the more difficult it is to test and debug, causing an increase in complexity, and making it harder to maintain. Thus this can only mean that the lines of code are basic, meaning only the validity of the parameters needs to be assessed.

Algorithms are ported to a single programming language, i.e. C for TTSA; and RTA and Heurisic from Matlab codes (TORSCHE) (Sucha et al., 2006).

The pseudo codes for each are found in the appendices section (refer to Appendix -C). In order to find LOC, the algorithms are ported to C. It was difficult to implement the complete functionality especially the heuristic search, and therefore, in this case the actual comparison is undertaken using the original codes.

The test running time is implemented in Matlab (Mathworks, 2010) and Visual C++ (Microsoft, 2010). The time required for the code to run, was measured using a timer function.

Table 7.1 Methods used for measuring the test running time, for the TTSA, RTA and heuristic algorithms.

Schedulability Test Algorithms	Programming Environment	Measurement Method	Number of Tasks
TTSA	Visual C++	Custom Timer Function	2, 10, 50, 80, 100
RTA	Matlab	Custom Timer Function	2, 10, 50, 80, 100
Heuristic	Matlab	Custom Timer Function	2, 10, 50, 80, 100

As the programming environment used to run the schedulability test was different, the difference between the timer functions in Matlab and Visual C++ were identified and carried out by implementing the same logical function with a similar code size. The running times for both programming environments was compared. The results showed that, without being fully optimised, the Visual C++ running time was greater than Matlab by a factor 0.358.

7.5 Experiment setup

7.5.1 System specifications

The algorithms for the test running time were tested, on a desktop PC with a 2.66GHz Intel Core Duo Processor E6750, 2GB RAM, running Windows XP Professional (v2002 sp3), MATLAB R2010a, and Microsoft Visual C++ (2010 Ultimate).

7.5.2 Test set generation

The tests were carried out for the stated number of tasks, three times per test, with the average times calculated as described in the results. A task set was established prior to testing the algorithms. A successful task set was created using the Hit and Trial method in which it was schedulable by all of the three algorithms. A typical task set is depicted in Table 7.2, Table 7.3 and Table 7.4.

Table 7.2 The RTA sample task set.

Name	Processing Time	Period
t1	51	1000
t2	3000	5000
t3	2000	25000
t4	1000	25000
t5	1000	40000
t6	1000	40000
t7	1000	50000
t8	1000	50000
t9	1000	50000
t10	2000	80000

Table 7.3 Sample data of heuristic search schedulability test.

Name	Processing Time	Release Time
t1	3	10
t2	5	9
t3	5	7
t4	5	2
t5	9	0
t6	3	10
t7	5	9
t8	5	7
t9	5	2
t10	9	0

Table 7.4 Sample data of TTSA Schdelability test (Gendy, 2008).

Task	WCET (us)	Deadline (us)	Period (us)	Jitter (us)	Exclusion	Precedence	Distance (us)	Latency (us)
1	496	3964	4000	1614	Task A excludes Task C	Task A precedes Task C	Distance between Task A and Task C is 3335	Latency between Task A and Task C is 3921
2	828	4711	10000	9488				
3	64	3673	4000	67				

Table 7.2, Table 7.3 and Table 7.4 illustrate the time taken from when the test input starts to be processed to when the test output is produced. It was found that when a TTSA test is carried out the number of task sets are tested in order to determine their schedulability and run times are calculated. Conversely, the number of task sets in RTA are declared schedulable by each test when running times are calculated.

A set of task specifications including task execution time and period were required as the test input for the RTA and TTSA tests. However, for the latter, some additional inputs of tasks constraints such as jitter, distance, latency, precedence and exclusion were necessary also. The outputs from the RTA test were:

- The worst case response time of the task.
- The result of the schedulability of the tasks i.e. whether the tasks are schedulable or not.
- The list of schedulable tasks.

The outputs of TTSA test were:

- The list of schedulable tasks and their offsets.
- The list of unschedulable tasks.
- The tick interval.

The outputs of the heuristic search test were:

- The results of the schedulability of the tasks.
- The list of schedulable tasks.

In this test, periods were randomly distributed in the interval [Pmin, Pmax], where Pmin = 1 and Pmax = 100,000,000 unit of time; and the WCET, *Ci* were randomly distributed in [0, 10000]. The task characteristics and constraints were randomly generated in accordance to the test of the input requirements. The details of the numerical values used for parameters are shown in Table 7.5.

Table 7.5 Numerical values used for test running time analysis.

	Response Time Analysis for ETP	**Time Triggered Schedulability Analysis for TTC and TTP**	**Heuristic Search for ETC**
WCET	[0, 10000]	[0, 10000]	[0, 100]
Release Jitter	-	[0, 1000]	-
Period	[0, 100000000]	[0, 100000]	-
Deadline	-	[0, 10000]	
Distance	-	[1,10000]	-
Precedence	-	[0, 1]	-
Exclusion	-	[0, 1]	-
Latency	-	[1,10000]	-
Jitter	-	[0, 10000]	-
Scheduling Overhead	-	[0,100]	-
Release Time	-		[0,100]
Offset	-		

The execution time of the schedulability test for the sets of tasks was assessed using the RTA, TTSA and heuristic search (based on earliest computation first) algorithms. The measurements of the execution time were carried out using a simple (custom) schedule simulator, running on a desktop PC 2.66GHz Intel Core Duo CPU. The time taken from the start of the input being processed and produced is recorded. The setting for the TTSA test entails the number of task sets tested schedulable or not by each test with its running time calculated. In addition, the number of task sets declared schedulable by each test in the RTA and its running time are calculated. The results for the TTSA undertaken and the RTS test are found in Figure 7.2.

7.5.3 The evaluation platform

The study is conducted on a desktop PC 2.66GHz Intel Core Duo CPU running Windows XP. The code counter software that supports C/C++ and its header as well as assembly language is used to count the lines of code.

7.5.4 Measurement of the lines of code

A similar method to that mentioned in the previous experiment was used for measuring LoC for the implementation of the schedulability test.

7.6 Results for the cost of design

This section presents an analysis of the costs associated with verifying the timing constraints during the design phase.

7.6.1 Number of inputs required for the test

In Table 7.6 a comparison of the number of inputs required to perform the schedulability analysis, is tabulated. This delivers a comparison in the number of inputs required to perform a schedulability analysis. As can be seen, the RTA and heuristic search algorithms require four inputs; whilst the TTSA requires nine items of timing information in order to perform the schedulability test. The number of inputs was derived from an in depth studies of the RTA, TTSA, and Heuristic Search theories, and the TORSCHE software (Sucha et al., 2006) which implemented the algorithms in further depth. These are the only number of inputs required, since these are the parameters defined in the functions.

In order to meet these requirements, the theory of RTA (Audsley, 1993), TTSA (Gendy, 2008) and heuristic search (Stankovic, 1989) and the TORSCHE software (TORSCHE, 2001) which implemented the algorithms was studied in depth and the

number of inputs used was the number of inputs tabulated. As mentioned earlier, these are the only inputs required, since the parameters were defined in the functions.

Table 7.6 Number of inputs required for the schedulability tests analysis.

Response Time Analysis for ET-P	**Time Triggered Schedulability Analysis for TT-C and TT-P**	**Heuristic Search for ET-C**
• WCET • Release jitter • Period • Deadline	• WCET • Deadline • Period • Distance • Precedence • Exclusion • Latency • Jitter • Scheduling overhead	• WCET • Release time • Deadline • Precedence
4 No. of Inputs	10 No. of inputs	4 No. of Inputs
(Audsley et al., 1993).	(Gendy et al., 2008, Gendy and Pont, 2008a, Gendy and Pont, 2008b).	(Bletsas, 2007 and Buttazzo, 1997).

7.6.2 Lines of code (LOC) schedulability analysis algorithms

MATLAB implements RTA and Heuristic Search codes, which were ported to C, using the Matlab Coder (MatlabCoder, 2012). Although the ported code was not optimised, due to lack of time, and complexity of the code, just the ported code was considered in our comparisons. Therefore, the LOC is considered as a weak parameter in our comparative studies. However, TTSA was already written in C, by one of our research members, Gendy (2008).

Since TTSA was written in C language, therefore, for a fair comparison, RTA and heuristic search codes, were also ported to C. The lines of code computed are depicted in Figure 7.1. It is evident that the total LOC of the TTSA test is greater than the RTA

test by 61%. Conversely, the LOC of the heuristic search algorithm is higher than the RTA and TTSA.

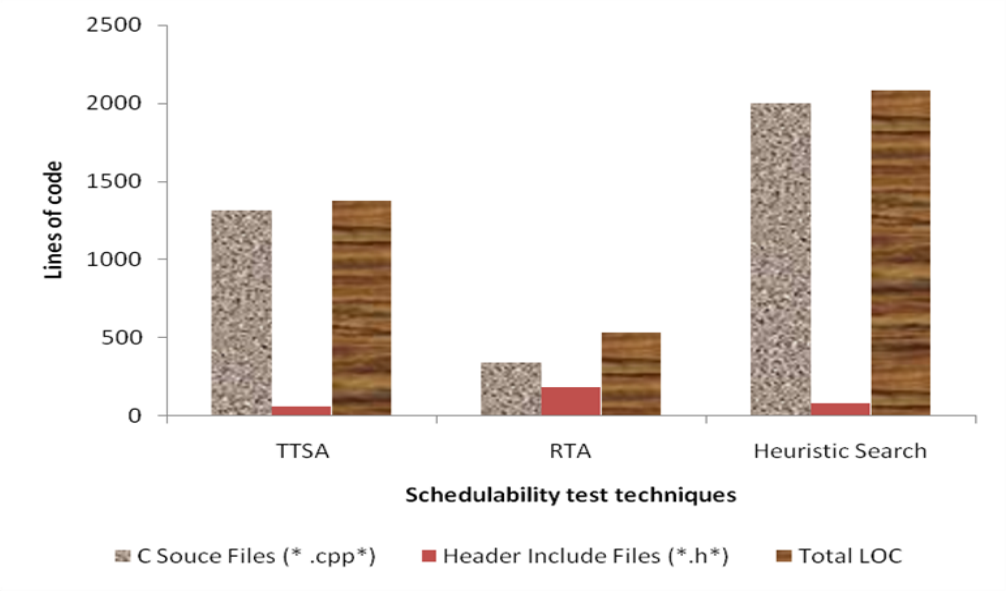


Figure 7.1 LOC of TTSA and RTA schedulability test.

An initial observation suggested that more LOC was required in the TTSA schedulability test when compared with the RTA schedulability test. However, in comparison with the TTSA algorithm, the heuristic search algorithm has more than 2000 LOC. This number is higher than TTSA's LOC. This is due to the fact that the search technique is rather complex for implementation, in order to consider timing constraints of each task such as release time and execution time.

7.6.3 Comparison of running time of the schedulability test algorithm

In order to explore the efficiency of the schedulability test, each schedulability test was tested from a small number of tasks to a large number of tasks.

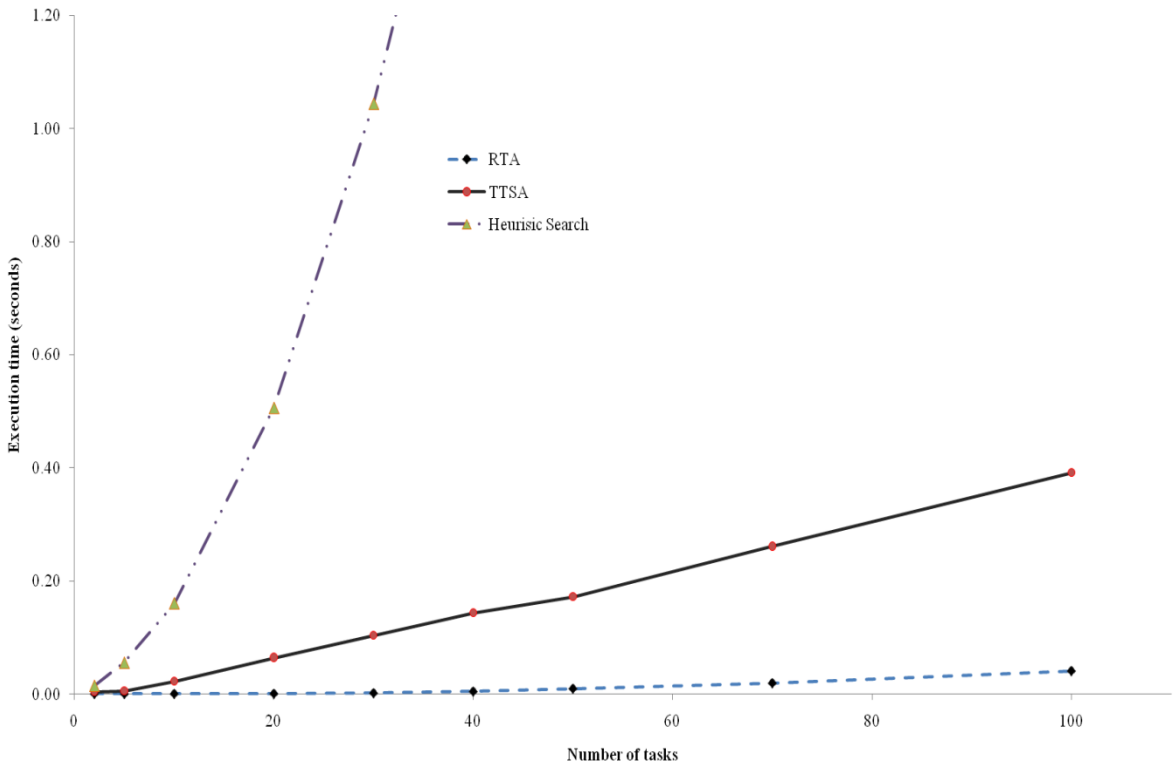


Figure 7.2 Comparison between the RTA, TTSA and heuristic search algorithm

As can be seen from Figure 7.2, there may be significant effects proceeding from the schedulability test execution time when the task constraints are included. The number of tasks in each test varied from 2 to 100. Clearly, as the number of tasks in an application increases, the execution time when calculating the worst-case response time and arranging the task ordering in finding the suitable offsets of the tasks rapidly increases with the schedulability test execution time.

It is worth noting that, although the RTA test had a faster execution time than the TTSA test; theoretically, the RTA alone has pseudo-polynomial complexity when the number of tasks increases (Bini, 2004). This is due to the increment in the number of steps or iterations in the innermost loop of the RTA test as a function of the number of tasks. For example, the RTA test used 5 iterations to calculate the worst-case response time of the 2 tasks and took 0.00175 microseconds for testing. When the number of tasks increases to 100, the number of iterations increases to 677825 with a total execution

time of 0.0404 microseconds. However, the effects of the number of tasks to the RTA's running time can be seen more evidently in the experimental results. In this experiment, it shows that when the number of tasks grows, the execution time for running RTA increases exponentially. Thus, the relationship for RTA and number of tasks is exponential rather than pseudo-polynomial as mentioned in the scheduling theory.

In contrast, the schedulability analysis of TTC and TTH required more time than the RTA test. For instance, in order to test the schedulability of 20 tasks, the RTA took 0.00268 microseconds to run, whereas the TTSA takes 0.18 seconds. The TTSA testing process applied a longer execution time to sort the tasks in accordance to their task constraints and scheduling techniques. The purpose of this process was to find suitable offsets of the tasks and to locate the appropriate tick interval in order to schedule the tasks. Unlike RTA, TTSA's execution time functions proportionally and increases by the number of tasks.

The results imply that that the pre-emptive schedulers' schedulability analysis generally outperforms the co-operative schedulers' schedulability analysis. However, the RTA test did not consider the task constraints, which affected the results of the scheduling theory and its implementation. Conversely, the TTSA test considered all the task constraints such as task jitter, precedence, overheads and latency, since the test was used with static scheduling. Thus, the scheduling theory for the TTSA test could have much closer analysis to its implementation realities. In a real scheduler implementation, processor attributes contribute to the amount of overhead and blocking projected. Thus, a small gap between scheduling theory and its implementation could be meaningful for validating the correctness of the timing properties of real-time applications.

As mentioned by Butazzo (Buttazzo, 2005a), the problem of finding a feasible schedule become NP-hard for a periodic task and non pre-emptive scheduling. Such scheduling algorithms adopted in the Spring kernel (Bletsas, 2007), as well as other researchers such as Burns (Burns et al., 1995), Jeffay (Jeffay et al., 1991b), Tindell (Tindell et al., 1992) and Short (Short, 2011). A heuristic search is usually used to make the algorithm computationally tractable. The search algorithm can be based on a heuristic search function that can behave as the First-Come-First-Serve, the Shortest-Job-First or the Earliest-Computation-First.

7.7 Discussion

The work presents the first evaluation results for four different software architectures. TTSA can be used to make a constructive schedulability test for TTC and TTH (TT with limited pre-emption) schedulers; whilst the RTA and heuristic search algorithm are used for ETP and ETC schedulers. The RTA schedulability analysis, commonly used for schedulability analysis for fixed priority scheduling approach, has a sufficiently short execution time in a small system domain. This can provide benefits to real-time applications which require online scheduling computation (Buttazzo, 2005a).

However, the RTA has pseudo-polynomial complexity and this affects the test performance when the number of tasks gets larger. Thus, RTA is unsuitable for large real-time applications which require online scheduling.

Figure 7.2 reveals that the TTSA takes a longer time to run the test than RTA. This is due to the fact that the TTSA finds a workable scheduler by using a heuristic but not an exhaustive search in addition to adding the offsets onto the tasks (Gendy, 2008). In fact, the test also includes all task constraints such as task jitter, precedence and overheads. The test requires more knowledge of timing constraints than the RTA and the heuristic

search algorithm. The standard RTA test and heuristic search test do not take into account such task constraints in its computations. By considering task constraints in the schedulability analysis, this could reduce the gap between scheduling theory and its implementation in real applications.

It is worth mentioning that the TTSA is a constructive schedulability analysis which is normally used in offline scheduling to ensure that all the timing constraints are met. Since the offline scheduling computes all the timing requirements before runtime, it makes the execution time unnecessary for TT architecture. In fact, more complex schedulability analysis can be applied to the test (Xu, 2006). Unlike TT architecture, ET architecture is normally associated with online scheduling. Although ET architecture schedulability analysis such as RTA has a fast execution time, this is not always true in all types of ET architecture. For ET architecture with co-operative scheduling, its schedulability analysis is based on a heuristic search (Butazzo, 2005). As can be seen from the results, the LOC of the heuristic search algorithm is higher than RTA and TTSA.

It is important to have a minimal execution time in the light of the schedulability test as well as suitability to use all the system's domain sizes for online scheduling or ET systems (Davis, 2008). It also has the least gap between scheduling theory and its implementation (Katcher, 1993). In order to minimise the gap between scheduling theory and its implementation, specifically for RTA, the analysis should be extended to account for operating system costs such as overheads and/or blocking costs. However, the computation will become more complicated and longer.

The LOC of TTSA represents the complexity of implementing the schedulability test for TT architecture. Conversely, the LOC of RTA represents the complexity for

implementing the test on a set of tasks running under ET architecture in order to determine whether it is schedulable or not. The results from Section 7.6.2 show that the TTSA program has higher LOC than the RTA program. This suggests that the use of ET architecture requires less effort for implementing as opposed to the TT architecture. However, with regards to the schedulability test design perspective, the TTSA schedulability test is facilitated by many automatic procedures contained in its programs. For example, it provides a suitable tick interval for the systems and appropriate offsets of the tasks to be scheduled. Conversely the RTA schedulability test is much simpler since it is only making use of a ceiling function to test the schedulability of a set of tasks.

7.8 Conclusion

This chapter presents the final evaluation of temporal verification of real-time scheduling behaviour based on the last hypothesis: *The cost of designing a system with a TT architecture is higher than that when designing an equivalent system with an ET architecture*. Scheduling theory is an important means to validate the timing correctness for real-time applications. Therefore, for this purpose, a few commonly used schedulability analyses performed in ET and TT scheduling algorithms were reviewed. Then, the effect of the execution time of the schedulability test for RTA, TTSA and heuristic search algorithms was discussed. This assessment provided a basic comparison approach for evaluating the cost performance for real-time software at the design phase. The cost of design of software architecture was evaluated by measuring the time taken to run the schedulability test and LOC of the schedulability test algorithm. The impact of the increments in the number of tasks for each metrics was demonstrated and recorded. The results prove that the hypothesis is not valid for all categories of ET and TT architecture. It was also shown that an ET with pre-emptive

scheduler incurs lower design costs in comparison to TT architecture. However, the current TT schedulability test shows that performance is better than that for the previous TT schedulability test, which is viewed as too fragile. In fact its complexity is less than the ETP schedulability test.

Chapter 8

Conclusions

8.1 Overview of the work conducted

The work described in this thesis began by exploring ways in which software architectures could be used to support the development of cost-effective and reliable embedded systems. Specifically, the initial aim of this study has been to determine whether the use of a time-triggered software architecture (Pont, 2001a) can provide any benefits to find a cost-effective solution to verify the system's timing behaviour at design, implementation and verification phases of the software development process.

An extensive comparison based on experimentation, for pre-emptive versus co-operative scheduling, was carried out, and the results have been promising, favouring co-operative schedulers. Although ET-based schedulers are not included in the experimental comparison, a similar outcome/result can be deduced as encountered throughout the TT-based study.

The second assessment, in terms of comparative studies, was undertaken to evaluate the testing cost implication on ET versus TT architecture for which two case studies have been carried out. Both were designed and implemented on a hardware, and their WCET's measured under isolated task test conditions, as well as the overall system's task test condition. The implemented case studies have verified that TT-based architecture is much quicker and easier to test, as compared to ET. The TTC within the TT architecture has proven to be the best performance in terms of testing.

In order to assess the impact of synchronisation methods on the testing, experimental hardware-based evaluation studies have been undertaken, based on Free-RTOS. This real-time operating system which supported the synchronisation algorithms is generally

not available in the customly developed embedded software architecture i.e. ET, or TT. Thus the easier the task can be isolated, the faster and more efficient the testing becomes. It has been found that TTH performs the best in comparison with the TTH, TTP and other custom RTOS. This is primarily due to the fact that TTH provides the lowest percentage difference in the timings of isolated tasks.

A method for evaluating embedded real-time software architecture was followed to assess three main phases: design, implementation and testing. The method was proposed to rapidly assess the overall cost involved: which can be used to specifically the ET and TT systems.

8.2 The efficacy of a software architecture evaluation approach

Three main effects of timing verification of embedded software were investigated: the schedulability test cost, the implementation cost and the testing cost. Based on these evaluations, it could help designers to make a wise decision in order to choose which architecture is best to use for developing a low-cost system.

8.2.1 Impact of software architecture on cost of design

Schedulability analysis is an important formal testing strategy applied to test whether a set of tasks is schedulable or not in a real-time scheduling algorithm. The most popular priority-based scheduling and dynamic scheduling schedulability test is based on CPU utilisation performance and response time analysis (RTA) (Liu and Layland, 1973; Burns and McDermid, 1994; Buttazzo, 2005a; and Bini et al., 2003). Conversely, for TT architecture, constructive schedulability analysis such as the heuristic search, branch and bound and time-triggered schedulability analysis (TTSA) are usually used (Xu and Parnas, 1990; Burns et al., 1995, Gendy and Pont, 2008a and Short, 2012).

The schedulability test is performed with timing properties information of tasks in mind, covering aspects including the task period, worst-case execution time and deadline. The costs of performing the schedulability test for TT pre-emptive and co-operative systems (represented by time-triggered schedulability analysis – TTSA (Gendy and Pont, 2008a)) and ET pre-emptive and co-operative systems (represented by response time analysis – RTA (Tindell et al., 1994 and Davis et al., 2008) and the heuristic search (Bletsas, 2007 and Buttazzo, 2005a)) are examined to identify which of these architectures provides less cost in performing the schedulability analysis of a set of tasks (see Chapter 7). The results have suggested that although the cost of schedulability analysis alone is not a perfect metric for comparing cost evaluation of the ET and TT architecture, it can still be used to estimate the cost of design for reliable embedded systems. It has also been noted that the cost is less in TT designs, as opposed to ET designs adopting co-operative scheduling strategy.

8.2.2 Impact of software architecture on cost of implementation

There is still a big gap between the schedulability analysis and the implementation results of real-time timing properties. Nevertheless, it is also impossible to take into account all overhead factors in the formal analysis. Audsley (Audsley et al., 1995), Burns (Burns and McDermid, 1994) and Katcher (Katcher et al., 1993) which improved the schedulability analysis for the TT and ET architecture by considering the context switch, blocking and scheduling overheads. The ET architecture produces more context switch overheads than the TT architecture while the TT architecture produces more blocking overheads than ET architecture (Katcher et al., 1993). The results presented in Chapter 6 have clearly shown that the effects of cost implementation of the TTP scheduler such as memory, LOC and processor utilisation are higher than the TTH and

TTC schedulers. In fact the overhead costs of the TTP architecture increase exponentially when the number of tasks increases. The results also suggest that the overall implementation cost can be reduced if the co-operative scheduling is used effectively.

8.2.3 Impact of software architecture on the cost of testing

As mentioned by Myers (1979) and Tsai et al (1989), the fault localisation represents perhaps 95% of the problem. Hence, the testing strategy focuses on the process of fault localisation in order to find the source(s) of faults at task level. The testing strategy in this study has observed the timing behaviour of the tasks when the task in isolation method is performed.

The hypothesis of this evaluation is that the testing will be easier for systems in which the timing data obtained for isolated and in-situ task is very similar. Two case studies have been presented (in Chapter 7) to evaluate the easiness of testing in the TT architecture and the ET architecture.

A system has been developed in different software architecture including the ET, off-the-shelf RTOS – FreeRTOS and TT architectures. The idea being to observe which architecture is more difficult to produce the WCET of task when the task needs to be tested in isolation. A test harness has been employed to help the isolated task to reproduce its WCET behaviour as in the complete system. In addition, inputs of the task, in this case, the events – should be identified so that expected outputs can be monitored. However, this is not the case for the ET architecture since it involves too many possible inputs or testing points, which have to be observed – the events can occur at any point in the system. Thus, it is difficult to reproduce a similar timing behaviour for isolated tasks under the ET architecture. Unlike the ET architecture,

observation points are easy to identify in TT designs – only during the tick interrupt (Schu\tz, 1993). In fact, the WCET of the isolated task produced during testing using the TT architecture was found to be 99.9% similar with the WCET of the task run in the complete system.

8.2.4 Effects of shared resources synchronisation mechanisms

Finally, in this study, the impact of synchronisation and inter-task communication on testing cost has been examined. As many synchronisation mechanisms include the application of the lock-based and lock-free techniques in the ET architecture and pre-emptive scheduling, it was important to evaluate which of these mechanisms could provide the minimum effort in testing tasks in isolation.

Many standard RTOS support several synchronisation mechanisms such as semaphore, mutex, disable interrupt and disable scheduling. Thus, the analysis of the effects used the RTOS platform to compare all those techniques. It has shown that the usage of semaphore and mutex could increase the difference between the WCET of task in isolation and the WCET of task in the complete system. The effectiveness of the TTP synchronisation method has also been covered in this study as well as the message queue techniques. The TTP synchronisation method has also provided a large overhead that could make task timing behaviour hard to reproduce.

However, using the TTH scheduler with double-buffer techniques – in the absence of the synchronisation locking methods, the timing behaviour obtained for isolated and in-situ task has been very similar.

Although in most cases, tasks can be isolated, however, based on the analysis, tasks which have used message queue as inter-task communication methods could not be easily isolated. In order to perform the message queue techniques, it requires the higher

priority task sent and another task, usually of lower priority, to receive and process it. It is impossible to isolate the lower priority task of the message queue since the queue must firstly be filled in with the higher priority task's data. Thus, it implies that the message queue techniques are not an appropriate method of communication for testing tasks in isolation.

8.3 Evaluation of the software architecture analysis methods

The results of software architecture evaluation method on the cost of design, implementation and testing have shown an obvious comparison between the ET and TT systems as well as the pre-emptive scheduling and co-operative scheduling. We have summarised the differences between the reviewed methods for evaluating the software architecture. All approaches have focused on software, and are intended to be used by embedded software supplier organisations.

Each of the methods under evaluation has been distinguishable concerning the specific goal of the method. All the methods had the same overall goal, i.e. to compare the cost of software architecture.

In principle, Bate’s method, has been improved, by incorporating costs of implementation, and testing and verification. Thus even for evaluating the design costs, the two additional costs are to be assessed. This adds rigorous evaluation, but based on this thesis, these two costs are deemed essential for the evaluation model.

Kazman’s strategy is related to applying the ATAM approach while the architecture is still on paper. After attribute taxonomies of software architecture are developed, some screening questions are constructed in order to facilitate the comprehensive elicitation of relevant attribute-specific information. However, when using an evaluative approach,

it is very difficult to ensure complete coverage of trade-off comparisons that can be implemented, particularly for large systems.

Conversely, Bate's (2008) evaluation method provides a more quantitative comparison for evaluating software architecture. All objectives within the system which are derived from arguments and goal-oriented mechanisms are converted into weighted values. The results from each stage are combined with the weighted values in order to obtain the overall cost of the design modifications. Although Bate claimed that this approach could support maintainability of the system in a systematic way, it is possible to see that his evaluation method can confidently help designers to make design choices. Table 8.1 provides a comparison of evaluation methods for real-time software architecture.

Table 8.1 Comparison with current Evaluation approaches for Software Development

	Context						User	Content						Validation
Method/Author	**Specific goal**	**Method**	**Quality attributes (maintainability, reusability, modifiability, adaptability, development or operational cost, performance, reliabilitytestability and portability)**	**Stage of the development lifecycle**	**Application domain**	**Input/Output**	**Users' involvement**	**Cost of design**	**Cost of implementation**	**Cost of testing**	**Evaluation technique**	**Tool support**	**Method's activities**	**Validation**
SAAM, Kazman, 1993	Identify the potential SA risks	Scenario-based method	Single quality attribute: modifiablity analysis	After specification phase and before implementation phase	Combat and avionics	Requirements specifications, business drivers and software architecture descriptions.	Architects, designers, and end users	√	x	x	Purely scenario-based	SAAMTOOL	Use scenario profiles to categorise the generated scenarios	Validated in several domains
ALMA	Predicting modifiability cost based on risk assesment, maintenance cost prediction and softwre architecture comparison	Scenario-based method	Single quality attribute: modifiablity analysis	After specification phase and before implementation phase	Embedded systems, telecommunications, and information systems	Requirements specifications, business drivers and software architecture descriptions.	Designers only	√	x	x	Uses a variety of approaches depending on evaluation goals	Not available	Use scenario profiles to categorise the generated scenarios	Validated in several domains
PASA	To identify and mitigate performance related risks	Scenario-based method	Single quality attribute: performance analysis	After specification phase and before implementation phase	Embedded systems, telecommunications, and information systems	Requirements specifications, business drivers and software architecture descriptions.	Developers and Maintainers	√	x	x	Combines scenarios with performance modelling	Not available	Use cases and scenarios to identify performance goals	Validated in several domains
ATAM (Kazman,1994, 1999)	Identifies and analyses sensitivity and trade-off points as these can prevent the achievement of a desired quality attribute	Scenario-based and attribute model-based analysis technique	Multiple quality attributes	Requirement phase	Combat and avionics systems	Requirements specifications, business drivers and software architecture descriptions, trade-off points	Architects, designers, and end users	√	x	x	Integrates existent questioning and measuring techniques: attribute model-based analysis	Information provided by experts/reusing information derived during previous applications of the technique	provides a six-element framework to characterise quality attributes, and uses a utility tree for generating and classifying scenarios	Validated in several domains
Bate, 2008	To understand and evalutaing design trade-offs	Goal-oriented approach	Maintainability, reusability, modifiability of the system's design	Design trade off	Safety critical application	Requirements specifications, Top-level objectives	Designers and Maintaners	√	X	X	Use safety argumentation to build design arguments	Systematic means: simulated annealing and genetic algorithms	Convert design arguments to a quntifiable measure and a appropriate weighting applied	Based on case study
GBRAM (Anton, 1996)	To identify, elaborate, refine and organise goals for requirement specifications	Goal-oriented approach	Single quality attribute: modifiablity analysis	Requirement phase	Air Force Base (AFB)	Requirements specifications, business drivers and software architecture descriptions.	Analyst, Stakeholders	x	x	x	Combine scenario and goal analysis	Not available	Objectives are mined from existing sources (documentations and usecase), Goal analysis	Not yet validated on real projects
Proposed software architecture evaluation	To identify and compare cost of testing and verfication at design, imlementation and testing phase	Goal-oriented approach	Correctness, Testability, Verifiability	Design, Implementation, Testing	Embedded real-time systems	Requirements specifications, objectives, software architecture description, experimental-based results	Designers and developers	√	√	√	Use goal arguments, case study and measure cost of design, implementation and testing phase	Experimental means	Convert design arguments to experimental assesment	Validated at design (schedulability test), implementation (scheduler implementation) and testing phases (traffic light and FFT systems)

When using NIMSAD as a tool for comparative analysis, the following observations were made: it is possible to show results such as quality attributes, methods or techniques used and input/outputs when implementing software architecture evaluation, as categorised within the software architecture evaluation context. The 'User' section, only includes the category of user's involvement, since this represents the potential for those who can obtain benefits from the software architecture evaluation method. 'Evaluation techniques' and 'Method activities' partly overlap in content. For the analysis of 'Validation', there are combinations presenting maturity of method and method validation; in addition there are characterisations associated with the validation of methods as applicable to a real application.

Kazman (1994) and Bate (2008) provided a software architecture analysis based on trade-off analysis to identify a suitable design; however, in this study, a different angle of analysis was introduced which considers more comparative and quantitative evidence prior to the appropriate software architecture being chosen. The results of this can be clearly seen on plotted graphs for the ET and TT architecture, based upon scenario and measurement-based techniques. In addition, most other approaches do not consider implementation costs or testing costs. In this study, costs from three main development lifecycles have been considered based on the proposed hypotheses and experimental works. Thus, it is apparent that the results produced, such as lines of code, schedulability analysis running time, ease of isolating an individual task, have shown different characteristics for each software architecture and scheduling algorithm. However, more qualitative results need to be presented to compare the costs involved in designing a system using TT or ET architecture.

Systems under testing, can benefit from the proposed methods. The evaluation method can provide results that increase the confidence level of an evaluation. Indeed, an

analysis for a large system can be developed using an evaluation strategy so that the impact of software architecture can be observed as the system grows.

However, to compare the impact from different software architectures, one has to carefully identify matching factors for the purpose of evaluation. This can be accomplished by determining common arguments in software architectures for each evaluation criteria. Since most of the evaluation techniques used for this study are experimental-based, and there is a range of design choices this can be considered in embedded systems development, as more experimental evidence may prove to be useful.

8.4 Bridging the gap between TT and ET architecture in the testing of real-time systems

As mentioned in the introduction chapter, it is necessary to reduce time and effort when testing of real-time systems, particularly for safety critical applications. For example, if a nuclear power plant operation is breaking down, the source of the error must be identified immediately and efficiently, otherwise there is a risk of catastrophic damage to the environment. Hence, using these empirical studies and evaluations of the impact on software architecture across several software aspects, the gap in design, implementation and testing costs or efforts for TT and ET architecture can be filled.

It has been shown that TT architecture provides many benefits that make testing easier than the ET architecture particularly for large and complex systems. Systems with less overheads and lower cost and complexity can improve testing effort to enhance the safety of critical systems. Thus, it will take more time and effort to ensure that all the tasks are schedulable as well as to avoid occurrence of deadlock or synchronisation error during execution (Xu, 2007).

8.5 Limitations and future work

A software metric has been employed to evaluate the costs involved in TT and the ET systems. These assessments can be improved by providing more accurate schedulability analysis; such as the Hyperplanes Exact Test (HET) (Bini and Buttazzo, 2004) or the exact schedulability analysis for EDF scheduling, such as Quick Processor-demand (Zhang and Burns, 2009). Furthermore, the evaluation and analysis can be extended to study the effects of clock-synchronisation, parallel executions and varying communication latencies (Thane and Hansson, 2001). These can prove to be beneficial factors for assisting assessments.

In order to obtain more accurate analysis for overheads, more variations of the WCET of tasks could be used and the number of tasks in the analysis could be increased to over 100 tasks. Although there are many research studies that have been conducted to address this issue, there remain a lack of empirical studies covering comparisons between ET and TT architecture in light of schedulability analysis' running time, implementation costs, lines of code and tasks in isolation for distributed systems.

The impact of micro architecture influence is evaluated by incorporating cache and pipeline, in the system testing.

8.6 Conclusion

In conclusion, evaluations testing real-time systems in different software architectures are vital and can provide numerous benefits for researchers and designers of embedded real-time applications (such as a nuclear reactors, military and space shuttle systems). However, verifying that a system is operating correctly, particularly as regards timing properties, can present major challenges. Thus, ways to analyse the impact of ET and TT architecture in reaction to pre-emptive and co-operative scheduling on the cost of

design, implementation and testing can assist designers to choose ideal software architectures.

The project described in this thesis has made three major contributions to the field of testing of the embedded real-time systems using event-triggered architectures and time-triggered architectures. These are summarised below:

Firstly, a comparative analysis of pre-emptive and co-operative schedulers was done, assessing the effects of overheads, memory and CPU utilisation, lines of code and the number of pre-emptions.

Secondly, a novel analysis was carried out to assess the ease of reproducing similar timing data for an isolated task, in comparison to a complete system.

Thirdly, scalability analysis was carried out for a schedulability test, assessing the effects on running time test, and other related costs, for ET and TT architecture, in response to the increase in the number of tasks.

Finally, an integrated software architecture evaluation approach was introduced to analyse the impact of a TT and ET architecture on the cost of design and implementation, and the testing of real-time embedded systems.

The research has successfully bridged the gap between TT and ET architecture within the embedded software testing perspectives in light of verification at the design phase. However, there are still many software effects that have not been included, although this would provide more convincing evaluations, particularly for modern microcontroller architecture. In fact, finding the source of errors is the most complicated consideration for software designers, particularly those working on large and complex real-time systems. Hence, it is important to choose a cost-effective

software architecture that can be guaranteed to be efficient across the design, implementation and testing phases.

Appendix -A

Design tools/Language	***Description***
Code Counter (Code, 2006).	*Code Counter Pro software is a source code counter program for Windows that can count several types of source code - including Java, JSP, C or C++, VB, PHP, HTML, Delphi or Pascal, ASM, XML, and COBOL* (Code, 2006). The software will produce a report that can be exported into excel or HTML files. http://www.geronesoft.com/
Labview for Instrument Control (National, 2010)	*LabVIEW is a development environment for problem solving and accelerated productivity. NI LabVIEW for Instrument Control software is one of Labview's products, which helps to acquire data and provides extensive libraries for signal processing and data visualisation (National, 2010). The power of LabVIEW software and IDNet instrument-specific drivers can help to automate third-party instruments to create reusable measurement solutions.* http://www.ni.com/labview/applications/instrument-control/
RapidiTty Builder (RapidiTTy, 2010)	*RapidiTTy Builder provides IDE to implement embedded C programming. The toolsets includes compilers, substantial code libraries, one or more real-time operating systems, and full support for detailed timing analysis. It has C and / or Ada compilers, detailed C and Ada code examples, with code libraries, a user-friendly editor with sophisticated function hyper-linking and code completion, a debugging framework with all the usual features - instruction stepping mode, setting breakpoints, viewing internal registers and memory contents (RapidiTTy, 2010).* http://www.ttesystems.com/products/rapiditty_builder
TORSCHE (TORSCHE, 2007,	*TORSCHE (Time Optimisation, Resources, SCHEduling) Scheduling Toolbox for Matlab is*

Sucha et al., 2006)	*a freely (GNU GPL) available toolbox developed at the Czech Technical University in Prague, Faculty of Electrical Engineering, Department of Control Engineering (Sucha et al., 2006).* *The current version of the toolbox has a variety of areas of scheduling: scheduling on dedicated processors/parallel processors, cyclic scheduling and real-time scheduling. Furthermore, particular attention is dedicated to graphs and graph algorithms, due to their large interconnection with scheduling theory. The toolbox offers various scheduling/graph algorithms, a useful graphical editor of graphs, an interface for Integer Linear Programming and an interface to TrueTime. (MATLAB/Simulink based simulator of the temporal behaviour).*
Olimex board (Olimex, 2009) https://www.olimex.com/Products/ARM/NXP/LPC2378-STK/	*LPC-2378STK is a starter kit, which uses a Microcontroller LPC2378 from NXP. This microcontroller supports various serial interfaces such as USB 2.0, UART, CAN, audio input and output, JTAG, Ethernet, TFT display and SD/MMC cardholder on this board. All this makes it possible to build a diversity of powerful applications to be used in a wide range of situations.*
MATLAB Coder (MatlabCoder, 2012)	*MATLAB Coder™ generates standalone C and C++ code from MATLAB® code. The source code generated is portable and readable. MATLAB Coder supports a subset of core MATLAB language features, including program control constructs, functions, and matrix operations. It can generate MEX functions that accelerate computationally intensive portions of MATLAB code and verify the behaviour of the generated code. Features for the MATLAB Coder include: ANSI/ISO compliant C and C++ code generation, MEX function generation for fixed-point and floating-point math, Project management tools for specifying entry points, input data properties, and other code-generation configuration options, Static or dynamic memory allocation for variable-size data. Code generation support for many functions and*

	System objects™ in Communications System Toolbox™, DSP, System Toolbox™, and Computer Vision System Toolbox™, Support for common MATLAB language features, including matrix operations, subscripting, program control statements (if, switch, for, while), and structures. *(MatlabCoder, 2012)*
Embedded C Programming (Pont, 2006)	*C is a powerful system programming language, and C++ is an excellent general purpose programming language, with modern bells and whistles. C's strengths in embedded system development greatly outweigh its weaknesses. It may not be an ideal language for developing embedded systems, but it is unlikely that a 'perfect' language will ever be created (Pont, 2006).*
Microsoft Visual C++ (Microsoft, 2010)	*Microsoft Visual C++ is Microsoft's implementation of the C and C++ compiler and associated languages-services and specific tools for integration with the Visual Studio IDE. It can compile either in C mode or C++ mode. For C, it follows the ISO C standard with parts of C99 spec along with MS-specific additions in the form of libraries. For C++, it follows the ANSI C++ spec along with a few C++11 features. Microsoft positions Visual C++ for development in native code, or in code that contains both native as well as managed components. Visual C++ supports COM, as well as the MFC library. Visual C++ can also use the Visual Studio forms designer to design UI graphically, and can be used with the Windows API. It also supports the use of intrinsic functions, i.e. functions recognised by the compiler itself and not implemented as a library (Microsoft, 2010).*
FreeRTOS (Barry, 2011)	*FreeRTOS TM is a market leading RTOS from Real Time Engineers Ltd. that supports 33 architectures. It is professionally developed, strictly quality controlled, robust, supported, and free to embed in commercial products. FreeRTOS has become the de facto standard RTOS for microcontrollers; this has been*

	achieved by removing common objections to using free software, and in so doing, providing a truly compelling free software model (Barry, 2011). http://www.freertos.org/RTOS.html

Appendix -B

B.I A Tabularised Summary of Literature Review

	1971	1973	1974	1980	1982	1982	1983	1984	1985	1986	1986	1987	1988	1989	1989	1989	1989	1990	1990	1990	1990
Characteristics of systems	Brat	LnL	Der	Leu	Mel	Leu	Mok	Carl	Zhao	Cheng	JnP	Leh	Sta	Kop	Leh	Bak	Chet	Tok	Sha	Loc	Leh
Software architecture																					
ET-task activation		X	X	X		X					X	X			X		X		X		
TT-task activation																					
Scheduling																					
Preemptive		X	X	X		X	X					X			X		X		X	X	X
Non pre-emptive																					
Dynamic priority (EDF)		X	X	X			X		X	X			X				X				
Fixed Priority (RM)		X				X					X	X			X					X	X
Preruntime														X							
Table driven																X					
Cyclic Executive					X			X								X					
Tick scheduling																					
Time-Driven																					
Feasibility test																					
Heuristic test	X									X											
Utilization based test		X																	X	X	X
Response time analysis											X										
Pseudo Polynomial test				X																	
Polynomial complexity																					

	1990	1990	1990	1991	1991	1991	1991	1992	1992	1992	1992	1993	1993	1993	1993	1995	1994	1994	1994	1995	1995
Characteristics of systems	Xu	Chn	Bar	Jeff	Bak	Tsai	Aud	Loc	Xu	Jeff	Scw	Bar	Ara	Xu	Aud	Katc	Yu	Tind	Tind	GM	Tind
Software architecture																					
ET-task activation		X	X		X						X	X			X	X					
TT-task activation													X	X		X					
Scheduling																					
Preemptive			X	X	X	X	X				X	X			X	X			X		
Non pre-emptive				X				X					X			X		X		X	X
Dynamic priority (EDF)		X	X	X	X	X				X	X	X								X	
Fixed Priority (RM)					X	X	X	X							X	X		X	X		X
Preruntime	X							X	X					X							
Table driven																					
Cyclic Executive								X													
Tick scheduling															X						
Time-Driven																X					
Feasibility test																					
Heuristic test	X								X												
Utilization based test				X														X			X
Response time analysis															X	X		X	X		X
Pseudo Polynomial test			X																		
Polynomial complexity																					

	1995	1995	1995	1995	1995	1995	1996	1996	1996	1997	1998	1999	2000	2000	2000	2000	2000	2001	2001	2002	2003
Characteristics of systems	Katt	Tind	Sta	Spu	Burn	Burn	Burn	Mok	Mok	Han	Sjo	Che	Dev	Mat	Xu	Lu	Liu	Ekel	Bini	Shei	Bril
								Rooi													
Software architecture																					
ET-task activation	X	X		X		X	X		X	X	X		X	X	X	X	X		X		X
TT-task activation						X									X						
Scheduling																					
Preemptive	X			X		X	X		X	X	X		X	X		X	X		X	X	
Non pre-emptive																				X	
Dynamic priority (EDF)	X			X									X							X	
Fixed Priority (RM)	X	X				X	X		X		X			X	X	X	X		X		
Preruntime															X						
Table driven																					
Cyclic Executive					X	X		X													
Tick scheduling						X															
Time-Driven																					
Feasibility test																					
Heuristic test					X								X					X			
Utilization based test									X			X	X			X				X	
Response time analysis	X	X		X		X	X				X			X							X
Pseudo Polynomial test										X			X								
Polynomial complexity																	X	X			

	2003	2003	2003	2003	2003	2004	2004	2005	2005	2005	2006	2006	2006	2006	2006	2007	2007	2007	2008	2008	2008
Characteristics of systems	Bini	Coo	Bate	Cof	Xu	Dob	Bini	Bini	Butt	Bar	Push	Bar	Xu	Nil	Lu	Bril	Butt	Lu	Lind	Ram	Dav
Software architecture																					
ET-task activation	X	X	X				X	X	X					X	X			X	X		X
TT-task activation					X						X								X		
Scheduling																					
Preemptive	X	X	X	X					X			X	X	X	X			X			
Non pre-emptive										X		X					X			X	
Dynamic priority (EDF)		X							X	X											
Fixed Priority (RM)	X	X	X	X		X	X	X	X						X			X			X
Preruntime					X						X		X								
Table driven																					
Cyclic Executive			X								X										
Tick scheduling																					
Time-Driven																					
Feasibility test																					
Heuristic test					X								X								
Utilization based test	X	X	X				X														
Response time analysis			X												X	X		X		X	X
Pseudo Polynomial test																					
Polynomial complexity	X						X					X									

	2008	2008	2009	2009	2009	2010	2010	2011	2011	2011	2012
Characteristics of systems	Aym	Shor	Pont	Zha	Yao	Pont	Bert	Min	Yao	Shor	Shor
Software architecture											
ET-task activation				X			X	X	X	X	X
TT-task activation	X	X	X			X				X	X
Scheduling											
Preemptive		X			X		X	X	X	X	X
Non pre-emptive	X	X	X		X	X				X	X
Dynamic priority (EDF)							X				
Fixed Priority (RM)				X	X						
Preruntime											
Table driven											
Cyclic Executive											
Tick scheduling											
Time-Driven											
non-EDF										X	X
Feasibility test											
Heuristic test	X										X
Utilization based test											
Response time analysis								X			
Pseudo Polynomial test					X						
Polynomial complexity											

Appendix -C

C.I Pseudo-code: Response time analysis (Davis, 2008):

```
rpev = 0;
r = initial_value();
while((r > rprev) && (r <= task[i].D)
      {
            rpev = r;
            r = task[i].C;
            for(j = 0; j < 1; j++)
            {
                  r+= ceiling(rpev, task[j].T)*tasks[j].C;
            }
      }
```

C.II Pseudo-code: TTSA schedulability analysis algorithm (Gendy, 2008):

```
START
Arrange tasks in order according to their deadlines (EDF)
//Common divisors of task periods in descending order
gcd(t) = (GCD,....,GCD), t = 1, 2, ....., m;
Sched_Strategy = (TTC, TTH);
// First check schedulability using TTC strategy
Sched_Strategy_Index = 1;
DO
      {
      Tick_Interval = GCD(Tick_Index);
      i = 1; Offset(t) = 0;
      Sched(i) = TRUE, Sched_Tasks = 1;
      Do
            {
            i++, offset[i] = 0;
            Do
            {
            Length_of_Major_Cycle = LCM(Period(k), k = 1,2,....,i
Max_Offset = Max(Offset(k)), k = 1,2,...,I;
Test_Period = 2* Length_of_Major_Cycle + Max_Offset;
Sched(i)    =    Check_Sched(i,    Test_Period,    Tick_Interval,
Sched_Strategy_Index)
IF (Sched(i) = TRUE)
      (Sched_Tasks ++;)
ELSE
      (Offset[i]++;)
}WHILE ((Offset[i]<Period[i]) and (Sched[i] = FALSE));
}WHILE (i<n);
IF (Sched_Tasks = n)
{print task offsets, tick interval, scheduler type; EXIT;}
ELSE
{Tick_index++;}
}WHILE(Tick_index <= m)
Sched_Strategy++; //Try the TTH strategy
}WHILE(Sched_Strategy <= 2);
Print list of the scheduled and unscheduled tasks;
END
```

C.III Heuristic search schedulability analysis algorithm (Stankovic, 1989):

```
Procedure Scheduler (task_set; task_set type; var schedule;
schedule;type; var schedulable;boolean);
VAR EAT*, EAT*, vector type;
BEGIN
      schedule := empty;
      schedulable := TRUE;
      EAT* := 0;
      WHILE (NOT empty (task_set)) AND (schedulable) DO
BEGIN
      calculate Test for each task T € task_set;
IF NOT strongly feasible (task_set, schedule)THEN
      schedulable := FALSE;
      ELSE
      BEGIN
      apply function H to each task in task_set;
      let T be the task with the minimum value of function H;
      Test = Test;
      task_set = task_set - T;
      schedule = append(schedule, T);//Append T to schedule
      calculate new values of EAT
      END
END
END
```

Appendix -D

The timing behaviours of tasks that are used to visualise task timing behaviour when assisting the production of the implementation cost analysis, are shown here.

D.I Experimental results of impact of number of tasks

Timing analysis diagrams that are used to observe task timing behaviour and implementation costs for 5, 20, 50 and 100 tasks are shown below. Three main software architectures were used in this experiment: TTC, TTH and TTP.

D.II *TTC*

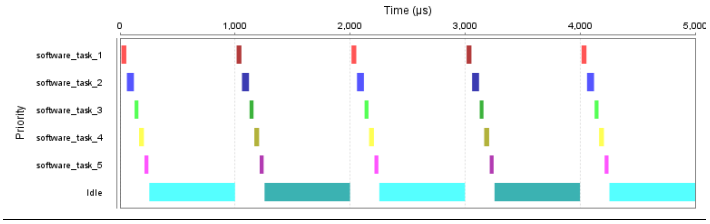


Figure D.1 Impact of no. of tasks for 5 tasks on TTC.

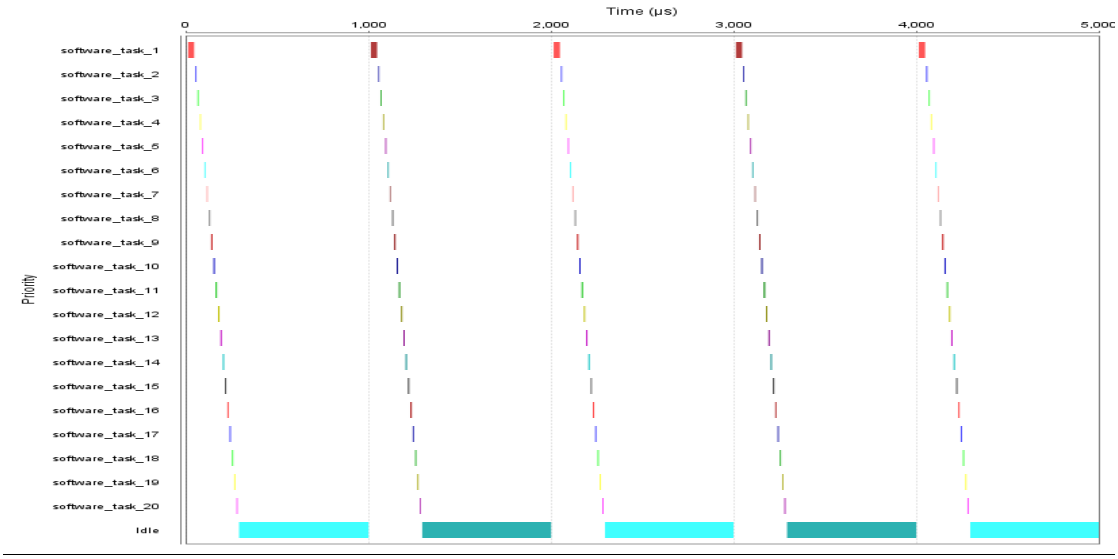


Figure D.2 Impact of no. of tasks for 20 tasks on TTC

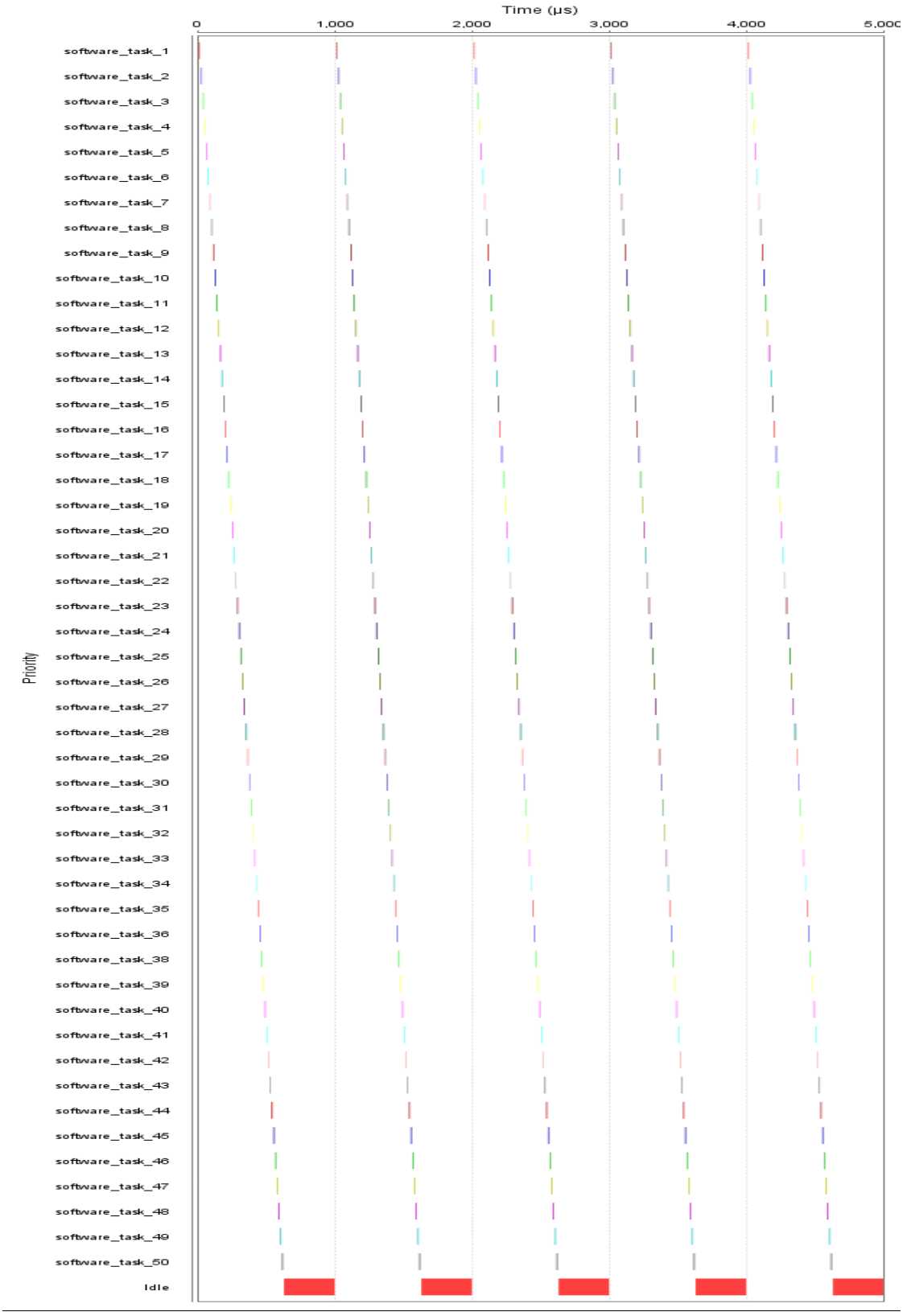


Figure D.3 Impact of no. of tasks for 50 tasks on TTC

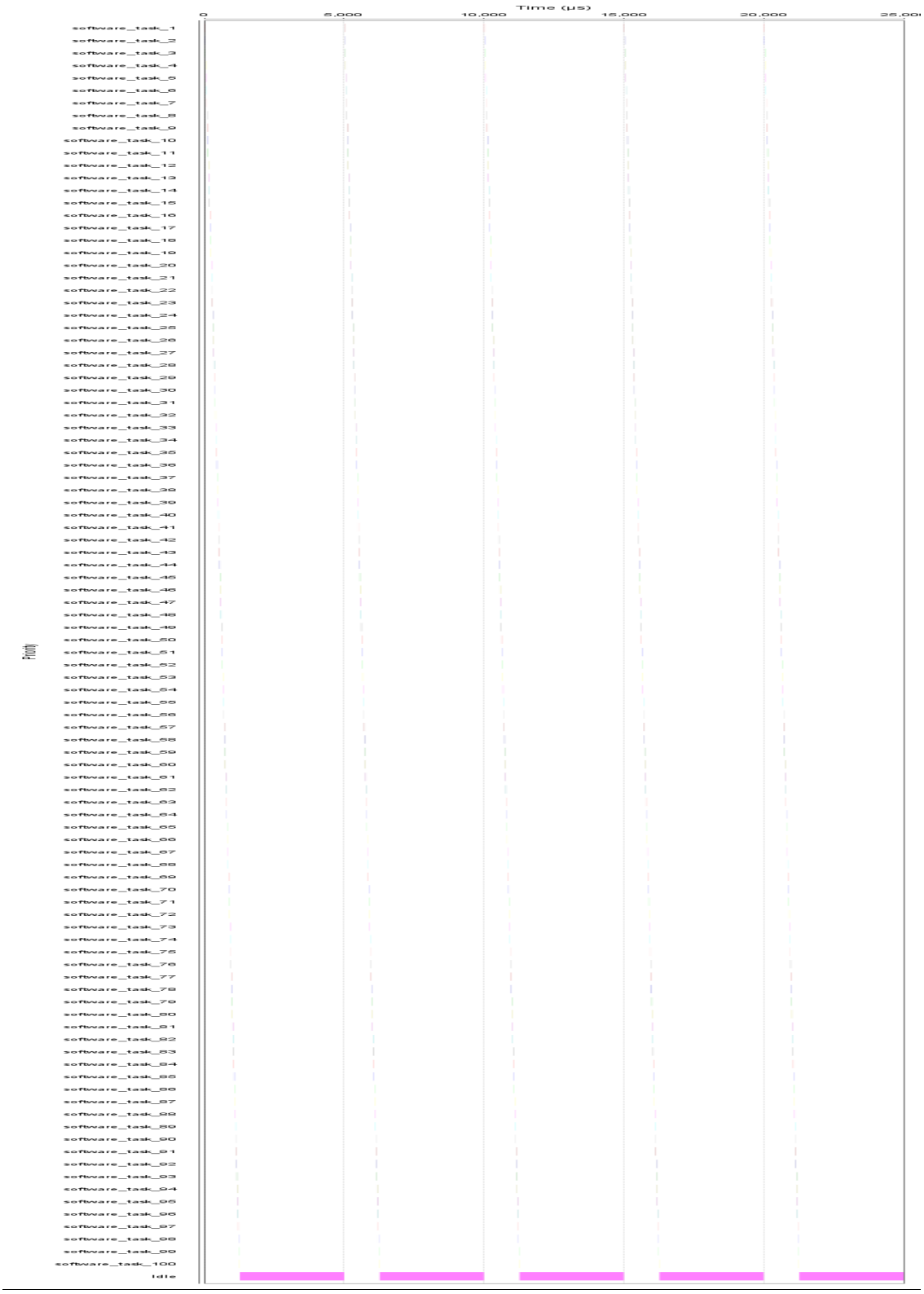


Figure D.4 Impact of no. of tasks for 100 tasks on TTC

D.III *TTH*

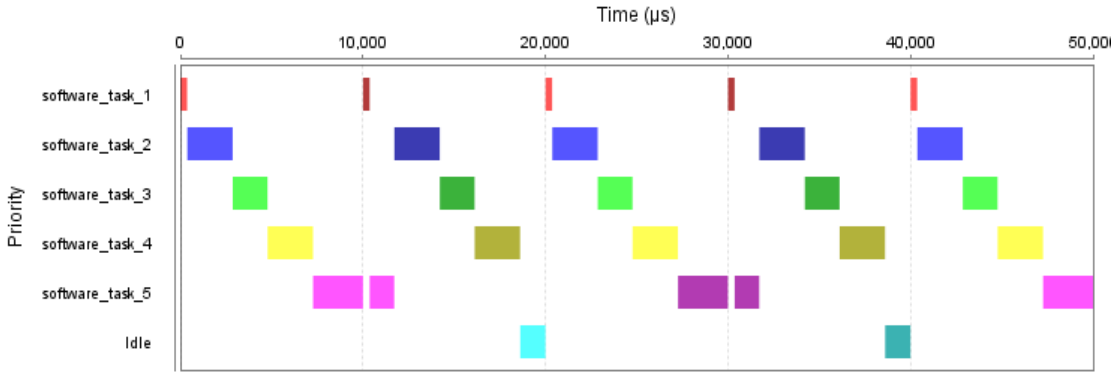


Figure D.5 Impact of no. of tasks on tth for 5 tasks.

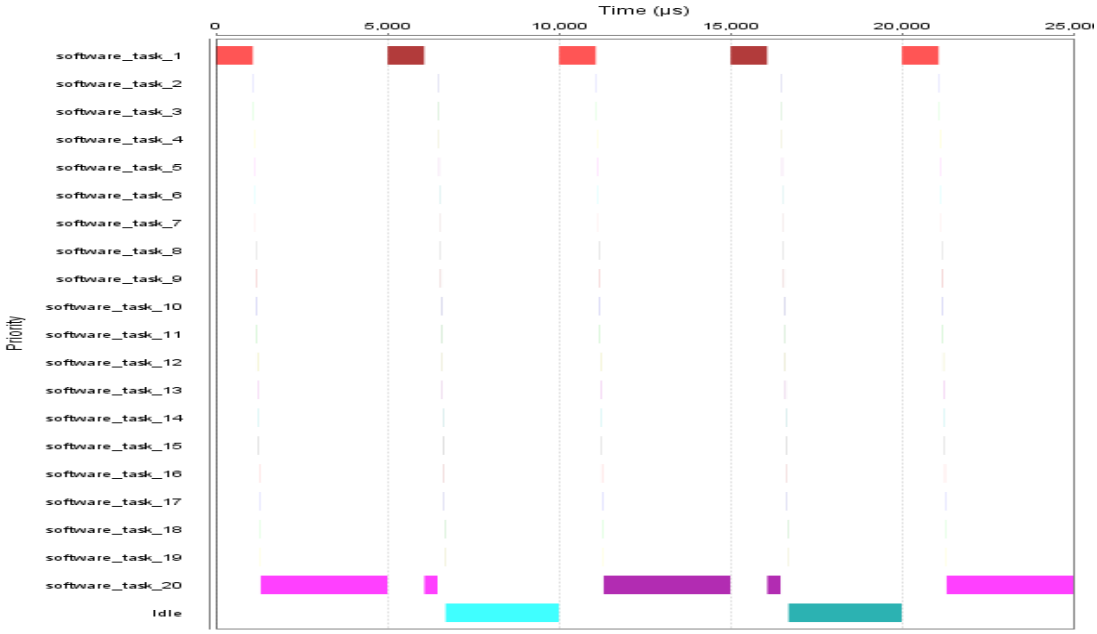


Figure D.6 Impact of no. of tasks for 20 tasks.

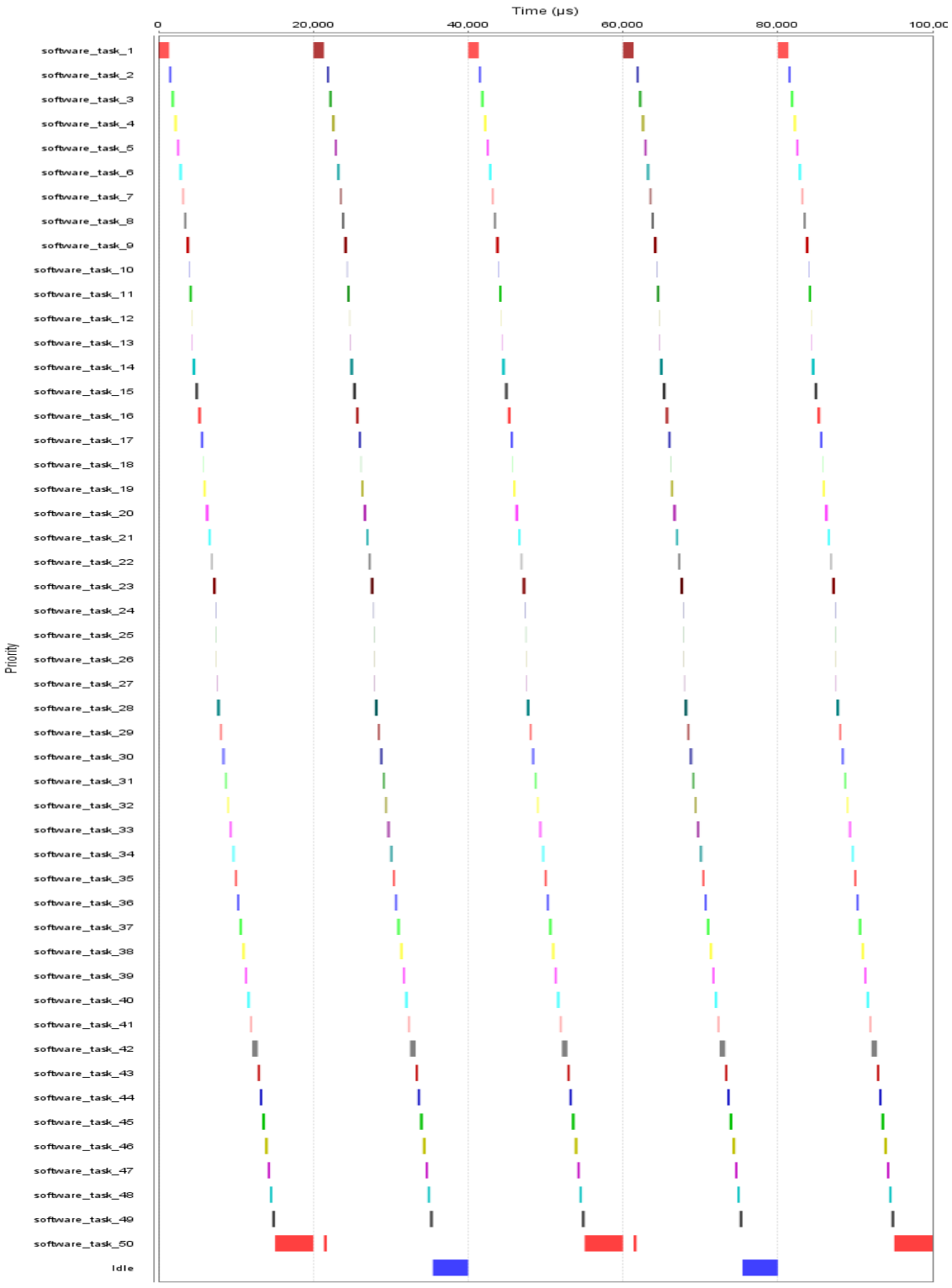


Figure D.7 Impact of no. of tasks on tth for 50 tasks.

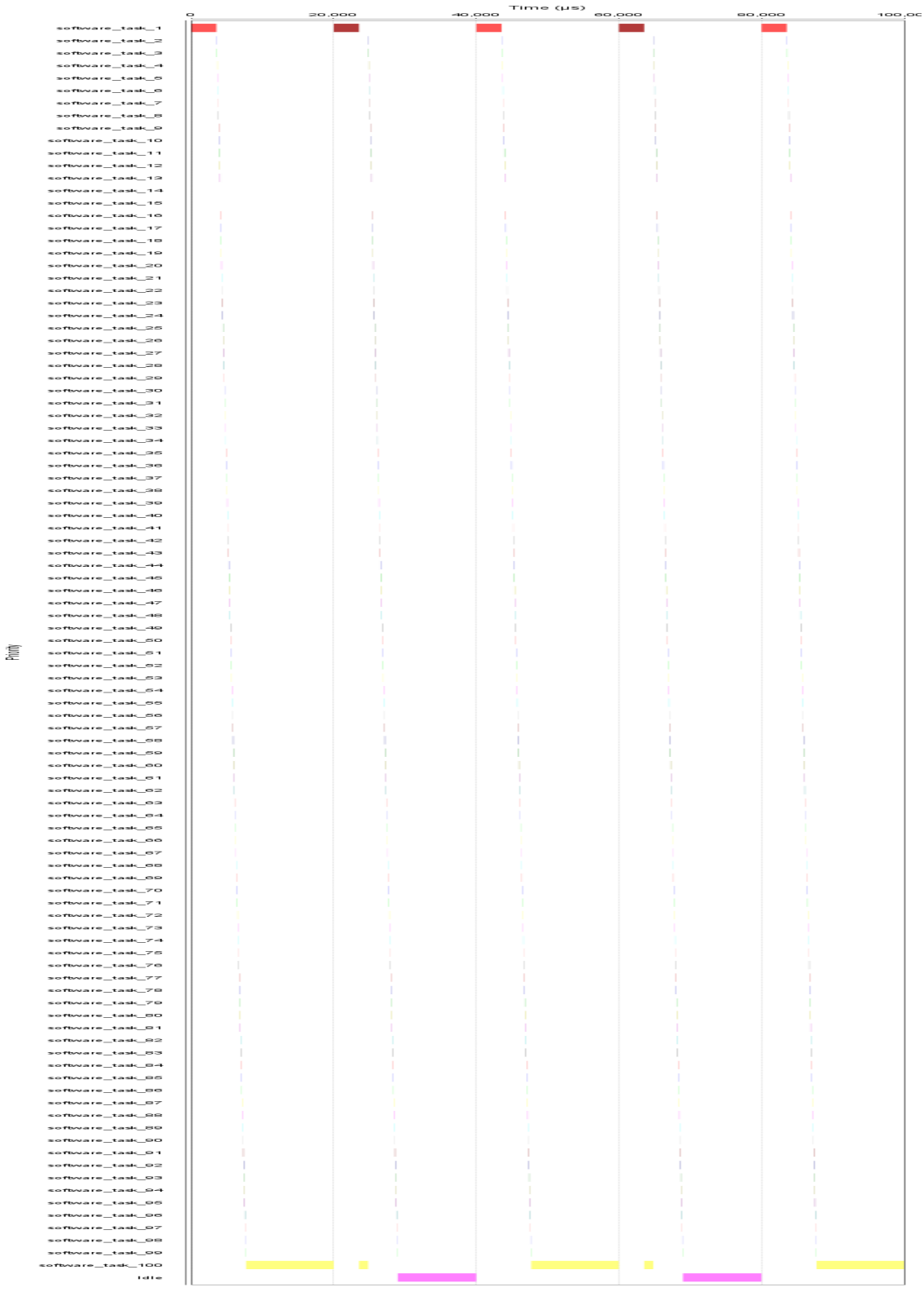


Figure D.8 Impact of no. of tasks for 100 tasks.

D.IV *TTP*

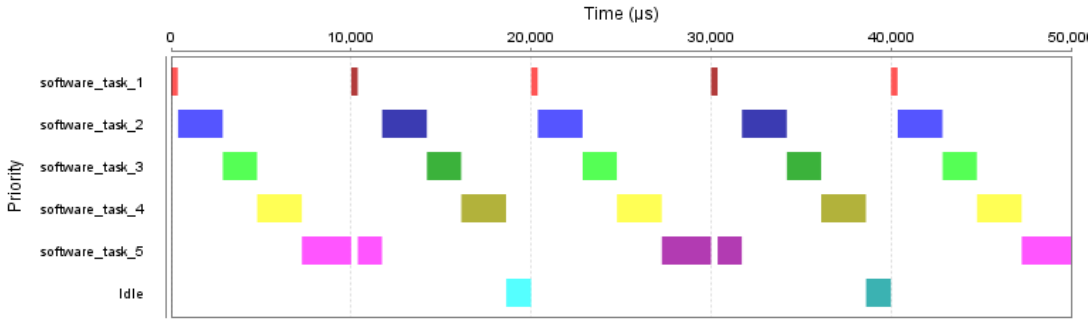


Figure D.9 Impact of number of tasks on TTP for 5 tasks.

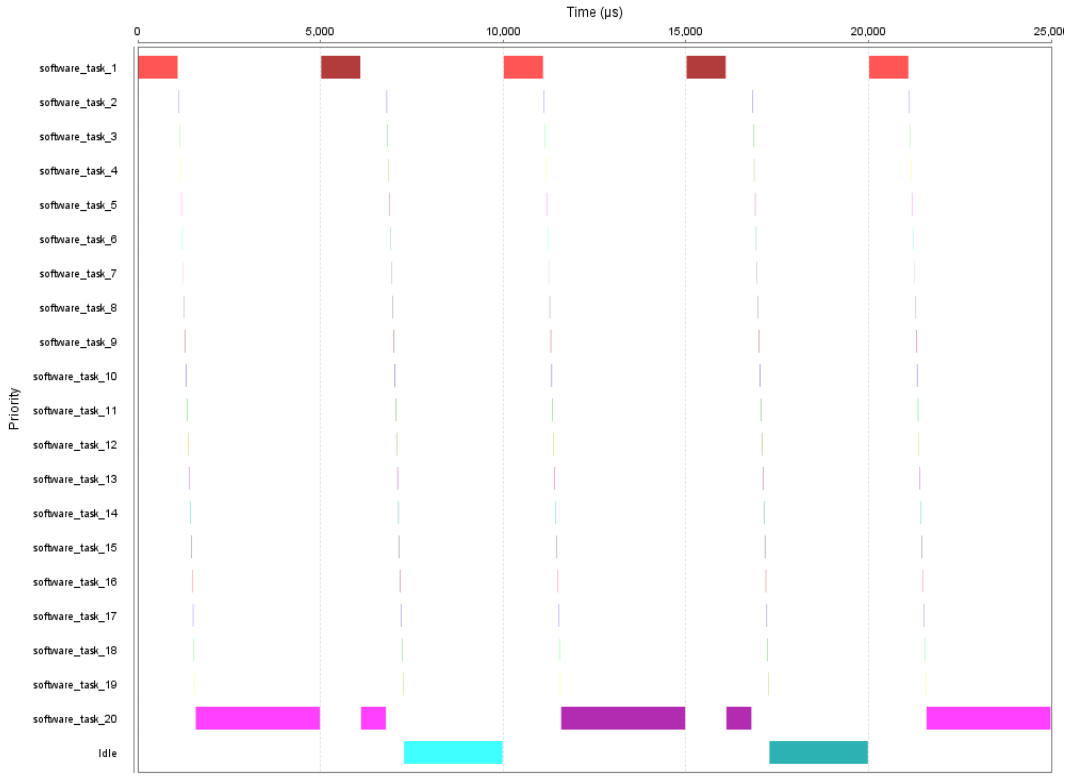


Figure D.10 Impact of no. of tasks on TTP for 20 tasks.

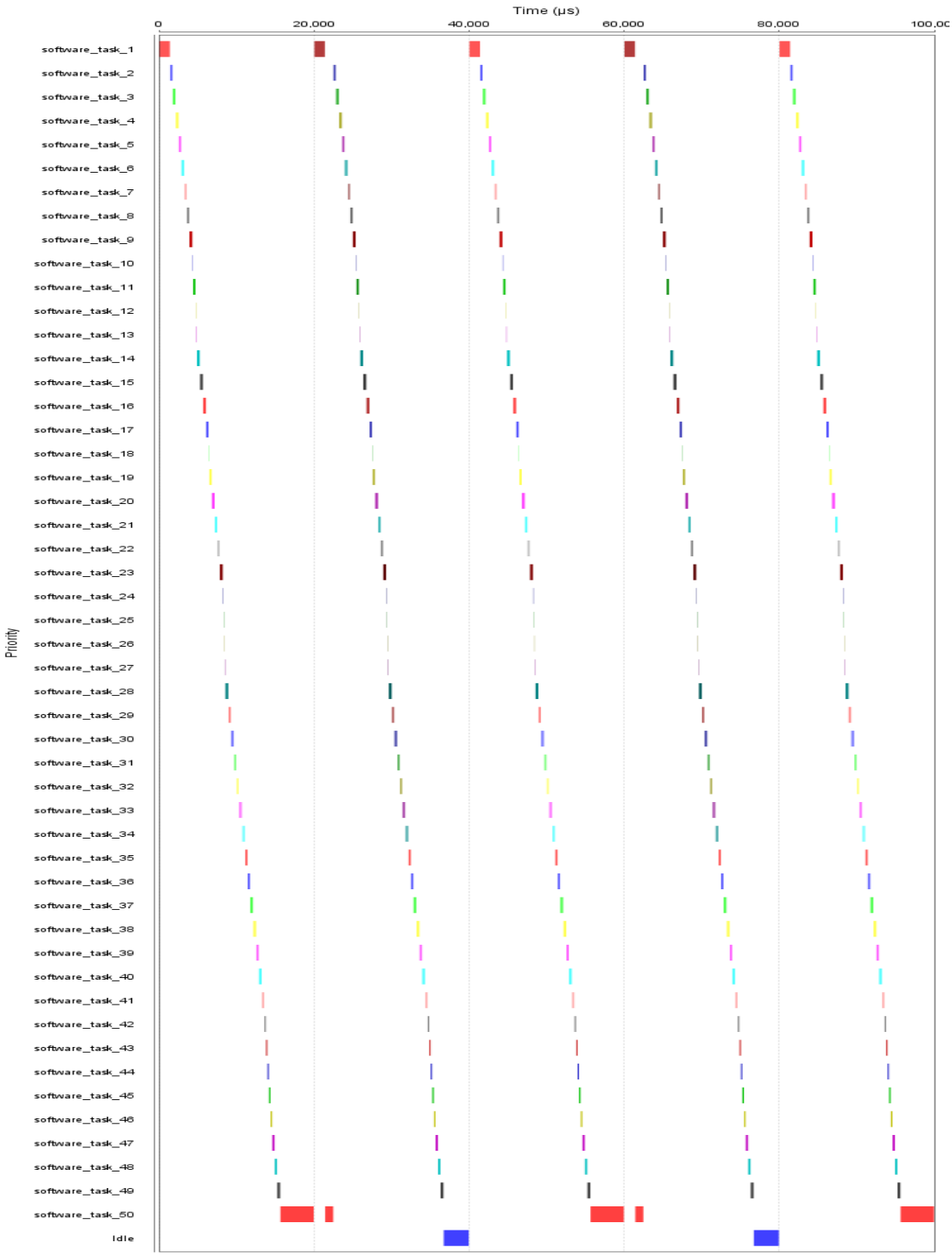


Figure D.11 Impact of no. of tasks on TTP for 50 tasks.

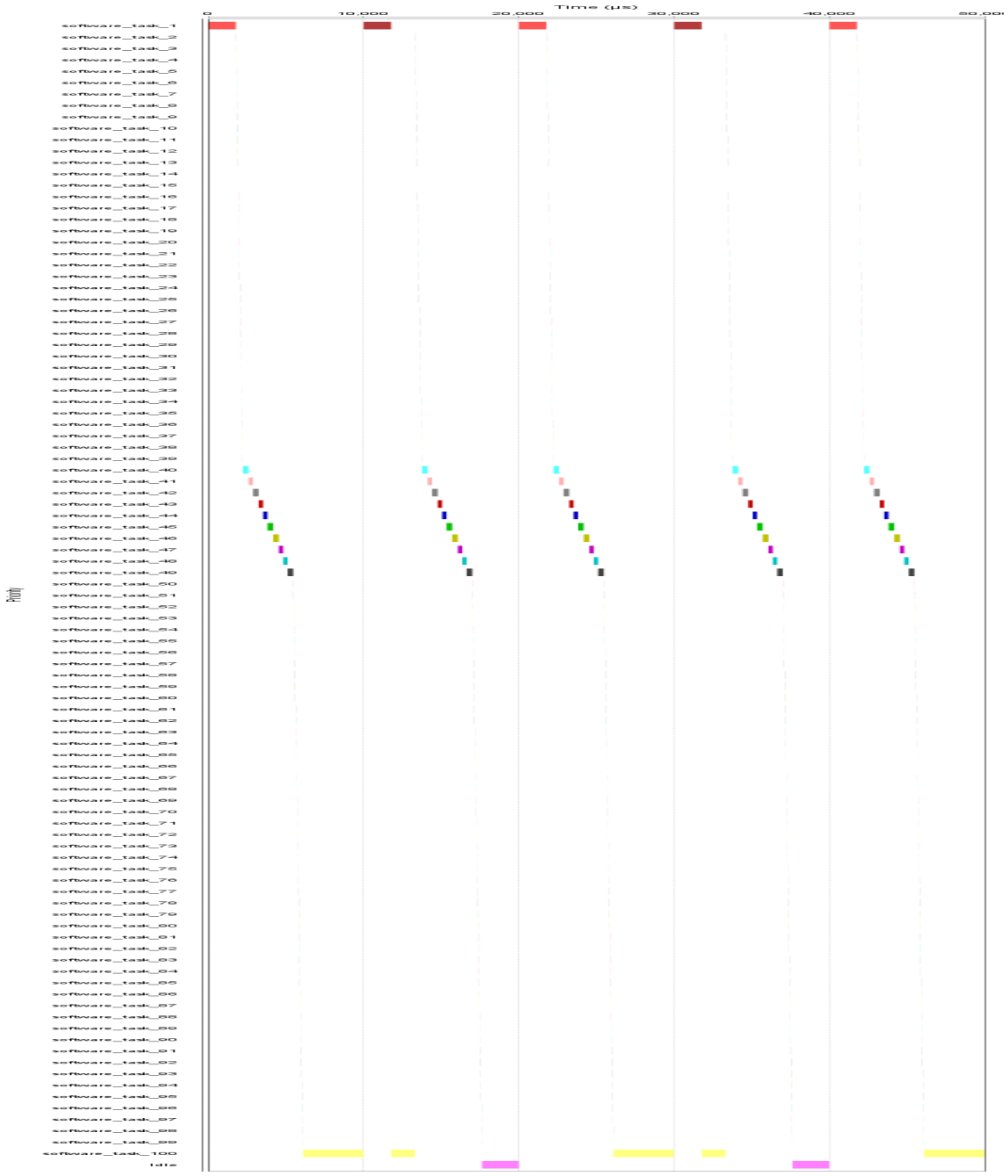


Figure D.12 Impact of no. of tasks on TTP for 100 tasks.

D.V Experimental results of impact of number of pre-emption

Sample results for the impact of pre-emption for 20 tasks are presented below. The analysis started from 0 pre-emption and ran up to 19 pre-emptions.

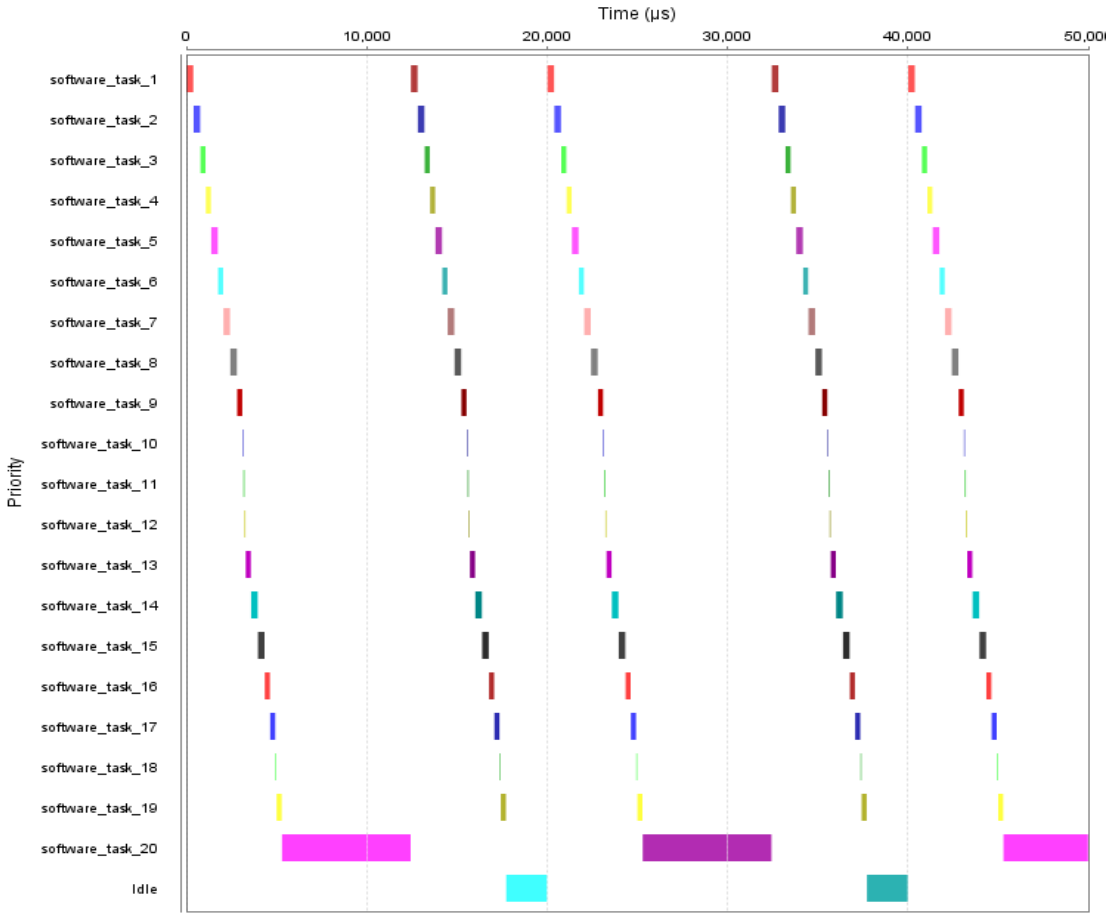


Figure D.13 Impact of 0 pre-emption.

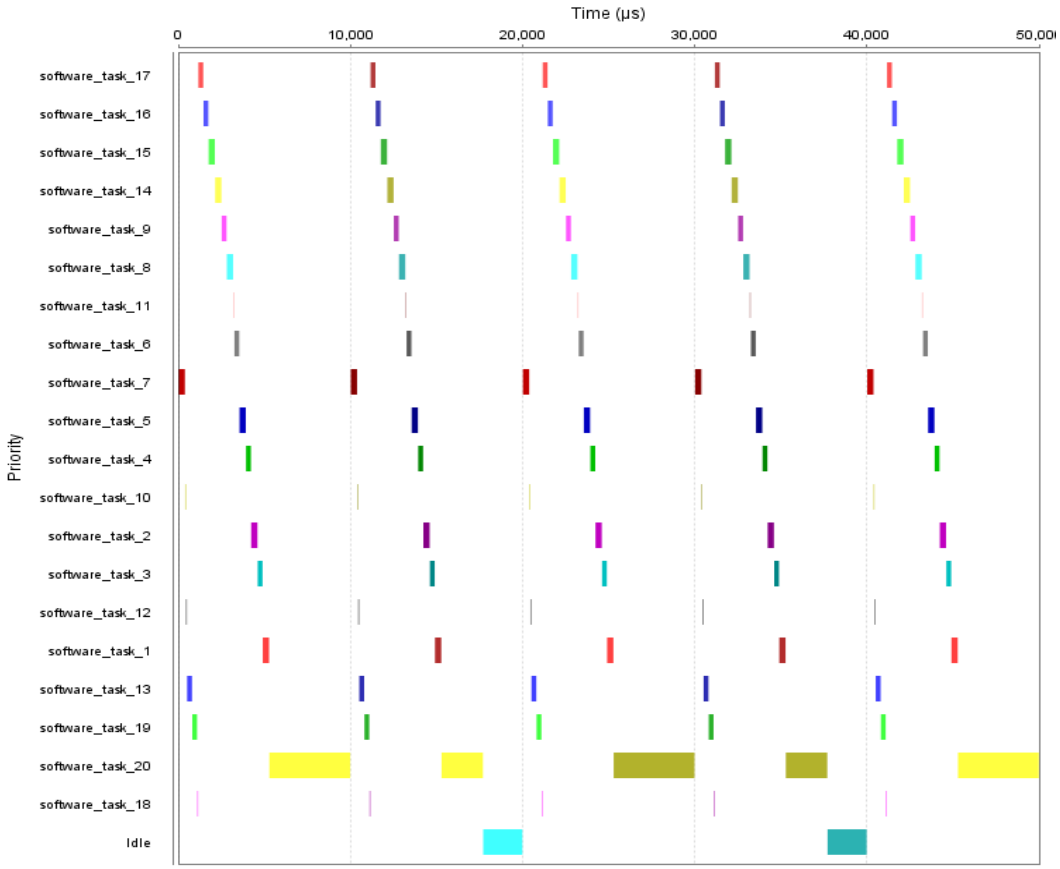


Figure D.14 Impact of 19 pre-emptions

Appendix -E

E.I Labview Tools used in timing measurements:

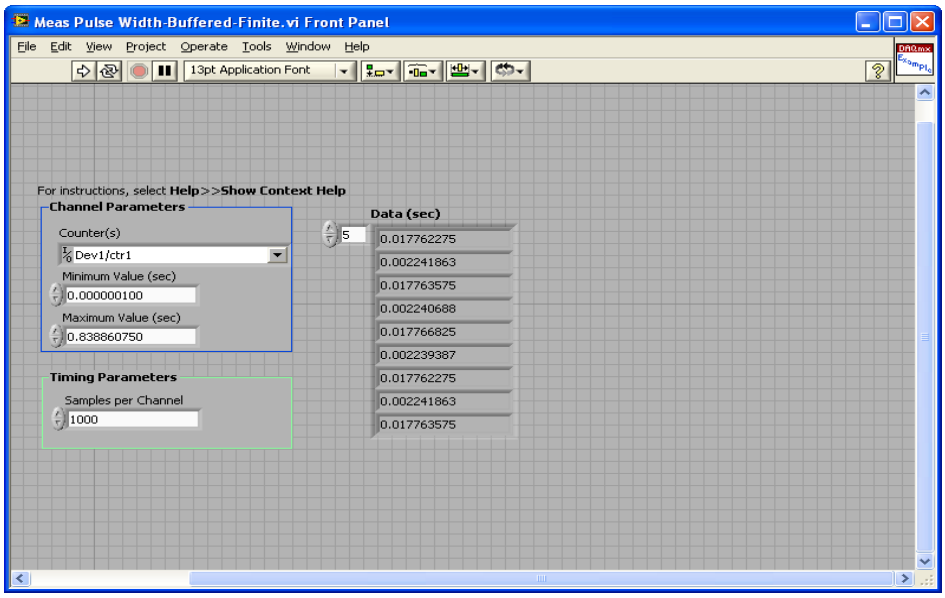


Figure E.1 Labview front end developed for measurement

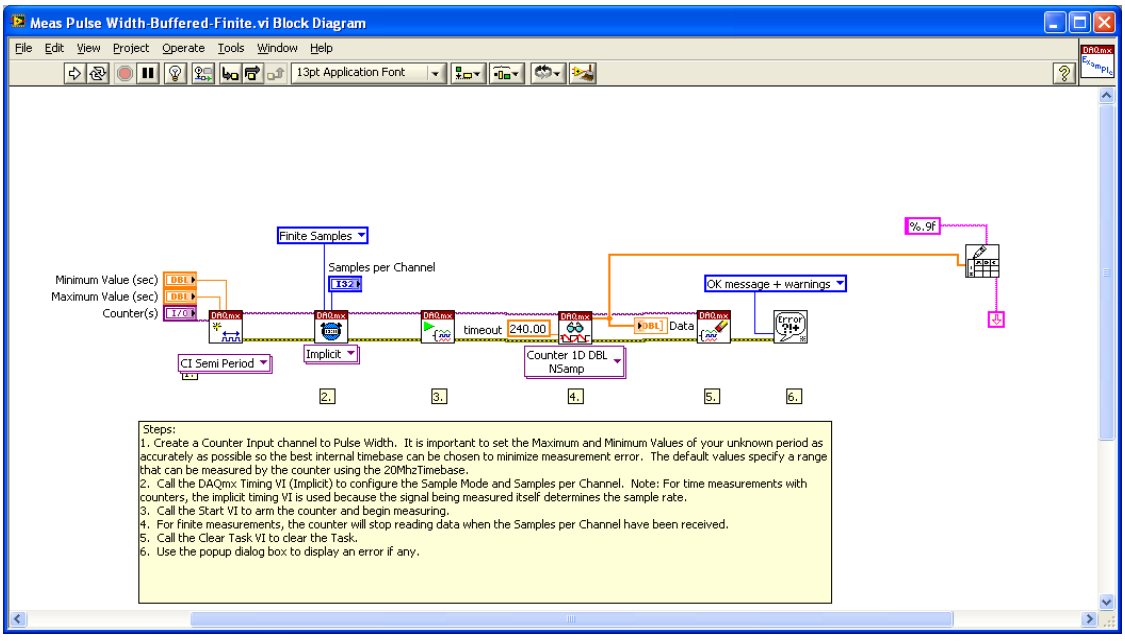


Figure E.2 LabView block diagram of measurement tool developed for timing analysis.

Appendix -F

F.I Aim of this pilot study

The aim of the pilot study is to explore the difficulties encountered when localising faults for a simple switch system.

F.II Target system specification

An Olimex LPC2129 ARM board was used as the target system platform. This board contains the programs needing to be debugged. In this case, the designer should invent a method to remotely debug the target system's peripherals, such as an LED system (represents the system's output) and a switch system (represents the system's input). Both peripherals are controlled by a scheduler with a time-triggered (TT) architecture.

The target system runs a 'switch-poll' task, which periodically checks for a switch pin and observes if it is pressed on or released. If the system detects that the push-button switch is pressed or released, the LED system will react by turning on or off the light. By using the time-triggered cooperative (TTC) scheduler, all the tasks are then pre-determined before execution. The target system runs the switch-poll task, which is invoked every 20 milliseconds. The other task is the LED response task, which responds to the switch state's transition. This task is invoked every 10 milliseconds.

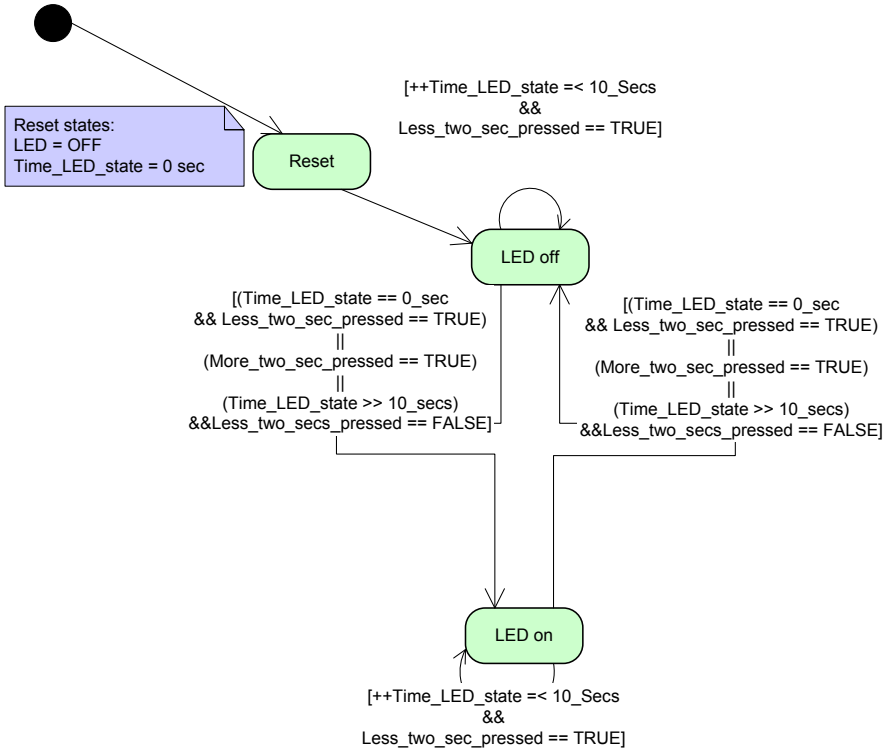


Figure F.1 LED state diagram

Initially, the LED is in the ‘off’ state. If the switch is pressed and released within 2 seconds, the LED turns on and remains on for 10 seconds, then the LED switches off again. The switch will remain on during the 10-second period, even if the switch is pressed again. In another scenario, if the LED is in the ‘off’ state and the switch is pressed and released for more than 2 seconds, the LED remains on permanently. Similar principles are used when the LED is in the ‘on’ state. The specification of the system is shown in Figure F.1.

A failure is an event that denotes a deviation between the actual service and the service intended. The system should run exactly as defined in the system specifications. For example, the switch system responds by turning the LED on for 10 seconds if it is pressed and released for less than 2 seconds. When the LED turns on for more or less than 10 seconds, the system is considered to be operating under faulty conditions. System specifications become the main reference for designing fault models. A system that does not fulfil its specification is a failed system.

F.III *Switch system test cases*

The test cases for the system are developed based on the target's specifications. The test cases contain all possible inputs (pre-conditions) for the system and all expected outputs. Black-box testing or requirement-based testing is implemented to check whether the system does what the specification says. Results from the testing procedure may be used to assess and investigate inaccuracies in the system. Testing will also uncover faults that may then be removed, thus increasing system dependability.

In order to obtain meaningful data through monitoring systems, some testing procedures are required. Table A.1 shows a list of tests carried out for the switch system.

Table F.1 Switch test cases

Testing	Description
Test 1	Switch is pressed for 200 milliseconds
Test 2	Switch is pressed for 1.9 seconds
Test 3	Switch is pressed for 3 seconds
Test 4	Switch is pressed for 6 seconds

The sample of expected results for the test cases:
Test 1: if pre condition = OFF, LED turns on for 10 seconds and turns off.
Test 2: if pre condition = OFF, LED turns on for 10 seconds and turns off.
Test 3: if pre condition = OFF, LED turns on and remains on
Test 4: if pre condition = OFF, LED turns on and remains on
The details of the test cases are mentioned in Appendix F.IV. A successful test case is one that shows that a program does not do what it is designed to do. One of the aims of the debugging processes is to find errors, as a result of a successful test case. Based on test results, the main sources of errors can be identified.

F.IV Conclusion

Here a pilot study for the testing system and the switch system with TT software architecture has been discussed. The development of the system begins with a fault analysis of the target system specifications. FMEA and FTA designs then become a foundation for the KB system, which is important in the fault localisation process. The system also facilitates online testing and monitoring of systems. The sample of outputs (monitoring and fault diagnosis processes) is shown in this chapter. There are only two faults diagnosed amongst the many potential faults, which could ultimately lead to the need to prolong the development time. The results from the empirical study suggest that further investigations of methods applied to predict a wide range of faults are required.

F.V Test cases for the switch system

F.VI *LED testing*

Case	LED Testing	Pre-Conditions	Consequences of states	Operation	Post-conditions
1	Press and release the switch for 200ms	LED is OFF Less_two_sec_pressed = FALSE Time_LED_state = 0 sec	1) Less_two_sec_pressed = TRUE 2)if (++Time_LED_state =< 10 secs) ~ LED turns ON 3)if Time_LED_state >> 10 secs ~ LED turns OFF ~Less_two_sec_pressed = FALSE	LED turns ON for 10 secs and then LED turns OFF	LED is OFF Less_two_sec_pressed = FALSE Time_LED_state = 0 sec
2	Press and release the switch for 2 seconds	LED is OFF Less_two_sec_pressed = FALSE Time_LED_state = 0 sec	1) Less_two_sec_pressed = TRUE 2)if (++Time_LED_state =< 10 secs) ~ LED turns ON 3)if Time_LED_state >> 10 secs ~ LED turns OFF ~Less_two_sec_pressed = FALSE	LED turns ON for 10 secs and then LED turns OFF	LED is OFF Less_two_sec_pressed = FALSE Time_LED_state = 0 sec
3	Press and release the switch for 3 seconds	LED is OFF More_two_sec_pressed = FALSE Time_LED_state = 0 sec	1) More_two_sec_pressed = TRUE 2) LED turns ON	LED turns ON permanently	LED is ON More_two_sec_pressed = FALSE Time_LED_state = 0 sec
4	Press and release the switch for 6 seconds	LED is OFF More_two_sec_pressed = FALSE	1) More_two_sec_pressed = TRUE 2) LED turns ON	LED turns ON permanently	LED is ON More_two_sec_pressed = FALSE

		Time_LED_state = 0 sec			Time_LED_state = 0 sec
Case	**LED Testing**	**Pre-Conditions**	**Consequences of states**	**Operation**	**Post-conditions**
5	Press and release the switch for 200ms	LED is ON Less_two_sec_pressed = FALSE Time_LED_state = 0 sec	1) Less_two_sec_pressed = TRUE 2)if (++Time_LED_state =< 10 secs) ~ LED turns OFF 3)if Time_LED_state >> 10 secs ~ LED turns ON ~Less_two_sec_pressed = FALSE	LED turns OFF for 10 secs and then LED turns ON	LED is ON Less_two_sec_pressed = FALSE Time_LED_state = 0 sec
6	Press and release the switch for 2 seconds	LED is ON Less_two_sec_pressed = FALSE Time_LED_state = 0 sec	1) Less_two_sec_pressed = TRUE 2)if (++Time_LED_state =< 10 secs) ~ LED turns OFF 3)if Time_LED_state >> 10 secs ~ LED turns ON ~Less_two_sec_pressed = FALSE	LED turns OFF for 10 secs and then LED turns ON	LED is ON Less_two_sec_pressed = FALSE Time_LED_state = 0 sec
7	Press and release the switch for 3 seconds	LED is ON More_two_sec_pressed = FALSE Time_LED_state = 0 sec	1) More_two_sec_pressed = TRUE 2) LED turns OFF	LED turns OFF permanently	LED is OFF More_two_sec_pressed = FALSE Time_LED_state = 0 sec
8	Press and release the switch for 6 seconds	LED is ON More_two_sec_pressed = FALSE	1) More_two_sec_pressed = TRUE 2) LED turns OFF	LED turns OFF permanently	LED is OFF More_two_sec_pressed = FALSE

		Time_LED_state = 0 sec			Time_LED_state = 0 sec
9	Press and release the switch several times	LED is OFF Less_two_sec_pressed = FALSE Time_LED_state = 0 sec	1) Less_two_sec_pressed = TRUE 2)if (++Time_LED_state =< 10 secs) ~ LED turns ON 3)if Time_LED_state >> 10 secs	LED turns ON for 10 secs and then LED turns OFF	LED is OFF Less_two_sec_pressed = FALSE Time_LED_state = 0 sec

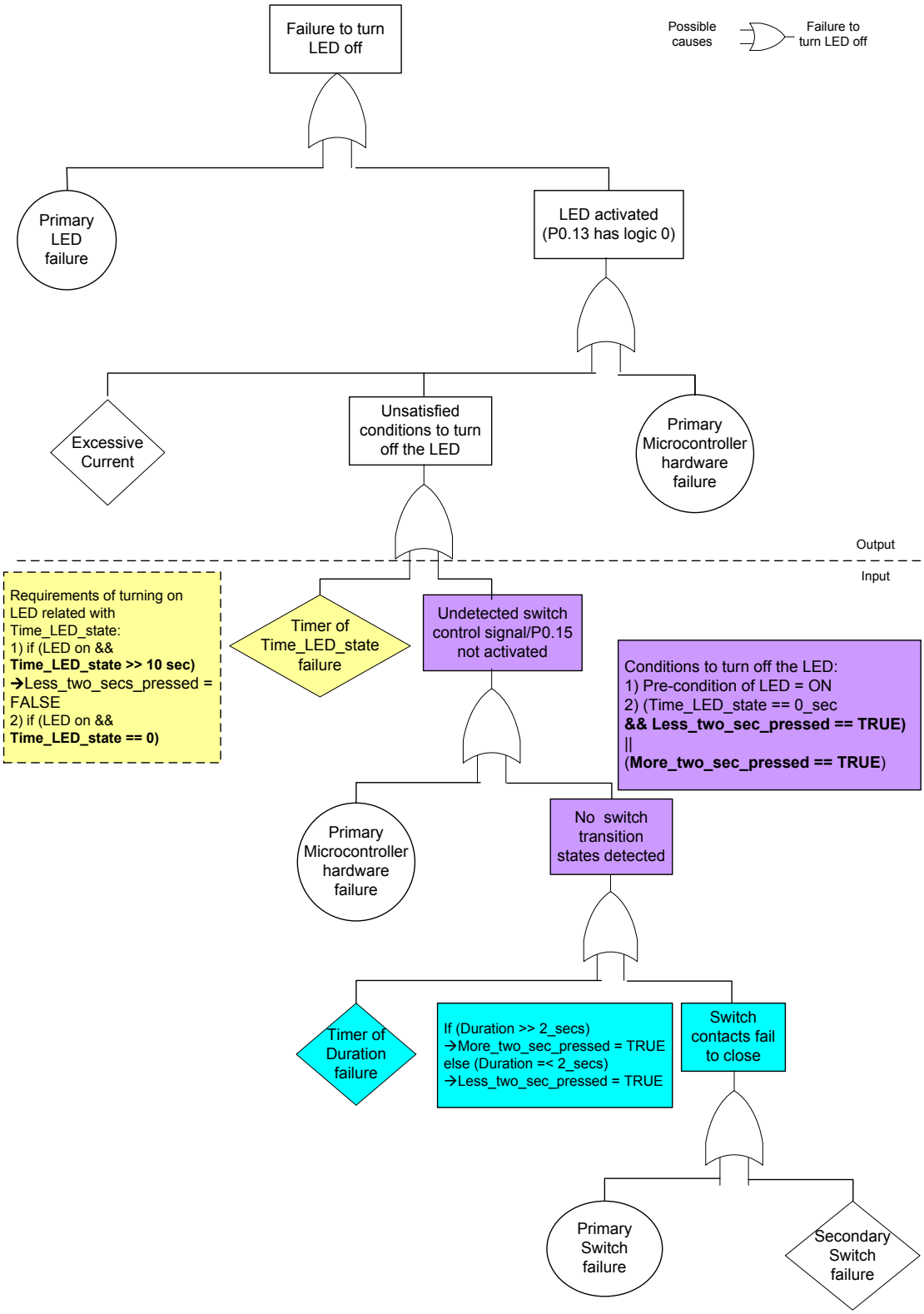
F.VII *Switch testing*

Case	Switch Testing	Pre-Conditions	Consequences of states	Operation	Post-conditions
1	Press and release the switch for 200ms	Duration = 0; Less_two_sec_pressed = FALSE	1) if (switch =! release) ~++Duration 2) if (Duration =< 2_seconds) ~ Duration = 200ms	The switch is pressed less than 2 seconds	Duration = 0; Less_two_sec_pressed = TRUE
2	Press and release the switch for 2 seconds	Duration = 0; Less_two_sec_pressed = FALSE	1) if (switch =! release) ~++Duration ~ Duration = 200ms 2) if (Duration =< 2_seconds) Less_two_sec_pressed = TRUE	The switch is pressed less than 2 seconds	Duration = 0; Less_two_sec_pressed = TRUE
3	Press and release the switch for 3 seconds	Duration = 0; Less_two_sec_pressed = FALSE More_two_sec_pressed = FALSE	1) if (switch =! release) ~++Duration ~ Duration = 3 seconds 2) if (Duration =< 2_seconds) Less_two_sec_pressed = TRUE 3) if (Duration >> 2_seconds) More_two_sec_pressed = TRUE Less_two_sec_pressed = FALSE	The switch is pressed more than 2 seconds	Duration = 0; More_two_sec_pressed = TRUE

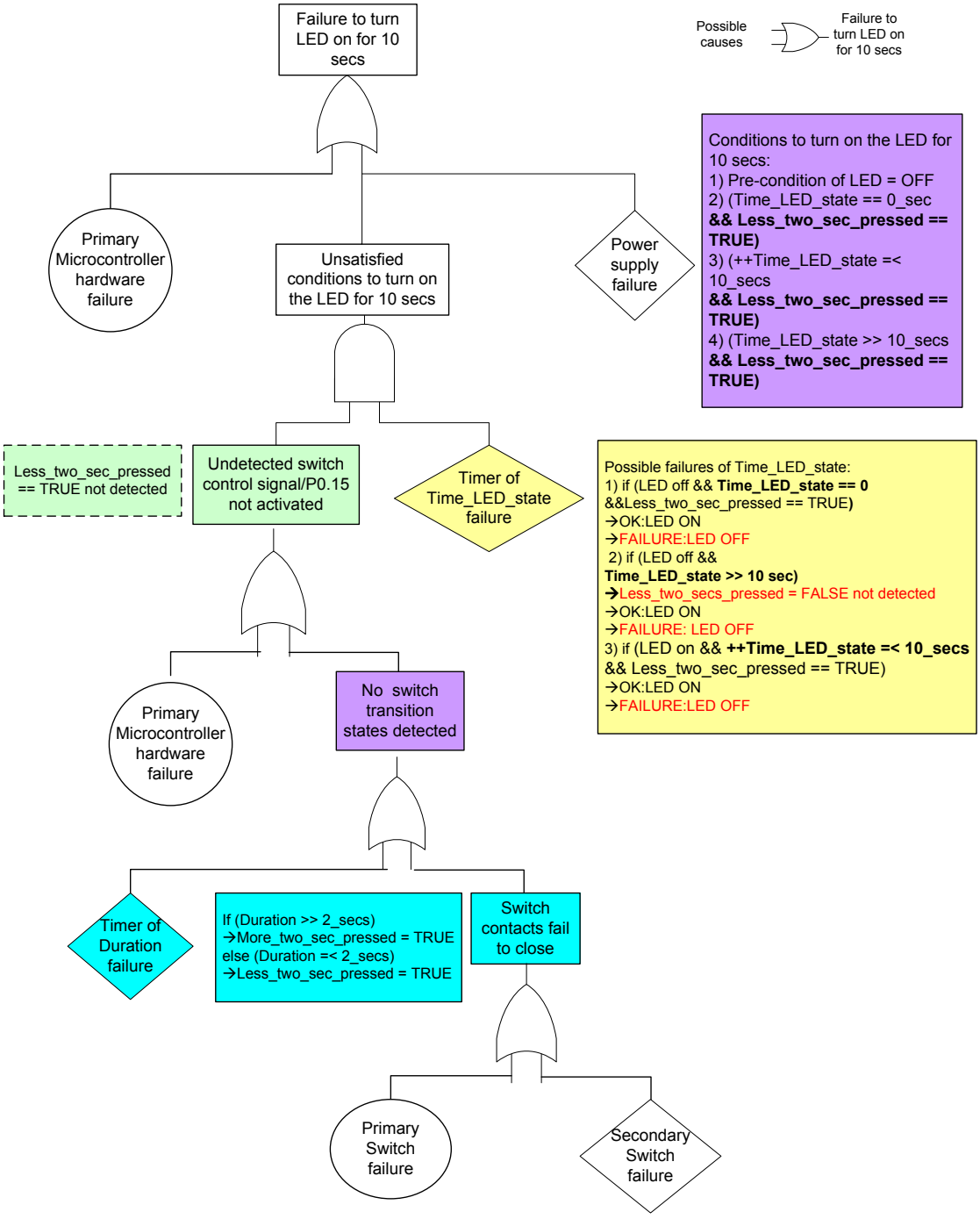
Case	Switch Testing	Pre-Conditions	Consequences of states	Operation	Post-conditions
4	Press and release the switch for 6 seconds	Duration = 0; Less_two_sec_pressed = FALSE More_two_sec_pressed = FALSE	1) if (switch =! release) ~++Duration ~ Duration = 6 seconds 2) if (Duration =< 2_seconds) Less_two_sec_pressed = TRUE 3) if (Duration >> 2_seconds) More_two_sec_pressed = TRUE Less_two_sec_pressed = FALSE	The switch is pressed more than 2 seconds	Duration = 0; More_two_sec_pressed = TRUE

F.VIII Fault Tree Analysis (FTA) for the switch system

F.IX *Failure to turn the LED off*



F.X *Failure to turn the LED on for 10 seconds*



References

ABELLA, J., QUI\, E., \#241, ONES, CAZORLA, F. J., SAZEIDES, Y. & VALERO, M. 2011. RVC: a mechanism for time-analyzable real-time processors with faulty caches. *Proceedings of the 6th International Conference on High Performance and Embedded Architectures and Compilers.* Heraklion, Greece: ACM.

ALBRECHT, A. J. & GAFFNEY, J. E., JR. 1983. Software Function, Source Lines of Code, and Development Effort Prediction: A Software Science Validation. *IEEE Transactions on Software Engineering,,* SE-9**,** 639-648.

ANDERSON, J. H., RAMAMURTHY, S. & JEFFAY, K. 1995. Real-time computing with lock-free objects. *Tech. Rep. TR95-021*.

ANDERSON, J. H., RAMAMURTHY, S. & JEFFAY, K. 1997a. Real-time computing with lock-free shared objects. *ACM Trans. Comput. Syst.,* 15**,** 134-165.

ANDERSON, J. H., RAMAMURTHY, S., MOIR, M. & JEFFAY, K. 1997b. Lock-free transactions for real-time systems. *Real-Time Databases: Issues and Applications*.

ANTON, A. I. 1996. Goal-based requirements analysis, *Proceedings of the Second International Conference on Requirements Engineering,* 15-18 Apr 1996. 136-144.

ARAKAWA, H., KATCHER, D. I., STROSNIDER, J. K. & TOKUDA, H. 1993. Modeling and validation of the real-time Mach scheduler. *Proceedings of the 1993 ACM SIGMETRICS conference on Measurement and modeling of computer systems.* Santa Clara, California, United States: ACM.

ATHAIDE, K. F., PONT, M. J. & AYAVOO, D. 2008. Shared-clock methodology for time-triggered multi-cores. *Concurrent Systems Engineering Series,* 66**,** 149-162.

AUDSLEY, N., BURNS, A., RICHARDSON, M., TINDELL, K. & WELLINGS, A. J. 1993. Applying new scheduling theory to static priority pre-emptive scheduling. *Software engineering journal,* 8**,** 284-292.

AUDSLEY, N. C., BURNS, A., DAVIS, R. I., TINDELL, K. W. & WELLINGS, A. J. 1995. Fixed priority pre-emptive scheduling: An historical perspective. *Real-Time Systems,* 8**,** 173-198.

AYAVOO, D., PONT, M. J., SHORT, M. & PARKER, S. 2007. Two novel shared-clock scheduling algorithms for use with 'Controller Area Network' and related protocols. *Microprocessors and Microsystems,* 31**,** 326-334.

BABAR, M. A. & GORTON, I. 2004. Comparison of scenario-based software architecture evaluation methods. *11th Asia-Pacific Software Engineering Conference,*, 30 Nov.-3 Dec. 2004. 600-607.

BAKER, T. P. 1991. Stack-based scheduling of realtime processes. *The journal of Real- Time Systems,* Vol. 3, pp. 67-99.

BARRY, R. 2011. FreeRTOS Reference Manual - API Functions and Configuration Options. Real Time Engineers Ltd.

BARUAH, S. K. & CHAKRABORTY, S. 2006. Schedulability analysis of non-preemptive recurring real-time tasks, In proceeding of: 20th International Parallel and Distributed Processing Symposium (IPDPS 2006), Proceedings, 25-29 April 2006, Rhodes Island, Greece.

BASILI, V. R. 1980. MODELS AND METRICS FOR SOFTWARE MANAGEMENT AND ENGINEERING. *Instrumentation in the Pulp and Paper Industry, Proceedings,* 1**,** 278-289.

BATE, I. J. & BURNS, A. 1997. A dependable distributed architecture for a safety critical hard real-time system. *IEE Colloquium (Digest)***,** 1/1-1/5.

BATE, I.J. 1998. "Scheduling and timing analysis for safety critical real-time systems", PhD thesis, University of York, UK.

BATE, I. 2008. Systematic approaches to understanding and evaluating design trade-offs. *Journal of Systems and Software,* 81**,** 1253-1271.

BATE, I. & BURNS, A. 2003. An integrated approach to scheduling in safety-critical embedded control systems. *Real-Time Systems,* 25**,** 5-37.

BEHRMANN, G., DAVID, A. & LARSEN, K. 2004. A Tutorial on Uppaal. *In:* BERNARDO, M. & CORRADINI, F. (eds.) *Formal Methods for the Design of Real-Time Systems.* Springer Berlin Heidelberg.

BENGTSSON, P., LASSING, N., BOSCH, J. & VAN VLIET, H. 2004. Architecture-level modifiability analysis (ALMA). *Journal of Systems and Software,* 69**,** 129-147.

BERNAT, G., COLIN, A. & PETTERS, S. M. 2002. WCET analysis of probabilistic hard real-time systems. *In: Proceedings of the 23rd IEEE Real-Time Systems Symposium,* Austin, TX. 279-288.

BERTOGNA, M., BUTTAZZO, G., MARINONI, M., YAO, G., ESPOSITO, F. & CACCAMO, M. 2010. Preemption points placement for sporadic task sets. Brussels. 251-260.

BERTOGNA, M., XHANI, O., MARINONI, M., ESPOSITO, F. & BUTTAZZO, G. 2011. Optimal selection of preemption points to minimize preemption overhead. *Real-Time Systems (ECRTS), 2010 22nd Euromicro Conference,* Porto. 217-227.

BINI, E. & BUTTAZZO, G. C. 2004. Schedulability analysis of periodic fixed priority systems. *IEEE Transactions on Computers,* 53**,** 1462-1473.

BINI, E., BUTTAZZO, G. C. & BUTTAZZO, G. M. 2003. Rate monotonic analysis: The hyperbolic bound. *IEEE Transactions on Computers,* 52**,** 933-942.

BLETSAS, K. 2007. *Worst-case and best-case timing analysis for real-time embedded systems with limited parallelism,* York, University of York, Dept. of Computer Science.

BOUCKÉ, N., WEYNS, D., SCHELFTHOUT, K. & HOLVOET, T. 2006. Applying the ATAM to an Architecture for Decentralized Control of a Transportation

System. *In:* HOFMEISTER, C., CRNKOVIC, I. & REUSSNER, R. (eds.) *Quality of Software Architectures.* Springer Berlin Heidelberg.

BR©ÞUNL, T. 2006. *Embedded robotics : mobile robot design and applications with embedded systems,* Berlin, Springer.

BRIAND, L. C., LABICHE, Y. & SHOUSHA, M. Stress testing real-time systems with genetic algorithms. 2005. *In:* BEYER, H. G., O'REILLY, U. M., ARNOLD, D., BANZHAF, W., BLUM, C., BONABEAU, E. W., CANTU-PAZ, E., DASGUPTA, D., DEB, K. & ET AL., eds.,Washington, D.C., 1021-1028.

BURNS, A., HAYES, N. & RICHARDSON, M. F. 1995. Generating feasible cyclic schedules. *Control Engineering Practice,* 3**,** 151-162.

BURNS, A. & MCDERMID, J. A. 1994. Real-time safety-critical systems: Analysis and synthesis. *Software engineering journal,* 9**,** 267-281.

BUTTAZZO, G. & KUO, T. W. 2009. Guest editorial: Special issue on real-time systems - Part II. *IEEE Transactions on Industrial Informatics,* 5**,** 1-2.

BUTTAZZO, G. C. 1997. *Hard real-time computing systems : predictable scheduling algorithms and applications,* Boston ; London, Kluwer Academic Publishers.

BUTTAZZO, G. C. 2005a. *Hard real-time computing systems : predictable scheduling algorithms and applications,* New York, Springer.

BUTTAZZO, G. C. 2005b. Rate Monotonic vs. EDF: Judgment day. *Real-Time Systems,* 29**,** 5-26.

CHAN, K. L. & PONT, M. J. 2010. Real-time non-invasive detection of timing-constraint violations in time-triggered embedded systems. Bradford. 1978-1986.

CHO, Y., ZERGAINOH, N.-E., YOO, S., JERRAYA, A. & CHOI, K. 2007. Scheduling with accurate communication delay model and scheduler implementation for multiprocessor system-on-chip. *Design Automation for Embedded Systems,* 11**,** 167-191.

CHU, W. K., ZHANG, F. M. & FAN, X. G. 2007. Measurement of real-time performance of embedded Linux systems. *Xi Tong Gong Cheng Yu Dian Zi Ji Shu/Systems Engineering and Electronics,* 29**,** 1385-1388+1401.

CLEGG, C. J. 2008. FreeRTOS / SafeRTOS in a Medical Device. *EmbeddedRelated.com.*

CODE, C. 2006. Code Counter Pro Software. v1.32 ed.: GeroneSoft, US.

COOK, R. A. & SPEAR, A. J. 1998. Back to Mars: The Mars Pathfinder mission. *Acta Astronautica,* 41**,** 599-608.

COOLING, J. E. 2001. *Software engineering for real-time systems,* Harlow, Addison-Wesley.

DAVIS, R. I., ZABOS, A. & BURNS, A. 2008. Efficient exact schedulability tests for fixed priority real-time systems. *IEEE Transactions on Computers,* 57**,** 1261-1276.

DERDERIAN, K., HIERONS, R., HARMAN, M. & GUO, Q. 2010. Estimating the feasibility of transition paths in extended finite state machines. *Automated Software Engineering,* 17**,** 33-56.

DURKIN, T. 1998. The Vx_files: What the media couldn't tell you about Mars Pathfinder. *Robot Science and Technology,* 1**,** 3.

FENTON, N. E. & PFLEEGER, S. L. 1997. Software Metrics: A Rigorous and Practical Approach. *Software Metrics: A Rigorous and Practical Approach*.

FOHLER, G., LENNVALL, T. & BUTTAZZO, G. 2001. Improved handling of soft aperiodic tasks in offline schedule real-time systems using total bandwidth server. *Emerging Technologies and Factory Automation, 2001. Proceedings. 2001 8th IEEE International Conference on*. Antibes-Juan les Pins, Sweden. 151-157.

GARLAN, D. 2000. Software architecture: a roadmap. *Proceedings of the Conference on The Future of Software Engineering.* Limerick, Ireland: ACM.

GENDY, A. K., LEI, D. & PONT, M. J. 2008. Improving the performance of time-triggered embedded systems by means of a scheduler agent. Las Vegas, NV. 57-63.

GENDY, A. K. & PONT, M. J. 2008a. Automatically configuring time-triggered schedulers for use with resource-constrained, single-processor embedded systems. *IEEE Transactions on Industrial Informatics,* 4**,** 37-46.

GENDY, A. K. & PONT, M. J. 2008. Towards a generic "single-path programming" solution with reduced power consumption. *ASME Conference Proceedings 2007.* Las Vegas, NV. 65-71.

HA, X, NSEL, J., ROSE, D., HERBER, P. & GLESNER, S. 2011. An Evolutionary Algorithm for the Generation of Timed Test Traces for Embedded Real-Time Systems. *Software Testing, Verification and Validation (ICST), 2011 IEEE Fourth International Conference on*, 21-25 March 2011. 170-179.

HANIF, M., PONT, M.J. AND AYAVOO, D. 2008. Implementing a simple but flexible time-triggered architecture for practical deeply-embedded applications. *in Proceedings of the 4th UK Embedded Forum*.

HARMAN, M., MANSOURI, S. A. & ZHANG, Y. 2012. Search-based software engineering: Trends, techniques and applications. *ACM Comput. Surv.,* 45**,** 1-61.

HUGHES, Z. M. & PONT, M. J. 2008. Reducing the impact of task overruns in resource-constrained embedded systems in which a time-triggered software architecture is employed. *Transactions of the Institute of Measurement and Control,* 30**,** 427-450.

HUNT, C. & JOHN, B. 2012. *Java performance,* Upper Saddle River, NJ, Addison-Wesley.

IQBAL, M., ARCURI, A. & BRIAND, L. 2012. Combining Search-Based and Adaptive Random Testing Strategies for Environment Model-Based Testing of Real-Time Embedded Systems. *In:* FRASER, G. & TEIXEIRA DE SOUZA, J. (eds.) *Search Based Software Engineering.* Springer Berlin Heidelberg.

JAYARATNA, N. 1994. *Understanding and Evaluating Methodologies: NIMSAD, a Systematic Framework*, McGraw-Hill, Inc.

JEFFAY, K., STANAT, D. F. & MARTEL, C. U. 1991. On non-preemptive scheduling of period and sporadic tasks. R*eal-Time Systems Symposium Proceedings.,* Twelfth, 4-6 Dec 1991 1991a. 129-139.

JEFFAY, K., STANAT, D. F. & MARTEL, C. U. 1991b. On non-preemptive scheduling of periodic and sporadic tasks. *IEEE Real-time Systems Symposium***,** 129-139.

JOSEPH, M. & PANDYA, P. 1986. Finding Response Times in a Real-Time System. *The Computer Journal,* 29**,** 390-395.

KACPRZAK, M., NABIAŁEK, W., NIEWIADOMSKI, A., PENCZEK, W., PÓŁROLA, A., SZRETER, M., WOŹNA, B. & ZBRZEZNY, A. 2008. VerICS 2007 - a Model Checker for Knowledge and Real-Time. *Fundamenta Informaticae,* 85**,** 313-328.

KATCHER, D. I., ARAKAWA, H. & STROSNIDER, J. K. 1993. Engineering and analysis of fixed priority schedulers. *IEEE Transactions on Software Engineering,* 19**,** 920-934.

KAZMAN, R., BASS, L., WEBB, M. & ABOWD, G. 1994. SAAM: a method for analyzing the properties of software architectures. *Proceedings of the 16th international conference on Software engineering.* Sorrento, Italy: IEEE Computer Society Press.

KAZMAN, R., KLEIN, M. & CLEMENTS, P. 1999. Evaluating software architectures for real-time systems. *Annals of Software Engineering,* 7**,** 71-93.

KAZMAN, R., KLEIN, M. & CLEMENTS, P. 2000. *ATAM: Method for Architecture Evaluation*.

KIM, S.-K., MIN, S. L. & HA, R. Efficient worst case timing analysis of data caching. *In Proceedings of the 2nd IEEE Real-Time Technology and Applications Symposium*, 1996 Brookline, MA, USA, 230-240.

KITCHENHAM, B. 2010. What's up with software metrics? - A preliminary mapping study. *Journal of Systems and Software,* 83**,** 37-51.

KOPETZ, H. 1991. Event-triggered versus time-triggered real-time systems. *Proceedings of the International Workshop on Operating Systems of the 90s and Beyond*. Springer-Verlag London, UK ©1991

KOPETZ, H. Why time-triggered architectures will succeed in large hard real-time systems. Proceedings of the Fifth IEEE Computer Society Workshop on Future Trends of Distributed Computing Systems, 1995 Cheju Island, South Korea., 2-9.

KOPETZ, H. 1997. *Real-time systems : design principles for distributed embedded applications,* Boston ; London, Kluwer Academic.

KURIAN, S. & PONT, M. J. 2007. The maintenance and evolution of resource-constrained embedded systems created using design patterns. *Journal of Systems and Software,* 80**,** 32-41.

LABROSSE, J. J. 2002. *MicroC/OS-II : the real-time kernel,* Lawrence, Kan., CMP Books.

LABROSSE, J. J. 2008. *Embedded software,* Amsterdam; Boston, Mass., Elsevier/Newnes.

LAPLANTE, P. A. & JOHN WILEY & SONS. 2004. *Real-time systems design and analysis,* Hoboken, N.J., Wiley.

LAUREN, M. K. 2001. FRACTAL METHODS APPLIED TO DESCRIBE CELLULAR AUTOMATON COMBAT MODELS. *Fractals,* 09**,** 177-184.

LEHOCZKY, J., SHA, L. & DING, Y. The rate monotonic scheduling algorithm: exact characterization and average case behavior. Real Time Systems Symposium, 1989., Proceedings., 5-7 Dec 1989 1989. 166-171.

LEUNG, J. Y. T. & WHITEHEAD, J. 1982. On the complexity of fixed-priority scheduling of periodic, real-time tasks. *Performance Evaluation,* 2**,** 237-250.

LINDGREN, P., ERIKSSON, J., AITTAMAA, S. & NORDLANDER, J. 2008. TinyTimber, reactive objects in C for real-time embedded systems. Munich. 1382-1385.

LINDSTRÖM, B., OFFUTT, J. & ANDLER, S. F. 2008. Testability of dynamic real-time systems: An empirical study of constrained execution environment implications. *International Conference on Software Testing, Verification, and Validation.* Lillehammer. 112-120.

LIU, C. L. & LAYLAND, J. W. 1973. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *J. ACM,* 20**,** 46-61.

LIU, J. W. S. 2000. *Real-time systems,* Upper Saddle River, N.J. ; London, Prentice Hall.

LOCKE, C. D. 1992. Software architecture for hard real-time applications: Cyclic executives vs. fixed priority executives. *Real-Time Systems,* 4**,** 37-53.

MANDRIOLI, D., MORASCA, S. & MORZENTI, A. 1995. Generating test cases for real-time systems from logic specifications. *ACM Trans. Comput. Syst.,* 13**,** 365-398.

MARWEDEL, P. 2003. *Embedded system design,* Boston, Kluwer Academic Publishers.

MATHWORKS, T. 2010. Matlab: The Language of Technical Computing. US: The Mathworks Incoporation.

MATLABCODER 2012. Matlab Coder: Generate C and C++ code from MATLAB code. The Mathworks Incorporation.

MICROSOFT 2010. Microsoft® Visual Studio® 2010 Professional US: Microsoft Corporation.

MIN-ALLAH, N., YONG-JI, W., JIAN-SHENG, X. & LIU, J. 2007. Revisiting fixed priority techniques. *International Conference, EUC 2007*.Taipei.

MORENO, G. A., SMITH, C. U. & WILLIAMS, L. G. 2008. Performance analysis of real-time component architectures: A model interchange approach. *Journal Special Issue on Software and Performance.* 115-126.

MUHAMMAD, A. & PONT, M. J. A time-triggered communication protocol for CAN-based networks with a fault-tolerant star topology. 2010. *IEEE 10th*

International Conference on Computer and Information Technology (CIT), Bradford. 2347-2354.

NAHAS, M., PONT, M. J. & SHORT, M. 2009. Reducing message-length variations in resource-constrained embedded systems implemented using the Controller Area Network (CAN) protocol. *Journal of Systems Architecture,* 55**,** 344-354.

NATIONAL, C. I. 2010. NI LabVIEW Full Development System for Windows 2009. Austin: National Instruments Corporation.

NOURCH, #232, ELLEUCH, N., KHALFALLAH, A. & AHMED, S. B. 2007. ArchMDE approach for the development of embedded real time systems. *Proceedings of the 12th international conference on Reliable software technologies.* Geneva, Switzerland: Springer-Verlag.

NXP 2011. LPC2377/78 Single-chip 16-bit/32-bit microcontrollers; 512 kB flash with ISP/IAP, Ethernet, USB 2.0, CAN, and 10-bit ADC/DAC. *Product datasheet.* U.K: NXP Semiconductor.

OBERMAISSER, R., PETI, P. & KOPETZ, H. Virtual networks in an integrated time-triggered architecture. 2005 Sedona, AZ. 241-253.

OLIMEX 2009. LPC-2378STK development board. United Kingdom: OLIMEX Ltd.

PONT, M. J. 2001a. *Patterns for time-triggered embedded systems : building reliable applications with the 8051 family of microcontrollers,* Harlow, Addison-Wesley.

PONT, M. J. 2002. *Embedded C,* London, Addison-Wesley.

PONT, M. J. 2003. Supporting the development of time-triggered co-operatively scheduled (TTCS) embedded software using design patterns. *Informatica (Ljubljana),* 27**,** 81-88.

PONT, M. J. 2008. Applying time-triggered architectures in reliable embedded systems: Challenges and solutions. *Elektrotechnik und Informationstechnik,* 125**,** 401-405.

POTOCKI DE MONTALK, J. P. Computer software in civil aircraft. Computer Assurance, 1991. COMPASS '91, Systems Integrity, Software Safety and Process Security. *Proceedings of the Sixth Annual Conference on,* 24-27 Jun 1991 1991. 10-16.

PUSCHNER, P. & NOSSAL, R. Testing the results of static worst-case execution-time analysis. *In:* ANON, ed., 1998 Madrid, Spain. IEEE, 134-143.

RAPIDITTY 2010. RapidiTTy® Builder development tools. Leicester, Leicestershire, LE1 7EA, United Kingdom.: TTE Systems Ltd.

ROSENBERG, J. 1997. Some misconceptions about lines of code. *International Software Metrics Symposium, Proceedings***,** 137-142.

SCHELER, F. & SCHROEDER-PREIKSCHAT, W. 2006. Time-Triggered vs. Event-Triggered: A matter of configuration? *Model-Based Testing, ITGA FA 6.2 Workshop on and GI/ITG Workshop on Non-Functional Properties of Embedded Systems, 2006 13th GI/ITG Conference -Measuring, Modelling and Evaluation of Computer and Communication (MMB Workshop)***,** 1-6.

SCHILD, K. & WÜRTZ, J. 2000. Scheduling of Time-Triggered Real-Time Systems. *Constraints,* 5**,** 335-357.

SCHNEIDER, J. Œ. 2003. *Combined schedulability and WCET analysis for real-time operating systems,* Aachen, Shaker Verlag GmbH.

SCHUTZ, W. 1993. *The testability of distributed real-time systems,* Boston, Mass. ; London, Kluwer Academic.

SHA, L., RAJKUMAR, R. & LEHOCZKY, J. P. 1990. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on Computers,* 39**,** 1175-1185.

SHA, L., RAJKUMAR, R., SON, S. H. & CHANG, C.-H. 1991. A real-time locking protocol. *IEEE Transactions on Computers,* 40**,** 793-800.

SHEN, V. Y., CONTE, S. D. & DUNSMORE, H. E. 1983. SOFTWARE SCIENCE REVISITED: A CRITICAL ANALYSIS OF THE THEORY AND ITS EMPIRICAL SUPPORT. *IEEE Transactions on Software Engineering,* SE-9**,** 155-165.

SHEPARD, T. & GAGNE, J. A. M. 1991. A pre-run-time scheduling algorithm for hard real-time systems. *IEEE Transactions on Software Engineering,* 17**,** 669-677.

SHEPARD, T. & GAGNE, M. A. 1990. Model of the F18 mission computer software for pre-run-time scheduling. *IEEE 10th International Conference on Distributed Computing Systems*, Paris, Fr. 62-69.

SHORT, M. 2011. On the implementation of dependable real-time systems with non-preemptive EDF. *In: Lecture Notes in Electrical Engineering Volume 90, 2011, pp 183-196 Springer* AO, S. L. & GELMAN, L. (eds.).

SHORT, M. 2012. Analysis and redesign of the 'TTC' and 'TTH' schedulers. *Journal of Systems Architecture,* 58**,** 38-47.

SHORT, M. & PONT, M. J. Hardware in the loop simulation of embedded automotive control systems. 2005 Vienna. 226-231.

SHORT, M., PONT, M. J. & FANG, J. 2008. Exploring the impact of task preemption on dependability in time-triggered embedded systems: A pilot study. Proceedings - *Euromicro Conference on Real-Time Systems*, Prague. 83-91.

SPURI, M. & BUTTAZZO, G. 1996. Scheduling aperiodic tasks in dynamic priority systems. *Real-Time Systems,* 10**,** 179-210.

STANKOVIC, J. A. & RAJKUMAR, R. 2004. Real-Time Operating Systems. *Real-Time Systems,* 28**,** 237-253.

SUCHA, P., KUTIL, M., SOJKA, M. & HANZALEK, Z. 2006. TORSCHE Scheduling toolbox for Matlab. Computer Aided Control System Design,*IEEE International Symposium on Intelligent Control*, 4-6 Oct. 2006 2006. 1181-1186.

THANE, H. & HANSSON, H. 2001. Testing distributed real-time systems. *Microprocessors and Microsystems,* 24**,** 463-478.

THANE, H., THANE, H. & HANSSON, H. 2000. Using deterministic replay for debugging of distributed real-time systems for debugging of distributed real-

time systems. *In:*12th Euromicro Conference on Euromicro RTS, 2000. 265-272.

TINDELL, K. W., BURNS, A. & WELLINGS, A. J. 1992. Allocating hard real-time tasks: An NP-Hard problem made easy. *Real-Time Systems,* 4**,** 145-165.

TINDELL, K. W., BURNS, A. & WELLINGS, A. J. 1994. An extendible approach for analyzing fixed priority hard real-time tasks. *Real-Time Systems,* 6**,** 133-151.

TORSCHE, T. 2007. TORSCHE Scheduling Toolbox for Matlab. Release 0.4.0 ed. Department of Control Engineering: Czech Technical University in Prague.

TRACEY, N., CLARK, J., MANDER, K. & MCDERMID, J. 2000. Automated test-data generation for exception conditions. *Software: Practice and Experience,* 30**,** 61-79.

TSAI, J. J. P. & BI, Y. 1991. Timing errors in real-time systems and their detection. International Proceedings Symposium on Software Reliability Engineering. 1991, 17-18 May 1991. 116-123.

TSAI, J. J. P., FANG, K.-Y. & BI, Y.-D. 1990a. On real-time software testing and debugging. Chicago, IL, USA. Publ by IEEE, 512-518.

TSAI, J. J. P., FANG, K.-Y. & CHEN, H.-Y. 1989. Knowledge-based debugger for real-time software systems based on a non-interference testing architecture. *In In Proc. 13th IEEE Intl. Computer Software.* Orlando, FL, USA. Publ by IEEE, 642-649.

TSAI, J. J. P., FANG, K.-Y., CHEN, H.-Y. & BI, Y.-D. 1990b. Noninterference monitoring and replay mechanism for real-time software testing and debugging. *IEEE Transactions on Software Engineering,* 16**,** 897-916.

WEGENER, J., STHAMER, H. & POHLHEIM, H. 1999. Testing the temporal behavior of real-time tasks using extended evolutionary algorithms. *Proceedings - Real-Time Systems Symposium*, Phoenix, AZ, USA. IEEE, 270-271.

WILHELM, R., ENGBLOM, J., ERMEDAHL, A., HOLSTI, N., THESING, S., WHALLEY, D., BERNAT, G., FERDINAND, C., HECKMANN, R., MITRA, T., MUELLER, F., PUAUT, I., PUSCHNER, P., STASCHULAT, J., STENSTR\, P. & \#246 2008. The worst-case execution-time problem\—overview of methods and survey of tools. *ACM Trans. Embed. Comput. Syst.,* 7**,** 1-53.

WILLIAMS, L. G. & SMITH, C. U. 1998. Performance evaluation of software architectures. *Proceedings of the 1st international workshop on Software and performance.* Santa Fe, New Mexico, USA: ACM.

WILLIAMS, L. G. & SMITH, C. U. 2002. PASASM: A method for the performance assessment of software architectures. *WOSP '02 Proceedings of the 3rd international workshop on Software and performance.* 179-189.

XU, J. 2003. Making software timing properties easier to inspect and verify. *IEEE Software,* 20**,** 34-41.

XU, J. A software architecture for complex real-time embedded systems. *Proceedings of International Conference on the 2nd Mechatronic and Embedded Systems and Applications,* IEEE/ASME 2007.

XU, J. & PARNAS, D. L. 1990. Scheduling processes with release times, deadlines, precedence, and exclusion relations. *IEEE Transactions on Software Engineering,* 16**,** 360-369.

XU, J. & PARNAS, D. L. 1993. On satisfying timing constraints in hard-real-time systems. *IEEE Transactions on Software Engineering,* 19**,** 70-86.

XU, J. & PARNAS, D. L. 2000. Priority scheduling versus pre-run-time scheduling. *Real-Time Systems,* 18**,** 7-23.

YAO, G., BUTTAZZO, G. & BERTOGNA, M. 2009. Bounding the maximum length of non-preemptive regions under fixed priority scheduling. *15th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*. Beijing. 351-360.

YAO, G., BUTTAZZO, G. & BERTOGNA, M. 2011. Feasibility analysis under fixed priority scheduling with limited preemptions. *Real-Time Systems,* 47**,** 198-223.

ZHANG, F., BURNS, A. & BARUAH, S. 2010. Sensitivity analysis for EDF scheduled arbitrary deadline real-time systems. *IEEE 16th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)* Macau. 61-70.

ZHANG, F. X. & BURNS, A. 2009. Improvement to quick processor-demand analysis for EDF-scheduled real-time systems. *IEEE 18th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA),* Dublin. 76-86.