Monitoring and designing predictable time-triggered software architecture for real-time embedded applications

Thesis submitted for the degree of

Doctor of Philosophy

at the University of Leicester

by

Kam L. Chan
B.Eng., M.Sc. (Sheffield)

Department of Engineering

University of Leicester

2012
Monitoring and designing predictable time-triggered software architecture for real-time embedded applications

Kam L. Chan

Abstract

Modern industrial applications often employ embedded processors – programmed with dedicated software – to perform some tasks in real time. In many such designs, the software running on the processor is developed according to rigorous industry standards. Even with such state-of-the-art designs, problems can occur "in the field" due to unforeseen circumstances – such as electromagnetic interference – that could undermine the underlying assumptions made at design time. It therefore remains essential to monitor the system at run-time, in order to detect any deviations from the required system behaviour.

However, monitoring embedded systems is a far from trivial process, not least because such systems are becoming increasingly complex. Also, variations in task execution times are likely to occur, and can have negative impacts on system predictability particularly in the presence of jitter-sensitive tasks. In addition, it is rarely possible (or cost-effective) to add precise monitoring capabilities to a system which has not been developed from the outset with such requirements in mind.

The work described in this thesis seeks to address these issues by introducing and evaluating a “predictable time-triggered” (pTT) framework that combines: a pTT scheduling algorithm and a hardware-based pTT monitor. A novel jitter-reduction technique in a pTT algorithm allows jitter-sensitive tasks to be executed in constant periods without the need to re-compute the entire task schedule. The studies reported in this thesis show that, with existing jitter-reduction methodologies, a pTT algorithm can provide extremely predictable temporal behaviour. The studies also show that the novel low-cost pTT monitor – that operates by monitoring fluctuations in the processor power consumption through a simple hardware interface – adds an additional level of safety by allowing run-time errors to be detected at a time resolution of microseconds. These findings provide sufficient evidence that the pTT framework could be an appropriate model for safety-critical system design.
Acknowledgements

Throughout the journey of my PhD study, there are numerous people to whom I am grateful for their help and support.

First and foremost, I would like to express my sincere thanks and appreciation to my supervisor, Professor Michael J. Pont, not only for his superlative guidance and unconditional support, but also for giving me many amazing opportunities that I doubt I would have otherwise.

My gratefulness also goes to Dr. Alistair McEwan and Dr. Michael Short who provided constructive comments and useful suggestions at my PhD transfer viva voce examination, Dr. Fernando Schlindwein who gave me insightful advice in the early stage of this study, as well as my examiners Professor Hani Hagras and Professor Tanya Vladimirova who provided invaluable feedback and recommendations to improve this thesis.

Of course, none of the research presented in this thesis would have been carried out without the financial support provided by the Innovative electronics Manufacturing Research Centre (IeMRC) as part of their studentship programme for which I am particularly grateful.

I would also like to thank all my past and present colleagues at the Embedded Systems Lab, in particular, Dr. Devaraj Ayavoo, Dr. Ricardo Bautista-Quintero, Dr. Ayman Gendy, Dr. Zemian Hughes, Dr. Terra Phatrapornnant and Ms. Huiyan Wang, for their support, encouragement and fruitful discussions.
Finally, I would like to thank my partner, Miss Yuenyee Wong, for her love and forbearance throughout the many years of my studies. I dedicate this thesis to her.
Table of Contents

ABSTRACT .......................................................................................................................... I
ACKNOWLEDGEMENTS ................................................................................................. II
TABLE OF CONTENTS ................................................................................................. IV
LIST OF FIGURES ........................................................................................................ VII
LIST OF TABLES ........................................................................................................... VIII
LIST OF PUBLICATIONS .............................................................................................. IX
LIST OF ABBREVIATIONS ........................................................................................... X
LIST OF SYMBOLS AND UNITS ................................................................................ XII

CHAPTER 1  INTRODUCTION ........................................................................................ 1
  1.1  MONITORING AND PREDICTABILITY OF EMBEDDED SYSTEMS ....................... 3
  1.2  RESEARCH AIMS AND SCOPE ........................................................................... 6
  1.3  THESIS STRUCTURE ......................................................................................... 7
  1.4  CONCLUSIONS .................................................................................................. 10

CHAPTER 2  REAL-TIME EMBEDDED DESIGN ............................................................. 11
  2.1  THE PRINCIPLES OF REAL TIME SYSTEMS ..................................................... 11
      2.1.1  Real time tasks ......................................................................................... 11
  2.2  TIMING ANALYSIS FOR REAL TIME TASKS ................................................... 13
  2.3  TASK JITTERS .................................................................................................. 15
      2.3.1  Release jitter ............................................................................................. 15
      2.3.2  Period jitter ............................................................................................... 17
      2.3.3  Execution jitter .......................................................................................... 18
  2.4  THE DEADLINE CONSTRAINT VIOLATIONS AND TASK OVERRUNS .............. 18
  2.5  THE PRINCIPLES OF TIME-TRIGGERED SCHEDULING ................................. 20
  2.6  TIME-TRIGGERED CO-OPERATIVE SCHEDULING ......................................... 21
  2.7  TIME-TRIGGERED HYBRID SCHEDULING ....................................................... 23
  2.8  TIME-TRIGGERED RATE-MONOTONIC SCHEDULING ...................................... 25
  2.9  DISCUSSION ..................................................................................................... 27
  2.10 CONCLUSION .................................................................................................... 29

CHAPTER 3  TECHNIQUES FOR PREDICTABILITY AND RELIABILITY .................... 31
  3.1  JITTER MINIMISATION METHODOLOGIES .................................................... 31
      3.1.1  The single-path programming paradigm ................................................... 32
      3.1.2  Sandwich delays ....................................................................................... 33
      3.1.3  The code balancing technique ................................................................. 34
      3.1.4  The planned pre-emption technique ......................................................... 36
  3.2  RUN-TIME MONITORING AND ERROR DETECTION ....................................... 37
      3.2.1  Software-based monitoring .................................................................... 39
      3.2.2  Hardware-based monitoring ................................................................... 42
      3.2.3  Hybrid monitoring .................................................................................. 44
  3.3  DISCUSSION ...................................................................................................... 47
      3.3.1  A remaining jitter problem ....................................................................... 48
      3.3.2  The strengths and weaknesses of the monitoring approaches ................. 50
  3.4  CONCLUSION .................................................................................................... 51

CHAPTER 4  NON-INVASIVE MONITORING ............................................................... 53
  4.1  INTRODUCTION ............................................................................................... 53
  4.2  PROCESSOR POWER PREDICTIONS ............................................................... 55
  4.3  TOWARDS A NOVEL HARDWARE-BASED MONITORING ............................... 57
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.4</td>
<td>THE NON-INVASIVE SAFETY AGENT</td>
<td>58</td>
</tr>
<tr>
<td>4.5</td>
<td>IMPLEMENTING THE NON-INVASIVE SAFETY AGENT</td>
<td>59</td>
</tr>
<tr>
<td>4.5.1</td>
<td>Hardware configuration</td>
<td>60</td>
</tr>
<tr>
<td>4.5.2</td>
<td>Software design</td>
<td>61</td>
</tr>
<tr>
<td>4.6</td>
<td>ASSESSING THE NON-INVASIVE SAFETY AGENT</td>
<td>63</td>
</tr>
<tr>
<td>4.6.1</td>
<td>Hardware configuration of the monitored system</td>
<td>63</td>
</tr>
<tr>
<td>4.6.2</td>
<td>Software configuration of the monitored system</td>
<td>64</td>
</tr>
<tr>
<td>4.6.3</td>
<td>Desktop monitor</td>
<td>64</td>
</tr>
<tr>
<td>4.6.4</td>
<td>The non-invasive safety agent module</td>
<td>65</td>
</tr>
<tr>
<td>4.6.5</td>
<td>Descriptions of the experiments</td>
<td>66</td>
</tr>
<tr>
<td>4.6.6</td>
<td>Experiment 1: fixed test cases</td>
<td>67</td>
</tr>
<tr>
<td>4.6.7</td>
<td>Experiment 2: random test cases</td>
<td>69</td>
</tr>
<tr>
<td>4.6.8</td>
<td>Case study: extended test duration and system power failure</td>
<td>71</td>
</tr>
<tr>
<td>4.7</td>
<td>DISCUSSION</td>
<td>73</td>
</tr>
<tr>
<td>4.8</td>
<td>CONCLUSION</td>
<td>74</td>
</tr>
<tr>
<td>5.1</td>
<td>INTRODUCTION</td>
<td>76</td>
</tr>
<tr>
<td>5.2</td>
<td>KEY CHALLENGES</td>
<td>78</td>
</tr>
<tr>
<td>5.3</td>
<td>IMPLEMENTING THE NON-INVASIVE SAFETY AGENT EXTENSION</td>
<td>79</td>
</tr>
<tr>
<td>5.4</td>
<td>CASE STUDY: A WIRELESS ELECTROCARDIOGRAPH DEVICE</td>
<td>81</td>
</tr>
<tr>
<td>5.4.1</td>
<td>Implementing the wireless electrocardiograph device</td>
<td>82</td>
</tr>
<tr>
<td>5.4.2</td>
<td>Software-based monitoring techniques</td>
<td>85</td>
</tr>
<tr>
<td>5.4.3</td>
<td>Evaluation and results</td>
<td>87</td>
</tr>
<tr>
<td>5.5</td>
<td>DISCUSSION</td>
<td>89</td>
</tr>
<tr>
<td>5.6</td>
<td>CONCLUSION</td>
<td>91</td>
</tr>
<tr>
<td>6.1</td>
<td>INTRODUCTION</td>
<td>92</td>
</tr>
<tr>
<td>6.2</td>
<td>A NOVEL JITTER-REDUCTION TECHNIQUE</td>
<td>93</td>
</tr>
<tr>
<td>6.3</td>
<td>THE PREDICTABLE TIME-TRIGGERED CO-OPERATIVE ALGORITHMS</td>
<td>95</td>
</tr>
<tr>
<td>6.4</td>
<td>THE PREDICTABLE TIME-TRIGGERED CO-OPERATIVE MONITOR</td>
<td>97</td>
</tr>
<tr>
<td>6.4.1</td>
<td>Detecting time-based errors</td>
<td>98</td>
</tr>
<tr>
<td>6.5</td>
<td>CASE STUDY AND EVALUATION</td>
<td>102</td>
</tr>
<tr>
<td>6.5.1</td>
<td>Hardware platform setup</td>
<td>103</td>
</tr>
<tr>
<td>6.5.2</td>
<td>Case Study: bubble sort</td>
<td>103</td>
</tr>
<tr>
<td>6.5.3</td>
<td>Evaluation of the predictable time-triggered monitor</td>
<td>109</td>
</tr>
<tr>
<td>6.6</td>
<td>DISCUSSION</td>
<td>114</td>
</tr>
<tr>
<td>6.7</td>
<td>CONCLUSIONS</td>
<td>115</td>
</tr>
<tr>
<td>7.1</td>
<td>INTRODUCTION</td>
<td>117</td>
</tr>
<tr>
<td>7.2</td>
<td>THE PREDICTABLE TIME-TRIGGERED RATE-MONOTONIC SCHEDULING</td>
<td>118</td>
</tr>
<tr>
<td>7.2.1</td>
<td>Handling jitter-sensitive tasks</td>
<td>118</td>
</tr>
<tr>
<td>7.2.2</td>
<td>Integration of the jitter-reduction techniques</td>
<td>119</td>
</tr>
<tr>
<td>7.3</td>
<td>THE PREDICTABLE TIME-TRIGGERED RATE-MONOTONIC MONITOR</td>
<td>121</td>
</tr>
<tr>
<td>7.3.1</td>
<td>Time-based error detection</td>
<td>124</td>
</tr>
<tr>
<td>7.4</td>
<td>CASE STUDY AND EVALUATION</td>
<td>126</td>
</tr>
<tr>
<td>7.4.1</td>
<td>Case study: a signal sampler</td>
<td>127</td>
</tr>
<tr>
<td>7.5</td>
<td>EVALUATION OF THE PREDICTABLE TIME-TRIGGERED RATE-MONOTONIC MONITOR</td>
<td>129</td>
</tr>
<tr>
<td>7.5.1</td>
<td>Overview of the pTRM system’s operations and timings</td>
<td>129</td>
</tr>
<tr>
<td>7.5.2</td>
<td>Simulation results</td>
<td>131</td>
</tr>
<tr>
<td>7.6</td>
<td>DISCUSSION</td>
<td>135</td>
</tr>
<tr>
<td>7.7</td>
<td>CONCLUSION</td>
<td>137</td>
</tr>
<tr>
<td>8.1</td>
<td>SUMMARY OF CONTRIBUTIONS</td>
<td>138</td>
</tr>
</tbody>
</table>

CHAPTER 8 CONCLUSIONS AND FUTURE WORK ......................................... 138
List of Figures

Figure 1.1: Safety-critical embedded systems .......................................................... 3
Figure 1.2: A conceptual model of the Predictable Time-Triggered Framework .......... 6
Figure 2.1: The operation of a periodic task ............................................................ 12
Figure 2.2: The operation of a sporadic task ........................................................... 13
Figure 2.3: A periodic task with deadlines ............................................................... 19
Figure 2.4: The model of time-triggered co-operative scheduling ............................. 22
Figure 2.5: The model of time-triggered hybrid scheduling .................................... 24
Figure 2.6: The model of time-triggered rate-monotonic scheduling ...................... 26
Figure 2.7: Overload situation in time-triggered co-operative scheduling ............... 28
Figure 3.1: Sandwich delays, .................................................................................... 33
Figure 3.2: Pseudo code of the code balancing technique, ...................................... 35
Figure 3.3: The planned preemption technique in TT rate-monotonic scheduling ...... 37
Figure 3.4: Software-based monitoring .................................................................... 39
Figure 3.5: Traditional Hardware-based monitor .................................................... 43
Figure 3.6: Hybrid monitoring ................................................................................. 45
Figure 3.7: Jitter in TT co-operative scheduling with the sandwich delay technique .... 49
Figure 3.8: Jitter in TT rate-monotonic scheduling with the sandwich delay technique . 50
Figure 4.1: The power measurement of a TT co-operative system ......................... 58
Figure 4.2: The methodology of the non-invasive safety agent ............................... 59
Figure 4.3: The connection diagram of the NISA module ........................................ 60
Figure 4.4: The program flowchart of the NISA module ......................................... 62
Figure 4.5: The connection diagram of the experimental setup ............................... 65
Figure 4.6: Tasks with an additional 0.8 ms time delay ......................................... 68
Figure 4.7: Tasks with an additional 0.9 ms time delay ......................................... 69
Figure 4.8: Task T3 executing with a 0.6 ms randomly added time delay .................. 70
Figure 4.9: Task T3 executing with a 0.8ms randomly added time delay ................. 71
Figure 5.1: The power measurement of a TTC-DVS system ................................... 79
Figure 5.2: The processed power measurement of a TTC-DVS system ................... 80
Figure 5.3: A flag-driven software-based monitoring technique .............................. 86
Figure 5.4: A timer-driven software-based monitoring technique ......................... 87
Figure 6.1: The proposed jitter-reduction technique ............................................. 95
Figure 6.2: The predictable time-triggered co-operative scheduling ...................... 96
Figure 6.3: The processor power signal of a predictable time-triggered co-operative system ................................................................. 98
Figure 6.4: The operations of the pTTC monitor .................................................... 99
Figure 6.5: An example for computing task violation points .................................... 102
Figure 6.6: A comparison of different jitter minimisation techniques ...................... 105
Figure 6.7: The power signals of different TT co-operative systems with a new task (Task 4) ................................................................. 108
Figure 6.8: The result for the task overrun detection ability of the pTTC monitor ...... 111
Figure 6.9: The result for the deadline violation detection ability of the pTTC monitor .... 113
Figure 7.1: The concept of predictable time-triggered rate-monotonic scheduling ...... 120
Figure 7.2: Operations of pTTRM monitor ($\tau_j$ is jitter-sensitive) ....................... 123
Figure 7.3: The start times of the sampling task .................................................... 128
Figure 7.4: Operational timings of the pTT rate-monotonic system ....................... 130
Figure 7.5: The simulation result of the pTT rate-monotonic monitor ..................... 133
Figure A.1: Test bed of the temperature experiment .............................................. 151
Figure A.2: Unstable temperature environments ................................................. 152
Figure A.3: Temperature experiment 1 (+9°C to +89°C) ....................................... 153
Figure A.4: Temperature experiment 2 (-14°C to +86°C) ..................................... 154

B.1: The front panel of the virtual instrument for task start time measurements .... 156
B.2: The block diagram of the virtual instrument for task start time measurements ... 157
B.3: The front panel of the virtual instrument for processor power measurements ... 158
B.4: The block diagram of the virtual instrument for processor power measurements ... 159
List of Tables

Table 2.1: A task set of time-triggered co-operative scheduling .............................................. 22
Table 2.2: A task set of time-triggered hybrid scheduling .......................................................... 24
Table 2.3: A task set of time-triggered rate-monotonic scheduling .......................................... 26
Table 3.1: Seven watchdog patterns .............................................................................................. 44
Table 3.2: A time-triggered co-operative task set that contains jitter ........................................... 49
Table 3.3: A time-triggered rate-montonic task set that contains jitter ......................................... 49
Table 4.1: A task set of the fixed time-delay experiment ................................................................. 67
Table 4.2: Task information of a test set in the second experiment ............................................... 69
Table 5.1: The CPU frequency and supply voltage mapping of the ECG device ............................ 83
Table 5.2: The task set of the wireless ECG device ....................................................................... 84
Table 5.3: Memory requirement for the wireless ECG device ......................................................... 88
Table 6.1: The task set of bubble sort – adapted from (Gendy and Pont, 2007) ............................... 104
Table 6.2: A jitter level comparison of different jitter-reduction techniques ................................. 107
Table 6.3: A comparison data of different jitter-reduction techniques ......................................... 107
Table 7.1: The task set of a signal sampler ..................................................................................... 127
Table 7.2: A comparison data of the TT and the pTT rate-monotonic schedulings ....................... 129
Table A.1: The task set for the case study ....................................................................................... 150
Table A.2: The temperature sensing task of the non-invasive safety agent .................................... 151
List of Publications

A number of papers which have been published and submitted during the course of the work described in this thesis are listed below in chronological order. Please note that the contents of some of these papers have been adapted for presentation in this thesis: where applicable, a footnote at the beginning of a chapter indicates that material from one or more papers has been included.

Patent:


Journal and conference publications:


# List of Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADC</td>
<td>Analogue to Digital Converter</td>
</tr>
<tr>
<td>AET</td>
<td>Actual Execution Time</td>
</tr>
<tr>
<td>BCET</td>
<td>Best-Case Execution Time</td>
</tr>
<tr>
<td>COTS</td>
<td>Commercial-Off-The-Shelf</td>
</tr>
<tr>
<td>CPU</td>
<td>Central Processing Unit</td>
</tr>
<tr>
<td>DAQ</td>
<td>Data Acquisition</td>
</tr>
<tr>
<td>DVS</td>
<td>Dynamic Voltage Scaling</td>
</tr>
<tr>
<td>ET</td>
<td>Event-Triggered</td>
</tr>
<tr>
<td>ISR</td>
<td>Interrupt Service Routine</td>
</tr>
<tr>
<td>JI</td>
<td>Jitter-Insensitive</td>
</tr>
<tr>
<td>JS</td>
<td>Jitter-Sensitive</td>
</tr>
<tr>
<td>MCU</td>
<td>Microcontroller</td>
</tr>
<tr>
<td>NISA</td>
<td>Non-Invasive Safety Agent</td>
</tr>
<tr>
<td>NISA-DVS</td>
<td>Non-Invasive Safety Agent for DVS system</td>
</tr>
<tr>
<td>PP</td>
<td>Planned Pre-emption</td>
</tr>
<tr>
<td>pTT</td>
<td>Predictable Time-Triggered</td>
</tr>
<tr>
<td>pTTC</td>
<td>Predictable Time-Triggered Co-operative</td>
</tr>
<tr>
<td>pTTRM</td>
<td>Predictable Time-Triggered Rate-Monotonic</td>
</tr>
<tr>
<td>RTC</td>
<td>Real Time Clock</td>
</tr>
<tr>
<td>SD</td>
<td>Sandwich Delay</td>
</tr>
<tr>
<td>TT</td>
<td>Time-Triggered</td>
</tr>
<tr>
<td>TTC</td>
<td>Time-Triggered Co-operative</td>
</tr>
<tr>
<td>TTH</td>
<td>Time-Triggered Hybrid</td>
</tr>
<tr>
<td>TTDM</td>
<td>Time-Triggered Deadline-Monotonic</td>
</tr>
<tr>
<td>Abbreviation</td>
<td>Description</td>
</tr>
<tr>
<td>--------------</td>
<td>-----------------------------------</td>
</tr>
<tr>
<td>TTRM</td>
<td>Time-Triggered Rate-Monotonic</td>
</tr>
<tr>
<td>WCET</td>
<td>Worst-Case Execution Time</td>
</tr>
<tr>
<td>WCRT</td>
<td>Worst-Case Response Time</td>
</tr>
</tbody>
</table>
List of Symbols and Units

Symbols

\( \Phi_i \) Set of jitter-sensitive tasks which are released at tick \( i \)

\( \Delta \) A short period of time (integer)

\( \gamma_i \) Set of jitter-sensitive tasks which are released at tick \( i \)

\( \Gamma \) Set of ready to run tasks

\( D_{\tau_i} \) Deadline of task \( \tau_i \)

\( DStart_{\tau_i} \) Deferred start time of a jitter-sensitive task \( \tau_i \)

\( I \) Tick interval

\( m \) A priority value of a task

\( n \) A priority value of a task

\( O_{\tau_i} \) Offset of task \( \tau_i \)

\( Overhead_f \) The scheduling overhead which is located before a task execution

\( Overhead_r \) The scheduling overhead which is located after a task execution

\( p \) A priority value of a task

\( P \) The next planned pre-emption point

\( PT \) Planned pre-emption point for a tick

\( R_{\tau_i} \) Remaining time of task \( \tau_i \) to reach the end of its sandwich delay

\( SD_{\tau_i} \) Sandwich Delay of task \( \tau_i \)

\( Start_{\tau_i} \) Start time of task \( \tau_i \)

\( T_{\tau_i} \) Period of task \( \tau_i \)

\( t_{s_{\tau_i}} \) Tick number where task \( \tau_i \) starts

\( \tau_i \) A task

\( VP_{\tau_i} \) Predicted Violation Point of task \( \tau_i \)
\( \psi_i \)  Set of jitter-sensitive tasks which are released at tick \( i \)

**Units**

<table>
<thead>
<tr>
<th>Unit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ms</td>
<td>Millisecond</td>
</tr>
<tr>
<td>( \mu s )</td>
<td>Microsecond</td>
</tr>
<tr>
<td>mV</td>
<td>Millivolt</td>
</tr>
<tr>
<td>KHz</td>
<td>Kilohertz</td>
</tr>
<tr>
<td>MHz</td>
<td>Megahertz</td>
</tr>
<tr>
<td>bps</td>
<td>bit per second</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

The growth of transistor density on integrated circuits continues to enable more advanced computer hardware to allow sophisticated software programs to operate. The use of computers today is not only for solving complex mathematical problems but also, and increasingly for controlling physical systems in various domains such as aerospace, automotive and medical instrumentation. Most often, modern physical systems are composed of interconnected elemental dynamic objects, such as sensors and actuators. These objects allow the control software to interact with the physical environment at run time. The control software, however, is required to collect and process data from sensors, and to generate control signals to actuators. All these tasks have to be done in time to prevent operational failures.

Designing software for precise timing applications can be challenging; since the execution time of a computational task varies. Thus, the upper bound estimates of the execution time for tasks are always used in the software designs. Based on these estimates, temporal properties of software can be examined at design time through the use of formal verification techniques (Felder and Pezzè, 2002; Cook and Koskinen et al., 2011). However, this cannot guarantee the absence of temporal errors in software at run time (Raju and Rajkumar et al., 1992; Goodloe and Pike, 2010); since it is practically difficult to accurately determine the upper time bounds for task executions (Wegener and Mueller, 2001; Puschner and
Burns, 2002a; Gustafsson, 2008; Wilhelm and Engblom et al., 2008; Harmon and Schoeberl et al., 2012), and hence they could be underestimated.

Further problems may occur in “hostile” environments (Jahanian and Goyal, 1990). For example, embedded systems may operate in an environment where electromagnetic interference (EMI) is present. EMI can be generated from various sources – such as electric motors, switches, etc. – and may interfere with computer hardware operations and corrupt data (Campbell, 1995; Campbell, 1998; Coulson, 1998; Ong and Pont et al., 2001; Ong and Pont, 2002). Consequently, embedded systems may experience transient errors or even a failure which can result in a “system crash” (i.e. cease functioning entirely).

Previous research (Kandasamy and Hayes et al., 2005; Yu and Ren et al., 2006; Goodloe and Pike, 2010) has suggested that faulty hardware may also cause software malfunctions, and can lead to a deadly consequence in safety-critical systems: for instance, the Turkish Airlines accident that involved a Boeing 737-800 flying from Istanbul to the Netherlands in 2009, was caused by a faulty altimeter that passed an erroneous altitude reading to the automatic throttle control system, resulting in reduction of engine thrust. The plane crashed, causing 8 deaths and 117 injuries (The Dutch Safety Board, 2010).

It is clear that the impact of failure in safety-critical embedded systems could be devastating. Monitoring such a system at run time is therefore remained essential, in order to detect any deviations from the required system behaviour – and apply appropriate mitigation / recovery strategies – before the system reaches a
dangerous operating state. Unfortunately, monitoring safety-critical embedded systems often presents real practical difficulties (Thane, 2000b; Zhu and Dwyer et al., 2009); since control software often operates on a small computer processor which is deeply embedded within the system it controls (Wolf, 2008). Figure 1.1 shows two typical examples of such systems: an anti-lock braking system and a brain pacemaker. Like other embedded systems, these two systems have limited interfaces to communicate with other devices. Moreover, the hardware interfaces are not designed for run-time monitoring. In addition, embedded software design focuses mostly on satisfying functional and non-functional requirements for systems, but how well an embedded system can be monitored at run-time is usually not considered.

![Image of anti-lock braking system and brain pacemaker](image)

**Figure 1.1: Safety-critical embedded systems**

*Images sources: Left (2CARPROS, 2011); Right (Brown University, 2011)*

### 1.1 Monitoring and predictability of embedded systems

Monitoring software executions in safety-critical embedded systems at run time is the subject of interest in this thesis. Run-time monitoring is an abnormality detection methodology. It observes an embedded system constantly, and often,
with intent to examine whether the behaviour of the system is in line with our expectations (Thane, 2000b; Maler and Nickovic et al., 2005; Goodloe and Pike, 2010).

Information about the monitored system is usually collected and analysed over time (Dodd and Ravishankar, 1992; Mansouri-Samani, 1995). The quality of the collected data is a dominant factor influencing the analytical results. Unbiased and deterministic data which truly reflect the actual behaviour of the monitored system could provide a pertinent and accurate result (Thane, 2000a; Obermaisser and Peti, 2001). Contrarily, vague and ambiguous data are prone to misinterpretation of the system’s behaviour, and could lead to an erroneous conclusion. It is important to emphasise that the monitoring process may interfere with the monitored system’s processes (Mansouri-Samani, 1995; Shobaki, 2004). This “probe effect” often has a negative impact on the quality of monitoring results (McDowell and Helmbold, 1989).

The quality of monitoring results can also be affected by the degree of knowledge that monitoring systems possess about the software system which is being monitored (Thane, 2000b). A monitoring system must able to characterize the behaviour of the monitored system. Preferably, every action at a given time of the monitored system can be determined. Hence, the temporal predictability of the monitored system’s behaviour has strong influence on the results of system monitoring. The temporal predictability of a system refers to the degree of precision with which it is possible to predict the satisfaction of the system’s
timing requirements (Stankovic and Ramamritham, 1990). Clearly, this property is important for real-time embedded software design (Schoebel, M. et al., 2011).

As embedded systems are becoming increasingly complex, designing predictable embedded software requires careful consideration when choosing software architecture for development (Gebhard, G. et al., 2011). Embedded systems are often designed using two common software architectures: event-triggered (ET) and time-triggered (TT) (Kopetz, 1991; Scarlett and Brennan, 2006; Scheler and Schröder-Preikschat, 2006). Briefly, in an ET architecture, the activation of tasks is stimulated by specific events either periodically, sporadically or aperiodically. The behaviour of an ET system is often difficult to predict. In a TT architecture, all events are related to the passage of time, in which tasks are scheduled to execute either periodically or at specific points in time.

It can be argued that TT architectures represent a “safe subset” of more general ET architectures. Indeed, a TT architecture which involves a collection of periodic tasks can have predictable timing behaviour (Baker and Shaw, 1989; Locke, 1992). However, this may not be the case in some situations. For example, variations in the execution time, start time and period of a task can introduce uncertainty in the system’s timing behaviour which makes it very difficult to be analysed and monitored. Also, improper task scheduling will induce unpredictable behaviour to a TT system and could result in a system failure. Further discussion for this subject will be presented in Section 3.3.1 of this thesis.
As ET architectures have unpredictable system behaviour, it will not be further explored in this thesis. Instead, our focus on monitoring and designing predictable embedded software will be based on TT architectures.

1.2 Research aims and scope

The research work described in this thesis seeks to provide safety-critical embedded system design with a framework that allows real-time tasks to be scheduled at an extremely low level of jitter, and allows time-based errors in embedded systems to be detected efficiently and accurately. These goals involve two objectives:

1. To design a monitoring technique which would allow rapid detection (preferably prediction) of timing errors in TT embedded systems
2. To provide a software solution to improve predictability of TT scheduling algorithms and facilitate run-time monitoring of task timing behaviour

Figure 1.2: A conceptual model of the Predictable Time-Triggered Framework
Figure 1.2 illustrates a conceptual model of the proposed framework – namely the Predictable Time-Triggered (pTT) Framework. The pTT framework combines two components: a pTT scheduling algorithm and a pTT monitor. A pTT scheduling algorithm is derived by applying jitter-reduction techniques to an existing time-triggered scheduling algorithm, in an attempt to improve its predictability. A matched pTT monitor aims to observe the run-time behaviour of the pTT scheduling algorithm, based on the temporal information of tasks given at design time and the task information extracted at run time from the power consumption of the processor where the pTT scheduling is running.

It is important to emphasise that the primary focus of a pTT monitor is on the detection of time-based errors in TT embedded systems which employ the tick-driven approach. Logical errors in embedded software and any non TT architecture are considered beyond the scope of this thesis.

1.3 Thesis structure

The remainder of this thesis is organised as follows:

Chapter 2 provides essential background material that is required to understand the research work described in this thesis. It begins with some basic principles of real-time systems, and then discusses the key issues of real-time systems. The focus of this chapter then shifts to time-triggered (TT) system design. It provides the concept of TT and reviews some TT scheduling algorithms including time-triggered co-operative (TTC) scheduling, time-triggered hybrid (TTH) scheduling and time-triggered rate-monotonic scheduling (TTRM). It also discusses the
advantages and disadvantages of these algorithms with respect to the key issues and temporal predictability of real-time systems.

Chapter 3 reviews some existing techniques in the areas of improving temporal predictability for TT embedded systems. These include the single-path programming paradigm (SPP), the sandwich delay technique (SD), the code-balancing technique (CB1) and the planned-preemption technique (PP). It also reviews previous work on the subject of run-time monitoring for real-time embedded systems. The areas of software-based, hardware-based and hybrid monitoring approaches with some existing techniques will be discussed.

Chapter 4 presents a novel monitoring methodology that can monitor the temporal behaviour of an embedded system based on the system’s task schedule and processor power consumption. This chapter begins with a brief revisit of the TTC scheduling discussed in Chapter 2. The related work in the area of processor power prediction is then briefly reviewed. After this, the proposed monitoring methodology is elucidated. We then present a non-invasive monitoring technique – the Non-Invasive Safety-Agent (NISA) technique – which utilises the proposed methodology to detect time-based errors in a TTC system. An evaluation and a case study with an extensive duration (7-day) for the NISA technique are explained. In the evaluation of the NISA technique, several hundreds of test cases with different noise levels were applied to the monitored system to assess the NISA technique. The selected results of the evaluation and the case study are reported and discussed.
Chapter 5 presents an extension of the NISA technique, namely the NISA-DVS technique. The NISA-DVS technique was designed specifically for monitoring a TTC system equipped with a static dynamic voltage scaling scheme (TTC-DVS). The NISA-DVS technique was implemented and assessed utilising a case study where a 3-channel wireless electrocardiogram (ECG) device – that is scheduled by a TTC-DVS scheduler – was employed as the monitored system. Like the evaluation of the NISA technique, time-based errors and power faults are injected to the ECG device. The evaluation results are reported at the end of this chapter.

Chapter 6 presents the proposed predictable time-triggered co-operative (pTTC) framework. A novel jitter-reduction technique and the concept of the pTTC scheduling are explained. After this, a detailed description of the pTTC monitor is provided. It then presents a case study and the evaluation of the pTTC scheduling and the pTTC monitor. The results of the case study and the evaluation are reported and discussed.

Chapter 7 presents a study of the possibility to extend the predictable time-triggered (pTT) framework to a time-triggered pre-emptive scheduling algorithm. It begins with a detailed description of an extension of the proposed jitter-reduction technique, and then presents the predictable time-triggered rate-monotonic (pTTRM) scheduling. After this, the pTTRM monitor is discussed, followed by a case study and the evaluation of the pTTRM scheduling and the pTTRM monitor. The results of the case study and evaluation are discussed.
A conclusion of this study is drawn in Chapter 8, where a summary of the contributions of this research work and possible future work are presented.

1.4 Conclusions

This chapter has discussed the background and motivation of this research study. It has emphasised the importance of temporal predictability of an embedded system for system monitoring, and highlighted some issues that could lead embedded systems to exhibit unpredictable behaviour.

The conceptual model of a predictable time-triggered (pTT) framework has also been introduced. The ultimate goal of this study is to realise this conceptual framework to provide extremely temporal predictable behaviour for safety-critical embedded system design, and to allow software run-time errors to be detected rapidly.
Chapter 2
Real-time embedded design

This chapter serves as a preamble to the core of this thesis which provides a description of some basic principles pertaining to real-time embedded system design. It also provides an introduction to time-triggered (TT) scheduling and reviews some fixed-priority TT scheduling algorithms. These include time-triggered co-operative (TTC), time-triggered hybrid (TTH) and time-triggered rate-monotonic (TTRM) algorithms.

2.1 The principles of real time systems

A system is said to be real-time if its correctness depends not only on the logical results of computation, but also on the time at which the results are produced (Laplante, 1996; Stankovic, 1996). Failure to produce the results in time may have varying degrees of consequences depending upon the characteristics of the system involved: for a hard real-time system, the consequences can be catastrophic (Kopetz, 1997; Buttazzo, 2005), but for a soft real-time system, it will not cause any serious damage. Nonetheless, real-time systems must be timely in order to prevent any negative impact (Gebhard, G. et al. 2011).

2.1.1 Real time tasks

Real-time systems are often controlled by software operating on embedded computer processors. Such real-time software is usually composed with a collection of tasks, each of which is a sequence of computer instructions that are executed by the embedded processor (Buttazzo, 2005). However, the start of a
task execution is often delayed after the release of the task, i.e. \( \text{task release time} \leq \text{task start time} \). Also, the first release of a task is not necessarily at the same time as the system starts. The first release of a task is called the “offset” of the task. It is important to note that, throughout this thesis, the release time of a task is always considered the same as the start of a system tick where it is released. When more than one task is released at the same tick, the task with the lowest priority value among the released tasks has the highest priority to execute.

A real-time task can be released at a regular time interval repeatedly. Such a task is called a “periodic task”, and the time interval between two successive releases is the “task period”. The operation of a periodic task is illustrated in Figure 2.1, where the offset of Task A equals to 2 with a task period equals to 1. This means that Task A will be first released at Tick 2, and executes once in every tick after it is released. Note that a task may also be executed on a “one-shot” basis (Pont, 2001).

![Figure 2.1: The operation of a periodic task](image-url)
A real time task may be released at variable time intervals repeatedly, but has a minimum time interval between two successive releases. Such a task is called a sporadic task (Spuri and Buttazzo, 1996). The operation of a sporadic task is illustrated in Figure 2.2, where the offset of Task A equals to 0 and its minimum arrival time equals to 1.

![Figure 2.2: The operation of a sporadic task](image)

An aperiodic task is a real-time task which is activated by a specific event which can happen at any time. The arrival of an aperiodic task may interfere with other tasks in the systems. Aperiodic tasks usually have soft deadlines or no deadlines (Spuri and Buttazzo, 1996).

### 2.2 Timing analysis for real time tasks

As real-time systems must be timely, task executions are often governed by stringent temporal constraints in accordance with the system specifications. However, this requires good knowledge of task execution times: the lower bound and the upper bound of task execution times correspond to the duration to execute the shortest and the longest execution paths. They are respectively called the best-
case execution time (BCET) and the worst-case execution time (WCET) of the
task (Gustafsson, 2008; Wilhelm and Engblom et al., 2008).

The WCET estimation of a task is considerably more important for the task to satisfy the deadline constraint (Wegener and Mueller, 2001; Gergeleit and Nett, 2002; Puschner and Burns, 2002a; Santos, 2008). However, precisely determining the WCET of a task is practically difficult. This is not only due to the non-determinism in microprocessors’ hardware temporal behaviour (for example, unknown number of cache misses and pipeline hazards), but also due to the difficulty of determining the longest execution path from a task which contains a large number of possible execution paths (Puschner, 2002; Puschner, 2003).

Timing analysis methodologies are already available for the WCET estimation of a task: direct measurement methods and measurement-based methods obtain temporal information of tasks from either the actual hardware of the monitored system or a simulator (Wilhelm and Engblom et al., 2008). Although these methods are commonly used in industry, the acquisition of the WCET of a task is not always guaranteed.

Alternatively, static timing analysis may be able to guarantee a safe approximation of the WCET of a task (i.e. equal to or greater than the true WCET of a task) (Gustafsson, 2008; Wilhelm and Engblom et al., 2008). Unlike measurement-based methods, this method does not require the actual hardware measurements of the monitored system, but requires the code of tasks and the
abstract model of the monitored system’s processor for the WCET estimation (Wegener and Mueller, 2001; Wilhelm and Engblom et al., 2008). If the model is accurate, the WCET of a task is guaranteed not to exceed the safe upper time bound. However, creating an accurate model for a processor is difficult and laborious (Gustafsson, 2008; Wilhelm and Engblom et al., 2008). Gernot G., et al. focused on software predictability of embedded systems and attempted to identify existing software coding rules to improve software predictability with respect to static timing analysis (Gernot G., et al. 2011).

Overall, adding a safe margin to the actual WCET estimation of a task is necessary to ensure the task not to exceed its WCET estimation at run time. However, processor time will be wasted if the safe margin is set to be too large, or vice versa. The trade-off between safety and resources will depend on the need of the application under development. Nevertheless, care must be taken when selecting a safe margin for the WCET estimation of a task.

2.3 Task jitters

Jitter can be defined as “the deviation of a timing event from its ideal occurrence in time” (Ou and Farahmand et al., 2004). In real-time systems, there are different types of jitters which are discussed in this section.

2.3.1 Release jitter

In Section 2.1.1, it was mentioned that a task’s release time is rarely the same as its start time. It is very often that the start time of a task differs from one instance to another. Such a variation in the start time of a task is called release jitter and
the **absolute release jitter** (ARJ) is defined in equation (2.1) as “the maximum deviation among all task instances” (Buttazzo, 2005).

$$ARJ(\tau_i) \equiv \max(Start_{\tau_{ik}} - Release_{\tau_{ik}}) - \min(Start_{\tau_{ik}} - Release_{\tau_{ik}}), \forall i, k \in \mathbb{N}_0$$  \hspace{1cm} (2.1)

where $Start_{\tau_{ik}}$ is the start time of task $\tau_i$ at the $k^{th}$ tick, $Release_{\tau_{ik}}$ is the release time of task $\tau_i$ at the $k^{th}$ tick and $\mathbb{N}_0$ is the set of non-negative integers.

Task release jitter can be induced by different factors, such as improper task synchronization (i.e. a task is blocked by another task occasionally) and variations in the execution times of previous tasks (Locke, 1992). It can cause a significant negative impact on system predictability and correctness. For example, when more than two periodic sampling tasks with different rates are released at the same time, the data sampled by some of these tasks may contain errors, and are likely to be meaningless if the release jitter level is higher than 10% (Cottet and David, 1999).

In this particular example, the sampling tasks are clearly **jitter-sensitive** which will fail to operate correctly even if they can complete their execution in time. Hence, such a jitter-sensitive task needs to start at the same point in time every invocation. However, there are some **jitter-insensitive tasks** in real time systems. The start time of a jitter-insensitive task is not a concern as long as it can complete before the deadline. For the release jitter problem related to jitter-sensitive tasks, we will further discuss in Section 3.3.1.
2.3.2 Period jitter

As tasks are assumed to be released at the beginning of each tick in this thesis, task periods will remain constant if the tick interval is fixed. However, the actual task period varies from one to another in the presence of task release jitter or tick interval jitter. The actual task period is defined as “the distance between the start time of two consecutive task instances”, and the absolute period jitter (APJ) is defined in equation (2.2) as “the maximum deviation between the ideal task period and all actual task periods” which coincides with the definition of input jitter defined in (Gendy, 2009).

\[ APJ(\tau_i) \equiv \max (T_{\tau_i}^{\text{max}} - T_{\tau_i}, T_{\tau_i} - T_{\tau_i}^{\text{min}}), \forall i, k \in \mathbb{N}_0 \] \hspace{1cm} (2.2)

\[ T_{\tau_i}^{\text{max}} \equiv \max (Start_{\tau_i,k+1} - Start_{\tau_i,k}), \forall i, k \in \mathbb{N}_0 \] \hspace{1cm} (2.3)

\[ T_{\tau_i}^{\text{min}} \equiv \min (Start_{\tau_i,k+1} - Start_{\tau_i,k}), \forall i, k \in \mathbb{N}_0 \] \hspace{1cm} (2.4)

where \( T_{\tau_i}^{\text{max}} \) and \( T_{\tau_i}^{\text{min}} \) are the maximum and the minimum of the actual task periods of task \( \tau_i \), \( T_{\tau_i} \) is the ideal period of task \( \tau_i \), \( k \) is the tick number and \( \mathbb{N}_0 \) is the set of non-negative integers.

Note that, the period jitter of a task should have the same level as the release jitter of the task under a normal situation. However, these two jitter levels may differ to each other if jitter is present in the tick interval.
2.3.3 Execution jitter

As discussed in Section 2.2, the execution time of a task is difficult to predict; since it may vary from one instance to another. This is due to non-deterministic processor hardware performance such as caching, pipelining and accessing memory (Edgar, 2002; Puschner and Burns, 2002a; Schoeberl, M. et al., 2011), and changes of execution paths in software programs in response to the inputs at run time (Puschner and Burns, 2002a; Wilhelm and Engblom et al., 2008). The variation in the execution time of a task is called execution jitter, and the absolute execution jitter (AEJ) is defined in equation (2.5) as “the maximum deviation of the execution interval among all instances” (Buttazzo, 2005).

\[ AEJ \equiv \max(End_{\tau_i,k} - Start_{\tau_i,k}) - \min(End_{\tau_i,k} - Start_{\tau_i,k}) \]  

(2.5)

where \( End_{\tau_i,k} \) is the end time of task \( \tau_i \) at the \( k^{th} \) tick.

From the outset of this thesis, temporal predictability has been emphasised as a crucial factor for real-time systems to satisfy their temporal requirements. In the presence of execution jitter, temporal predictability may be difficult to achieve. The execution jitter problem will be further explored in Chapter 3.

2.4 The deadline constraint violations and task overruns

In real-time system design, temporal constraints are always required for ensuring tasks to execute timely. As discussed in previous sections, a real-time task must complete its execution before the deadline. This section will formally define a
The deadline constraints of tasks are considered to be the most important constraints in real-time applications. As Figure 2.3 shown, an **absolute deadline** of a task is measured from the start of the system (i.e. system time = 0), and a **relative deadline** of a task is measured from the start of the task period. It is important to note that the term “deadline” is used to refer to task relative deadlines throughout this thesis.

A **deadline violation** occurs when a task execution exceeds its deadline, as shown in Figure 2.3 where Task A violates the deadline at tick 2. While the deadline of a task is not necessarily the same as the task’s WCET, task may overrun its WCET but may not violate its deadline. This situation is also depicted in Figure 2.3. Such a **task overrun** is also considered as a temporal error. As discussed,
An underestimate of task WCETs or external disturbances can result in both deadline violations and task overruns.

### 2.5 The principles of time-triggered scheduling

There are numerous ways in which we can describe (and distinguish) different software architectures which are employed in embedded computer systems. The architecture which forms the focus of this thesis is usually described as “time triggered” (TT). When saying that a computer system has a time-triggered architecture we mean that we can determine in advance – before the system begins executing – exactly what it will do at every moment of time in which it is running.

Completely defined TT behaviour is – certainly – difficult to achieve in practice. Even so, existing TT algorithms have attempted to achieve this goal by employing a tick-driven approach. Tick-driven scheduling employs a timer to generate an interrupt signal periodically to divide the timeline into infinite equal time slices (Tindell and Burns et al., 1994; Audsley and Bate et al., 1996; Pont, 2002). This ensures that the scheduler takes control of the processor to perform task scheduling in every fixed interval (or every tick). Hence, tasks are always released at the beginning of a tick. The granularity of the tick interval is derived from the system requirements.

A task $\tau_i$ in TT scheduling is usually characterised by a 4-tuple $(T_{\tau_i}, O_{\tau_i}, D_{\tau_i}, p_{\tau_i})$ where $T_{\tau_i}$ is the period, and $O_{\tau_i}$ is the offset, and $D_{\tau_i}$ is the deadline and $p_{\tau_i}$ is the execution priority. A TT algorithm will schedule tasks to execute in accordance
with this information. In tick-driven TT approach which is the subject of interest in this thesis, the execution rates of tasks are required to be a multiple of the (tick) timer interrupt rate. The following sections will introduce some fixed-priority TT scheduling algorithms.

### 2.6 Time-triggered co-operative scheduling

Time-triggered co-operative (TTC) scheduling which has been described as a “cyclic executive” (Baker and Shaw, 1989; Locke, 1992), is considered as the closest approximation of a “perfect” TT algorithm in the TT family. It is a “pre-run-time” or an “offline” scheduling algorithm (Xu and Parnas, 1993; Xu and Parnas, 2000) which means that all scheduling decisions are made at design time. Hence, the task schedule of a TTC system is known *a priori*.

It involves a collection of periodic tasks which operate co-operatively (or “non-pre-emptively”). The deadline for a task in TTC scheduling is assumed to be located at the end of the tick (i.e. before the next tick timer interrupt) where the task is released; and the task priority is implicitly expressed by the task index value – the lower the index value, the higher the execution priority – in the task array which is used for storing task information.

The tick timer in TTC scheduling is designed to enable an interrupt at the rate equal to the lowest common multiple of all the execution rates of the tasks in the system. Task pre-emption is not allowed, and hence tasks will run to completion once they have started (Liu and Layland, 1973; Kalinsky, 2001; Pont, 2001; Pont, 2002). In a common scenario, the scheduler will set the processor into an idle
state to wait for the next timer interrupt when no task is executing (or pending for execution) in order to conserve energy. This algorithm is also known as a “multi-rate executive” for periodic tasks (Kalinsky, 2001). Figure 2.4 depicts the above-described TTC algorithm with the task set shown in Table 2.1. Since Task $\tau_0$ has the lowest index value, it has the highest priority to execute. Thus, it will execute before Task $\tau_1$ and Task $\tau_2$ when they are released simultaneously. Task $\tau_1$ and Task $\tau_2$ will execute after the completion of Task $\tau_0$. As the tasks execute co-operatively, race conditions will not occur even if tasks share the same resources.

<table>
<thead>
<tr>
<th>Task</th>
<th>Offset (tick)</th>
<th>Period (tick)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\tau_0$</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>$\tau_1$</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>$\tau_2$</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

Table 2.1: A task set of time-triggered co-operative scheduling

As embedded applications often have aperiodic (or sporadic) events generated from external devices, TTC scheduling handles such events by using devoted tasks to periodically check if the corresponding events occur or not (Baker and...
Shaw, 1989; Locke, 1992; Kalinsky, 2001). However, such a polling mechanism is considered a waste of resources. Also, it is expected that the responsiveness of TTC scheduling is not as quick as other TT pre-emptive approaches (Allworth and Zobel, 1981); since a task to service an external device may be blocked by other co-operative tasks.

In addition, TTC scheduling is not designed to schedule a long duration task (for instance, task execution time > tick interval). Although, it is possible to schedule a long duration task by re-allocating the task’s deadline to the tick where it is expected to complete, it will further reduce the responsiveness of the system if other tasks are present in the subsequent ticks. This will also introduce jitter in the scheduler tick – further discussion for this issue will be presented in Section 3.1.4.

### 2.7 Time-triggered hybrid scheduling

TTC scheduling can indeed provide a highly predictable behaviour to embedded systems even in the presence of aperiodic events, if all design assumptions – such as task WCETs are precisely estimated and correct processor speed is chosen – are not violated (Stewart and Khosla, 1997). However, when embedded applications require better system responsiveness than TTC scheduling, the TTH scheduling can be employed.

TTH scheduling supports a set of co-operative tasks with a single, short duration pre-emptive task which has the highest execution priority (Pont, 2001; Maaita and Pont, 2005). It is sometimes described as a “multi-rate executive with interrupts”
(Kalinsky, 2001) in which an aperiodic event can be serviced by the preemptive task if the event has occurred in the form of interrupt at the previous tick. It also can be used for scheduling a critical, jitter-sensitive periodic task in the presence of a long duration task. For example, a data analyzer may have to perform signal processing calculations (i.e. a long task) such as the Fast Fourier Transform (FFT). At the same time, the analyzer may need to take a sample every tick (i.e. a jitter-sensitive short task).

<table>
<thead>
<tr>
<th>Task</th>
<th>Offset (tick)</th>
<th>Period (tick)</th>
<th>Deadline (tick)</th>
<th>Preemptive</th>
</tr>
</thead>
<tbody>
<tr>
<td>τ₀</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>Y</td>
</tr>
<tr>
<td>τ₁</td>
<td>0</td>
<td>3</td>
<td>1</td>
<td>N</td>
</tr>
<tr>
<td>τ₂</td>
<td>0</td>
<td>3</td>
<td>1</td>
<td>N</td>
</tr>
</tbody>
</table>

**Table 2.2: A task set of time-triggered hybrid scheduling**

![Diagram](image)

**Figure 2.5: The model of time-triggered hybrid scheduling**

Figure 2.5 shows the TTH algorithm to schedule the task set in Table 2.2. In this particular task set, TTC scheduling would fail to schedule Task τ₀ to meet the deadline, even if it has highest execution priority; since Task τ₂ will run to completion and block the execution of Task τ₀. In TTH scheduling, this problem
is addressed. As we can see in Figure 2.5, Task $\tau_0$ preempts Task $\tau_2$ in every invocation. However, a locking mechanism (such as semaphore) is required to prevent simultaneous access to a shared resource by tasks.

### 2.8 Time-triggered rate-monotonic scheduling

TTH scheduling is a useful single pre-emptive task solution for less complex systems. When the complexity of an embedded system grows, multiple pre-emptive tasks are likely to be required.

Fully pre-emptive approach is often considered as a viable solution to handle sporadic (or aperiodic) events (Bate, 1998; Kalinsky, 2001). The responsiveness of this approach is achieved by using a priority scheme to control task executions in which a task is permitted to pre-empt the execution of any lower-priority task in the system (Laplante, 1996). As tasks scheduled by a fully pre-emptive scheduler can be switched from one to another at any point in time, it is difficult to predict the system behaviour (Pont, 2001). This makes such an event-triggered approach much less popular than its time-triggered pre-emptive counterpart in hard real time system design.

For example, the rate-monotonic (RM) scheduling algorithm (Liu and Layland, 1973) is a time-triggered, pre-emptive algorithm in which task priorities are fixed and inversely proportional to their period. Liu and Layland demonstrated that – when using this algorithm – every task will meet its deadline if the total CPU utilization is $\leq 69\%$; all tasks are independent of each other; the deadline of each
task is equal to its period; the WCET of all tasks is known; and context switching time is ignored (Liu and Layland, 1973; Bate, 1998).

Figure 2.6 illustrates the operation of TTRM scheduling for the task set in Table 2.3. Since Task $\tau_0$ has the shortest periodic time, it has the highest priority to execute and preempt the execution of Task $\tau_2$. The execution of Task $\tau_2$ is also preempted by Task $\tau_1$ at tick 2, as Task $\tau_1$ has a shorter periodic time than Task $\tau_2$. Like TTH scheduling, a locking mechanism is required in TTRM scheduling, unless tasks are independent and have no shared resources.

<table>
<thead>
<tr>
<th>Task</th>
<th>Offset (tick)</th>
<th>Period = Deadline (tick)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\tau_0$</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>$\tau_1$</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>$\tau_2$</td>
<td>0</td>
<td>3</td>
</tr>
</tbody>
</table>

Table 2.3: A task set of time-triggered rate-monotonic scheduling

It should be emphasised that the TTRM system shown in Figure 2.6 which was used in the work described in this thesis, was implemented using the tick-driven approach which differs from other implementations such as scheduling based on a
real-time clock (RTC). Although scheduling using the tick-driven approach is less responsiveness than the RTC approach, the tick-driven approach is more flexible than the RTC approach in term of reusability (Audsley and Bate et al., 1996).

Nonetheless, RM scheduling, regardless of the implementation, will have more scheduling overhead than TTC scheduling; since additional scheduling time is needed due to the increase of complexity associated with context switches (Locke, 1992; Phatrapornnant, 2007).

2.9 Discussion

The beauty of TT scheduling is its predictable behaviour in which tasks are (periodically) released at particular points in time. While we take the view that comparatively simple TT architectures (such as TTC) are often a practical solution with modern processors (Pont, 2001), we do not pretend that they are universally applicable. Where a TTC approach is employed, there is one key failure mode which must be borne in mind: the task overrun. This may create a “domino effect” which makes the subsequent tasks miss the deadline (Buttazzo, 2005; Hughes and Pont, 2008). The entire schedule will be disrupted as a result of such an overload condition (Buttazzo, 2002). This situation can be a result of an underestimate of a task’s WCET or external disturbances.

Figure 2.7 illustrates this overload situation in a TTC system, where the execution of Task $\tau_0$ exceeds its WCET at tick 1. This leads to a violation of the deadline of
Task $\tau_2$. A domino effect starts to take place and results in an overload situation where the subsequent tasks are likely to miss the deadlines.

![Overload situation in time-triggered co-operative scheduling](image)

**Figure 2.7: Overload situation in time-triggered co-operative scheduling**

Task overruns and deadline violations can put any TT embedded system into a dangerous situation. Thus, such timing errors must be detected as early as possible, in order to apply appropriate mitigation / recovery strategies. However, detecting timing errors can be challenging, as briefly discussed in Section 1.1, particularly in the presence of task jitter. This problem is getting more challenging in TTRM scheduling when long duration tasks are present; since there is uncertainty in the task schedule: tasks may release at a tick and finish at any subsequent tick depending on previous task executions and the number of task preemptions during the execution of the task. When many tasks exhibit jitter, it is extremely challenging to determine the system behaviour at a given point in time.

As Maaita stated in his thesis: “*In real-time systems, predictability is inversely proportional to the levels of task jitter experienced in the system*” (Maaita, 2008).

Hence, real-time systems tend to violate the timing requirements in the presence of high jitter level. This has been discussed as an example in Section 2.3.1 where,
in the presence of jitter, TT systems could fail to operate correctly even if all deadline constraints are satisfied.

2.10 Conclusion

Real-time system design is a specialist area of software engineering which involves many concepts and theories. The materials covered in this chapter are only relevant to the work described in this thesis.

Among the materials which have been presented in this chapter, we have reviewed three different TT scheduling algorithms. However, TTH was not considered in the work described in thesis; since there is no difference between TTH scheduling and TTRM scheduling with one pre-emptive task.

We have also repeatedly emphasised the impact of task jitter on system predictability, and discussed the difficulty in estimating task WCETs. It is clear that when task jitter level is high, task WCETs are underestimated or electromagnetic interference is encountered, the behaviour of TT systems becomes difficult to predict. This is in apparent contradiction to the original idea of TT which has been stated in Section 2.5: “… a computer system has a time-triggered architecture we mean that we can determine in advance – before the system begins executing – exactly what it will do at every moment of time in which it is running”. This motivates the research study described in this thesis.

In Chapter 3, we will review some existing jitter-minimisation methodologies to improve system predictability. We also look three conventional monitoring
approaches and some existing time-based error detection techniques that could help to improve system reliability and add more confidence to real-time embedded systems.
Chapter 3

Techniques for predictability and reliability

In Chapter 2, we discussed the task jitter problem which will cause significant negative impact on the temporal predictability of TT embedded systems. This will likely decrease the system reliability and may result in run-time errors. If such errors remain undetected, actions cannot be taken; they could further develop into a critical failure.

This chapter attempts to address the jitter problem by reviewing various jitter-minimisation methodologies, including the “single-path programming paradigm” (SPP), “sandwich delays” (SD), the “code-balancing technique” (CB1) and “the planned preemption technique” (PP). We also review previous work on the subject of run time monitoring for real-time embedded systems. The areas of software-based, hardware-based and hybrid monitoring approaches with some existing techniques are included.

3.1 Jitter minimisation methodologies

Maintaining temporal predictability of real-time systems is important (Schoeberl et al., 2011), in particular when task jitter is high. Jitter minimisation methodologies are useful to mitigate this adverse effect. This will improve temporal predictability and result in better reliability for TT embedded systems.
3.1.1 The single-path programming paradigm

The major challenge in WCET analysis is that many execution paths may exist within a task. The task executes differently in accordance with different input data. This makes it difficult to estimate the WCET of the task.

The single-path programming paradigm (SPP) proposed by Puschner aims to transform a task with multiple execution paths into a task with a single execution path. This can be achieved by replacing input-data dependencies in the control flow using predicated code instead of branch code (Puschner, 2002). In predicated execution, instructions are associated with predicates, and are executed if the predicate is true; otherwise a “NOP” (or no operation) instruction is executed (Puschner and Burns, 2002a; Puschner, 2002; Puschner, 2003). Schoeberl et al. explored a temporal predictable chip-multiprocessor system based on the SPP technique (Schoeberl et al., 2009). Their proposed system was tested with 3 processor cores on a field-programmable gate array (FPGA) board and proven to be capable to deliver repeatable and predictable timing behaviour. However, some instructions (such as conditional move) used in the sequential code require specific processors (such as a Motorola M-Core processor) to support the instructions. Also, the execution time of a single-path task converted from a multiple-path task will be much longer than the multiple-path task. Hence, the power consumption of the SPP will likely increase. Also, Gebhard et al. argued that the SPP will produce complex software and impair the worst-case behaviour (Gebhard et al., 2011); since some instructions in a single-path program will always be fetched but not be executed.
3.1.2 Sandwich delays

A sandwich delay (Pont and Kurian et al., 2009) is a jitter-reduction technique to minimize task release jitter. It involves the use of a hardware timer to ensure that variations in a task execution time will not affect the start of next tasks. This enables tasks to start at the same point in time every period. A sandwich delay (SD) is defined in equation (3.1).

\[ SD_{\tau_i} = WCET_{\tau_i} + \text{Safe Margin} \]  

(3.1)

Figure 3.1 illustrates the SD technique, where the start of Task \( \tau_2 \) is postponed due to the SD of Task \( \tau_1 \). This allows Task \( \tau_2 \) to execute in a constant period of time. A SD for a task can be implemented by starting a timer at the start of the task, and waiting for the timer to reach a pre-determined time value after the completion of the task. Ideally, the processor should be set to idle while waiting for the timer to expire, to conserve energy consumption. Alternatively, a sandwich delay can be achieved by using the tick timer’s match function (if it is supported) to trigger task executions at specific points in time within a scheduler tick. This will significantly reduce the run-time overhead caused by the timer operations in practice. It is worthwhile to note that both of the approaches are
usually implemented using the timer interrupt service, if it is provide by the processor.

Phatrapornnant and Pont have demonstrated the effectiveness of this simple jitter minimisation technique (implemented using a hardware timer) to reduce jitter levels in TTC embedded systems which employ a static dynamic voltage scaling scheme (DVS) (Phatrapornnant and Pont, 2006). However, improper use of the SD technique (for example: underestimate of the WCET of tasks) will make the system unreliable when a timer interrupt is involved; since the timer interrupt (for a SD) to wake up the processor will occur before the processor is set into the idle state. Subsequent tasks at the same tick have to wait until the arrival of the next timer interrupt.

### 3.1.3 The code balancing technique

The code balancing technique CB1 (Gendy, 2009) is based on the idea of sandwich delay to insert sufficient idle time delay to minimize (task execution) jitter caused by, for example, irregular loops or conditional statements within a task. Figure 3.2 shows the pseudo code of the CB1 algorithm for balancing a “for-loop”. It assumes that the maximum number of iterations of the loop is known *a priori*. Before entering to the loop, a hardware timer will start. The number of iterations of the loop is observed while it is executing. As soon as the loop stops, the timer will stop. By using equation (3.1), it evaluates the difference between the time requires for maximum iterations and the time taken for the number of iterations which has been performed. This value plus a safe margin will be used to set up an interrupt before setting the processor into the idle state.
where MAX is the maximum number of iterations, \(x\) is a loop control variable, \(Time()\) is a time function which returns the time taken for a number of iterations equal to the given parameter, i.e. the time taken for \(x\) iterations is equal to \(Time(x)\).

It is noticeable that the CB1 technique may involve with multiple timer operations; since it evaluates and inserts idle time for a task every time when a loop or a conditional block of code within a task finishes. This will increase runtime overhead, in particular, when a task have multiple loops and conditional statements. In addition, a tiny round-off error in the idle time estimation will be accumulated. As a result, the CB1 technique is less effective than the SPP technique in term of jitter minimization. However, it consumes less power than the SPP technique (Gendy, 2009).

\[
Time(MAX - x) = \frac{(MAX-x)\cdot Time(x)}{x} \tag{3.1}
\]

\begin{verbatim}
Start timer;
for (i = 0; i < x; i++)
{
   //loop body
}
Stop timer;
Time(x) = timer count;
Time(MAX - x) = (MAX - x) * Time(x) / x;
Reset timer;
Adjust timer interrupt to occur after time:
Time_till_next_int = Time(MAX - x) + "Safety margin";
Send the microcontroller to power saving mode;
\end{verbatim}

**Figure 3.2:** Pseudo code of the code balancing technique, adapted from: (Gendy, 2009)
3.1.4 The planned pre-emption technique

The planned pre-emption (PP) technique is a jitter minimisation technique for interrupt latency jitter (Maaita and Pont, 2005). Interrupt latency is a delay after an interrupt occurs. It is inevitable since a processor needs to perform number of operations, such as to change the operating mode, save the current program status and perform a branch operation to interrupt vectors, before servicing an interrupt event. Each of these operations may take a small amount of time to complete, but will contribute to the interrupt latency.

Interrupt latency jitter is often due to interrupts that occur when the processor is executing different instructions; since different instructions may require different number of clock cycles to complete. Clearly, when the processor is executing a multiple-cycle instruction, an interrupt has to wait longer than when the processor is executing a single-cycle instruction.

In a tick-driven scheduling algorithm, the tick timer will generate an interrupt at the beginning of every tick. However, if the time span of a task execution is longer than the tick interval, it is likely that the tick timer will suffer from jitter in the interrupt latency. This is because the tick timer will interrupt the processor while an instruction (which could be a multiple-cycle or a single-cycle instruction) of the task is being executed. Consequently, tick jitter will be encountered.
The PP technique aims to suspend any task execution and sets the processor to idle, before the tick interrupt occurs. As shown in Figure 3.3 where TTRM scheduling is employed, the long task $\tau_3$ is always suspended by the PP to allow the tick timer interrupt to wake up the processor at the same state in every tick. The interrupt latency jitter is therefore minimised.

In the next section, we review the methodologies for run-time monitoring and error detection that are indispensable for achieving system reliability.

### 3.2 Run-time monitoring and error detection

Embedded systems are commonly used in modern industrial application to perform safety-critical tasks (Laplante, 1996; Bate, 1998; Goodloe and Pike, 2010) in real time. Such hard real-time systems have to operate timely to prevent system failures; since all properties in safety-critical real-time systems are effectively safety properties (Rushby, 2007; Goodloe and Pike, 2010). However, hard real-time systems are often designed and tested based on a set of assumptions (Stewart and Khosla, 1997). Overconfidence on such systems is likely to result in more serious consequences. For example, the Therac-25 medical device was used for radiation therapy in North America. It involved six massive radiation
overdoses which resulted in deaths or serious injuries between 1985 and 1988 (Leveson and Turner, 1993). The cause of those accidents was often attributed to software errors. Indeed, the software procedure substituted for traditional hardware interlocks that provided independent safety mechanisms, failed to function correctly. However, some of the accidents could be avoided if those hardware interlocks were not removed or an independent monitoring system was employed to trigger a recovery mechanism in response to a detected error. Perhaps, over-confidence in the ability of software should take the major responsibility for those accidents. This lesson underlines the importance of embedded software monitoring which could make the difference between life and death.

Unlike logical errors which can be minimized using debugging tools at design time, time-based errors are impossible to detect at design time (Tokuda and Kotera et al., 1990). This may be due to unrealistic assumptions, external disturbances or underestimate of the WCET. As the WCET guarantee can be abolished at run time as a result of, for example, malicious interference such as “cyber-attack”, monitoring real-time systems is essential to ensure run-time errors to be detected.

Over the last decades, many studies have been conducted on monitoring software behaviour of real-time systems at run time. The approaches taken can be divided into three categories: [i] Hardware-based monitoring; [ii] Software-based monitoring; and [iii] Hybrid monitoring. This section provides a review of these monitoring approaches.
3.2.1 Software-based monitoring

Software-based monitoring approaches are generally considered to be simple and flexible (Obermaisser and Peti, 2001). Instrumentation codes – also known as “software sensors” - are inserted at the points of interest in the software program running on the monitored system (Mansouri-Samani, 1995; Shobaki, 2004). Figure 3.4 shows a typical example of software-based monitoring approach where the instrumentation codes start a timer before a task execution and check if the task is overrun at its completion. If an overrun occur, the overrun task’s index value and the overrun tick number are stored before the program is terminated, otherwise the program continues.

![Diagram of Software-based monitoring](image)

**Figure 3.4: Software-based monitoring**
Software-based monitoring approach requires the monitored system to share its resources with the monitoring system. Thus, the capability of the monitoring system is limited by the availability of the system resources (Tokuda and Kotera et al., 1990). Also, the monitoring process will inevitably produce some degree of interference with the task executions at run time. Many software-based monitoring techniques have been developed in an attempt to minimise the impact of this “probe effect” on the monitored system.

3.2.1.1 A real time monitor

A real time (ART) monitor was proposed by Tokuda et al. (Tokuda and Kotera et al., 1990) as a permanent part of a monitored system: this prevents having to remove the software probes at a late stage in the system development. The ART monitor was composed with three parts: an event tap, a reporter and a visualizer. The event tap is used to record information; the reporter periodically sends the event message to the visualizer on a remote host. The visualizer obtains the message and displays the message on the host screen with an interface to allow the user to perform remote debugging.

The ART monitor was demonstrated on a mixed scheduling policy which utilises an integrated time-driven scheduler: hard real-time tasks were scheduled to execute based on the rate-monotonic priority scheme, and soft aperiodic tasks were scheduled to execute using a deferrable server. The ART monitor successfully detected time-based errors based on the notion of time encapsulation, that is to encapsulate each task’s timing error (if any) within the task. This was achieved by using the so-called “time fence” mechanism. This mechanism will
trigger a recovery operation if a task has not completed after its WCET. Although
time-based errors can be detected and handled, the monitor required significant
resources from the monitored system.

Thane (Thane, 2000b) further explored Tokuda’s approach and proposed to build
a monitoring system as a permanent part of a monitored system. Extensive testing
for this approach was conducted, see also (Pohlack and Dobel et al., 2006). In
Thane’s subsequent paper (Thane, 2000a), he suggested that software probes
which were placed within a ‘temporal firewall’ could be removed from the
monitored system without producing (temporal) side-effects: this is only possible
if the monitored system is either statically scheduled or employs a fixed-priority
scheduler (Thane, 2000a).

3.2.1.2 Inline software sensors
Mahrenholz (Mahrenholz, 2001) proposed that data values from software sensors
should only be stored if they cannot be reconstructed: he also suggested the use of
inline software sensors (instead of function calls). These two techniques can
reduce the spatial and temporal impacts of software monitoring approach.
However, the probe effect of software-based monitoring cannot be completely
eliminated (Tokuda and Kotera et al., 1990).

3.2.1.3 Task Guardian
Hughes and Pont introduced a task overrun handling technique, “Task guardian”
(TG), for TT embedded systems (Hughes and Pont, 2008). It was evaluated on
an embedded system which employed a TTH scheduler described in Section 2.7.
The TG employed a software monitoring approach for task overrun detection, but uses slightly different policies to handle an overrun case in a co-operative task and a pre-emptive task.

When a co-operative task overrun is detected, the co-operative task will be terminated and a backup task will be executed immediately. All executions of subsequent tasks are postponed by one tick interval. However, the overrunning task is allowed to continue its execution, if no other tasks are pending to run at the same tick interval. When a pre-emptive task overrun is detected, the overrunning pre-emptive task will be terminated and a backup pre-emptive task will be executed immediately but no task execution will be postponed.

The evaluation showed that the TTH system which employs TG can deliver predictable behaviour, even in the event of task overruns.

3.2.2 Hardware-based monitoring

Hardware-based monitoring approaches can address the software probe effect. However, they do this at the cost of additional hardware.

3.2.2.1 Traditional hardware-based monitor

Tsai et al. (Tsai and Chen, 1990) employed a complex architecture which was physically connected to the monitored system in order to “snoop” for signals from its buses, memory system and I/O channels, as shown in Figure 3.5. The monitoring system then used this information to imitate the monitored system and thereby obtain its state. Such a traditional hardware-based monitoring approach is
highly dependent on the architecture of the monitored system and requires sophisticated hardware (Haban and Wybranietz, 1990; Dodd and Ravishankar, 1992). Moreover, it is difficult to measure signals from some buses or memory ports, due to the miniaturisation and integration of components in modern microcontrollers (Thane, 2000b; Gergeleit, 2001). In addition, a limited number of port pins in microcontrollers makes it more difficult to extract data from the monitored systems (Shobaki, 2004).

3.2.2.2 Hardware watchdog

A watchdog timer is a simple hardware-based monitoring alternative that can handle a range of faults in a real-time embedded system. The mechanism of a watchdog timer is straightforward: it will induce system reset if it is not refreshed on time. This mechanism will operate periodically once the watchdog is enabled.

Pont and Ong introduced seven different watchdog patterns (Pont and Ong, 2002), as shown in Table 3.1, to detect failures or errors on system recovery. It was

![Figure 3.5: Traditional Hardware-based monitor](image-url)
designed for TTC embedded systems. These seven patterns can be used individually or selectively depending on the problem encountered.

Watchdog timers are often used with long timeout periods which may not be suitable for many applications. Also, using watchdog timers improperly may reduce the system reliability which may lead to multiple resets.

<table>
<thead>
<tr>
<th>Name of the pattern</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Watchdog Recovery</td>
<td>It helps to determine the cause of reset after it restarts.</td>
</tr>
<tr>
<td>Scheduler Watchdog</td>
<td>It is used for detecting if the scheduler has stopped operating</td>
</tr>
<tr>
<td>Program-flow Watchdog</td>
<td>It helps to deal with EMI-related problems</td>
</tr>
<tr>
<td>Reset Recovery</td>
<td>It employs an external watchdog timer to allow the system to restart after a watchdog-induced reset. However, it cannot determine the cause of reset; since only the on-chip watchdog can have this ability.</td>
</tr>
<tr>
<td>Fail-Silent Recovery</td>
<td>It shuts down the system after a watchdog-induced reset.</td>
</tr>
<tr>
<td>Limp-Home Recovery</td>
<td>It allows the system to run using a simple version of original algorithm after a watchdog-induced reset.</td>
</tr>
<tr>
<td>Oscillator Watchdog</td>
<td>It is used for dealing with the system’s oscillator failures. The watchdog has its own resistor-capacitor (RC) oscillator running at low frequency. The watchdog will be invoked if the system’s oscillator frequency is lower than the RC oscillator.</td>
</tr>
</tbody>
</table>

*Table 3.1: Seven watchdog patterns*

3.2.3 *Hybrid monitoring*

A third alternative is a “hybrid” monitoring approach which uses a combination of software and hardware (Shobaki, 2004), and can alleviate some of the shortcomings of both approaches (Mansouri-Samani, 1995). In a hybrid design, as shown in Figure 3.6, a software monitor captures events of interest in the
monitored system and passes information through an I/O interface to a dedicated external hardware device for further analysis. Since the error detection mechanism is migrated to the external device, the number of instrumentation codes in the monitored system, therefore, is reduced: this is particularly useful for monitored systems with scarce resources. Since the monitoring system consumes fewer resources from the monitored system, the interference it produced is accordingly minimized.

3.2.3.1 The test and measure processor monitor

Haban and Wybranietz (Haban and Wybranietz, 1990) demonstrated the test and measure processor (TMP) monitoring system in a distributed system. The TMP monitoring system was designed to allow users to have better knowledge about the run-time behaviour of their applications. The TMP was considered as an
integral part of each node in a multicomputer system. It was responsible for monitoring, recording and evaluating the activities of the host node. The TMP can communicate with other TMP, and send its results to the monitored host system or the central monitoring station. Haban claimed that the TMP monitoring system had no significant effect on the performance (below 0.1%) of the monitored system. This hybrid approach can also reduce the number of hardware connections between the monitored and the monitoring systems (Haban and Wybranietz, 1990; Thane, 2000b). However, a specifically tailored hardware platform is still required (Gergeleit, 2001).

3.2.3.2 Scheduler agent

Gendy and Dong et al. (Gendy and Dong et al., 2007) introduced a novel architecture, the main processor – scheduler agent (MP-SA), in an attempt to generate a feasible task schedule automatically for a TTC scheduling system. The architecture employed two microcontrollers: one of them was the main processor (MP) to operate the task schedule for executing tasks. Another one acted as a scheduler agent (SA) which was used for computing a feasible task schedule from the run-time information of the MP. The SA was connected to the MP to allow communication between them. This also enabled software probes in the MP to provide tasks’ information to the SA (i.e. a hybrid monitoring approach).

This technique required knowledge of task specifications, such as the WCETs and the periods. However, the initial release times for all tasks were set to 0, i.e. their first releases were at the first tick. The task specifications were first stored in the MP and then sent together with the tick interval to the SA. The SA then signalled
the MP to execute number of dummy tasks (i.e. no task is running) in different
ticks and measured the idle time in each tick. By obtaining the difference
between the tick interval and the idle time, the scheduler overhead was computed.

Based on the this result in conjunction with the previously obtained task
specifications, the SA attempted to generate a feasible task schedule by re-
calculating the initial release time of each task, the initial task execution order and
the tick interval. Once the MP received the new task schedule, it reconfigured the
scheduler in accordance to the task schedule and restarted all operations.

The SA continued to observe the timing behaviour of all tasks in the MP. If any
violation of timing constraints is detected, the SA would re-generate a feasible
task schedule based on the actual task execution times. In the worst case scenario
that no feasible task schedule was generated, the SA signalled the MP to terminate
all operations.

Although the SA can detect run-time errors and create a task schedule for the MP,
it may not be possible to use this technique at run time unless a backup system is
present; since recalculating a task schedule may take a sufficient long time.

3.3 Discussion

In this chapter, we reviewed four different jitter minimisation techniques and three
monitoring approaches have been reviewed. This section discusses a common
jitter problem which cannot be addressed by the reviewed jitter minimisation
techniques. We also discuss the strength and weaknesses of the reviewed monitoring approaches.

3.3.1 A remaining jitter problem

In Section 3.1, we have reviewed four different jitter minimisation methodologies. These methodologies can minimise different types of jitter for TT embedded systems. For example, task execution jitter and task release jitter can be minimised by the SD, the SPP and the CB1 techniques, while the PP technique can prevent tick jitter.

However, the reviewed methodologies cannot cope with task jitter that is due to the task schedule. A typical example has been briefly considered in Section 2.3.1. In this particular example, multiple (periodic) tasks with different execution rates are released simultaneously. Undoubtedly, this will cause task release jitter even if any of the aforementioned jitter minimisation methods is applied.

Figure 3.7 and Figure 3.8 illustrates this common scenario, where the task sets in Table 3.2 and Table 3.3 are respectively scheduled by a TTC scheduler and a TTRM scheduler. The figures show that the SD technique has been applied to both systems, but cannot prevent task release jitter to occur. In Figure 3.7, Task τ3 – scheduled by a TTC scheduler – executes after Task τ1 and Task τ2 at tick 0 but only after Task τ1 at tick 2 and tick 4. As a result, Task τ3 executes at irregular periods of time. This also happens at TTRM scheduling. As shown in Figure 3.8, Task τ2 executes after Task τ1 at tick 0 and tick 6, but at tick 3 it executes as the first task in the tick.
<table>
<thead>
<tr>
<th>Task</th>
<th>Offset (tick)</th>
<th>Period (tick)</th>
</tr>
</thead>
<tbody>
<tr>
<td>τ1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>τ2</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>τ3</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>τ4</td>
<td>0</td>
<td>4</td>
</tr>
</tbody>
</table>

Table 3.2: A time-triggered co-operative task set that contains jitter

<table>
<thead>
<tr>
<th>Task</th>
<th>Offset (tick)</th>
<th>Period (tick)</th>
</tr>
</thead>
<tbody>
<tr>
<td>τ1</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>τ2</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>τ3</td>
<td>0</td>
<td>4</td>
</tr>
</tbody>
</table>

Table 3.3: A time-triggered rate-monotonic task set that contains jitter
The same result can be envisaged when applying the CB1 technique or the SPP technique to the system. This is because the SD, the CB1 and the SPP techniques operate on task level, but the release jitter in this particular problem is induced at scheduling level. Even if the execution times of tasks are fixed at constant values, it cannot affect the scheduling decision in which different subset of tasks in the systems may be simultaneously released at some ticks. Clearly, jitter minimization techniques that operate at task level could not handle task jitter that is caused at scheduling level.

### 3.3.2 The strengths and weaknesses of the monitoring approaches

In general, the three monitoring approaches described in this chapter have their own strengths and weaknesses in different situations. Of course, the run-time behaviour of the monitored system should not be affected by the monitoring process, in particular, if the monitored system is safety-critical. In this situation where the degree of system interference is concerned, hardware-based approach is the optimal solution for run-time monitoring (Gergeleit, 2001).
However, production cost is often a primary consideration for consumer product manufacturing. Under this consideration, software-based monitoring approach would be the best candidate. In the situation when the resources of the monitored system are tightly constrained, sacrificing some of the coverage areas and abilities of the software-based monitoring system are inevitable, in order to minimize interference with the monitored system (Mahrenholz, 2001). Nevertheless, without sufficient and good quality information about monitored systems, false positive (or false negative) results may be obtained. Instead of removing some components (for analyzing, recording, etc.) in the software-based monitoring system, hybrid monitoring approach addresses this issue by migrating them to an external hardware monitoring device (Mahrenholz, 2001).

In brief, selecting a run-time monitoring system for real-time embedded systems is “requirement-oriented” – hence, there is no universal monitoring solution (to date) which can satisfy all requirements of different real-time embedded systems. The work described in this thesis, however, was specifically aiming to explore novel and low-cost techniques to monitor safety-critical TT embedded systems with jitter constraints, and ultimately to provide a framework that has no negative influence from the monitoring process to support safety-critical system design.

### 3.4 Conclusion

This chapter has provided reviews of four jitter minimisation techniques. Using these techniques, the behaviour of tasks is more predictable. In particular, when the SD technique is employed, the behaviour of tasks is very close to what a perfect TT system was defined in Section 2.5.
However, there is a jitter problem that cannot be addressed by the SD technique and the other reviewed techniques. This jitter problem has been identified and elucidated in Section 3.3.1. In Section 6.2 of Chapter 6, a novel jitter minimization technique that operates on scheduling level is proposed to address this problem. This technique is utilised in conjunction with the SD technique to form the basis of a pTT system.

This chapter has also reviewed different monitoring approaches and techniques. As the aims of this research study are concerned, software-based monitoring is excluded from this research work; since it will induce interference to the monitored system. In addition, when the monitored system is malfunctioning, the monitor resides within the monitored system is likely to be affected. The next chapter will introduce a novel non-invasive hardware-based monitoring approach which does not have these problems.
Chapter 4

Non-invasive monitoring

This chapter introduces a non-invasive monitoring approach that is designed specifically for embedded systems which employ the time-triggered co-operative (TTC) scheduling described in Section 2.6.

This chapter begins with a brief discussion of the choice of monitoring approaches and software architecture for safety-critical systems. It then reviews previous work on processor power predictions, followed by an introduction of the principle of the proposed non-invasive monitoring approach. The implementation of this monitoring approach for a TTC system is also presented with the results of the evaluation and case studies to demonstrate the feasibility of the proposed monitoring approach.

4.1 Introduction

As the result of failures in safety-critical embedded systems can be fatal, selecting a monitoring approach and software architecture for such systems requires careful consideration. When selecting software architecture for safety-critical embedded designs, temporal predictability is one of the most important criteria which has been emphasised repeatedly in this thesis. For instance, event-triggered (ET) software architecture has been excluded from our consideration in the beginning of this thesis, due to its unpredictable behaviour that is likely to cause violations of deadlines. Also, since it is difficult to analyse the temporal behaviour of tasks, run-time errors are unlikely to be detected.
Alternatively, time-triggered (TT) software architecture is considered as an appropriate choice for safety-critical embedded designs; since TT software architecture can provide predictable temporal behaviour which maximises the chances that the system will operate correctly at run time. It also increases the possibility for the detection of run-time errors if they occur.

In Section 2.5, the definition of a perfect TT model has been stated. Amongst the TT family, the closest approximation to this model involves a collection of periodic tasks which operate co-operatively. Such time-triggered co-operative (TTC) scheduling, which has been introduced in Section 2.6, is often considered as a practical solution for safety-embedded system designs. However, as run-time errors may occur unexpectedly, monitoring the run-time behaviour of safety-critical systems is essential.

Applying software-based monitoring to an embedded software system will introduce additional computational overhead and may change the actual behaviour of the system. More important, it may not be capable to report errors when the monitored system is halted by errors or failures. The Clementine spacecraft incident in 1994 (Binkley and Cheng et al., 2005) was a typical example: a software error was encountered, but the software watchdog algorithm could not execute due to the control computer had “crashed”.

Thus, monitoring systems are necessary to work independently from the monitored system in safety-critical system monitoring. It is preferable that the
monitor process should have no interference to the monitored system’s operations. As discussed in Section 3.3.2, hardware-based monitoring would be the best candidate to achieve this goal. However, it is difficult to measure internal signals from a modern microcontroller.

In this chapter, we explore a novel hardware-based monitoring approach to detect time-based errors in a TTC embedded system based on the processor power consumption.

### 4.2 Processor power predictions

A modern embedded processor contains in the order of million logic gates implemented in the Complementary Metal Oxide Semiconductor (CMOS) technology (Raja and Agrawal et al., 2005). When a CMOS gate is powered up, it starts to consume power, regardless it is active or not. When it is active, the power consumed by the gate is called dynamic power; while it is inactive, the power dissipated due to leakage current is called static power. Clearly, dynamic power is much higher than the static power.

From an embedded processor perspective, a computer program is simply a sequence of instructions. The processor executes different instructions by applying or removing electrical charge to different sets of logic gates. Thus, the energy consumed by the processor will depend on the instructions being carried out (Tiwari and Malik et al., 1994; Tiwari and Malik et al., 1996; Talarico and Rozenblit et al., 2005). In other words, the power consumption of a processor will
vary when a computer program is being executed. This effect provides the underlying foundation to support the work described in this thesis.

Many research studies have been conducted to estimate the power consumption of software programs executing in embedded systems, as embedded devices are often power constrained. Nikolaids and Laopoulos (Nikolaidis and Laopoulos, 2002) proposed a power consumption measurement framework to achieve the estimation based on an instruction-level power consumption model. This model is pre-generated by measuring the base cost of all instructions and the power overhead of a sequence of different instructions. It also takes into account different factors which may affect the power consumption of a software program, such as cache misses and pipeline stalls.

Chang et al. (Chang and Shen et al., 2006) introduced a hardware-based simulation approach to identify the highest power consumption component in a software program. The software runs on a FPGA-based processor where two counters are embedded in the processor to record the number of times that each functional unit has been accessed and the number of data bits which has changed. The values of these counters are gathered by an ARM9-based processor and subsequently sent to a desktop PC for analysis. The analysis involves estimating the power consumption of the software based on the data from the counters. Although the power measurement is imprecise, the system can identify the software components which consume most power – and does so much faster than a software simulator.
Tiwari et al. (Tiwari and Malik et al., 1996) used a measurement-based technique to obtain the average current of instructions in the development of instruction power model for a 486Dx2 CPU which was mounted on a mobile personal computer evaluation board. The obtained power model had a set of base costs of instructions which were all average values. The power model was rather precise which provided useful insight into the average power consumption in processors.

Overall, previous research on the estimation of power consumption has focused on fulfilling power (and particularly battery life) requirements for embedded systems. Unlike the work described in this thesis, the aim has not been to detect software timing errors.

### 4.3 Towards a novel hardware-based monitoring

It was discussed that the instantaneous power consumed by processors during operations will change from one instruction to another. However, it is challenging to discriminate instructions from processor power consumption; since modern processors often employ multi-stage pipeline architecture in which a number of instructions are processed simultaneously in every clock cycle (Tiwari and Malik et al., 1994; Tiwari and Malik et al., 1996). In addition, input data will have significant influence to the power consumption of an instruction (Tiwari and Malik et al., 1996; Chang and Kim et al., 2002). It would be extremely challenging to perform instruction identification in run time. This is because modern processors can perform tens of millions of instructions per second. Processing such a large amount data in run time not only requires substantial computational effort and speed, but also a large memory space (Shobaki, 2004).
However, it would be much appropriate to determine if a processor is active or not, from its power consumption. The instantaneous power consumption of a processor, as shown in Figure 4.1, will be at high level when it is active, but low level when it is inactive or idle. This effect was exploited by the proposed monitoring technique, namely the non-invasive safety agent (NISA).

![Figure 4.1: The power measurement of a TT co-operative system](image)

**Figure 4.1: The power measurement of a TT co-operative system**

### 4.4 The non-invasive safety agent

In a real-time embedded system that employs the TTC scheduling algorithm described in Section 2.6, the processor’s power consumption will fluctuate significantly over time, because of the periodic use of idle mode. In other words, the processor’s power consumption will drop after task executions in every tick if no violation of deadline occurs.

Based on this observation, the NISA technique can detect deadline violations by comparing a fixed threshold value with the processor’s power consumption around the common deadline of task executions, i.e. before the end of a scheduler tick. As shown in Figure 4.2 where the processor power consumption of a TTC
system is illustrated, the processor power consumption rises above the threshold when a tick interrupt occurs. This indicates that the processor is active. Based on the task schedule provided at design time, the NISA technique recognizes if a task or a set of tasks is released in a specific tick. By comparing the processor power consumption with the threshold, it can detect the completion of task executions. Since the processor is set to the idle state after the completion of task executions, the processor power consumption drops below the threshold. In this example, no violation of deadline occurs.

![Figure 4.2: The methodology of the non-invasive safety agent](image)

However, if the processor is not set into the idle state before the next tick timer interrupt, the common deadline is violated\(^1\). The NISA technique will report the tick number and the set of tasks which are scheduled to run at that tick.

### 4.5 Implementing the non-invasive safety agent

This section presents the implementation of the NISA technique which includes the hardware and software designs.

\(^1\) Please note that we assume that tasks themselves do not enter an idle state unless otherwise stated.
4.5.1 **Hardware configuration**

The NISA technique can be implemented on any microcontroller which has an on-chip analogue-to-digital converter (ADC) for obtaining the power signal from a monitored processor. Unlike traditional hardware monitoring methods described in Section 3.2.2.1, the NISA module does not require any complex architecture to obtain internal signals from the monitored system, but requires only a simple hardware interface – such as a resistor (typically 1 Ω) – to obtain the power consumption of the monitored processor, as shown in Figure 4.3.

![Figure 4.3: The connection diagram of the NISA module](image)

However, a signal conditioning module that allows differential voltage inputs – such as a MAX4372 current-sense amplifier (Maxim Integrated Products Inc., 2009) – is required if the on-chip ADC only supports single-ended ADC inputs. In a situation when a microcontroller does not have an ADC, an external ADC can be used. Overall, the implementation cost for the NISA is still at relatively low cost. Furthermore, it does not require for any software modification in the monitored system, and have no influence to the software operations of the monitored system at run time.
4.5.2 Software design

The monitoring software of the NISA technique presented in this section was designed to start before the monitored system. As shown in Figure 4.4, as soon as the NISA module starts, it begins searching for the initialisation power trace of the monitored system from the power signal. If the monitored system is not powered on in time, the time-out flag will be set. The NISA module will report this situation. If the initialisation power trace is found, it will start to search for a rising edge. This rising edge indicates the start of the first tick in the monitored system.

While the rising edge is found, the NISA module records the tick number and set itself into the idle state after it starts a timer. The timer is utilized to generate an interrupt signal every 1/x of the tick interval to increase a counting variable by 1, where x is an integer constant and the tick interval is divisible by x.

When the counting variable is equal to x-1, the NISA module will wake up and start searching for the next rising edge, i.e. the monitor process only starts before the common task deadline. Once the rising edge is found, the timer will restart immediately. The NISA module will reset the counting variable and update the current tick number before returning to the idle state. The same operation will be performed in the next cycle.
However, the rising edge could not be found when a task misses the deadline. In this situation, for experimental purposes, the NISA module will continue to increase the counting variable rather than to trigger a system recovery operation. The counting variable will be reset when its value equal to $x$, i.e. at the end of the tick. In addition, the current tick number will be sent to the host computer to be displayed before the NISA module returning to the idle state.

It is noticeable that the synchronization between the monitored system and the NISA module is done when every rising edge is found. In addition, the NISA
does not require sampling over the entire tick interval, but instead it only samples 
over a short period of time at the end of a tick. Accordingly, the NISA module is 
able to save computational power and memory space.

4.6 Assessing the non-invasive safety agent

In this section, the ability of the NISA technique on detecting deadline violations 
in TTC embedded systems is illustrated through a set of empirical tests and a case 
study. For the purpose of demonstrating the flexibility of the NISA technique in 
hardware selection, two different microcontrollers were chosen to perform the 
tests and the case study respectively.

It is important to note that a LabVIEW (Bishop, 2003) software monitor program 
was also created on a host computer to provide a visualisation of the monitored 
system’s power measurement. This allowed the results obtained by the NISA 
module from the monitored system to be verified.

4.6.1 Hardware configuration of the monitored system

The monitored system was implemented on an NXP LPC2106 microcontroller 
(NXP Semiconductors, 2003b) mounted on an Ashling EVBA7 evaluation board 
(Ashling Microsystems, 2003). The Ashling board provides two separated power 
supplies to the LPC2106 microcontroller: a 1.8V supply to the processor and a 
3.3V supply to the I/O ports. The LPC2106 microcontroller is a commercial-off- 
the-shelf (COTS) microcontroller, contains a range of peripherals and an ARM7- 
TDMI CPU core which is one of the market leading embedded processors.
Note that the processor speed was set to 29.4912MHz and the 1.8V supply to the processor was monitored in the experiments.

4.6.2 Software configuration of the monitored system

The monitored system was embedded with a TTC scheduler described in Section 2.6, to schedule five different tasks to execute. These five tasks are:

1. Blinking four LEDs,
2. Reading an I/O port,
3. Reading the real-time clock and storing the current time to a buffer,
4. Sending data in a buffer using UART, and
5. Sending data using SPI (Serial Peripheral Interface bus).

Note that the scheduler tick interval was set to 1 millisecond.

4.6.3 Desktop monitor

The desktop monitor was implemented on a host computer equipped with a NI PCI-6035E data acquisition – DAQ – card with an on-board 16-bit analogue and digital converter (ADC). The 200 KHz ADC gives a uniform 5 microsecond sampling interval. As depicted in Figure 4.5, it was connected to the sampling resistor through a National Instruments CL-68LP connected block to sample the analogue power signal from the monitored system.
4.6.4 The non-invasive safety agent module

The NISA module comprised a Keil MCB2100 evaluation board (Keil - an ARM company, 2006) mounted with a 32-bit ARM7-based LPC2129 microcontroller (NXP Semiconductors, 2007). The LPC2129 microcontroller contains an internal 10-bit ADC with a sampling rate up to 400 KHz which allows an analogue signal from 0 to 3.3 V to be measured.

Hence, there are 1024 quantization levels between 0 to 3.3 V and each level has a value of 3.23 mV approximately. This resolution was insufficient for the NISA module to determine the state of the monitored system. This was because the difference of the voltage drops across the sampling resistor between the “IDLE” state (~5 mV) and the “ACTIVE” state (~16 mV) of the monitored system’s processor, is too small (only ~11mV, i.e. 3 quantization levels) which could be easily corrupted by noise. Additionally, the LPC2129 only supports for single-ended ADC inputs. Thus, a signal conditioning module was needed.
A signal conditioning module – a low-cost differential amplifier – was deployed between the NISA module and the monitored system, as shown in Figure 4.5. The power signals from the two ends of the sampling resistor was amplified by 10 times, and then subtracted by each other to obtain the amplified voltage drop across the resistor before feeding to the ADC of the NISA module. Note that a threshold value of 8 mV was empirically chosen to determine whether the monitored system was running or idling.

4.6.5 Descriptions of the experiments

The NISA technique was evaluated by performing several short duration tests – 1000 ticks – in two different experiments on the monitored system. Both of the experiments contained a number of test sets to test the ability of the NISA technique on the detection of deadline violations. In each test set, there were different test cases. Each test case employed a unique set of task parameter values, such as task periods and offsets.

In the first experiment, a fixed time delay was added to all tasks in each test set after a specific tick. Different time delay values from 0 to 90% of the tick interval were used in different cases in a test set (i.e. 10 cases in a test set). With this delay, tasks might fail to complete their executions before the deadline (i.e. the processor would not be placed into the idle state).

In the second experiment, most of the settings from the first experiment were adopted. However, in order to bring the experiment closer to reality, a fixed time delay (from 10 to 90% of the tick interval) was added at a random time to a
selected task in each test set (i.e. 9 test cases in a set). This was achieved by using a look-up table to store a set of randomly pre-generated Boolean values, each of which was associated with each tick in the monitored system: the time delay was added to the selected task only if it was scheduled to execute in the tick that had a value of one ("true").

The first experiment had 48 test sets containing a total of 480 tests that covered the cases for adding different time delays into the first tick (Tick 0), the last tick (Tick 999) and the ticks that are selected randomly between these two ticks. Note that the ticks, in which a time delay was added in the second experiment, were selected in the same manner. However, only 270 tests in 30 different test sets were performed, since these were felt to be sufficient to guarantee the quality of the experimental results.

4.6.6 Experiment 1: fixed test cases

Table 4.1 shows the task set of the monitored system in one of the test sets selected from the first experiment.

<table>
<thead>
<tr>
<th>Task ID</th>
<th>Task</th>
<th>Offset (ms)</th>
<th>Period (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>T0</td>
<td>SPI_Update</td>
<td>1</td>
<td>50</td>
</tr>
<tr>
<td>T1</td>
<td>UART1_O_UPDATE</td>
<td>5</td>
<td>75</td>
</tr>
<tr>
<td>T2</td>
<td>Blink_Leds</td>
<td>0</td>
<td>20</td>
</tr>
<tr>
<td>T3</td>
<td>Display_RTC_Update</td>
<td>10</td>
<td>70</td>
</tr>
<tr>
<td>T4</td>
<td>Switch_Read</td>
<td>7</td>
<td>30</td>
</tr>
</tbody>
</table>

Table 4.1: A task set of the fixed time-delay experiment
In this test set, a fixed amount of time delay was added to all task execution times after tick 989. The monitored system’s behaviour was captured by the desktop monitor. Two of the test results are selected and shown in Figure 4.6 and Figure 4, for each respective time delay – 0.8 ms and 0.9 ms – between tick 997 and tick 1001.

![Graph of the voltage across the sampling monitor of CPU CORE](image)

**Figure 4.6: Tasks with an additional 0.8 ms time delay**

In Figure 4.6, the processor power measurement in tick 997 indicated that the processor was idling at the end of the tick (as it is below the 8mV threshold). This provided evidence that task T4 completed its execution before the deadline (i.e. the start of tick 998). In tick 998 and tick 999, no task was scheduled to run based on the task schedule. The processor power measurement rose above the threshold in the beginning of these two ticks was due to the scheduler overhead. In tick 1000, task T2 executed normally as the power measurement dropped below the threshold before the arrival of tick 1001. However, task T0 missed the deadline at tick 1001; since the processor power measurement was above the threshold throughout the tick. As the processor remained active, no evident for the completion of task T0 can be found. The NISA module confirmed that task T0 violated the deadline in tick 1001.
In Figure 4.7, the processor power measurement did not drop below the 8mV threshold in tick 997, tick 1000 and tick 1001. As the processor remained active in these ticks, no evident for the completion of the task executions (T4, T2 and T0) in these ticks can be found. This confirmed that tasks T4, T2 and T0 failed to finish before the deadlines in tick 997, tick 1000 and tick 1001, respectively. These deadline violation events were also detected by the NISA module.

4.6.7 Experiment 2: random test cases

Table 4.2 shows the task set of the monitored system in one of the test sets selected from the second experiment.

<table>
<thead>
<tr>
<th>Task ID</th>
<th>TASKS</th>
<th>Offset (tick)</th>
<th>Periodic Time (tick)</th>
</tr>
</thead>
<tbody>
<tr>
<td>T0</td>
<td>Switch_Read</td>
<td>1</td>
<td>20</td>
</tr>
<tr>
<td>T1</td>
<td>UART1_O_UPDATE</td>
<td>2</td>
<td>60</td>
</tr>
<tr>
<td>T2</td>
<td>SPI_Update</td>
<td>5</td>
<td>35</td>
</tr>
<tr>
<td>T3</td>
<td>Blink_Leds</td>
<td>15</td>
<td>60</td>
</tr>
<tr>
<td>T4</td>
<td>Display_RTC_Update</td>
<td>20</td>
<td>55</td>
</tr>
</tbody>
</table>

Table 4.2: Task information of a test set in the second experiment
In this test set, task T3 was the selected task in which a fixed amount of time delay might be added to its execution time; this was dependent on the random numbers associated with the ticks where task T3 was scheduled to run. Figure 4.8 and Figure 4.9 respectively show the results of two different test cases (with 0.6 ms and 0.8 ms time delays randomly added to tasks) in the test set. In order to permit comparisons to be made, each of the results was organized in three segments, showing different ticks where task T3 was scheduled to run.

Figure 4.8: Task T3 executing with a 0.6 ms randomly added time delay

In Figure 4.8 where 0.6 ms time delay was randomly added to task T3, the left segment shows that task T3 executed normally in tick 15 as the power measurement dropped below the 8 mV threshold before the start of tick 16. From the middle segment, it can be determined that task T3 missed the deadline in tick 75, as the processor remained active in this tick. According to the task schedule, the NISA module reported that a deadline violation involved with three tasks – T2, T3 and T4 – occurred in tick 75. The segment on the right shows that task T3 completed its execution before the deadline in tick 195 based on the same reason as task T3 in tick 15.
Figure 4.9: Task T3 executing with a 0.8ms randomly added time delay

In Figure 4.9, task T3 again ran normally in tick 15 and tick 195, as shown respectively in the left and the right segments. Unlike the middle segment in Figure 4.8, tasks T2, T3 and T4 did not miss the deadline at tick 75 in this test, as the processor was set into the idle state before the beginning of tick 76.

4.6.8 Case study: extended test duration and system power failure

In this case study, the NISA technique was evaluated over an extended test period and the ability to detect power failure. A microcontroller equipped with a high-resolution ADC was employed to implement the NISA module. This minimised the need for external hardware to connect the NISA module to the monitored system.

4.6.8.1 Descriptions of the case study

The NISA technique in this case study was implemented on an Olimex ADuC-P7026 development board (Olimex Ltd, 2005) mounted with an ARM7-based ADuC7026 microcontroller (Analog Devices, 2007). The ADuC7026 microcontroller is equipped with a 12-bit ADC with a sampling rate up to 1 MHz.
This device supports differential ADC inputs with an adjustable ADC reference voltage which makes it particularly suitable for the implementation of the NISA technique.

The NISA technique described in Figure 4.4 was implemented in this case study. However, it was configured to observe the monitored system continuously for a seven-day period (604800000 ticks). The set of tasks described in Table 4.2 was employed by the monitored system. Again, the tasks were scheduled by a TTC scheduler. However, only one of these tasks was forced to overrun (once) in the week.

A time delay which is equivalent to 90% of the tick interval was added to task T4 – see Table 4.2 – at the 522720020th tick. With this time delay, a task overrun was guaranteed to occur in this tick.

The NISA module was also adapted by adding a conditional code to test if the power measurement of the monitored system is dropped below the idle state’s voltage level. We simulated power failure in the monitored system by unplugging its power cable, to test the NISA module after the extended period test was completed.

4.6.8.2 Results

Over the seven-day period of this test, the selected task T4, which was the only task forced to overrun the deadline once, was detected at the 522720020th tick.
No other (false) positive results were obtained. Also, the power failure in the monitored system was detected by the NISA module.

This simple case study provided evidence that the NISA technique is not only able to be used for a longer period of continuously monitoring, but also capable to detect power failure in a TTC embedded system. The case study also illustrated that the NISA technique could be implemented at very low cost (with minimal external hardware).

4.7 Discussion

The experiments and the case study demonstrated that the most critical timing constraint violation (i.e. missing a deadline) in TTC scheduling can be detected by the NISA technique. However, a false alarm will occur in the situation when the completion of task executions is equal or very close to the deadline. In this situation, the processor may not have enough time to enter into the idle state. However, allowing this situation will have serious impact, in practice, on the temporal predictability of the system as discussed in Section 3.1.4. Thus, the common deadline for task executions must be set at the point in time before the arrival of the next timer interrupt.

It is also noticeable that the NISA technique is incapable to identify the cause of a violation. For example, when one of the tasks in a task chain overruns its WCET (i.e. an occurrence of a task overrun) for some reasons, it is impossible for the NISA technique to identify the overrun task; since the processor remains active when the task chain is executing – hence, the processor’s power consumption will
not drop. Perhaps, the root cause of this limitation is due to the nature of hardware-based monitoring approach which only observes signals passively from the monitored systems. As a result, detailed run-time information of task executions cannot be captured. However, the monitoring approach of the NISA technique provides an alternative approach to detect time-based errors in TTC embedded systems, and a light on the horizon for hardware-based monitoring approach to detect run time errors in modern embedded systems.

### 4.8 Conclusion

This chapter has presented the non-invasive safety agent (NISA) technique, which is a novel hardware-based monitoring approach to detect deadline violations in TTC embedded systems based on the observation of the processor power consumption.

The implementation of the NISA technique is simple which does not require a complex architecture like traditional hardware-based monitoring techniques (see Section 3.2.2.1), but only a microcontroller with one resistor. This hardware configuration not only minimises the number of external hardware but also reduces the number of connections between the external monitor and the monitored system.

The results of the evaluations and case study show that the most critical timing constraint violation (i.e. a violation of deadline) in TTC embedded systems can be detected by the NISA technique. This provides sufficient evidence for the
feasibility of the monitoring approach of the NISA technique, despite the limitations discussed in Section 4.7.

In the next chapter, we present an extension of the NISA technique to detect deadline violations in embedded systems which have more complex power measurements than TTC systems. A study for the limitations of the NISA technique and the possible approach to overcome these limitations will be discussed in Section 5.5.
Chapter 5
An extension of the non-invasive monitoring

It was demonstrated in Chapter 4 that it is feasible to detect deadline violations in TTC scheduled embedded systems by monitoring fluctuations in the processor power consumption. This chapter introduces an extension of the NISA technique to monitor TTC embedded systems which employ a static dynamic voltage scaling (and dynamic frequency scaling) – DVS – scheme. In order to distinguish this extension from the NISA technique, we name it as the NISA-DVS technique. The NISA-DVS technique was demonstrated on a realistic case study and evaluated against other software-based monitoring techniques. The results of this case study are also reported in this chapter.

5.1 Introduction

Portable embedded systems are usually power constrained due to limited battery life. However, embedded software for such portable systems is becoming increasingly complex and more advanced which requires more powerful processors to operate (Pillai and Shin, 2001).

Many research studies have been conducted in attempts to resolve this conflict. For example, some software techniques aim to conserve energy by setting the processor into an idle state when no operation is required to perform. A more sophisticated technique – a dynamic voltage scaling (DVS) scheme – based on equation (5.1) to conserve energy by changing the processor clock frequency and supply voltage dynamically according to the workload either predicted at design...
time – i.e. offline (or static) DVS – or computed at run time – i.e. online (or aggressive) DVS.

\[ P = CV^2 f \]  \hspace{1cm} (5.1)

where \( P \) is the power needed for a CMOS gate to switch its state from one to another, \( C \) is the switching capacitance, \( V \) is the operating voltage and \( f \) is the switching frequency.

Safety-critical embedded applications, like other embedded applications, may also require an energy saving scheme in order to maximize the battery life. As temporal predictability is paramount for safety-critical systems, the energy saving technique must also be predictable. Offline techniques are often preferred over online technique to achieve high predictability. Thus, a static DVS scheme would be an appropriate choice for safety-critical software design.

However, safety-critical systems which employ a static DVS scheme also require a monitoring system to detect errors at run time. In this chapter, an extension of the NISA technique – namely the NISA-DVS technique – for TTC embedded systems equipped with a static DVS scheme is presented. The NISA-DVS technique was evaluated in a realistic system. Specifically, a 3-channel wireless electrocardiograph (ECG) system was implemented as the monitored system in the case study.

The ECG device was previously employed as a test-bed for the experiments described in Phatrapornnant and Pont’s paper (Phatrapornnant and Pont, 2006). The so-called “TTC-jDVS” algorithm used in the monitored system has a
mechanism to minimise jitter that is introduced when the processor clock frequency is changed by the DVS scheme. This algorithm is described elsewhere in Phatrapornnant and Pont’s paper and will not be explored further in this thesis.

5.2 Key challenges

It is expected that the power consumption of a processor will fluctuate when a software program is being executed. However, when a DVS scheme is employed, the processor power consumption will fluctuate more frequently and significantly. As shown in Figure 5.1 where the power measurement of a TTC-DVS system is illustrated, the processor clock frequency and supply voltage change in different ticks based on the DVS scheme. Unlike systems which operate at a single processor speed which has two processor power consumption levels (see Figure 4.1 in Section 4.3), the processor power consumption in DVS systems has different levels as a result of changes in the processor clock frequency. This increases the difficulty to monitor the run time activities of task executions.

Specifically, one of the key challenges to monitor the TTC-DVS system arises from the difficulty to determine the processor state from its power consumption when the clock frequency is at a low speed. As shown in Figure 5.1, the difference between the active power measurement and the idle power measurement of the processor at a low clock frequency (10 MHz) is relatively small (approximately 2 mV) which can easily be corrupted by noise.
Another key challenge comes from the nature of the DVS scheme, in which the processor supply voltage and clock frequency change frequently at run time. As shown in Figure 5.1, the active power measurement of the processor at a low clock frequency (10 MHz) is lower than the idle power measurement of the processor at a high clock frequency (50 MHz). It is impossible to determine the processor state using a single threshold value like the NISA technique described in Section 4.3.

**Figure 5.1: The power measurement of a TTC-DVS system**

At a low CPU frequency, the difference in the CPU power consumption between CPU ACTIVE and CPU IDLE is relatively small.

The idle CPU power consumption at a high frequency is higher than the active CPU power consumption at a low frequency.

5.3 Implementing the non-invasive safety agent extension

Given the wide range of power signal levels which is to be monitored in a TTC-DVS system, it is necessary to employ a signal conditioning module to process the power signal before data analysis is performed. Specifically, the power signal needs to be amplified and filtered in order to make the processor state at a low clock frequency distinguishable by the NISA-DVS technique.
The power signal shown in Figure 5.1 was amplified approximately 25 times and then filtered by a second-order Butterworth low-pass filter with a 10 KHz cut-off frequency. The processed power signal, shown in Figure 5.2, becomes smoother and the processor state at a low clock frequency (10 MHz) is distinguishable; since the difference between the idle power measurement and active power measurement of the processor at 10 MHz clock frequency was increased from 2 mV to 60 mV.

The NISA-DVS technique copes with another challenge described in Section 5.2 by dynamically adjusting the threshold value in response to the changes in the processor clock frequency, as shown in Figure 5.2. This is achievable; since a static DVS scheme is considered in this research study. This means that the processor clock frequency and the supply voltage at each tick can be known at design time. Thus, a set of threshold values in response to the clock frequency changes in different ticks can be pre-defined and stored in a lookup table within

![Figure 5.2: The processed power measurement of a TTC-DVS system](image)

This figure illustrates the processed power measurement of a TTC-DVS system, with thresholds adjusted accordingly to identify the system state at different processor clock frequencies. The system state at low processor clock frequency becomes distinguishable.
Chapter 5 – An extension of the non-invasive safety agent

the NISA-DVS module. Note that the number of threshold values must cover the major cycle of the task set in the monitored system.

The software procedures for the NISA technique shown in Figure 4.4 at Section 4.5.2 can be adapted for use in the NISA-DVS technique. However, the threshold value of the NISA-DVS technique must be changed accordingly at the start of every tick of the monitored system; assuming no deadline violation has occurred.

In a situation when a deadline violation occurs (i.e. no rising edge is found), the change of the threshold value will depend on the processor speed in the present tick and the previous tick (where the violation occurred):

- If the pre-defined processor speed in the present tick is higher than the speed in the previous tick, the threshold will be changed when a rising edge is found,
- If the pre-defined processor speed in the present tick is lower than the speed in the previous tick, the threshold will be changed when a falling edge is found,
- If both ticks have the same processor speed, the threshold is not necessary to be changed.

5.4 Case study: A wireless electrocardiograph device

This case study employed a 3-channel wireless electrocardiograph (ECG) device as the monitored system to evaluate the NISA-DVS monitoring system and two simple software-based monitors. This section begins with a brief description of the ECG system, and then presents the evaluation and the results.
5.4.1 Implementing the wireless electrocardiograph device

An ECG device enables diagnosis of heart disease by recording the quantized electrical signal of heartbeats. An ECG device with 12 leads (standard leads, precordial leads and augmented limb leads) is, generally, employed in hospitals to obtain the heartbeat signal from patients at a minimum sampling rate at 250 Hz. An ECG with three standard leads (Lead I, Lead II and Lead III) is normally considered sufficient for an initial diagnosis and was adopted in this study.

5.4.1.1 Hardware configuration

According to the description given in Phatrapornnant and Pont’s paper (Phatrapornnant and Pont, 2006), the wireless ECG was reconstructed for this study. It was again implemented by the Ashling evaluation board mounted with an ARM7 LPC2106 microcontroller. An external 12-bit ADC was connected to the microcontroller through a serial peripheral interface bus (SPI). It was used to sample electrical signals from the three standard leads at 500Hz.

The sampled data were transmitted periodically using the “RS-232” protocol through a ‘HandyCore’ Bluetooth communication module (HandyWave USA, version 2.0) to a host computer for display. For efficient data transfer, the communication speed between the Bluetooth module and the host computer was set to 115200 bps. The Bluetooth module was connected to the LPC2106 microcontroller through a SPI bus.
For the processor frequency scaling, the phase-locked loop (PLL) in the LPC2106 microcontroller allows the processor clock frequency to be changed by modifying the PLL multiplier. The PLL multiplier has six levels from 1 to 6. With a 10 MHz external crystal oscillator, the clock frequency of the processor can be scaled from 10 MHz to 60 MHz respectively. The PLL multiplier was modified at the start of each tick according to a look-up table which stored a list of pre-defined PLL multiplier values associated with different ticks.

In different clock frequencies, the processor requires different supply voltage levels. The voltage levels were determined empirically by measuring the minimum voltage at which the processor still worked properly and then adding a 10 percent safe margin. The processor frequency and supply voltage mapping is shown in Table 5.1.

<table>
<thead>
<tr>
<th>CPU frequency (MHz)</th>
<th>10</th>
<th>20</th>
<th>30</th>
<th>40</th>
<th>50</th>
<th>60</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU supply voltage (V)</td>
<td>1.021</td>
<td>1.032</td>
<td>1.065</td>
<td>1.141</td>
<td>1.252</td>
<td>1.373</td>
</tr>
</tbody>
</table>

*Table 5.1: The CPU frequency and supply voltage mapping of the ECG device*

The processor supply voltage was also controlled by the processor itself. At the start of each tick, the processor will send a command signal to an external digital to analogue converter (DAC) through a SPI bus to request for a specific processor supply voltage for the current tick. While the processor is performing frequency scaling, the DAC generates a reference voltage according to the command signal to control a DC-to-DC converter to provide the required voltage to the processor. The whole frequency/voltage scaling process takes around 200 microseconds
5.4.1.2 The scheduler settings

The wireless ECG device in this study employed a TTC scheduler to schedule five different tasks to execute. The task model of the ECG device is shown in Table 5.2. The tick interval was set to 1 millisecond.

<table>
<thead>
<tr>
<th>Task</th>
<th>Offset (tick)</th>
<th>Period (tick)</th>
<th>WCET (μs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data acquisition</td>
<td>0</td>
<td>2</td>
<td>70</td>
</tr>
<tr>
<td>Data transfer</td>
<td>1</td>
<td>2</td>
<td>400</td>
</tr>
<tr>
<td>Read control switch status</td>
<td>0</td>
<td>100</td>
<td>20</td>
</tr>
<tr>
<td>Wireless connection check</td>
<td>0</td>
<td>200</td>
<td>20</td>
</tr>
<tr>
<td>Output to system status LED</td>
<td>0</td>
<td>400</td>
<td>30</td>
</tr>
</tbody>
</table>

**Table 5.2: The task set of the wireless ECG device**

The WCET of tasks were measured at the maximum processor clock frequency (60 MHz). The WCET of tasks are important for the estimation of the processor clock frequency and supply voltage in each tick to ensure that the processor is sufficient to complete all task executions before the deadline.

In this task set, only 10 MHz, 20 MHz and 50 MHz processor clock frequencies were used in every major cycle (400 ticks). Specifically,

- 50 MHz processor speed was used for executing the “Data transfer” task,
- 20 MHz processor speed was used at the first tick of every major cycle to execute other 4 tasks,
- 10 MHz processor speed was used at the rest of the ticks in a major cycle.

It is noticeable that the “Data transfer” task had the WCET of 400 microseconds at 60 MHz processor speed, and the tick interval was set to 1000 microseconds.
At 30MHz processor speed, the WCET of the task is equal to ~800 microseconds which would be sufficient for the task to meet the deadline. However, the scheduler overhead and the time required for frequency and voltage scaling consume a large amount of computational time (~200 microseconds). This has been mentioned in Section 5.4.1.1. Thus, it was appropriate to choose 50 MHz processor speed to execute the task.

In Section 4.7, the issue related to the common deadline of tasks in TTC scheduling was discussed. This issue will lead to a false alarm triggered by the NISA module when the completion of task executions is very close to the deadline. In order to overcome this issue, the common deadline in each tick was placed at 15 microseconds before the start of next tick. This gave sufficient time for the processor to enter into the idle state under normal operations.

5.4.2 Software-based monitoring techniques

Software-based monitoring approach has been widely discussed in literature (see Section 3.2.1). In this case study, the proposed monitoring system was evaluated against two different software-based monitoring techniques.

5.4.2.1 A flag-driven software monitor

The first software-based monitoring technique was implemented by inserting a flag (instrumentation codes) into the software of the ECG device. Figure 5.3 illustrates the operation of this monitoring technique. As the figure shows, the state of this flag is “set” before task scheduling at the start of each tick, and “unset” before the processor entering to the idle state. The flag is examined in the
start of each tick to determine if a deadline has been violated in the previous tick. As the flag is set at \( T_1 \), a violation of deadline is detected. In response to the violation, the software monitor triggers a routine to send the related information (such as the tick number) to the host computer through the RS232 interface.

\[ T_0 \quad T_1 \quad T_2 \quad T_3 \quad T_4 \quad \text{Time} \]

A violation of deadline is detected!!

**Figure 5.3: A flag-driven software-based monitoring technique**

### 5.4.2.2 A timer-driven software monitor

Software-based monitoring technique using a timer-driven approach to detect task overrun events has been described in literature (Pont and Ong, 2002; Harbour and Rivas, 2003). The second monitoring system in this case study adopted this approach as shown in Figure 5.4. In each tick, a timer starts before task executions, and stops either after the completion of task executions before the deadline (i.e. no deadline is missed) or when the timer interrupt occurs at the deadline (i.e. the deadline is missed). An occurrence of the timer interrupt in a tick indicates that the deadline in this tick has been detected. This situation can be seen at the end of tick \( t \) in Figure 5.4 where the timer interrupt occurs at the deadline. This activates the interrupt service routine to send information related...
to the violation to the host computer through the RS232 interface. This technique is often described as a “software watchdog”.

5.4.3 Evaluation and results

The evaluations were intended to simulate human errors in the ECG device for the purpose of testing the monitoring techniques described in this chapter. In the first evaluation, a timing fault was injected to a selected task which required a high processor speed to execute. This was done by reducing the processor clock frequency to 10 MHz at the tick where the selected task was scheduled to run. In this case study, the “Data transfer” task in Table 5.2 was selected.

In the second evaluation, a power failure caused by insufficient processor supply voltage was simulated: the processor supply voltage was reduced at the voltage scaling step to halt the processor.

Figure 5.4: A timer-driven software-based monitoring technique
5.4.3.1 Comparison of memory requirement

In both evaluations, the instrumentation codes for the software monitoring systems were inserted into the faulty ECG device in turn after the evaluation of the NISA-DVS monitor. The memory requirement for the wireless ECG device for each monitoring technique is shown in Table 5.3.

The memory requirement of the ECG device was increased by 2% when the flag-driven software-based monitor was employed. When the timer-driven monitor was employed, the memory requirement of the ECG device was increased by 3.6%. The memory requirement of the ECG device remained the same when the NISA-DVS monitor was employed.

<table>
<thead>
<tr>
<th></th>
<th>Memory requirement (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>The ECG device</td>
<td>18148</td>
</tr>
<tr>
<td>The ECG device with the NISA-DVS monitor</td>
<td>18148</td>
</tr>
<tr>
<td>The ECG device with the flag-driven monitor</td>
<td>18512</td>
</tr>
<tr>
<td>The ECG device with the timer-driven monitor</td>
<td>18796</td>
</tr>
</tbody>
</table>

Table 5.3: Memory requirement for the wireless ECG device

5.4.3.2 Detecting a deadline violation

In the first evaluation, the selected task – the data transfer task in Table 5.2 – failed to meet the deadline when the processor clock frequency was dropped to 10 MHz. This created a “domino effect” leading to the tasks in the following ticks to miss the deadline. This situation was detected by all monitoring techniques in this case study. However, these three monitoring techniques had different response times:
- The flag-driven monitor detected the violation at the subsequent tick which was approximately 18.5µs after the violation occurred,
- The NISA-DVS detected the violation at the end of the tick which was approximately 15µs after the violation occurred, and
- The timer-driven monitor (software watchdog) detected the violation at the deadline which was approximately 1µs after the violation occurred.

5.4.3.3 Detecting a power failure

In the second evaluation, the only technique which was capable to report the failure was the NISA-DVS technique. All software-based monitoring techniques in this evaluation did not report any error messages to the host; since the processor of the monitored system had halted, hence error reporting was impossible.

5.5 Discussion

This case study provided evidence that the NISA-DVS technique has a distinct advantage when compared to the software-based monitoring techniques in term of the ability to operate (and detect) when the processor of the monitored system has failed. This ability is important which allows the NISA-DVS technique to trigger a recovery mechanism, such as a system reset or switching to a backup system, when the processor of the monitored system is halted.

Although the time for the NISA-DVS technique to respond a deadline violation was approximately 14 µs slower than the software watchdog in this case study,

---

2 As the CPU operation stopped, no power would be consumed by the CPU and - therefore - a rising edge could not be found. It would also be possible to identify this error, since the power measurement of the CPU in this error is less than the power measurement when the CPU is idling.
this does not mean that timing error detection based on the processor power consumption is slower than software-based monitoring techniques. In fact, the detection mechanism of the NISA-DVS technique can be modelled like a watchdog. This will make the response time of the NISA-DVS technique in line with the software watchdog. This enhancement will be further discussed in Section 6.4.1.

Nevertheless, the NISA-DVS technique inherits the limitation of the NISA technique (described in Section 4.7) which is incapable to identify an overrun task in a task chain. Software-based monitoring techniques, on the other hands, have more advantages in this aspect. Instrumentation codes can be easily inserted between task executions for monitoring purposes. However, substantial increase of memory and computational demands can be envisaged when additional software monitoring features are integrated into the system. This would not be suitable for systems which have a high utilization task set, or significant overheads like DVS systems.

As discussed in 3.3.2, hybrid monitoring approach can complement the disadvantages of software-based monitoring and hardware-based monitoring. If hybrid monitoring approach is employed, more run time information about task executions can be obtained. Therefore, the limitation of the NISA and NISA-DVS techniques can be overcome. However, it may have a negative effect on the monitored systems; since instrumentation codes still required to be inserted into the monitored system’s software.
This possibility to integrate the non-invasive monitoring technique into a hybrid monitoring approach and minimise any negative effect on the monitored system will be considered in the following chapters of this thesis.

5.6 Conclusion

This chapter presented an extension of the novel non-invasive safety agent (NISA) technique. This extension, or the “NISA-DVS” technique, is designed to detect deadline violations in time-triggered co-operative (TTC) embedded systems which employ a static dynamic voltage scaling (DVS) scheme. Like the NISA technique, the violation detection of the NISA-DVS technique is achieved based on the monitored system’s task schedule – provided at design time – and instantaneous power consumption of the processor – measured at run time.

Despite the processor power measurement of the TTC-DVS system frequently fluctuates in various voltage levels, the NISA-DVS technique can cope with such dynamical changes and detect deadline violations if they occur in the monitored system. This ability has been demonstrated in the case study where a realistic test bed – a 3-channel ECG device – was employed.

The NISA-DVS technique has a distinct advantage, which is inherited from the NISA technique, in detecting power failures or other hardware faults that can halt the operation of the processor in the monitored system. This ability cannot be achieved by any software-based monitoring technique, as illustrated in the case study. The next chapter will present a novel jitter-reduction technique and the predictable time-triggered scheduling.
Chapter 6

The predictable time-triggered co-operative framework

The benefit of employing a time-triggered co-operative (TTC) scheduling algorithm has been repeatedly emphasized in this thesis. However, a task jitter problem discussed in Section 3.3.1 makes it impossible to maintain a low-level of task jitter for multiple jitter-sensitive tasks. This chapter presents the predictable time-triggered co-operative (pTTC) framework which can address this problem and allow time-based errors to be detected. It begins with a detailed description of a novel jitter-reduction technique which is proposed to address the task jitter problem. The predictable time-triggered co-operative (pTTC) scheduling algorithm is then described. Followed by this, the run-time monitor of the pTTC framework is presented. A case study to assess the jitter level of the pTTC framework against other jitter-minimisation techniques, and an evaluation of the error detection ability of the pTTC framework are also presented. The results are discussed before the conclusion of this chapter.

6.1 Introduction

In Section 2.3, it was discussed that the temporal uncertainty of time-triggered scheduling algorithms was due to variations in task start times, task execution times and interrupt latency. For example, variations in the execution time of a task can cause release jitter in subsequent task executions. This jitter problem can be addressed by using the sandwich delay technique or the code balancing CB1 technique; since such task jitter is introduced at task level. However, when task
release jitter is introduced at scheduling level, such as the jitter problem discussed in Section 3.3.1, jitter minimisation techniques that operates at task level cannot cope with. Thus, a mechanism to minimize task release jitter for this problem is essential. In this chapter, a novel jitter-reduction technique that operates at scheduling level is proposed for this purpose.

A predictable time-triggered co-operative scheduling algorithm – i.e. the pTTC algorithm – can be derived by applying the proposed jitter-reduction technique in conjunction with the sandwich delay (SD) technique on a TTC scheduling algorithm. The pTTC algorithm can schedule real-time tasks in an extremely temporal predictable manner which is particularly suitable for embedded applications with jitter-sensitive tasks, such as high-precision data acquisition applications. With the pTTC run-time monitor, the pTTC framework is established. If a run-time error occurs in the pTTC system, the pTTC monitor can detect it at a time resolution of microseconds.

### 6.2 A novel jitter-reduction technique

The proposed jitter-reduction technique for the jitter problem described in Section 3.3.1, involves deferment of task executions within the same tick. The deferred start time of jitter-sensitive (JS) tasks can be determined at design time using equation (6.1) iteratively from the lowest priority JS task.

\[
D_{\text{Start}}_{tm} = \begin{cases} 
D_{\text{Start}}_{tn} - WCET_{tm}, & \forall \tau_m, \exists \tau_p : (\tau_m, \tau_p, \tau_p \in \Phi_t) \land (m < n < p) \\
I - WCET_{tm}, & \text{otherwise}
\end{cases}
\]  

(6.1)
where $DStart_{\tau_m}$ is the deferred start time of a JS task $\tau$, $I$ is the tick interval, $m, n$ and $p$ are priority values, $WCET_{\tau_m}$ is the worst-case execution time of $\tau$, $\Phi_i$ is the set of JS tasks which are released at tick $i$, $\tau_n$ has the highest priority in the set of JS tasks ($\gamma_i$) which have lower priority than $\tau_m$, i.e. for any $\gamma_i \in \Phi_i$ and for any $\psi_i \in \gamma_i : (\forall \tau_n \in \gamma_i (\exists \tau_m \in \Phi_i \land \tau_m \notin \gamma_i) : m < n) \land (\forall \tau_p \in \psi_i (\exists \tau_n \in \gamma_i \land \tau_m \notin \psi_i) : n < p) \rightarrow m < n < p$.

As the start of a JS task can be deferred up to the point that is equal to the tick interval minus its WCET, higher priority tasks cannot interfere with the execution of the JS task. This is certainly true in time-triggered co-operative scheduling; since the sum of the WCETs of tasks released in a tick must be smaller than the tick interval to avoid a violation of deadline. In addition, lower priority jitter-insensitive (JI) tasks at the same tick can utilise the original time slots of the JS tasks to reduce impact on the overall schedulability.

In Section 3.3.1, we have shown that the TTC scheduling equipped with the sandwich delay technique failed to minimise the released jitter for the jitter-sensitive task ($\tau_3$) in the task set shown in Table 3.2. Figure 6.1 illustrates the ability of the proposed jitter-reduction technique using this task set. The figure shows that all instances of the jitter-sensitive task ($\tau_3$) are deferred to start according to equation (6.1). The lower-priority jitter-insensitive task ($\tau_4$) is allowed to execute before Task $\tau_3$. As the start of Task $\tau_3$ no longer depends on the completion of other tasks at the same ticks, it can execute at the same point in tick every period. Accordingly, the proposed jitter-reduction technique overcomes the task jitter problem that is introduced at scheduling level and
depicted in Figure 3.7 of Section 3.3.1. It is important to note that scheduling overhead is ignored in this section for ease of explanation.

6.3 The predictable time-triggered co-operative algorithms

In Section 6.2, we have shown the proposed jitter-reduction technique to address the jitter problem that is introduced at scheduling level. However, this technique cannot solve task jitter that is introduced at task level. In order to achieve extremely predictable temporal behaviour, we introduce the predictable time-triggered co-operative (pTTC) algorithm.

As briefly mentioned in Section 6.1, a pTTC scheduling algorithm can be derived by applying the proposed jitter-reduction technique in conjunction with the sandwich delay technique on the TTC scheduling algorithm. Deriving the pTTC scheduling algorithm must first apply the sandwich delay technique on all tasks in the TTC system.

As described in Section 3.1.2, when applying a sandwich delay to a task, the WCET of the task must be overestimated, typically added with a safe-margin.
This prevents execution jitter produced by the task to affect the start of a subsequent task. By substituting the sandwich delay time to the WCET of a task in equation (6.1) and calculating the start time for each task in the system, the pTTC scheduling is established.

Figure 6.2 shows the pTTC scheduling algorithm to schedule the task set shown in Table 3.2 of Section 3.3.1. Like the proposed jitter-reduction technique, all instances of the jitter-sensitive task (Task \(\tau_3\)) are deferred to start. The lowest priority jitter-insensitive task (Task \(\tau_4\)) is allowed to execute before Task \(\tau_3\). The figure also shows that all tasks in the system are “wrapped” individually by a sandwich delay. This minimised task release jitter introduced at task level. For example, Task \(\tau_2\) that always executes after Task \(\tau_1\) is no longer affected by Task \(\tau_1\).

As Task \(\tau_4\) is jitter insensitive, it is not necessary to consider its jitter level since it can tolerate task jitter. However, if Task \(\tau_4\) is jitter sensitive, the start of Task \(\tau_4\) can be deferred using equation (6.1) to eliminate the release jitter. As such, the
Chapter 6 – The predictable time-triggered co-operative framework

pTTC scheduling algorithm can achieve extremely predictable temporal behaviour.

It is important to note that, as discussed in Section 3.1.2, a task overrun (i.e. task’s actual execution time (AET) is greater than its sandwich delay) can disrupt the task schedule when the sandwich delay technique is employed; since the timer interrupt to wake up the processor at the end of the sandwich will occur when the overrun task is still executing. At the completion of the overrun task, the processor will enter to the idle state. As no mechanism to wake up the processor within the tick, subsequent tasks at the same tick have to wait until the next tick interrupt occurs. Surely, the common deadline will be violated, even if subsequent tasks have enough time to complete their executions before the deadline. In order to prevent this situation, the wake-up time to mark the end of a task’s sandwich delay is calculated at the completion of the task execution\(^3\). If a task overrun occurs, the processor will not be set into the idle state. This will allow a subsequent task to start its execution after the overrun task at the same tick.

6.4 The predictable time-triggered co-operative monitor

The non-invasive monitoring approach discussed in Section 4.4 demonstrated the feasibility of utilising the processor power consumption to detect timing constraint violations in a TTC embedded system, but has certain limitations. This is because insufficient run time information of the system can be obtained from the processor power measurements. However, the non-invasive monitoring approach is ideal

\(^3\) A sandwich delay is usually set up before a task starts.
for embedded systems which employ a pTTC scheduling algorithm; since the extremely predictable behaviour of pTTC scheduling allows deterministic temporal information about individual tasks in the system to be monitored from the processor power consumption.

As shown in Figure 6.3, a short idle period (i.e. the processor power consumption is below the threshold) is always present between tasks in the power measurement of a pTTC system, due to task execution times are “safely” overestimated – i.e. tasks are individually wrapped by a sandwich delay. With the task schedule in hand, the pTTC monitor can identify information about the start (a rising edge in the power measurement) and the end (a falling edge in the power measurement) of individual tasks in the pTTC system.

![Figure 6.3: The processor power signal of a predictable time-triggered co-operative system](image)

### 6.4.1 Detecting time-based errors

The pTTC monitor is designed for embedded systems that employ pTTC scheduling. With a priori knowledge of the task schedule and tasks’ timing information of a pTTC system, the monitoring process of the pTTC monitor can
be carried out straightforwardly and non-intrusively at run time. As 
aforementioned, a short idle period is always present between tasks. Therefore, 
the start and the end of a task can be identified by scanning the processor power 
measurement for the rising edge and the falling edge corresponding to the task 
execution.

As shown in Figure 6.4, the pTTC monitor first scans for a rising edge in the 
processor power measurement of the pTTC system to locate the start of a tick. 
Once the rising edge is found, it evaluates the next task to be run according to a 
_priori_ knowledge of the pTTC system. After the next task is evaluated, the 
monitor will start scanning for a falling edge at the point in time before the BCET 
of the task is reached. An absence of a falling edge in the period of time between 
the BCET and the WCET of the task indicates an occurrence of a task overrun. 
On the other hand, a presence of a falling edge (or the processor power 
consumption is below the threshold) before the BCET of the task is also 
considered as a timing fault. When a falling edge is found, the monitor will 
evaluate the next task to be run and perform the same operations again. In the 
case when no task is due to run, the monitor will set an interrupt at the time before
the end of the current tick. It will scan for a rising edge for the start of next tick when the interrupt occurs.

The synchronisation between the pTTC monitor and the pTTC system is achieved periodically when a rising edge for the start of a tick is detected. As the monitor has to keep track of the monitored system’s tick number, the increment of the tick number in the monitor can be performed after the synchronization. Additionally, the jitter level of the monitored system’s tick interval can also be measured at the same time if it is a concern.

The mechanisms in the pTTC monitor to detect a task overrun and a deadline violation can be modelled like a “watchdog timer” which must be fed before specific points. For instance, two timer interrupts in the pTTC monitor can be enabled when the start of a task is detected. These two interrupts are set to occur respectively at the end of the task’s sandwich delay and at the “violation point” (VP) where a deadline violation is anticipated. These two interrupts will be disabled if a falling edge is detected before the first interrupt (i.e. at the end of the task’s SD) occurs. Otherwise, the first interrupt will occur. This indicates a task overrun. Similarly, if there is no falling edge found before the VP, the second interrupt will occur which indicates that a deadline violation is detected or anticipated.

It is important to note that a released task will execute immediately after a task overrun, as described in Section 6.3. This means that the pTTC monitor cannot determine the end of the overrun task and the start of the released task. In order to
allow the monitor to maintain a deterministic view of task behaviour, a minor adjustment of the pTTC scheduling algorithm is required: a short period of idle time ($\Delta$) is inserted after the completion of an overrun task. This will allow the monitor to detect the start time of the task after a task overrun. The VP of a task in each tick can be computed at the start of the task using equation (6.2).

$$VP_{t_i} = I - (\delta + \sum_{j \in \Gamma} SD_{t_j})$$ \hspace{1cm} (6.2)

where $I$ is the tick interval and $\Gamma$ is the set of tasks which are released at the present tick but have not executed, $\delta$ is a small integer time, $\delta < \Delta$ and $\Delta$ is an idle time that will added to the end of an overrun task.

The rationale behind equation (6.2) is that each task in the pTTC system has a fixed timeslot to execute as it is “wrapped” by a sandwich delay. Thus, the time required for all released tasks in a tick to complete is known. The computation of task violation points in equation (6.2) is virtually placing all released tasks from the end of the tick, as shown in Figure 6.5, and setting the violation point for each task at the end of its sandwich delay minus a small integer time ($\delta$). Since $\delta$ is smaller than the idle time ($\Delta$) which will be added after an overrun task, a task execution reaching its violation point at run time means that the sandwich delay time of the last task in the task chain will exceed the deadline. Although the last task could finish within its sandwich delay, the slack time within the sandwich delay is also considered as part of the task execution; since no task is allowed to execute in the sandwich delay of other tasks.
It is important to emphasize that the violation point calculation in this study is an exact and direct calculation without assumptions. Any task execution reaching the violation point, a deadline violation is guaranteed to occur.

In addition to the detection of deadline violations, monitoring the jitter level for a jitter-sensitive task is also possible; since tasks are released only at the beginning of a tick, and the start time of a task can be obtained by taking a timestamp at the rising edge when the task starts. The release jitter for the task, therefore, can be calculated. Also, the start times of two successive instances of the task can be used for evaluating the period jitter, but it is necessary to take the tick interval jitter into account; since it will affect the period of the task even if release jitter is absence.

### 6.5 Case study and evaluation

The evaluation process for the pTTC framework was divided into two parts: (1) the evaluation of task jitter level, and (2) the evaluation of the ability of run-time error detection of the framework.

In the first part of the evaluation, the proposed pTTC scheduling in the framework was evaluated against the TTC scheduling, the TTC scheduling equipped with the
CB1 technique and the TTC scheduling equipped with the SPP. These four techniques were implemented in turn on a hardware platform, and assessed using a case study. The task set in the case study generated a high level noise to examine if the pTTC algorithm and other jitter-minimisation techniques can maintain a low level of jitter for a jitter-sensitive task in the system. In the second part of the evaluation, the pTTC system in the first part of the evaluation was re-used and injected with timing faults to assess the error detection ability of the pTTC monitor. The assessment of this evaluation is based on a simulation using the processor power measurement of the faulty pTTC system.

### 6.5.1 Hardware platform setup

The hardware platform for the evaluation comprised an Ashling EVAB7 evaluation board on which was mounted an NXP LPC2106 32-bit ARM7-based microcontroller (NXP Semiconductors, 2003a). The processor speed was set to 60MHz. In the first part of the evaluation, the jitter level of each implementation of the jitter minimisation techniques was measured using a 20MHz counter/timer in a National Instruments PCI-6035E data acquisition (DAQ) card (National Instruments, 2000). In the second part of the evaluation, the processor power measurements of the pTTC system was obtained using the 200 KHz analogue signal sampler of the DAQ card from a sampling resistor that was connected in series with the power input to the processor, as described in Section 4.6.3.

### 6.5.2 Case Study: bubble sort

The sorting time required for a computer to process a set of elements into a specific order is determined by number of factors, such as the number of elements
to be sorted and the degree of sorting complexity. The bubble sort algorithm employed in this case study varies these two factors to generate a high level of task release jitter at run time.

<table>
<thead>
<tr>
<th>ID</th>
<th>Task</th>
<th>Offset (tick)</th>
<th>Period (tick)</th>
<th>Jitter sensitivity</th>
<th>BCET (µs)</th>
<th>WCET (µs)</th>
<th>Sandwich delay (µs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Sort complexity</td>
<td>0</td>
<td>2</td>
<td>N</td>
<td>37.70</td>
<td>38.50</td>
<td>80</td>
</tr>
<tr>
<td>1</td>
<td>Sort length</td>
<td>0</td>
<td>2</td>
<td>N</td>
<td>2.00</td>
<td>2.35</td>
<td>40</td>
</tr>
<tr>
<td>2</td>
<td>Bubble sort</td>
<td>1</td>
<td>2</td>
<td>N</td>
<td>6.90</td>
<td>259.80</td>
<td>300</td>
</tr>
<tr>
<td>3</td>
<td>Jitter test</td>
<td>1</td>
<td>2</td>
<td>Y</td>
<td>82.85</td>
<td>82.90</td>
<td>130</td>
</tr>
</tbody>
</table>

Table 6.1: The task set of bubble sort – adapted from (Gendy and Pont, 2007)

Table 6.1 shows the task set of the bubble sort algorithm where the number of elements to be sorted and the degree of sorting complexity are respectively controlled by Task 0 (“Sort complexity”) and Task 1 (“Sort length”). Specifically, Task 0 arranges the elements to be sorted either in fully sorted or in fully unsorted. Task 1 changes the number of elements to be sorted incrementally from a low value to a high value and then start over again. These changes will cause large variations in the execution time of the bubble sort task (Task 2) at run time. As a result, the subsequent task – Task 3 (“Jitter test”) – will suffer from a high level of release jitter.

This task set enabled the evaluation of the proposed pTTC scheduling algorithm versus the code balancing (CB1) technique and the single-path programming paradigm (SPP) technique in term of the jitter minimisation ability. We, in turn, applied these techniques to the TTC system to schedule this task set at 1 millisecond tick interval and measured the release jitter of Task 3. Several
experiments had been performed in which different numbers and values of elements were selected to generate different amount of release jitter to Task 3. This section presents a representative result of these experiments where 10 elements with the values from 2 to 9 were selected in this experiment. It is worthwhile to note that different implementations and configurations of this task set were employed in other literature (Puschner and Burns, 2002b; Gendy and Pont, 2007) for the evaluation of the SPP and the CB1 techniques.

---

**Figure 6.6: A comparison of different jitter minimisation techniques**

Figure 6.6 shows the start times of 10000 invocations of Task 3 scheduled by (a) The TTC scheduling, (b) The TTC scheduling with the CB1 technique (TTC_CB1), (c) The TTC scheduling with the SPP technique (TTC_SPP) and (d)
The proposed pTTC scheduling. As we expected, under the TTC scheduling, Task 3 suffered from significant release jitter where the start time variance equal to 4984.17, due to variations in the execution time of Task 2. The release jitter of Task 3 was improved when using the TTC_CB1 scheduling, where the start time variance was reduced to 817.03. The start time variance of Task 3 was further reduced to 57.99 in the TTC_SPP scheduling. This confirms that the SPP technique has better jitter-reduction ability than the CB1 technique, which concurs to the results presented in Gendy’s thesis (Gendy, 2009). However, the release time of Task 3 was still influenced by Task 2. The proposed pTTC scheduling eliminated the influence of Task 2 by scheduling Task 3 – based on equation (6.1) in Section Figure 6.2 – to start execution at a pre-determined (fixed) point in time that was greater than the completion time of Task 2. Figure 6.6 (d) shows that, under the pTTC scheduling, the start time variance of Task 3 was 0, and hence the release jitter of Task 3 was minimised.

With the reference to Table 6.2 where the absolute release jitter levels of the jitter-reduction techniques are reported, the release jitter level of the TTC scheduling was reduced by 64.09% when the code balancing (CB1) technique is employed, 90.8% when the single-path programming paradigm was employed and 99.98% when the pTTC scheduling was employed. Clearly, the pTTC scheduling has an outstanding jitter-reduction ability.

It is noticeable that the start of Task 3 was also delayed in both the TTC_CB1 scheduling and the TTC_SPP scheduling. This was due to the “side-effect” that substantial overhead was inevitably introduced to Task 2. With reference to Table
6.3, the execution time of Task 2 was increased by 107.8 times at the best case and 3.2 times at the worst case when Task 2 was scheduled by the TTC_CB1 scheduling. When it was scheduled by the TTC_SPP scheduling, the execution of Task 2 was increased by 113.7 times at the best case and 3.1 times at the worst case. Unlike the CB1 technique and the SPP technique that operate on task-level, the proposed pTTC scheduling addresses the task release jitter problem on scheduling level which has no impact to the task execution times.

<table>
<thead>
<tr>
<th>Scheduling algorithm</th>
<th>Absolute release jitter (µs)</th>
<th>Start time variance (σ²)</th>
</tr>
</thead>
<tbody>
<tr>
<td>The TTC algorithm</td>
<td>252.85</td>
<td>4984.17</td>
</tr>
<tr>
<td>The TTC_CB1 algorithm</td>
<td>90.8</td>
<td>817.03</td>
</tr>
<tr>
<td>The TTC_SPP algorithm</td>
<td>23.25</td>
<td>57.99</td>
</tr>
<tr>
<td>The pTTC algorithm</td>
<td>0.05</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 6.2: A jitter level comparison of different jitter-reduction techniques

<table>
<thead>
<tr>
<th>Scheduling algorithm</th>
<th>Memory consumption (Bytes)</th>
<th>The best case and the worst case execution times of Task2 (µs)</th>
<th>Scheduler overhead (µs) in one tick</th>
</tr>
</thead>
<tbody>
<tr>
<td>The TTC</td>
<td>2870</td>
<td>6.90, 259.80</td>
<td>39.65</td>
</tr>
<tr>
<td>The TTC_CB1</td>
<td>3862</td>
<td>743.55, 834.35</td>
<td>39.65</td>
</tr>
<tr>
<td>The TTC_SPP</td>
<td>3294</td>
<td>784.65, 807.90</td>
<td>39.65</td>
</tr>
<tr>
<td>The pTTC</td>
<td>4042</td>
<td>6.90, 259.80</td>
<td>85.30</td>
</tr>
</tbody>
</table>

Table 6.3: A comparison data of different jitter-reduction techniques
Although the overhead of the pTTC scheduler was larger than the TTC scheduler’s by 2.15 times in this particular case study, it was negligible when compared with the impact on the task execution times in the TTC_CB1 scheduling and the TTC_SPP scheduling. This can be illustrated in a situation when tasks need to be added into the system. Figure 6.7 shows that the pTTC scheduling was capable to maintain its jitter reduction ability and satisfy all real-time constraints when a periodic task (Task 4) – with the same offset and period
as Task 2 and Task 3 and the WCET equal to 60.5 µs – was added. Whilst both the TTC_CB1 scheduling and the TTC_SPP scheduling failed to meet the timing requirements, i.e. task overruns occurred. However, in this case study, the pTTC scheduling required 5% and 22% more memory space than the CB1 technique and the SPP technique, respectively. Table 6.2 and Table 6.3 summarize the above-described evaluation data of the TTC, the TTC_CB1, the TTC_SPP and the proposed pTTC scheduling algorithms.

6.5.3 Evaluation of the predictable time-triggered monitor

This section presents the evaluation of the predictable time-triggered (pTTC) monitor described in Section 6.4. The focus of this evaluation was on three aspects: (1) task-overrun detection, (2) detection of deadline violations, and (3) task jitter monitoring. It was undertaken using a simulation based on the processor power measurements of the task set shown in Figure 6.7. This task set is equivalent to the task set shown in Table 6.1 plus an addition task (Task 4). The parameter values of Task 4 have been described in Section 6.5.2.

6.5.3.1 Task overrun detection

For the task overrun detection in the pTTC system, we simulated a transient error situation: without re-estimation of Task 2’s sandwich delay time, the WCET of Task 2 (the Bubble Sort Task) was lengthened by adding one more element into the sorting list. In the worst case, Task 2 will overrun its sandwich delay time, but not lead to a violation of the deadline.
It is important to note that the pTTC monitor needs to start before the monitored system to allow the state of the system to be monitored consistently. As shown in Figure 6.8, it scans the processor power measurement of the monitored system. After the initialisation power profile is found, it searches for the rising edge that represents the start of the first tick (Tick 0). Once the start of a tick is detected, it determines if a task is due to run based on the task schedule. If no task is due to run, the monitor will enable an interrupt at the time before the arrival of next tick, and then enter into the idle state. It will wake up at the interrupt point and scan for the start of next tick. If a task is due to run, the monitor will enable two timer interrupts. One of the interrupts will be enabled to occur at the end of the task’s sandwich delay for the detection of a task overrun. This interrupt will be disabled when a falling edge that represents the end of the task is found, normally before it occurs. Otherwise, a task overrun is detected. Another timer interrupt will be enabled to occur at the time before the BCET of the task for the start of scanning the falling edge. However, if the duration of the task is too short, the monitor will not enable the interrupt but continue to search for the falling edge.

The result of the evaluation to assess the task overrun detection ability of the pTTC monitor is shown in Figure 6.8. For ease of visualisation, we only show the interrupt points for the overrun detection of Task 2. As the figure shows that a task overrun was detected in Tick 39; since no falling edge that represents the end of Task 2, was found before the interrupt for the task overrun detection.
6.5.3.2 Monitoring task jitter levels

Monitoring task jitter levels for the jitter-sensitive task – Task 3 – was done by taking a time stamp when the rising edge of Task 3 was found. With reference to Figure 6.8, a rising edge for each instance of Task 3 was found at 870 microseconds after the synchronization between the system and the monitor in the pTTC framework. Hence, no task released jitter is detected for Task 3.
6.5.3.3 Detecting deadline violations

As mentioned in Section 6.4.1, the detection of a violation of deadline in the pTTC monitor employs the same mechanism as the detection of a task overrun. When a rising edge that represents the start of a task is detected, the pTTC monitor will enable a timer interrupt to occur at the point in time \((VP_{t_j})\) where a violation of the deadline is detected or anticipated. Akin to the detection of a task overrun, the timer interrupt for the detection of a deadline violation must be disabled before the \(VP_{t_j}\), or otherwise the violation is anticipated (or detected).

In this evaluation, we employed the same task set as the evaluation of task overrun detection, in which the execution time of Task 2 was lengthened to overrun its sandwich delay in Tick 39. In addition, the execution time of Task 4 in Tick 39 was also prolonged to make Task 3 to violate the deadline. The result of this evaluation is shown in Figure 6.9, where the \(VP_{t_2}\) (for Task 2) and the \(VP_{t_4}\) (for Task 4) were computed online using equation (6.2) described in Section 6.4.1. For example, the \(VP_{t_2}\) for each instance of Task 2 and the \(VP_{t_4}\) for each instance of Task 4 were calculated as follows:

For Task 2,

\[
VP_{t_2} = I - \left( \delta + \sum_{t_j \in F} SD_{t_j} \right) = 1000 - (40 + 130 + 130) = 700 \mu s
\]

for all instances of Task 2, where \(I = 1000 \mu s\), \(\delta = 40 \mu s\), \(SD_{t_2} = 130 \mu s\) and \(SD_{t_4} = 130 \mu s\)

For Task 4,

\[
VP_{t_4} = I - \left( \delta + \sum_{t_j \in F} SD_{t_j} \right) = 1000 - (40 + 130) = 830 \mu s
\]

for all instances of Task 4, where \(I = 1000 \mu s\), \(\delta = 40 \mu s\), \(SD_{t_3} = 130 \mu s\)

112
Figure 6.9 also shows that the interrupt for detecting the overrun of Task 2 occurred at tick 39, but the interrupt for $VP_{T2}$ did not occur; since a falling edge was found in the processor power measurement. However, the interrupt for $VP_{T4}$ occurred at 830µs in tick 39 indicated that the deadline would be missed.
6.6 Discussion

In the first part of the evaluation, the proposed pTTC scheduling had demonstrated its jitter reduction ability, in which the jitter level of the jitter-sensitive task was minimized in the case study. From the results, it is easy to recognize that the pTTC scheduling had better effect, in term of jitter-reduction, than the TTC_CB and the TTC_SPP scheduling algorithms. In addition, the proposed pTTC scheduling was capable to accommodate additional tasks and maintain the stability and jitter-level of the system; whilst the TTC_CB scheduling and the TTC_SPP scheduling failed to comply with the timing constraints when a new task was added to the system.

It is also noticeable that the TTC scheduling was unable to handle jitter-sensitive tasks in the case study. Although re-schedule the task set may help to overcome this problem in some situations, it may be difficult to find a feasible schedule which meets all timing requirements (as it is a NP-hard problem). Perhaps, it would be much easier to apply the pTTC scheduling on a task schedule which has satisfied all deadline constraints but not yet fulfilled the jitter requirements.

However, the pTTC scheduling requires more memory space and has more scheduling overhead than TT scheduling algorithms. This, as aforementioned, can be seen as a trade-off between schedulability and predictability of the system. As tasks’ start times are known a priori under the pTTC scheduling, it is more predictable and easier to be monitored at run time than tasks scheduled by the TTC scheduling.
In the second part of the evaluation, the simulation results show that the pTTC monitor was capable to detect task overruns and deadline violations (at a time resolution of microseconds).

Overall, the pTTC framework can maintain task jitter at a minimum level which makes the system behaviour more predictable. It also supports rapid detection of time-based errors at run time. However, the monitoring process of the pTTC framework, like other (intrusive) monitoring techniques, is limited by the availability of the monitored system’s processor. This means that the pTTC framework may not be applicable to real-time systems which already have very tight timing constraints (or high processor utilization). However, if system reliability is the primary concern and sufficient processor time is available, this framework may be more beneficial and desirable to some safety-related embedded designs than other real-time scheduling algorithms with a monitoring technique; since a monitoring technique often has a negative influence – rather than a positive influence like the proposed pTTC framework – on the timing predictability of the monitored system.

6.7 Conclusions

The work described in this chapter provided the predictable time-triggered co-operative framework, or the pTTC framework, for embedded system designs which require extremely predictable temporal behaviour and a rapid time-based error detection mechanism.
The pTTC scheduling algorithm in the framework has demonstrated much better effect than the code balancing (CB1) technique and the single-path programming paradigm (SPP) technique, in term of jitter reduction ability. It addressed the jitter problem discussed in Section 3.3.1 where the CB1 and the SPP techniques failed to cope with this problem. The pTTC monitor adds extra confidence to the pTTC framework where an occurrence of a time-based error can be detected in a time resolution of microseconds. It also allows jitter-level in the pTTC system to be monitored. This distinctive achievement of the pTTC framework is due to the extremely predictable temporal behaviour of the pTTC system that allows deterministic run-time information about task executions to be monitored.

Overall, the pTTC framework would be more beneficial and desirable to some safety-related embedded designs than other real-time scheduling algorithms with a monitoring technique; since a monitoring technique often has a negative influence – rather than a positive influence like the pTTC framework – on the temporal predictability of the system.

In the next chapter, we present the predictable time-triggered rate-monotonic framework.
Chapter 7

The predictable time-triggered rate-monotonic framework

Chapter 6 presented the predictable time-triggered co-operative (pTTC) framework which has an extremely predictable timing behaviour and allows time-based errors to be detected rapidly at run time. This chapter provides a study of the possibility to extend the proposed framework to a more complex time-triggered algorithm in order to support different software requirements. The time-triggered rate-monotonic (TTRM) algorithm described in Section 2.8 was selected for this study; since it supports task preemption. The predictable time-triggered rate-monotonic (pTTRM) framework is first presented followed by an evaluation and a case study to assess the pTTRM framework. The assessment was focused on examining the memory requirement, the ability of jitter-reduction and the ability of time-based error detection. The possibility for extending the proposed framework to other time-triggered algorithm is discussed based on the assessment results.

7.1 Introduction

The predictable time-triggered co-operative (pTTC) framework – presented in Chapter 6 – provides sufficient evidence that the framework is suitable for some safety-critical system design; since it supports critical jitter-sensitive tasks to operate at an extremely low-jitter level, and supports time-based error detection to enhance safety at run time. As the pTTC scheduling algorithm is derived from a TTC algorithm, it inherits the limitations of TTC scheduling. For instance, tasks
with duration greater than the tick interval are not allowed. A naïve solution which is to increase the tick interval to accommodate a long duration task may not be feasible; since it will block the executions of other tasks which have execution periods smaller than the duration of the task. Another solution is to divide the task into several small tasks and schedule them to execute in different ticks. This solution will increase the difficulty to obtain a feasible schedule\(^4\). Alternatively, using task preemption to temporarily suspend the long task when other tasks in the system are due to execute would be a better solution. However, time-triggered scheduling algorithms that support task preemption – such as the time-triggered rate-monotonic (TTRM) scheduling algorithm – are not less predictable as a TTC scheduling algorithm. In this chapter, we investigate if the proposed framework is generic enough to implement the TTRM scheduling algorithm in order to improve its temporal predictability.

7.2 The predictable time-triggered rate-monotonic scheduling

7.2.1 Handling jitter-sensitive tasks

Like the pTTC scheduling, the predictable time-triggered rate-monotonic (pTTRM) scheduling, requires the proposed jitter-reduction technique that has been described in Section 6.2. This technique involves deferment of jitter-sensitive (JS) tasks and assumes that:

(i) all tasks are independent,

(ii) all task periods \(T_{ri}\) are divisible by the tick interval \(I\),

(iii) the worst-case response time (WCRT) of a JS task \(\tau_m\) is less than or equal to the tick interval \(I\):

\(^4\) Finding a feasible schedule in TTC scheduling is a NP-hard problem.
\[ I \geq WCRT_{\tau_m} = (Start_{\tau_m} + WCET_{\tau_m}) - Release_{\tau_m}, \text{ and} \]

(iv) \[ I - WCRT_{\tau_m} \geq \sum_{\tau_n \in \Phi_j} WCET_{\tau_n}, \] where \( \Phi_j \) is the set of released JS tasks which has lower priority than the jitter-sensitive task \( (\tau_m) \) in tick \( j \) where \( \tau_m \) is also released.

Based on these assumptions, the start time of the deferred JS tasks can be computed iteratively from the lowest priority JS task at design time using equation (6.1) defined in Section 6.2. As the start of a JS task can be deferred up to the point that is equal to the tick interval minus its WCET, higher priority tasks cannot interfere with the execution of the JS task (in accordance to our assumptions). Also, lower priority jitter-insensitive (JI) tasks at the same tick can utilise the original time slots of the JS tasks to reduce the impact on the overall schedulability.

### 7.2.2 Integration of the jitter-reduction techniques

To derive the pTTRM scheduling algorithm, the sandwich delay technique must be applied to all tasks in the system individually before the proposed jitter-reduction technique can be applied. However, in TTRM scheduling, a long duration task may span across several ticks. Inevitably, the long task will be interrupted by the tick timer several times. This will cause interrupt latency variations which has been discussed in Section 3.1.4, and will lead to tick jitter. This will affect the temporal precision of task executions and will result in task jitter or even a violation of the deadline. For this reason, we adopted the “Planned Pre-emption” (PP) technique described in Section 3.1.4, in the pTTRM scheduling algorithm.
Unlike the TTRM scheduling described in Section 2.8 where task pre-emption requests are always examined at the interrupt service routine (ISR) after a tick timer interrupt occurs, tasks in the pTTRM scheduling may be pre-empted at a pre-determined point within a tick; since jitter-sensitive (JS) tasks are deferred to start, and lower-priority tasks are allowed to execute in the original timeslots of the deferred JS tasks. A lower priority task will be pre-empted when it reaches the start of a JS task. In order to prevent the interrupt latency jitter due to task pre-emption, we extended the idea of the PP technique to suspend the execution of a lower-priority task before a task pre-emption occurs. With respect to the task set in Table 3.3 and Figure 3.8 of Section 3.3.1, the concept of pTTRM scheduling is illustrated in Figure 7.1.

![Figure 7.1: The concept of predictable time-triggered rate-monotonic scheduling](image)

Figure 7.1 shows that all tasks scheduled by the pTTRM scheduling are “wrapped” individually by a sandwich delay. This minimised task release jitter introduced at task level. Also, the jitter-sensitive task (Task \( \tau_2 \)) is deferred to start at a fixed point in time every period by the proposed jitter-reduction technique.
Chapter 7 – The predictable time-triggered rate-monotonic framework

The long duration task (Task $\tau_3$) which has the lowest priority is allowed to execute in the original timeslot of Task $\tau_2$. However, Task $\tau_3$ is preempted when it reaches a pre-determined pre-emption point, such as before the start of Task $\tau_2$ and before the next tick interrupt. As Task $\tau_3$ is jitter insensitive, it is not necessary to consider its jitter level; since it has no effect on the system reliability. As such, the pTTRM scheduling algorithm, like the pTTC scheduling algorithm, can achieve extremely predictable temporal behaviour.

It is important to note that, as the sandwich delay technique is employed, a task overrun can completely disrupt the task schedule; since the processor will enter into the idle state after the wake-up timer interrupt occurs. The pTTRM scheduling employs the same approach as the pTTC scheduling to handle this problem: at the completion of a task, a timer interrupt to wake up the processor is enabled to occur at the end of the sandwich delay before the processor entering to the idle state. If a task overrun occurs, like the pTTC framework, a short period of idle time ($\Delta$) will be added after the completion of the overrun task. This allows the run-time monitor to maintain a deterministic view of task behaviour.

### 7.3 The predictable time-triggered rate-monotonic monitor

The basic principle for the predictable time-triggered rate-monotonic (pTTRM) monitor is identical to the pTTC monitor: rising and falling edges in the processor power consumption are utilised for determining the actual start and finish (or suspension) of tasks in accordance with the task schedule. The mechanisms to keep track of the tick number and perform synchronisation with the pTTRM
system are also the same as the pTTC monitor. It is necessary to note that scheduling overhead is ignored in this section for ease of explanation.

Although the task schedule in the pTTRM scheduling is known \textit{a priori}, it is important for the monitor to store the run time information of tasks (temporarily) at each tick; since the pre-determined start and end times of a task will no longer be valid if another task exceeds its sandwich delay, i.e. a task overrun occurs. The run time information of a released task required by the monitor includes (i) the tick number where the task actually starts and (ii) the remaining time for the task reaching the end of its sandwich delay. The tick number where a task actually starts can be obtained at the rising edge where the task starts. The remaining time for the task completion must be updated every time when the task is interrupted or preempted. These two run time information will be removed when the task fully completes its execution.

As shown in Figure 7.2, the pTTRM monitor starts scanning for a rising edge to locate the start of \textit{Tick x}. Once the rising edge is found, it will look at the list of the release tasks at the present tick and then compare the list with the current run-time time to evaluate the next ready-to-run task (i.e. Task $\tau_1$). By using equation (7.1), the start position for scanning a falling edge of the task can be obtained. The start scanning position is typical located at point in time before the BCET of the task or before the next preemption point.
Chapter 7 – The predictable time-triggered rate-monotonic framework

The \( pTTRM \) monitor then waits for the time reaching to the scanning point and starts scanning a falling edge. When the falling edge is found, it starts to search for a rising edge to locate the start of the next task (i.e. Task \( \tau_3 \)). When the rising edge is found, it uses equation (7.1) again to obtain the scanning point for a falling edge. As the BCET of the task is greater than the tick interval, the scanning point is at the start of the next pre-emption point. At the pre-emption point, the remaining time for Task \( \tau_3 \) is computed and stored. The first release tick number of Task \( \tau_3 \) is also stored. The monitor performs the same operations for the jitter-sensitive task (Task \( \tau_2 \)). At the next tick (Tick \( x+1 \)), Task \( \tau_3 \) is resumed. The

\[
\text{ScanFE}(\tau_i) =
\begin{cases} 
\text{Start}_{\tau_i} + R_{\tau_i} - (SD_{\tau_i} - \text{BCET}_{\tau_i}) - \Delta, & \text{if Start}_{\tau_i} + R_{\tau_i} - (SD_{\tau_i} - \text{BCET}_{\tau_i}) < P \\
\Delta, & \text{otherwise}
\end{cases} 
\] (7.1)

where \( R_{\tau_i} \) is the remaining time for the task to reach the end of its sandwich delay, \( \Delta \) is a short period of time and \( P \) is the next planned pre-emption point.
monitor uses equation (7.1) to obtain the scanning point and performs the same monitoring operations again. Until Task $\tau_3$ is resumed at Tick $x+3$, the monitor, again, uses equation (7.1) to obtain the scanning point. In this case, the monitor knows Task $\tau_3$ will finish at that tick and starts scanning at the point in time before its BCET. At the end of Task $\tau_3$, the run time information of Task $\tau_3$ is removed. The pTTRM monitor is then continuing to perform similar monitoring operations.

7.3.1 Time-based error detection

The pTTRM monitor uses the same overrun detection approach as the pTTC monitor. It is a timer interrupt-based approach where a timer interrupt is set to occur at the end of the sandwich delay of each task. In the situation when a task overrun is detected but the overrun task has not yet violated its deadline, the pTTRM monitor will continue to scan for a falling edge until reaching the point (VP) where a violation of (deadline or jitter) constraint is detected or anticipated. However, if a planned pre-emption occurs during the overrun task execution, the time consumed by the overrun task stored. On resumption of the overrun task, the monitor will continue to scan for a falling edge in the same manner as the above described.

The VP from the start (or resumption) of a task can be computed using equations (7.2), (7.3) and (7.4), where equation (7.2) is used for deriving the “earliest” time for the completion of the task which will make a lower priority task to violate its deadline. For example, three tasks – A, B and C – are scheduled to execute by the pTTRM algorithm (i.e. the deadline of a task equals to the period of the task) with tick interval 100 ms. The execution time of A, B and C are 50 ms, 30 ms and 140
ms, respectively. A’s period and B’s period are 100 ms, and C’s period is 200 ms. A’s priority is greater than B’s and B’s is greater and C’s. Task A, B and C are released at the first tick. Assume that A executes at time equal to 0 ms. It is obvious that the latest start time of B to meet its deadline is at time equal to 70 ms, and the latest start time for C to meet its deadline is at time equal to 60 ms. Therefore, the earliest time for the completion of A to prevent C to violate its deadline is 60 ms, despite A’s deadline is located at 100ms.

Equation (7.3) is used for deriving the start time of the jitter-sensitive (JS) task which has a lower priority and is the closest to the task that is under the estimation, and equation (7.4) selects the minimum value from the results of equations (7.2) and (7.3) as the VP of the task. Accordingly, the same mechanism for detection of deadline constraint violations described in Section 6.4.1 can be applied.

\[
f(\tau_i) = \begin{cases} 
\min \left( \min \left( T_{\tau_j} - \left( t^p - t^*_{\tau_j} \right) \times I \right) - \text{Start}_{\tau_i} - R_{\tau_i}, D_{\tau_i} \right), & \exists \tau_j \in \Gamma : j > i \\
D_{\tau_i}, & \text{otherwise}
\end{cases}
\]

\[
g(\tau_i) = \begin{cases} 
\min(\text{Start}_{\tau_k}), & \exists \tau_k \in \Psi : k > i \\
D_{\tau_i}, & \text{otherwise}
\end{cases}
\]

\[
VP_{\tau_i} = \min \left( f(\tau_i), g(\tau_i) \right) - \delta
\]

where \( \min \) is a function to select the smallest value from a set of numbers; \( i, j \) and \( k \) are the index of tasks, where the task index is inversely proportional to the task priority; \( \Gamma \) is the set of released tasks which have lower priority than \( \tau_i \); \( T_{\tau_j} \) is the period of \( \tau_j \); \( t^p \) is the present tick number; \( t^*_{\tau_j} \) is the tick number where \( \tau_j \)
starts; $I$ is the tick interval; $R_{\tau_j}$ is the remaining time for $\tau_j$ to complete its execution; $D_{\tau_i}$ is the deadline of $\tau_i$ and $D_{\tau_i} = PT$ (the planned pre-emption point of the tick where the deadline of $\tau_i$ is located); $\Psi$ is the set of released jitter-sensitive tasks which have lower priority than $\tau_i$; $Start_{\tau_k}$ is the start time of $\tau_k$; $\delta$ is a small integer and $\delta < \Delta$ where $\Delta$ is a small idle time period that will be added at the completion of an overrun task.

As discussed in Section 7.2.2, the pre-determined start times for task executions will become invalid after an occurrence of a task overrun. However, task information in each tick (for example, the number of released tasks and the task execution order) can still be estimated based on the given task parameters (such as the period and sandwich delay of tasks). With this information and the pertinent run time information (such as the current tick number, the uncompleted released tasks, etc.) in hand; the monitor can continue the monitoring process deterministically.

Monitoring the levels of task jitter and tick jitter in the pTTRM scheduling can be performed in the same manner as in the pTTC scheduling, where timestamps are taken at the start of two successive instances of a task and at the start of each tick.

### 7.4 Case study and evaluation

The evaluation process for the pTTRM framework, like the pTTC framework, was divided into two parts: (1) the evaluation of task jitter level, and (2) the evaluation of the ability of run-time error detection of the framework.
In the first part of the evaluation, the proposed pTTRM scheduling in the framework was evaluated if it could improve the jitter level of the TTRM scheduling. In the second part of the evaluation, the pTTRM system in the first part of the evaluation was re-used and injected with timing faults to assess the error detection ability of the pTTRM monitor. The assessment of this evaluation is based on a simulation using the processor power measurement of the faulty pTTRM system. It is necessary to note that this evaluation used the same hardware platform for the evaluation of the pTTC framework described in Section 6.5.1.

### 7.4.1 Case study: a signal sampler

In this case study, an analogue signal sampler was implemented using both the TTRM and the pTTRM scheduling algorithms. The task set of the signal sampler is shown in Table 7.1.

<table>
<thead>
<tr>
<th>ID</th>
<th>Task</th>
<th>Offset (tick)</th>
<th>Period (tick)</th>
<th>Jitter sensitivity</th>
<th>BCET (µs)</th>
<th>WCET (µs)</th>
<th>Sandwich delay (µs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Short Task</td>
<td>0</td>
<td>1</td>
<td>N</td>
<td>9.20</td>
<td>9.25</td>
<td>60</td>
</tr>
<tr>
<td>1</td>
<td>HV Task</td>
<td>0</td>
<td>2</td>
<td>N</td>
<td>41.25</td>
<td>263.25</td>
<td>315</td>
</tr>
<tr>
<td>2</td>
<td>Sampling Task</td>
<td>0</td>
<td>4</td>
<td>Y</td>
<td>18.45</td>
<td>18.50</td>
<td>80</td>
</tr>
<tr>
<td>3</td>
<td>Data transfer</td>
<td>1</td>
<td>4</td>
<td>N</td>
<td>25.45</td>
<td>25.50</td>
<td>85</td>
</tr>
<tr>
<td>4</td>
<td>Long Task</td>
<td>1</td>
<td>80</td>
<td>N</td>
<td>2000</td>
<td>2015</td>
<td>2030</td>
</tr>
<tr>
<td>5</td>
<td>Flash LED</td>
<td>1</td>
<td>250</td>
<td>N</td>
<td>3.25</td>
<td>4.15</td>
<td>60</td>
</tr>
</tbody>
</table>

Table 7.1: The task set of a signal sampler

---

5 The sandwich delay of all tasks includes 24.25µs scheduling overhead only (See Figure 7.4).
In this case study, an analogue signal was sampled by the sampling task (Task 2), and then transferred to a host computer (by Task 3) through the RS232 interface. The HV task (Task 1) was used to generate a high level of execution jitter, and has an execution priority higher than the jitter-sensitive task (Task 2). Since the periods of Task 1 and Task 2 were harmonic, Task 2 suffered from release jitter. The jitter level of Task 2 in both the TTRM and the pTTRM scheduling algorithms were measured and examined if the jitter level in the pTTRM scheduling algorithm had been improved. The result is shown in Figure 7.3 where the start time of Task 2 was sampled 10000 times. It shows that Task 2 suffered from significant release jitter in the TTRM scheduling, where the start time variance equal to 4457.07, due to variations in the execution time of Task 1. The jitter level of Task 2 is extremely low in the pTTRM scheduling where the start time variance equal to 0.

Table 7.2 summarizes the results of this case study. It shows that the absolute release jitter of the pTTRM scheduling is 99.97% less than the TTRM scheduling.
However, the memory requirement of the pTTRM scheduling is 99.87% more than the TTRM scheduling.

<table>
<thead>
<tr>
<th>Scheduling algorithm</th>
<th>Start time variance ($\sigma^2$)</th>
<th>Absolute release jitter Max – Min (µs)</th>
<th>Memory consumption (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>The TTRM scheduling</td>
<td>4457.07</td>
<td>186.75</td>
<td>5571</td>
</tr>
<tr>
<td>The pTTRM scheduling</td>
<td>0</td>
<td>0.05</td>
<td>11135</td>
</tr>
</tbody>
</table>

Table 7.2: A comparison data of the TT and the pTT rate-monotonic schedulings

### 7.5 Evaluation of the predictable time-triggered rate-monotonic monitor

This section presents an evaluation for the pTTRM monitor. Like the evaluation of the pTTC monitor, the evaluation of the pTTRM monitor focused on three aspects: (i) task overrun detection, detection of deadline violations, and (iii) task jitter monitoring.

It is important note that scheduling overhead is taken into account in this section, and this evaluation was undertaken using a simulation based on the power measurement of the signal sampler employed in the case study presented in Section 7.4.1. The hardware and software configurations were also the same as the case study.

#### 7.5.1 Overview of the pTTRM system’s operations and timings

It is important to underline that the variation of the scheduling overhead in the pTTRM monitor was minimized. As an example shown in Figure 7.4 where the scheduling overhead and the operation timings of this particular implementation
(the pTTRM scheduler) are summarized, the scheduler performs task scheduling at the beginning of a tick and then sets the processor into the idle state. When the pre-determined time (170 µs from the start of the tick) is reached, the processor will be woken up to execute the first task after ~16.5 µs.

At the end of a task, the scheduler will spend ~24.25 µs to calculate the wake up time of the processor before it is set into the idle state. Task scheduling (~87.50 µs) will take place after the processor is woken up at the end of a task’s sandwich delay. If no task is due to run, the scheduler will disable the planned pre-emption and then set the processor into the idle state to wait for the next tick arrival. If there is a task due to run, the task will start to execute without any delay.

![Figure 7.4: Operational timings of the pTT rate-monotonic system](image)

When a planned pre-emption occurs during a task execution, the scheduler will update the pre-empted task’s timing information (such as the remaining time for the task’s execution or sandwich delay). A planned pre-emption will occur at 50 µs before the tick timer interrupt or the start of a jitter-sensitive task.

In the case when there is not enough time to schedule a task to execute due to a planned pre-emption is nearby, i.e. the processor will be woken up within the range of 157.5µs (which is the sum of the scheduling overhead 87.5µs, the
planned pre-emption time 50µs and the safe margin 20µs) from the end of the tick or the start of a jitter-sensitive task, no task will be scheduled to execute.

7.5.2 Simulation results

7.5.2.1 Task overrun detection

For the task overrun detection, we simulated a situation when the WCET of a task is underestimated: the WCET of Task 0 (Short Task) was lengthened (between Tick 1 to Tick 3) to make it exceed its sandwich delay, but not to violate the deadline. Figure 7.5 shows the simulation result based on the real power measurement of the pTTRM signal sampler, to demonstrate the task-overrun detection ability of the pTTRM monitor.

The figure shows that the detection of overrun events for Task 0 was achieved by enabling a timer interrupt to occur at the end of its sandwich delay (i.e. at 245µs or the 49th sample after the synchronization between the system and the monitor). Since the pTTRM monitor could not find the falling edge before the interrupts occurred in Ticks 1, 2 and 3, the monitor confirmed that Task 0 overran its WCET at these ticks.

After the completion of Task 0, the monitor process continued to scan for the next task in the same manner, where the next rising edge was used to locate the start of the next task. The monitor then computed the time to scan for a falling edge using equation (7.1). For example, the scanning points for a falling edge of Task 4 (the Long Task) were derived at the start (or at the resumption) as follows.
At Ticks 1, 3, 5, 6 and 7:

The time required for Task 4 to complete was greater than the planned pre-emption point (i.e. \( \left\{ \text{Start}_{t_4} + R_{t_4} + \text{Overhead}_f + \text{Overhead}_r - \left( \text{SD}_{t_4} - \text{BCET}_{t_4} \right) \right\} > P \)), where \( \text{Overhead}_f \) is the scheduling overhead before the task starts and \( \text{Overhead}_r \) is the scheduling overhead after the task completion, so that,

\[
\text{ScanFE}(t_4) = P - \Delta = 950 - 15 = 935 \mu s \text{ (the 187th sample)}
\]

At Tick 2:

Since the time when the rising edge was found at 900 \( \mu s \) (180th sample) which was greater than 842.5 \( \mu s \) (the planned pre-emption point \( P = 950 \) \( \mu s \) – the scheduling overhead \( \text{Overhead}_f = 87.50 \) \( \mu s \) – safe margin 20\( \mu s \)), Task 4 was not scheduled to execute.

At Ticks 4 and 8:

The monitor knew that the jitter-sensitive task (Task 2) would execute in these two ticks. The start position for scanning the falling edge of Task 4 was calculated using the Task 2’s planned pre-emption point:

\[
\text{ScanFE}(t_4) = P - \Delta = 760 - 15 = 745 \mu s \text{ (the 149th sample)}
\]

At Tick 9:

The time required for Task 4 to complete was less than the next planned pre-emption point (i.e. \( \left\{ \text{Start}_{t_4} + R_{t_4} + \text{Overhead}_f + \text{Overhead}_r - \left( \text{SD}_{t_4} - \text{BCET}_{t_4} \right) \right\} < P \)) so that,

\[
\text{ScanFE}(t_4) = \text{Start}_{t_4} + R_{t_4} + \text{Overhead}_f + \text{Overhead}_r - \left( \text{SD}_{t_4} - \text{BCET}_{t_4} \right) - \Delta \\
\text{ScanFE}(t_4) = 415 + 88.25 + 87.5 + 24.25 - (2030 - 1975.75) - 15 \\
= 545.75 \mu s \text{ (the 109th sample)}
\]
Figure 7.5: The simulation result of the pTT rate-monotonic monitor
7.5.2.2 Detecting deadline constraint violations

Like the pTTC monitor described in Section 6.4.1, an interrupt for detecting a deadline constraint violation was also enabled in the monitor when the pTTRM system was performing task scheduling. This interrupt was set to occur at the point \( (VP_{t_i}) \) where the deadline violation was detected. The \( VP_{t_i} \) can be derived using equations (7.2), (7.3) and (7.4). As Figure 7.5 shows, the \( VP_{t_0} \) for each instance of Task 0 was set, and can be calculated as follows.

By equation (7.2): \( f(t_0) = D_{t_0} = PT = 950 \) for all ticks, where \( PT \) is the planned preemption of a tick.

At Tick 1, 2, 3, 5, 6, 7 and 9

By equation (7.3): \( g(t_0) = D_{t_0} = PT = 950 \mu s \)

By equation (7.4): \( VP_{t_0} = f(t_0) - \Delta = 950 - 15 = 935 \mu s \) (187\textsuperscript{th} sample)

At Tick 0, 4 and 8

By equation (7.3): \( g(t_0) = Start_{t_2} = 810 \mu s \)

By equation (7.4): \( VP_{t_0} = g(t_0) - \Delta = 810 - 15 = 795 \mu s \) (159\textsuperscript{th} sample)

Akin to the detection of a task overrun, the interrupt for detecting timing constraint violations must be disabled before the \( VP_{t_i} \), or otherwise a violation of deadline constraint is detected.

7.5.2.3 Monitoring task jitter levels

Monitoring task jitter levels for the jitter-sensitive task – Task 2 (Sample Task) – was done by taking a timestamp when the rising edge of Task 2 was found. With
reference to Figure 7.5, the rising edge for Task 2 at Ticks 0, 4 and 8 were found at 810μs (Sample 162) after the synchronization between the system and the monitor. Hence, no task released jitter is detected for Task 2.

Although the jitter level of Task 2 can be measured at run time, it is clear that the measured jitter-level for Task 2 may not be precise enough in this evaluation; since the sample rate of the monitor was only 200 KHz (i.e. a precision of ±5 μs). However, if high precision task jitter measurements are required, the pTTRM monitor can be implemented on a microcontroller equipped with a high speed ADC.

7.6 Discussion

The case study and the evaluation assessed the system and the monitor of the proposed pTTRM framework. In the case study, the pTTRM scheduling demonstrated its jitter reduction ability in which the jitter level of the jitter-sensitive task was almost reduced to zero. However, the pTTRM scheduling has more scheduling overhead and is double the size of memory requirement of the TTRM scheduling.

In the evaluation of the time-based error detection ability, the simulation result shows that the pTTRM monitor detected all task overrun events and deadline violations. These were achieved by monitoring the run time power measurement of the pTTRM system, the knowledge of the task schedule and, in particular, precise timing information about all tasks. The run time information about tasks in the pTTRM system is indeed easy to be obtained, and therefore the monitor can
perform run time detection of task overruns and deadline violations. Since the start times of tasks in the pTTRM are fixed, any violation of jitter requirements can also be detected.

Although the pTTRM monitor is capable to detect time-based errors, it is noticeable that the monitor is required to compute interrupt locations (on-line) for detecting task overruns and deadline violations. This suggests that the processing power required by the monitor is proportional to the number of tasks in the pTTRM system.

Overall, the pTTRM framework can indeed improve temporal predictability for the TTRM scheduling, and allow time-based errors to be detected efficiently. This suggests that the proposed framework is possible to extend to time-triggered scheduling algorithms which support task pre-emption. However, the scheduling overhead and the memory requirement of the pTTRM scheduling are substantially high. Although these can be seen as a trade-off between schedulability and predictability of the system, this will restrict the use of the pTTRM framework in embedded applications which already have tight timing constraints (or high processor utilization). Also, the interruption points for error detections require to be computed at run time. This will limit the use of the pTTRM monitor only to systems which have a small set of tasks. Surely, the limitations of the pTTRM framework require further enhancement.
7.7 Conclusion

This chapter presented a study of the possibility to extend the proposed framework to a time-triggered scheduling algorithm which supports task preemption. We implemented the time-triggered rate-monotonic (pTTRM) framework. The implementation included upgrading the proposed jitter-reduction technique described in Section 6.2 and integrating it with other existing techniques for the pTTRM scheduling. The evaluation results were promising where task jitter present in the TTRM scheduling was almost reduced to zero.

The implementation of the pTTRM monitor involved upgrading the equations to compute accurate times for data sampling and for setting up interrupts in order to detect time-based errors. The results of the evaluation show that all injected time-based errors are detected and the measurement of task jitter level at run time was also supported.

All the evidence provided in the evaluations suggests that the proposed framework is generic enough for the implementation of time-triggered preemptive algorithms. However, the memory requirement and the scheduling overhead are substantially high. These limitations need to be addressed. Otherwise, the pTTRM framework will only be suitable for safety-critical embedded applications that require low processor utilisation.
Chapter 8

Conclusions and future work

This conclusion chapter outlines the novelty contributions achieved by this research study. It also discusses the limitations and recommendations for future work.

8.1 Summary of contributions

The work described in this thesis is concerned with the issues of temporal predictability and run-time error detection in safety-critical embedded system design. It has emphasised that temporal predictability of embedded systems is utmost important not only for fulfilling the critical temporal requirements but also for allowing time-based errors to be detected at run time.

As time-triggered (TT) software architectures are more predictable than event-triggered (ET) architectures, this research study focused on TT embedded software designs. However, variations often appear in task execution times, start times and periods, which make it difficult to determine the exact behaviour of TT embedded systems, in particular, at run time. Even one of the most predictable time-triggered algorithms – the time-triggered co-operative (TTC) scheduling – suffers from the task jitter problem.

Existing jitter minimisation techniques can reduce task jitter level, but may not be capable to address task jitter that is introduced at scheduling level. Even if the task jitter problem is solved, embedded systems may fail to operate correctly at
run time. Adding a monitoring system can help to detect run-time errors, but is likely to introduce negative effect to the monitored system; since a monitoring system is seldom considered as a part of an embedded application during the development process. These motivated the research study reported in this thesis.

8.1.1 Designing a novel non-invasive monitoring

Designing monitoring systems are often confronted with the probe effect problem. This problem can be addressed by employing hardware-based monitoring approach to passively obtain information from the monitored system at run time. Traditional hardware-based monitoring approach achieves this by using complex hardware architecture to attach hardware probes to the buses of the monitored system’s processor. Unfortunately, this approach is obsolete for modern embedded processors; since all buses are encapsulated inside the processors, providing no interface for the connection to the buses.

The non-invasive monitoring described in Chapter 4 is a novel hardware-based monitoring methodology. It is designed to detect deadline violations in embedded systems which employ time-triggered co-operative (TTC) scheduling. Instead of connecting to the internal buses or an input/output port of the monitored system’s processor, it is connected to the power input of the processor using a simple and low cost hardware interface, such as a resistor.

By observing fluctuations in the processor power consumption in conjunction with prior knowledge of the task schedule, the non-invasive monitoring systems – namely the non-invasive safety agent (NISA) and the non-invasive safety agent
for dynamic voltage scale systems (NISA-DVS) – can detect deadline violations with a time resolution of microseconds in TTC embedded systems with (or without) a static DVS scheme.

In addition, the non-invasive monitoring techniques have a distinct advantage in detecting power failures or other hardware faults that can halt the operation of the processor in the monitored system. This ability cannot be achieved by any software-based monitoring technique as illustrated in Section 5.4.3.3.

8.1.2 Dealing with task jitter introduced at scheduling level

Although the TTC scheduling algorithm can provide a predictable temporal behaviour which guarantees tasks to execute at specific ticks, it cannot control tasks to execute at specific times within a tick. As task execution times usually vary from one instance to another, tasks will suffer from execution jitter and release jitter. Existing jitter-minimisation techniques are capable to reduce task jitter that is introduced at task level. However, when tasks with different execution rates are present in a task set, these tasks may start at different points in time every period. Such release jitter is introduced at scheduling level which cannot be addressed by jitter-minimisation techniques that operate at task level.

A novel jitter-reduction technique was introduced in Section 6.2 to cope with this jitter problem at scheduling level. Jitter-sensitive tasks are deferred to start within the same tick where they are released. An equation – equation (6.1) – is formulated in Section 6.2 for the computation of the deferred start times for jitter-sensitive tasks in embedded systems which employ the time-triggered co-
operative (TTC) scheduling and the time-triggered rate-monotonic (TTRM) scheduling. The evaluation results shown in Section 6.5.2 and Section 7.4.1 for the proposed jitter-reduction technique were promising in which task jitter level was almost reduced to zero in both the TTC and the TTRM scheduling.

### 8.1.3 Designing the predictable time-triggered framework

The novel jitter-reduction technique introduced in Section 6.2 addressed task jitter produced at scheduling level. It was used in conjunction with an existing jitter-minimisation technique – the sandwich delay technique – in order to reduce task jitter produced at task level. We applied these two jitter-reduction techniques to the time-triggered co-operative (TTC) algorithm to obtain an extremely low jitter scheduling algorithm. This novel scheduling algorithm is the predictable time-triggered co-operative (pTTC) algorithm.

The pTTC scheduling algorithm can schedule task executions in an extremely predictable temporal behaviour; since each task in the system executes in a predetermined timeslot every period. The absolute jitter level of the pTTC scheduling algorithm in the case study was 0.05 µs which reduced the release jitter in the TTC scheduling by 99.98%. In term of absolute release jitter level, the pTTC scheduling was 35% better than the code balancing (CB1) technique and 9% better than the single-path programming paradigm (SPP).

It is important to note that a small idle period is always present between task executions; since the sandwich delay technique is applied. This specific characteristic is particular suitable to be monitored using the non-invasive
monitoring approach described in Chapter 4; since the processor power consumption will rise when a task starts its execution, and drop when the task execution is complete.

We enhanced the non-invasive safety agent (NISA) by modelling its error detection mechanism like a “watchdog” which must be fed before the occurrence of an interrupt. An occurrence of this interrupt indicates that a time-based error is detected. This interrupt-driven approach which has been discussed in Section 5.5, improves the time to respond to a run-time error. This enhanced version of the NISA technique is the predictable time-triggered co-operative (pTTC) monitor. Unlike the watchdog timer which does not consider the task schedule and task jitter, the pTTC monitor can detect task overrun events, deadline constraint violations and jitter requirement violations in a pTTC system. It can also report the involved task and the tick number where the error occurred. These are achieved by collecting run time information of the monitored system with prior knowledge of the task schedule. In addition, equation (6.2) presented in Section 6.4.1 is defined for determining the deadline violation point of a task. A timer interrupt can be set to occur at this point to check if the deadline is violated.

The novel predictable time-triggered co-operative (pTTC) framework is established by combining the pTTC scheduling algorithm and the pTTC monitor. It was done by using a simple hardware interface (for instance, a resistor) to connect the monitor to the power input of the system’s processor. This framework is suitable for safety-critical embedded system design with jitter-sensitive tasks in a moderate processor utilisation task set; since it can provide an
extremely low jitter operating environment to task executions, and allows time-
based errors and power failure to be detected at run time.

### 8.1.4 Extending the predictable time-triggered framework

An attempt to extend the predictable time-triggered (pTT) framework had been made after the success of the pTTC framework. The predictable time-triggered rate-monotonic (pTTRM) framework, like the pTTC framework, consists of a scheduling algorithm and a run-time monitor. The pTTRM algorithm was derived by applying the proposed jitter-reduction technique in conjunction with two existing jitter-reduction techniques – the sandwich delay technique and the planned preemption technique – to an existing TTRM scheduling algorithm. The result of the case study showed that the absolute release jitter level in the pTTRM scheduling algorithm was almost reduced to zero and less than the TTRM algorithm by 99.97%.

The pTTRM monitor is a variant of the pTTC monitor and designed to detect time-based errors in a pTTRM system. It employs the same interrupt-driven approach as the pTTC monitor to detect time-based errors. Although the detection of task overruns is identical to the pTTC monitor, the calculation of the constraint violation points of a task requires more steps; since a pTTRM system supports jitter-sensitive tasks, task preemption and long duration tasks. Three equations – (7.2), (7.3) and (7.4) – have been formulated in Section 7.3.1 for deriving the constraint violation points for a task. Like the pTTC monitor, a timer interrupt can be set to occur at a violation point of a task to check if the deadline
of the task is violated. Measuring task jitter and tick interval jitter of a pTTRM system are also possible.

The evaluation results reported in this Section 7.4 suggested that the current version of the pTTRM framework is sufficient to support safety-critical embedded system design that has a low processor utilisation task set.

### 8.2 Limitations

This thesis has presented a study to provide a low-jitter operating environment and a run-time error detection mechanism to support safety-critical embedded system design. The result of this study is promising. However, there are some limitations to this study.

The non-invasive monitoring techniques described in Chapter 4 and Chapter 5 is designed to detect deadline violations in time-triggered co-operative (TTC) embedded systems. However, a false alarm will occur when a task completes its execution very close to the deadline; since the techniques detects the system state based on the level of the processor power consumption, and the deadline is set at the point in time just before the next tick interrupt. In this particular situation, the scheduler may not have enough time to set to processor into the idle state; since the arrival of the next tick will generate an interrupt to stop this process.

The run-time monitor in the predictable time-triggered co-operative (pTTC) framework addressed the limitation of the non-invasive monitoring technique; since a short period of idle time is present at the end of each task execution.
However, it relies on the *a priori* knowledge of task executions and precise measurement of the scheduler overhead. Imprecise measurement of task execution times and the scheduler overhead will lead to incorrect monitoring results. For the limitations of the pTTC scheduling algorithm, the results of the case study in 6.5.2 have shown that it required 40% more memory space than the TTC scheduling and the scheduling overhead increased by 2.15 times. These will restrict the use of pTTC framework in embedded system design that requires moderate processor utilisation.

The predictable time-triggered rate-monotonic (pTTRM) framework inherits the limitations from the pTTC framework. The evaluation results described in Section 7.4.1 have shown that the pTTRM scheduling algorithm required 99.87% more memory space than the TTRM scheduling, and had significant scheduling overhead (see Section 7.5.1). These mean that the pTTRM algorithm is only suitable for use with embedded system design that requires low processor utilisation.

### 8.3 Future work

This thesis has presented the predictable time-triggered frameworks that consist of a predictable time-triggered algorithm and a novel run-time monitoring technique. The future work of this research will need to address the limitations described in Section 8.2, and to explore possible ways to further improve the frameworks to support other scheduling algorithms. This section provides some suggestions for further research of this study.
8.3.1 **Addressing the limitations of the predictable time-triggered systems**

Scheduling overhead is a practical issue which affects system schedulability and reduces system predictability in the presence of jitter. In a predictable time-triggered (pTT) system, the durations of scheduling overhead are fixed. Thus, the behaviour of a pTT system is very predictable, but the scheduling overhead is substantial high. This can be seen as a trade-off between predictability and schedulability.

This trade-off problem may not be able to solve in software-level, but can be addressed in hardware-level. Implementing a pTT scheduling algorithm on a hardware platform can reduce or eliminate scheduling overhead (Kuacharoen and Shalan et al., 2003; Gupta and Mandal et al., 2010). This will also reduce the memory requirement for a pTT system.

8.3.2 **Extending the predictable time-triggered framework**

This thesis has only presented two predictable time-triggered (pTT) frameworks: the pTTC framework and the pTTTRM framework. However, there are many other TT algorithms which may be useful for specific applications. For example, the time-triggered hybrid (TTH) scheduling described in Section 2.7 and the time-triggered deadline monotonic (TTDM) scheduling. The pTT framework is clearly generic enough to extend to these scheduling algorithms.

8.3.3 **Ensuring the correctness of monitoring results**

Monitoring systems like other real-time embedded systems may fail to operate correctly. It is necessary to ensure a monitoring system to produce correct results
about the monitored system. However, it is pointless to monitor a monitoring system; since it will create an endless question: Who is going to monitor the monitor?

A simple solution which has been widely used for this kind of problems is to employ a hardware redundancy (or diversity) strategy. This will require at least three monitoring systems to monitor a monitored system simultaneously. A voting scheme between the monitoring systems must be established beforehand. The final result about the monitored system will be determined by majority vote of the monitoring systems.

### 8.4 Conclusions

The work described in thesis has accomplished its goal to deliver two frameworks to support safety-critical embedded system design. The predictable time-triggered frameworks provide a low-jitter operating environment to allow real-time tasks to execute in an extremely predictable manner, and a run-time error detection mechanism in case time-based errors occur at run time.

The key contributions of this study are summarized as follows:

- Development of a novel non-invasive monitoring approach and implemented in non-invasive monitoring techniques using this approach to detect deadline violations in time-triggered co-operative (TTC) embedded systems;
- Development of a novel jitter-reduction technique to address task release jitter introduced at scheduling level; and combined this technique with other existing jitter-minimisation techniques to derive the predictable time-
triggered co-operative (pTTC) algorithm and the predictable time-triggered rate-monotonic (pTTRM) algorithm, respectively from an existing time-triggered co-operative (TTC) algorithm and time-triggered rate-monotonic (TTRM) algorithm;

- Development of two predictable time-triggered monitors to detect time-based errors, such as deadline violations, task overruns and jitter-requirement violations in a matched predictable time-triggered system.
Appendix A

An investigation of the power measurement quality in an unstable temperature environment

This appendix provides an investigation of the impact of the ambient temperature changes on the non-invasive safety agents described in Section 4.4. This investigation is relevant to the work in this thesis.

A.1 Introduction

Resistance in any material will change with the temperature. This fact often affects the performance and reliability of electronic devices. The monitoring systems – such as the non-invasive safety agent and the predictable time-triggered monitor – rely on the processor power measurements obtained from a sensory resistor to detect time-based errors in a time-triggered embedded system. A change in the resistance of the resistor will have significant impact on the accuracy of the monitoring results. The following sections provide an investigation of this issue.

A.2 Possible Solutions for unstable temperature environment

There are few approaches which can be employed to minimize the impact of temperature changes on the monitoring results:-

(i) To employ a thermo-resisted container which provides a constant temperature environment to the monitor,
(ii) To employ a PT-100 platinum resistance thermometer (30K/-234C – 873K/600C) to sense the temperature and change the threshold of the monitor accordingly, or

(iii) To employ a low temperature coefficient resistor, such as a metal film resistor.

A.3 Case study: an unstable temperature environment

This case study is to test the effect on the power measurements obtained by the non-invasive safety agent from a time-triggered co-operative embedded system under an unstable temperature environment.

A.3.1 Hardware and software configurations of the monitored system

Evaluation board: The Ashling EVBA7 evaluation board
Microcontroller: The LPC 2106 (ARM7)
Scheduler: The TTC scheduler
Tick interval: One millisecond

<table>
<thead>
<tr>
<th>Task ID</th>
<th>Task</th>
<th>Offset (ms)</th>
<th>Period (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>T0</td>
<td>SPI_Update</td>
<td>1</td>
<td>50</td>
</tr>
<tr>
<td>T1</td>
<td>UART1_O_UPDATE</td>
<td>5</td>
<td>75</td>
</tr>
<tr>
<td>T2</td>
<td>Blink_Leds</td>
<td>0</td>
<td>20</td>
</tr>
<tr>
<td>T3</td>
<td>Display_RTC_Update</td>
<td>10</td>
<td>70</td>
</tr>
<tr>
<td>T4</td>
<td>Switch_Read</td>
<td>7</td>
<td>30</td>
</tr>
</tbody>
</table>

Table A.1: The task set for the case study
Appendix A – An investigation of the power measurement quality in an unstable temperature environment

A.3.2 Hardware and software configurations of the non-invasive safety agent

Evaluation board: The Olimex ADuC-P7026
Microcontroller: The ADuC7026 (ARM7)
Temperature sensor: Maxim/Dallas Semiconductor DS75LXS

**Sampling resistor:** Vishay PR02 1R ±5% metal film resistor (E24 series)

\[
\text{temperature coefficient} \leq \pm 250 \times 10^{-6}/K
\]

Scheduler: The TTC scheduler
Tick interval: 2 seconds (for temperature sensing only)

<table>
<thead>
<tr>
<th>Task</th>
<th>Offset (s)</th>
<th>Period (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Check temperature and take samples if necessary</td>
<td>0</td>
<td>4 (2 ticks)</td>
</tr>
</tbody>
</table>

Table A.2: The temperature sensing task of the non-invasive safety agent

Note: Fast \textsuperscript{I}C (400K) was utilised for the communication between the ADuC7026 microcontroller and the DS75LXS temperature sensor.

![Figure A.1: Test bed of the temperature experiment](image)
A.3.3 The experiment description

The power for both the non-invasive safety agent and the monitored system was supplied by a host PC through two USB cables. The ambient temperature was initially measured and stored in the safety agent as the threshold for power measurement data sampling. The threshold was subsequently compared with the current temperature. If the current temperature is greater than/equal to the threshold, the safety agent will start taking 2000 samples of the processor power consumption of the monitored system. The threshold was then increased by 10°C and waiting for the next comparison. Sampled power data were sent and stored in the host PC. The sampling process was terminated when the ambient temperature reached +90°C.

In the experiments, a freezer and an oven – for domestic use – were employed to change the ambient temperature (from -21°C to +90°C).

Figure A.2: Unstable temperature environments
A.4 Results

A.4.1 Experiment 1

Temperature range: +9°C – +89°C

Figure A.3: Temperature experiment 1 (+9°C to +89°C)
A.4.2 Experiment 2

Temperature range: -14°C – +86°C

Figure A.4: Temperature experiment 2 (-14°C to +86°C)

Note: This set of data at 46°C was corrupted by the short circuit due to the condensation of humidity on the surface of both systems.
A.5 Conclusions

The results of the case study show that changes in the ambient temperature at a reasonable range (for example, -21°C to 90°C) do not have a significant impact to the resistance of the sampling resistor – a low temperature coefficient metal-film resistor. Hence, the quality of the power measurement is sufficient for the non-invasive safety agent to identify the state of the monitored system. However, care must be taken in a sudden change of the ambient temperature; since the condensation of humidity will affect the operations of both the safety agent and the monitored system, as illustrated in Figure A.4 at 46°C.
Appendix B

The LabVIEW virtual instruments

B.1 Introduction

This appendix describes the “LabVIEW” virtual instruments (VIs) that were employed in the case studies and experiments in this research study.

B.2 The virtual instrument for release jitter measurement

Figure B.1 shows the front panel of the virtual instrument (VI) for the measurement of task start times in the experiments of this research work.

![Virtual Instrument for Task Start Time Measurements]

B.1: The front panel of the virtual instrument for task start time measurements

The input fields are described as follows:

1. The input counter field is used for selecting an on-board timer/counter
2. The number of measurements field is used for selecting a specific number of measurements
Appendix B – The LabVIEW models for the experimental measurements

After entering all values in the input fields, the VI can be started by pressing the start button in the LabVIEW tool bar. The results will print out in the output fields. The output fields – labelled “Min” and “Max” – are the results of the minimum value and the maximum value among all measurements. The output field – labelled “Variance” – is the variance of all measurements.

Figure B.2 shows the block diagram of the VI for the measurement of task start times. Each frame in the sequence structure is described as follows.

In Frame 1, all variables in this VI are initialised. These variables include two control variables (labelled “Done” and “Measuring”) and three variables for the outputs.

In Frame 2, the virtual channel is created by using the input counter value from the front panel and the selected value in the “Create Virtual Channel” block. In this VI, the “CI Pulse Width” is selected. This means that this VI is used to measure the width of a digital pulse. The “Timing” block allows the user to configure the number of samples to acquire and create a buffer when needed. In this VI, the “Implicit” and “Finite Samples” are selected. These mean that no
specific timing is defined for data sampling, and the number of samples depends on the input value from the front panel. The “Start Task” block is used to start the measurement from the selected channel. The “Read” block is used to read data samples from the selected channel. When data samples are read, the “Clear Task” block will release the resources that the task has been reserved.

In Frame 3, the data samples are filled into an array and passed to Frame 4.
In Frame 4, the maximum and the minimum values among all samples are computed.
In Frame 5, all data samples are stored in a file, and a histogram is generated and passed to the front panel to display. It also computes the variance of the data samples and then passes it to the front panel to display.

B.3 The virtual instrument for the processor power measurement

Figure B.3 shows the front panel of the VI for processor power measurement used in the experiments and case studies of this research work.

B.3: The front panel of the virtual instrument for processor power measurements
This VI can be started by pressing the start button in the LabVIEW tool bar and then pressing the “START” button in the front panel.

Figure B.4 shows the block diagram of the VI for the measurement of processor power measurements in the experiments of this study. This outer square in the figure is a “While Loop” structure. This loop is configured to stop if the conditional terminal has a value equal to “True”. The inner square in the figure is a “Case” structure. It is used for starting the measurement after the start button in the LabVIEW tool bar is pressed.

1. These blocks are the inputs from the front panel.

2. This block is the “DAQ Assistant Express VI”. It is used to create, configure and run tasks to read data samples. In this VI, it is configured to
read a voltage input signal using the on-board differential ADC. The sampling rate and number of samples are input from the front panel.

3. This block is to write the data samples to a file.

4. This block is used to select specific input signals. In this VI, it selected the signal index ‘0’ which is the input signal from the DAQ Assistant Express.

5. This block is used to stop the program if an error occurs during data sampling.

6. This block is used to stop the program if an error occurs during writing data into a file.

7. This block is used to convert a dynamic data input to a one dimension array of floating point numbers.

8. This block is used to store the array of floating point numbers in 7.

9. This block is a “For loop” structure. It is used to generate the x-axis values for the display.

10. This block convert the one dimension of x-axis values to a dynamic data output.

11. This block generates an X-Y graph from two dynamic data inputs (the x-axis values from 10 and the data samples from 4.

12. This block displays the data samples on the front panel

13. This block creates an error handler to display an error message if an error occurs.
Appendix B – The LabVIEW models for the experimental measurements

References


Bate, I. J. (1998), "Scheduling and timing analysis for safety critical real-time systems", PhD thesis, Department of Computer Science, University of York,


Appendix B – The LabVIEW models for the experimental measurements


HandyWave USA (version 2.0), "HCS-100 HandyCore-Serial user's manual ", http://www.handywave-usa.com/downloads/HCS_100_manual_v2.0_english.pdf.


Appendix B – The LabVIEW models for the experimental measurements


Appendix B – The LabVIEW models for the experimental measurements

Mansouri-Samani, M. (1995), "Monitoring of distributed systems", PhD thesis, Department of Computing, University of London Imperial College of Science, Technology and Medicine, UK.,


Appendix B – The LabVIEW models for the experimental measurements


Appendix B – The LabVIEW models for the experimental measurements

Raja, T., Agrawal, V. D. and Bushnell, M. L. (2005), "Variable input delay CMOS logic for low power design", *Proceedings of 18th International Conference on VLSI Design* pp. 598-605.


Appendix B – The LabVIEW models for the experimental measurements


Appendix B – The LabVIEW models for the experimental measurements


