University of Leicester

Techniques for scheduling time-triggered resource-constrained embedded systems

Thesis submitted for the degree of

Doctor of Philosophy

at the University of Leicester

by

Ayman K. G. Gendy

Embedded Systems Laboratory

Department of Engineering

University of Leicester

Leicester, UK

June 2009

Techniques for scheduling time-triggered resource-constrained embedded systems

Ayman K. G. Gendy

Abstract

It is often argued that time-triggered (TT) architectures are the most suitable basis for safety-related applications as their use tends to result in highly-predictable system behaviour. This predictability is increased when TT architectures are coupled with the use of co-operative (or "non pre-emptive") task sets.

Despite many attractive properties, such "time-triggered co-operative" (TTC) and related "time-triggered hybrid" (TTH) architectures rarely receive much attention in the research literature. One important reason for this is that these designs are seen to be "fragile": that is, small changes to the task set may require revisions to the whole schedule. Such revisions are seen as challenging and time consuming. To tackle this problem two novel algorithms (TTSA1 and TTSA2), which help to automate the process of scheduler selection and configuration, are introduced. While searching for a workable schedule, both the algorithms try to ensure that all task constraints are met, a co-operative scheduler is used whenever possible and the power consumption is kept as low as possible. The effectiveness of these algorithms is tested by means of empirical trials.

Both TTSA1 and TTSA2, like most of scheduling algorithms introduced in the literature, rely on knowledge of task worst-case execution time (WCET). Unfortunately, determining the WCET of tasks is rarely straightforward. Even in situations where accurate WCET estimates are available at design time, variations in task execution time, between its best-case execution time (BCET) and its WCET, may still affect the system predictability and/or violate task constraints. In an effort to address this problem, a set of code-balancing techniques is introduced. Using an empirical study it is demonstrated that these techniques help in reducing the variations in task execution time, and hence increase the system predictability. These goals are achieved with a reduced power-consumption overhead, compared to alternative solutions.

Acknowledgement

First of all, I would like to express my heartfelt gratitude to Professor Michael Pont. As my supervisor, he has provided me with invaluable guidance and constant support throughout this research project. I consider myself very fortunate for having found such an excellent supervisor. Without him, this thesis would not have been possible.

Next, I would like to thank people in Assiut University in Egypt (especially Prof. Ibrahim H., Prof. Doss M., Prof. Mahdy Y., and Prof. Sewisy A.) for their support and for awarding me the scholarship which gave me the opportunity to pursue my research at the University of Leicester.

I also would like to thank all my colleges at the embedded systems lab and TTE Systems Ltd, Ahmad A., Amir M., Athaide K., Chan K. , Bautista R., Edwards T., Hanif M., Hughes Z., Imran S., Key S., Kurian S., Kyriakopoulos I., Lei D., Lakhani F., Mearns D., .Nazri A., Rizvi S., Vidler P., Wang H., Dr. Ayavoo D., Dr. Das A., Dr. Fang J., Dr. Maaita A., Dr. McEwan A., Dr. Mwelwa M., Dr. Nahas M., Dr. Ong R., Dr. Phatrapornnant T., and Dr. Short M. for both their technical and moral supports. It was privilege to work with you all.

A special acknowledge to Chan K. (ESL, Leicester) for his help in making the power measurements, Lakhani F and Wang H. for their help in getting me the papers whenever I needed them, Dr. Nahas M. (ESL, Leicester) for his help in making the measurements of the scheduler overhead, Dr. Ayavoo D. (TTE Systems Ltd) for his help in proofreading earlier version of Chapter 1 of the thesis, and Dr. Demian P. (Loughborough University) for his help in proofreading of the thesis English.

More importantly, I would like to thank my beloved wife, Mary, for her support, encouragement and patience, which I needed the most. You sacrifice your dreams and did everything you can, whether you like or not, for preparing the environment I need during the whole period of my work in this thesis. I hope that God help me in being always beside you doing whatever I can to bring happiness to your loving heart.

The great thanks go to the youngest person who brought unbounded joy and happiness to my heat, my son Kyrillos. You were the light that shines in our small family and the source of smile which we need in difficult times.

Last but not least I would like to thank my mother, brother, sisters, mother in law, father in law and brothers in law for their constant support and prayers over the last few years. Thank you all for being there for me, as always. Special thanks to the soul of my father who departed while I was in middle of my work in this project. Your dream of seeing me getting my PhD degree was always in front of my eyes encouraging me especially in hard times.

I dedicate this thesis to

my wife, son, mum and the soul of my father

Table of contents

ACKNOWLEDGEMENT ..... **II**

TABLE OF CONTENTS ..... **I**

LIST OF FIGURES ..... **V**

LIST OF TABLES ..... **IX**

LIST OF PUBLICATIONS ..... **XII**

LIST OF ABBREVIATIONS, SYMBOLS AND UNITS ..... **XIV**

CHAPTER 1 INTRODUCTION **1**

1.1 INTRODUCTION 1

1.2 WHAT IS AN EMBEDDED SYSTEM? 2

1.3 WHAT IS A REAL-TIME SYSTEM? 3

1.4 DEVELOPING REAL-TIME SYSTEMS 4

1.5 SCHEDULING TIME-TRIGGERED SYSTEMS 6

1.6 CHALLENGES WITH SIMPLE TT ARCHITECTURE 7

1.7 AIMS OF THE THESIS 9

1.8 THESIS CONTRIBUTIONS 10

1.9 THESIS OUTLINE 11

1.10 CONCLUSIONS 12

CHAPTER 2 SCHEDULING STRATEGIES **13**

2.1 TASK CHARACTERISTICS 13

2.2 TASK CONSTRAINTS 15

2.3 SCHEDULING CRITERIA 19

2.4 CYCLIC EXECUTIVES 22

2.5 PRIORITY SCHEDULERS 24

2.6 DISCUSSION 40

2.7 CONCLUSIONS 43

CHAPTER 3 SCHEDULING ALGORITHMS **44**

3.1 THE FUNCTION OF SCHEDULING ALGORITHMS 44

3.2 CHOOSING THE RIGHT SCHEDULING STRATEGY 44

3.3 CHOOSING THE APPROPRIATE TASK ORDER/STARTING TIME 46

3.4 AUTOMATIC SCHEDULE GENERATION IN REAL-TIME SYSTEMS 48

3.5 PREVIOUS WORK DONE IN ESL FOR AUTO CODE GENERATION FOR TT SYSTEMS 52

3.6 DISCUSSION 54

3.7 CONCLUSIONS 54

CHAPTER 4 NECESSITY OF STABILISING TASK EXECUTION TIME 56

4.1 IMPACTS OF VARIATIONS OF TASK EXECUTION TIME 56

4.2 THE NEED FOR STABILISING THE TASK EXECUTION TIME 59

4.3 CHALLENGES WITH ESTIMATING THE TASK EXECUTION TIME 61

4.4 DEALING WITH EXECUTION TIME ERRORS 64

4.5 DISCUSSION 65

4.6 CONCLUSIONS 65

CHAPTER 5 CODE-BALANCING TECHNIQUES 67

5.1 TOWARD A FIXED EXECUTION PATH 67

5.2 THE SINGLE PATH PROGRAMMING PARADIGM 68

5.3 THE PROPOSED CB1 TECHNIQUES 69

5.4 PERFORMANCE OF THE CB1 TECHNIQUES 73

5.5 DISCUSSION 83

5.6 CONCLUSIONS 84

CHAPTER 6 ANALYSIS OF SCHEDULER/TASK CONFIGURATION 85

6.1 A CLOSE LOOK AT TTC AND TTH ARCHITECTURES 85

6.2 THE NEED FOR APPROPRIATE CONFIGURATION OF SCHEDULER/TASK PARAMETERS 89

6.3 EFFECTS OF TASK OFFSET 90

6.4 EFFECTS OF TICK INTERVAL 93

6.5 EFFECTS OF TASK ORDER 98

6.6 DISCUSSION 102

6.7 CONCLUSIONS 102

CHAPTER 7 TTSA1 ALGORITHM 104

7.1 TTSA1 FOR AUTOMATICALLY CHOOSING/CONFIGURING SCHEDULER/TASK PARAMETERS 104
7.2 EVALUATING THE TTSA1 ALGORITHM 112
7.3 DISCUSSION 121
7.4 CONCLUSIONS 122

CHAPTER 8 TTSA2 ALGORITHM 123

8.1 PROBLEMS WITH TTSA1 ALGORITHM 123
8.2 TTSA2 ALGORITHM 124
8.3 EVALUATING THE TTSA2 ALGORITHM 128
8.4 DISCUSSION 134
8.5 CONCLUSIONS 134

CHAPTER 9 DISCUSSION AND CONCLUSIONS 135

9.1 REASONS AND MOTIVATION OF THE THESIS WORK 135
9.2 A REVIEW OF THE CONTRIBUTIONS 136
9.3 LIMITATIONS AND FUTURE WORK 142
9.4 CONCLUSIONS 143

APPENDIX A TT ARCHITECTURES IMPLEMENTED IN ESL 144

A.1 TTC SCHEDULER 144
A.2 TTH SCHEDULER 147
A.3 CONCLUSIONS 148

APPENDIX B TTSA1 CASE STUDY (SCHEDULER AGENT) 149

B.1 THE BASIC SYSTEM DESCRIPTION AND FUNCTIONALITY 149
B.2 THE MP-SA ARCHITECTURE 150
B.3 THE MP-SA PERFORMANCE 152
B.4 CONCLUSIONS 161

APPENDIX C TTSA2 APPLICATION EXAMPLE 162

C.1 THE BASIC SYSTEM DESCRIPTION AND FUNCTIONALITY 162
C.2 TASK SPECIFICATIONS 162
C.3 TASK SCHEDULING ACCORDING TO ONE SEGMENT PER TASK 164
C.4 TASK SCHEDULING WITH CONSIDERING MULTIPLE SEGMENTS PER TASK 164

C.5 CONCLUSIONS 165

REFERENCES 166

List of Figures

FIGURE 1-1 INTEL 4004 MICROPROCESSOR AND BUSICOM 141-PF PRINTING CALCULATOR. THESE TWO IMAGES HAVE BEEN USED WITH PERMISSION FROM INTEL MUSEUM (INTEL, 2009). 2

FIGURE 1-2 RTOS BASIC SERVICE GROUPS, REDRAWN FROM (KALINSKY, 2005) (FIGURE 1). 5

FIGURE 2-1 TYPICAL TASK PARAMETERS. 15

FIGURE 2-2 CYCLIC EXECUTIVE SCHEDULER FOR THE TASK SET SHOWN IN TABLE 2-1. 24

FIGURE 2-3 RM SCHEDULE OF THE TASK SET SHOWN IN TABLE 2-2. 27

FIGURE 2-4 DM SCHEDULE FOR THE TASK SET SHOWN IN TABLE 2-3. 29

FIGURE 2-5 EDF SCHEDULE FOR THE TASK SET SHOWN IN TABLE 2-4. 31

FIGURE 2-6 LLF SCHEDULE FOR THE TASK SET SHOWN IN TABLE 2-5, ADAPTED FROM COTTET *ET AL.* (2002). 32

FIGURE 2-7 ILLUSTRATION OF THE PRIORITY INVERSION PHENOMENON FOR TASKS SHOWN IN TABLE 2-6, ADAPTED FROM SHA *ET AL.* (1990). 34

FIGURE 2-8 ILLUSTRATION OF USING THE PRIORITY INHERITANCE PROTOCOL FOR TASKS SHOWN IN TABLE 2-6, ADAPTED FROM SHA *ET AL.* (1990). 36

FIGURE 2-9 ILLUSTRATION OF THE DEADLOCK PHENOMENON FOR TASKS SHOWN IN TABLE 2-7, ADAPTED FROM SHA *ET AL.* (1990). 38

FIGURE 2-10 ILLUSTRATION OF USING THE PRIORITY CEILING PROTOCOL FOR TASKS SHOWN TABLE 2-7, ADAPTED FROM SHA *ET AL.* (1990). 39

FIGURE 3-1 INFEASIBLE RM SCHEDULER FOR TASK SHOWN IN TABLE 3-1, ADAPTED FROM XU AND PARNAS (2000). 45

FIGURE 3-2 FEASIBLE NON PRE-EMPTIVE EDF SCHEDULER FOR TASK SHOWN IN TABLE 3-1, ADAPTED FROM XU AND PARNAS (2000). 46

FIGURE 3-3 INFEASIBLE FIXED PRIORITY SCHEDULER FOR TASK SET SHOWN IN TABLE 3-2, ADAPTED FROM XU AND PARNAS (2000). 47

FIGURE 3-4 FEASIBLE SCHEDULER FOR TASK SET SHOWN IN TABLE 3-2, ADAPTED FROM XU AND PARNAS (2000). 47

FIGURE 4-1 TASK SCHEDULE WHEN TASKS SHOWN IN TABLE 4-1 RUN WITH THEIR BCETS. 57

FIGURE 4-2 TASK SCHEDULE WHEN TASKS SHOWN IN TABLE 4-1 RUN WITH THEIR WCETS. 58

FIGURE 4-3 TASK SCHEDULE FOR TASKS SHOWN IN TABLE 4-2. 59

FIGURE 5-1 CONVERTING *IF-THEN-ELSE* STRUCTURE TO SINGLE PATH, ADAPTED FROM (PUSCHNER AND BURNS, 2003)....68

FIGURE 5-2 PSEUDO CODE OF A BALANCED *FOR*-LOOP. 71

FIGURE 5-3 PSEUDO CODE OF A BALANCED *WHILE*-LOOP USED FOR WAITING FOR INPUT. 72

FIGURE 5-4 PSEUDO CODE OF A BALANCED *IF-THEN-ELSE* STRUCTURE, ADAPTED FROM (PUSCHNER AND BURNS, 2003)....73

FIGURE 5-5 THE CIRCUIT USED TO MEASURE THE SYSTEM POWER CONSUMPTION, COPIED FROM NAHAS (2009)....74

FIGURE 5-6 TRADITIONAL IMPLEMENTATION OF BUBBLE SORT. 76

FIGURE 5-7 SINGLE PATH IMPLEMENTATION OF BUBBLE SORT (ADAPTED TO WORK WITHOUT THE SUPPORT OF THE CONDITIONAL MOVE INSTRUCTION) (PART 1/2).....77

FIGURE 5-8 CB1 IMPLEMENTATION OF BUBBLE SORT (PART 1/2). 79

FIGURE 6-1 ILLUSTRATING THE OPERATION OF A TYPICAL (INTERRUPT-DRIVEN) TTC SCHEDULER IMPLEMENTATION. 87

FIGURE 6-2 ILLUSTRATION OF TTC SCHEDULE OF TASK SET SHOWN IN TABLE 6-2. 88

FIGURE 6-3 ILLUSTRATING THE OPERATION OF A TYPICAL TTH SCHEDULER IMPLEMENTATION, ADAPTED FROM MAAITA AND PONT (2005B), FIGURE 1. 89

FIGURE 6-4 TASK SCHEDULE FOR TASKS SHOWN IN TABLE 6-5. 92

FIGURE 6-5 TASK SCHEDULE FOR TASKS SHOWN IN TABLE 6-6. 93

FIGURE 6-6 TASK SCHEDULE FOR TASKS SHOWN IN TABLE 6-7 (WITH TICK INTERVAL = 2 *MS*). 94

FIGURE 6-7 TASK SCHEDULE FOR TASKS SHOWN IN TABLE 6-8 (WITH TICK INTERVAL = 1 *MS*). 95

FIGURE 6-8 TASK SCHEDULER FOR TASKS SHOWN IN TABLE 6-9 (WITH TICK INTERVAL = 2 *MS*). 96

FIGURE 6-9 TASK SCHEDULER FOR TASKS SHOWN IN TABLE 6-10 (WITH TICK INTERVAL = 1 *MS*). 96

FIGURE 6-10 INAPPROPRIATE TASK ORDERING FOR TASKS SHOWN IN TABLE 6-12. 99

FIGURE 6-11 APPROPRIATE TASK ORDERING FOR TASKS SHOWN IN TABLE 6-12. 100

FIGURE 6-12 POSSIBLE SCHEDULE FOR TASKS SHOWN IN TABLE 6-13. 101

FIGURE 6-13 JITTER-AWARE SCHEDULE FOR TASKS SHOWN IN TABLE 6-13. 102

FIGURE 7-1 FLOW CHART FOR THE TTSA1 ALGORITHM. 107

FIGURE 7-2 FLOW CHART FOR THE CHECK_SCHED() FUNCTION OF TTSA1 ALGORITHM. 108

FIGURE 7-3 NUMBER OF SCHEDULED TASK SETS (3 INTERDEPENDENT TASKS IN EACH SET). 119

FIGURE 7-4 NUMBER OF SCHEDULED TASK SETS (4 INTERDEPENDENT TASKS IN EACH SET). 119

FIGURE 7-5 NUMBER OF SCHEDULED TASK SETS (5 INTERDEPENDENT TASKS IN EACH SET). 120

FIGURE 7-6 NUMBER OF SCHEDULED TASK SETS (50 INTERDEPENDENT TASKS IN EACH SET). 120

FIGURE 8-1 PSEUDO CODE FOR THE TTSA2 ALGORITHM. 126

FIGURE 8-2 NUMBER OF SCHEDULED TASK SETS (3 INTERDEPENDENT TASKS IN EACH SET). 132

FIGURE 8-3 NUMBER OF SCHEDULED TASK SETS (4 INTERDEPENDENT TASKS IN EACH SET). 132

FIGURE 8-4 NUMBER OF SCHEDULED TASK SETS (5 INTERDEPENDENT TASKS IN EACH SET). 133

FIGURE 8-5 NUMBER OF SCHEDULED TASK SETS (50 INTERDEPENDENT TASKS IN EACH SET). 133

FIGURE A-1 THE MAIN FUNCTION OF A TTC SCHEDULER WHICH EXECUTES THREE PERIODIC TASKS, ADAPTED FROM KURIAN AND PONT (2007). 145

FIGURE A-2 THE DISPATCH FUNCTION OF A TTC SCHEDULER, ADAPTED FROM KURIAN AND PONT (2007). 146

FIGURE A-3 THE TICK ISR FUNCTION OF A TTC SCHEDULER, COPIED FROM KURIAN AND PONT (2007). 146

FIGURE A-4 THE TICK ISR FUNCTION OF A TTH, ADAPTED FROM (PONT, 2001). 148

FIGURE B-1 AN OVERVIEW OF THE MP-SA ARCHITECTURE. 150

FIGURE B-2 THE MP-SA HARDWARE. 153

FIGURE B-3 A SIMPLE SCHEDULE BASED ON THE ESTIMATED BCET AND WCET WITH 100 *MS* TICK INTERVAL. 157

FIGURE B-4 EFFECT OF INACCURATE ESTIMATIONS OF BCET AND WCET ON TASK BEHAVIOUR. 157

FIGURE B-5 EFFECT OF SCHEDULER OVERHEAD ON TASK BEHAVIOUR. 158

FIGURE B-6 TASK BEHAVIOUR WITH THE SCHEDULER PRODUCED BY THE MP-SA ARCHITECTURE. 158

FIGURE C-1 ILLUSTRATING OF THE FIRST 3 TICKS FOR THE TASKS SHOWN IN TABLE C-1 AND TABLE C-2 SCHEDULED BY TTC WITHOUT CONSIDERING TASK SEGMENTATION 165

FIGURE C-2 ILLUSTRATING OF THE FIRST 3 TICKS FOR THE TASKS SHOWN IN TABLE C-1 AND TABLE C-2 SCHEDULED BY TTC WITH CONSIDERING TASK SEGMENTATION (TTSA2 ALGORITHM). 165

List of Tables

TABLE 2-1 TASK SPECIFICATIONS FOR A SYSTEM RUNS WITH SIMPLE CYCLIC EXECUTIVE SCHEDULER. 23

TABLE 2-2 TASK SPECIFICATIONS FOR A SYSTEM SCHEDULED BY RM SCHEDULER. 27

TABLE 2-3 TASK SPECIFICATIONS FOR A SYSTEM SCHEDULED BY DM SCHEDULER. 29

TABLE 2-4 TASK SPECIFICATIONS FOR A SYSTEM SCHEDULED BY EDF SCHEDULER. 30

TABLE 2-5 TASK SPECIFICATIONS FOR A SYSTEM SCHEDULED BY LLF SCHEDULER. 32

TABLE 2-6 TASK SPECIFICATIONS FOR A SYSTEM WHICH MAY ENCOUNTER PRIORITY INVERSION 34

TABLE 2-7 TASK SPECIFICATIONS FOR A SYSTEM WHICH MAY SUFFER FROM DEADLOCK. 37

TABLE 3-1 TASK SPECIFICATIONS FOR A SYSTEM IN WHICH THE SCHEDULER STRATEGY DECISION AFFECTS TASK SCHEDULABILITY 45

TABLE 3-2 TASK SPECIFICATIONS FOR A SYSTEM IN WHICH TASK ORDER/STARTING TIMES AFFECTS TASK SCHEDULABILITY. 47

TABLE 4-1 AN EXAMPLE THAT SHOW THE EFFECT OF VARIATIONS OF EXECUTION TIME ON TASK ORDER. 57

TABLE 4-2 AN EXAMPLE THAT SHOW THE EFFECT OF VARIATIONS OF TASK EXECUTION TIME ON JITTER. 59

TABLE 5-1 MINIMUM, MAXIMUM, (MAXIMUM – MINIMUM), AND PERCENTAGE OF VARIATIONS (W.R.T. THE MAXIMUM) IN TASK EXECUTION TIME RESULTED FROM DIFFERENT IMPLEMENTATIONS OF BUBBLE SORT. 81

TABLE 5-2 MAXIMUM JITTER AND AVERAGE POWER CONSUMPTION RESULTED FROM DIFFERENT IMPLEMENTATIONS OF BUBBLE SORT. 81

TABLE 5-3 MINIMUM, MAXIMUM, (MAXIMUM – MINIMUM), AND PERCENTAGE OF VARIATIONS (W.R.T. THE MAXIMUM) IN TASK EXECUTION TIME RESULTED FROM DIFFERENT IMPLEMENTATIONS OF FIBONACCI. 82

TABLE 5-4 MAXIMUM JITTER AND AVERAGE POWER CONSUMPTION RESULTED FROM DIFFERENT IMPLEMENTATIONS OF FIBONACCI. 82

TABLE 5-5 MINIMUM, MAXIMUM, (MAXIMUM – MINIMUM), AND PERCENTAGE OF VARIATIONS (W.R.T. THE MAXIMUM) IN TASK EXECUTION TIME RESULTED FROM DIFFERENT IMPLEMENTATIONS OF MATRIX MULTIPLICATION. 83

TABLE 5-6 MAXIMUM JITTER AND AVERAGE POWER CONSUMPTION RESULTED FROM DIFFERENT IMPLEMENTATIONS OF MATRIX MULTIPLICATION. 83

TABLE 6-1 EXAMPLE OF TASK SPECIFICATIONS SCHEDULED BY TTC. 86

TABLE 6-2 EXAMPLE OF TASK SPECIFICATIONS WHICH CANNOT BE SCHEDULED BY TTC. 88

TABLE 6-3 TASK SPECIFICATIONS FOR A SYSTEM IN WHICH TASK OFFSETS ARE INAPPROPRIATE (TASK C MISSED ITS DEADLINE)..90

TABLE 6-4 TASK SPECIFICATIONS FOR A SYSTEM IN WHICH TASK OFFSETS ARE APPROPRIATE. (ALL TASKS MET THEIR DEADLINES)..91

TABLE 6-5 TASK SPECIFICATIONS FOR A SYSTEM IN WHICH TASK OFFSETS CAUSE HIGH JITTER. 91

TABLE 6-6 TASK SPECIFICATIONS FOR A SYSTEM IN WHICH TASK OFFSETS CAUSE LOW JITTER. 92

TABLE 6-7 TASK SPECIFICATIONS FOR A SYSTEM IN WHICH TICK INTERVAL IS INAPPROPRIATE (TASK B MISSED ITS DEADLINE)..94

TABLE 6-8 TASK SPECIFICATIONS FOR A SYSTEM IN WHICH TICK INTERVAL IS APPROPRIATE (ALL TASKS MET THEIR DEADLINES)...94

TABLE 6-9 TASK SPECIFICATIONS FOR A SYSTEM WITH AN INAPPROPRIATE TICK INTERVAL (TASK B SUFFERS FROM JITTER). 95

TABLE 6-10 TASK SPECIFICATIONS FOR A SYSTEM WITH AN APPROPRIATE TICK INTERVAL (ALL TASKS HAVE LOW JITTER; APPROXIMATELY ZERO). 96

TABLE 6-11 AVERAGE POWER CONSUMPTION (MW) USING DIFFERENT TICK INTERVALS. 97

TABLE 6-12 TASK SPECIFICATIONS FOR A SYSTEM WHERE AN INAPPROPRIATE TASK ORDERING AFFECTS SCHEDULABILITY. 99

TABLE 6-13 TASK SPECIFICATIONS FOR A SYSTEM WHERE AN INAPPROPRIATE TASK ORDERING AFFECTS JITTER. 101

TABLE 7-1 SAMPLE OF TASK SPECIFICATIONS AND CONSTRAINTS (SET OF 3 TASKS) 116

TABLE 7-2 NUMBER OF TRIAL AND THE TOTAL TIME 118

TABLE 8-1 TASK SPECIFICATIONS FOR A TASK SET THAT CANNOT BE SCHEDULED WITH TTC/TTH. 124

TABLE 8-2 TASK SPECIFICATIONS FOR A TASK SET THAT CAN BE SCHEDULED WITH TTC/TTH. 124

TABLE 8-3 NUMBER OF TRIALS AND THE TOTAL TIME. 131

TABLE B-1 TASK SPECIFICATIONS...156

TABLE B-2 EXTENDED TASKS' SPECIFICATIONS. 160

TABLE C-1 TASK SPECIFICATIONS FOR THE APPLICATION EXAMPLE. 163

TABLE C-2 TASK EXCLUSION RELATIONS FOR THE APPLICATION EXAMPLE 164

List of Publications

A number of papers were published during the course of the work described in this thesis. These are listed below (in reverse 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.

Gendy, A. and Pont, M.J. (2008a) "*Automatically configuring time-triggered schedulers for use with resource-constrained, single-processor embedded systems*," IEEE Transactions on Industrial Informatics, vol. 4, no.1, 37 - 46

Gendy, A. and Pont, M.J. (2008b) "*Automating the processes of selecting an appropriate scheduling algorithm and configuring the scheduler implementation for time-triggered embedded systems*," Lecture Notes in Computer Science, Computer Safety, Reliability, and Security, Volume 5219/2008, Springer Berlin / Heidelberg, 27th International Conference on Computer Safety, Reliability and Security, SAFECOMP 2008, 22-25 September 2008, Newcastle upon Tyne, UK , 440-453.

Gendy, A. and Pont, M.J. (2007) "*Towards a generic 'single-path programming' solution with reduced power consumption*", Proceedings of the ASME 2007 International Design Engineering Technical Conferences & Computers and Information in Engineering Conference (IDETC/CIE 2007), September 4-7, 2007, Las Vegas, Nevada, USA

Gendy, A., Dong, L. and Pont, M.J. (2007) "*Improving the performance of time-triggered embedded systems by means of a scheduler agent*", Proceedings of the ASME 2007 International Design Engineering Technical Conferences & Computers and Information in Engineering Conference (IDETC/CIE 2007), September 4-7, 2007, Las Vegas, Nevada, USA.

Gendy, A. and Pont, M.J. (2007) "*Power-aware software design for resource-constrained embedded systems*," Poster presentation at the Festival of Postgraduate Research, 29 July 2007, University of Leicester, Leicester, UK.

Gendy, A. and Pont, M.J. (2006) "*Selecting and implementing a custom scheduler for a reliable embedded application*," Poster presentation at the IET Postgraduate Workshop on Embedded Systems, 11th October 2006 (NEC, Birmingham, in conjunction with the Embedded Systems Show), and Festival of Postgraduate Research, 13 June 2006, University of Leicester, Leicester, UK.

Gendy, A. and Pont, M.J. (in preparation) "Analysis of scheduler/task configuration in time-triggered embedded systems".

Das, A., Lakhani, F., Gendy, A., and Pont, M.J. (accepted) "*Two simple patterns to support the development of reliable, real-time embedded systems*", EUROPLOP 2009 14th European Conference On Pattern Languages Of Program, July 08, 2009 - July 12, 2009, Irsee, Germany.

List of Abbreviations, Symbols and Units

Abbreviations

BCET	Best-Case Execution Time
CB1	Code-Balancing Techniques 1
CPU	Central Processing Unit
DM	Deadline Monotonic
DVS	Dynamic Voltage Scaling
ECG	Electrocardiogram
EDF	Earliest Deadline First
EMI	Electromagnetic Interference
ESL	Embedded Systems Laboratory
ET	Event-Triggered
GCD	Greatest Common Divisor
I/O	Input / Output
ISR	Interrupt Service Routine
LCM	Least Common Multiple
LLF	Least Laxity First
MP	Main Processor
PTTES	Pattern for Time-Triggered Embedded Systems
PC	Personal Computer
RM	Rate Monotonic

RTOS	Real-Time Operating System
SA	Scheduling Agent
SA_1	Segment 1 of Task A
SL	Super Loop
Time(x),	Time spent in performing "x" iterations
TT	Time-Triggered
TTC	Time–Triggered Co-operative
TTH	Time–Triggered Hybrid
TTSA1	Time–Triggered Scheduling Algorithm 1
TTSA2	Time–Triggered Scheduling Algorithm 2
WCET	Worst-Case Execution Time

Symbols

$AbsJitter(T_i)$	The absolute jitter of task T_i
C_i	Execution time of task T_i
$C_i^{(k)}$	The completion time of the kth invocation of task T_i.
D_i	Deadline of task T_i
L_i	Laxity of task T_i
N	Number of tasks
O_i	The offset of task T_i

P_i	Period of task T$_i$
$P_i^{(max)}$	The maximum time intervals between successive completions (or invocation) of task T$_i$;
$P_i^{(min)}$	The minimum time intervals between successive completions (or invocation) of task T$_i$
r_i	Release time of task T$_i$
$S_i^{(k)}$	The start time of the kth invocation of task T$_i$;
u	Unitization

Units

ms	Millisecond
mW	Milliwatt
s	Second
μs	Microsecond

Chapter 1
Introduction

In this introductory chapter, an overview of the work undertaken in this thesis is introduced and the importance of this area is discussed.[1]

1.1 Introduction

We live in a fast-changing world; new ideas are converted to practical products, which reach consumers over a short period of time. For example while it took about 47 years for the major inventions of the 19th century, like telegraph and photography, before their commercial use, the time span is shortened to about 33 years for most of the 20th century inventions; such as telephone and electric railroad, and it shrank even more, to be less than 20 years, for recent inventions, time span was approximately 13 years in case of the cellular telephone (Moore and Simon, 2000).

While the above phenomenon, fast time-to-market, is welcome and preferred, other considerations, such as increasing the reliability and reducing the cost of the product, have to be taken into account. Meeting all goals together is a very difficult task, for example it is concluded that meeting the "faster, better, and cheaper" strategy which was adopted by NASA in the early 1990s, need a lot of hard work and careful design which follow thorough on details, otherwise failure is likely to be the result (Musser, 1995; Gregory, 1996; David, 2000). One of the design considerations which can affect the system reliability is the design of the scheduler, that part of the system which decides when each task in the system should run or should use a resource (Zurawski, 2005). The scheduling problem which caused multiple system resets in the NASA Mars Pathfinder (Reeves, 1997) is a well known example of that kind of problem which may arise from scheduling design errors.

[1] Parts of this chapter have been published previously in Gendy and Pont (2008a)

The focus of this thesis is on the design of schedulers which can be used in low-cost safety-related embedded-systems.

1.2 What is an embedded system?

Embedded systems can be defined as "*information processing systems that are embedded into a larger product and that are normally not visible to the user*" (Marwedel, 2006). This means that, unlike desktop computer systems, embedded systems have more limited interactions with users and have limited resources (such as small size, low processing power, and small memory) which are dequate to complete their specific operation in the product where they reside.

The first appearance of such systems can be dated back to year 1971, the year in which the first microprocessor, the 4004, was produced by Intel to be used in a series of calculators produced by the Japanese company Busicom (Leventhal, 1979; Barr, 1999; Ganssle and Barr, 2003), as shown in Figure 1-1.

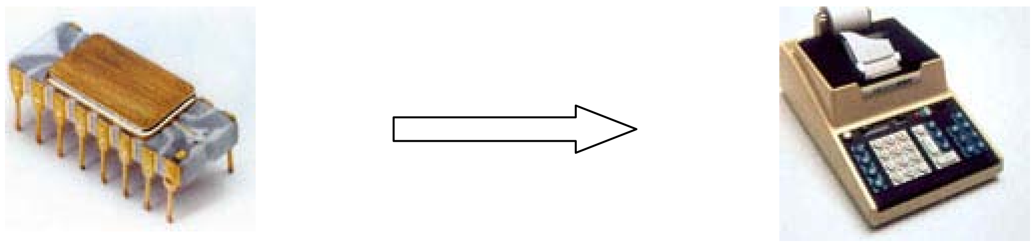


Figure 1-1 Intel 4004 microprocessor and Busicom 141-PF printing calculator. These two images have been used with permission from Intel Museum (Intel, 2009).

Over the years a great deal of work has been done in the development of the microprocessor which was going in two different trends. The first trend was based on building microprocessors to be used in general purpose desktop systems, including PC's and servers. The other trend was based on developing a special purpose microprocessor, typically called a microcontroller, to be used in embedded applications.

The main difference between a microprocessor and a microcontroller is that a microprocessor contains only a central processing unit (CPU) whereas a microcontroller contains a CPU in addition to memory and input/output (I/O) on the chip (Arnold, 2000).

The vast majority of processors sold every year go to the embedded market. For example in 2005 the total number of embedded processors sold was estimated to exceed 3 billion, compared to 200 million desktop computers and 10 million servers (John and David, 2007). This is a result of the increased demand for new devices, which normally include embedded microprocessor, to be used in every aspect of our modern lifestyle. Examples range from the simple devices used in home appliances (such as microwave ovens, mobile phones, washing machines), to the more sophisticated devices used in the medical sector (such as the mobile ECG), or the automotive industry (for example the Mercedes S-class has 63 microprocessors in it and 1999 BMW 7-series has 65) (Turley, 1999).

1.3 What is a real-time system?

Many embedded systems are also real-time systems. In real-time systems the execution of each function (or task) is constrained by a set of temporal requirements, such as task deadline, the time before which the task should finish its execution (Liu and Layland, 1973; Xu and Parnas, 1993; Tindell, 1994; Sandström and Norström, 2002; Buttazzo *et al.*, 2005). Thus real-time systems can be defined as "*computing systems that must react within precise time constraints to events in the environment*" (Buttazzo, 2005a). This means that the key factor that defines a real-time system is that it produces the correct output within the predefined time limit (Cheng, 2002; Laplant, 2004). Note that – contrary to common usage – this does not necessarily imply either that the system must be fast (Stankovic, 1988) or that the response time of the system must be short (Laplant, 2004; Buttazzo, 2005a), as the required response time depends on the application at hand.

According to Laplant (2004) and Buttazzo *et al.* (2005) real-time systems are usually classified into 3 categories (depending on the criticality of the tasks they perform): hard, firm, and soft real-time systems. They showed that in soft real-time systems the system keeps working at low level of performance in case of failure to meet response-time constraints, for example: automated teller machine. Whereas in firm real-time systems only a limited number of missed deadlines are allowed, but missing more than this lead to complete and catastrophic system failure, for example: an embedded navigation controller for autonomous robot weed killer. Finally in hard real-time systems a critical and complete system failure can result if a single deadline is missed, example: avionics weapons. To cope with these timing constraints careful designs (both in terms of hardware and software) must be employed in the development process of such applications.

1.4 Developing real-time systems

It can be inferred from the previous section that developers creating software for use in real-time systems face a very different set of challenges from those creating the majority of "desktop" applications. For example the time interval within which the desktop system should respond to a command may vary significantly without causing a major problem whereas even small levels of variation in task stating time, formally called " release jitter", (milliseconds or much less) in safety-related systems may prove life threatening in (for example) an industrial, automotive or medical system. Hence general desktop software architectures (e.g. desktop operating systems) are not suitable for safety-critical applications (Pont, 2001; Cheng, 2002; Buttazzo *et al.*, 2005; Marwedel, 2006).

Figure 1-2 shows the main services provided by the kernel of a real-time operating system, RTOS, (Kalinsky, 2005). One of the main components of any RTOS is the task scheduler (Cheng, 2002; Kalinsky, 2005; Marwedel, 2006); moreover "*a scheduler can be viewed as a simple operating system that allows tasks to be called periodically or (less commonly) on a one-shot basis*" (Pont, 2001).

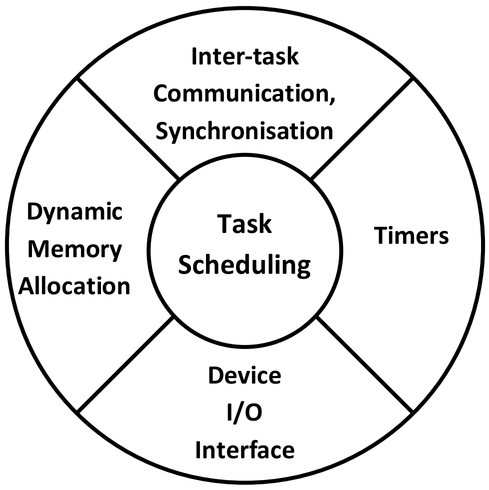

Inter-task Communication, Synchronisation
Dynamic Memory Allocation
Task Scheduling
Timers
Device I/O Interface

Figure 1-2 RTOS basic service groups, redrawn from (Kalinsky, 2005) (Figure 1).

Depending on the type of the application a wide range of software architecture can be used, from a simple scheduler to a complex full RTOS. However, it is desirable to keep system complexity as low as possible (Pont, 2001) as it may affect both the system reliability and cost. For example cost estimates of embedded software (from commencement to shipping) are quoted around US$15-30 per line of code, this increases up to $100 in military defence systems and to approximately $1,000 for highly critical applications, such as the Space Shuttle (Atkinson *et al.*, 2005). The increased system complexity may result in additional demand for hardware resources (such as memory and CPU processing power) and make the debugging process more difficult. For example making a tiny change, changing just three lines of code, when fixing a bug, in the several-million-line signalling program used in the local telephone systems in California and along the Eastern seaboard caused a breakdown in the system in 1991 (Joch and Sharp, 1995).

There are two common approaches used in scheduling real-time embedded systems: event-triggered (ET) schedulers and time-triggered (TT) schedulers. In ET architectures, tasks are invoked as a response to events represented by external interrupts (Albert, 2004; Scheler and Schröder-Preikschat, 2006), whereas in TT architectures tasks are invoked periodically under the control of a timer (Ludemann, 1983; Volz and Mudge, 1987; Ward, 1991; Kopetz, 1997; Pont, 2001). The ET architecture may be used in systems which have many aperiodic events whereas the TT architecture is the preferred choice for safety-related systems in which the task characteristics are known a priori (Kopetz, 1997; Domaratsky and Perevozchikov, 2000; Pont, 2001; Albert, 2004; Scheler and Schröder-Preikschat, 2006). The work in this thesis focuses on TT architectures.

For resource-constrained embedded systems, which have a very limited memory and CPU performance, a simple "time-triggered co-operative" (TTC) – a form of cyclic executive – scheduler (Baker and Shaw, 1988; Burns, 1995; Kopetz, 1997; Huang *et al.*, 2003; Gangoiti *et al.*, 2005), "*which has low run-time overhead*" (Huang *et al.*, 2003), is often used. Furthermore for safety-related applications which have hard real-time constraints, such as low jitter requirements, the TTC architectures demonstrate very low levels of task jitter (Locke, 1992), and can maintain their low-jitter characteristics even when techniques such as dynamic voltage scaling (DVS) are employed to reduce system power consumption (Phatrapornnant and Pont, 2006).

In most TT designs, an "offline" (also known as "pre-runtime", or "static") schedule is said to be the best choice (Xu and Parnas, 2000; Pont, 2001; Huang *et al.*, 2003; Xu, 2003; Gangoiti *et al.*, 2005).

1.5 Scheduling time-triggered systems

The specific implementation options which are considered here are a time-triggered co-operative (TTC) scheduler (a form of cyclic executive: e.g. Shaw (2001)), and a time-triggered "hybrid" (TTH) scheduler. Such architectures are employed frequently in low-cost control systems (e.g. automotive control: (Ayavoo *et al.*, 2005; Ayavoo, 2006)) and in condition-monitoring / fault diagnosis systems (e.g. Schlindwein *et al.* (1988)). A brief overview of these schedulers will be discussed in the following subsections.

1.5.1 Time-triggered cooperative scheduler (TTC)

The TTC implementation discussed in this work is based on the idea of executing each task in predefined time intervals which are derived from a scheduler tick. The scheduler tick is usually signalled by an interrupt associated with the (periodic) overflow of a hardware timer. At each tick the status of each task is updated and tasks which are due to run are dispatched. Then the processor is usually set to an "idle" (power saving) mode, where it will remain until the following tick (in order to reduce the system power consumption)

1.5.2 Time-triggered hybrid scheduler (TTH)

Despite some attractive features, a TTC solution is not always appropriate. For example the system cannot respond to an external critical event while executing specific task if the required response time is shorter than the worst-case execution time (WCET) of the running task plus the time required to handle the event (Allworth, 1981). In these cases a fully pre-emptive architecture such as the rate monotonic (RM) or the earliest deadline first (EDF) can be used (Liu and Layland, 1973). Such an approach provides flexibility (and possibly, portability), but it will also tend to increase the system complexity and overhead when compared to pre-run-time scheduling (Xu and Parnas, 2000; Xu, 2003).

In some designs the system responsiveness can be increased while maintaining the minimal resource requirements, by allowing a limited level of pre-emption in the system. This can be done by using what is called a "time-triggered hybrid" (TTH) scheduler (Pont, 2001; Maaita and Pont, 2005a), sometimes called a "multi-rate executive with interrupts" (Kalinsky, 2001). The TTH scheduler can be seen as a RM scheduler that supports a single, short, high priority, pre-empting task, and a collection of co-operative tasks (which have equal priorities lower than that of the pre-empting task).

The pre-empting task may be used for periodic data acquisition, typically by means of an analogue-to-digital converter or similar device. Such requirements are common in, for example, control systems (Buttazzo, 2005b), and applications which involve data sampling and Fast-Fourier transforms (FFTs) or similar techniques: an example is given in the work by Schlindwein *et al.* (1988).

1.6 Challenges with simple TT architecture

Two key challenges facing the developers of simple TTC and TTH designs are the schedule fragility (at design time) and the possibility of task overruns (at run time). These challenges are considered in this section.

1.6.1 The fragility of TTH and TTC designs

It has been shown that – during the design process – TTC / TTH designs are "fragile": that is, small changes to the timing of particular tasks can mean that the developer has to make substantial changes to the whole schedule (e.g. Shaw (2001)). Moreover, it has been demonstrated in previous studies that the problem of testing the schedulability and determining the scheduler and task parameters for a set of tasks for such a system is NP-hard (Brucker *et al.*, 1977; Baker and Shaw, 1988; Tindell *et al.*, 1992; Xu and Parnas, 1992; Xu and Parnas, 2000; Ekelin and Jonsson, 2001; Cucu and Sorel, 2004; Baruah, 2006). Inappropriate choices of parameters may mean that a given task set cannot be scheduled at all. Where the parameter set does ensure that all tasks are scheduled, inappropriate decisions may still lead to unnecessarily high levels of task jitter and / or to increased system power consumption. The focus in this thesis is therefore on developing ways in which the process of configuring TT schedulers for use in single-processor embedded systems can be automated.

1.6.2 Impact of long tasks during system execution

As discussed in the previous section, TTH architectures allow a designer to execute one or more tasks with long WCETs and also respond within a short time interval to external events. This solution can be effective, for many designs, if the WCET of every task is known at design time. Unfortunately, as many researchers have observed (Nett *et al.*, 1996; Domaratsky and Perevozchikov, 2000; Engblom and Ermedahl, 2000; Engblom and Jonsson, 2002; Gergeleit and Nett, 2002; Burguiere and Rochange, 2005; Deverge and Puaut, 2005; Kirner and Puschner, 2008), determining the WCET of tasks is rarely straightforward.

Lack of knowledge about WCETs is a problem which faces the developers of many embedded systems (not just those based on TTC / TTH designs). For example, as Gergeleit and Nett have noted: "*Nearly all known real-time scheduling approaches rely on the knowledge of WCETs for all tasks of the system*." (Gergeleit and Nett, 2002). Nonetheless, the fact that a TTC / TTH architectures employs static scheduling (and, even in the case of TTH, a very limited degree of pre-emption) means that – in the event of a task overrun – the problem may not even be detected (let alone resolved). This may have a serious impact on the system behaviour. For example, as Buttazzo has noted:

"[Co-operative] *scheduling is fragile during overload situations, since a task exceeding its predicted execution time could generate (if not aborted) a domino effect on the subsequent tasks*" (Buttazzo, 2005b).

As part of an effort to address these problems, the work presented here introduces a set of novel code-balancing techniques which helps to reduce variations in task execution time to a value equal to its WCET. Unlike other methods, these techniques can be adapted to be used with any hardware and have limited impact on system power consumption.

1.7 Aims of the thesis

Despite the attraction of using an offline (or pre-run time) schedulers, such as the TTC and TTH schedulers described above, "*Builders of real-time systems often use priority scheduling in their systems without considering alternatives*" (Xu and Parnas, 2000). The main reason beyond the above argument can be related to the "fragility" of the offline designs of schedulers. It is generally assumed that the effort involved in such a rescheduling process will be very significant. Such arguments have been used in the past as a reason for avoiding TT architectures.

While it can be argued that the issue of TTC/TTH fragility can be exaggerated in situations where appropriate design decisions are taken, it is true that re-scheduling may be required during the development and maintenance of TTC/TTH designs. Given that such a schedule re-design may be required, the aim of this thesis is to explore techniques which can reduce the effort involved in such a process via the use of novel scheduling algorithms.

The second aim of the project is to increase the predictability of systems which use TTC/TTH designs. The proposed scheduling algorithms introduced in this thesis attempt to find a workable schedule that satisfies all task constraints. Like other scheduling algorithms, the proposed algorithms introduced here rely on the availability of accurate estimates of the upper bound of task execution time at design time. Unfortunately determining WCET values is becoming more challenging as embedded designs become more complex and make use of faster and smaller processors and "system on chip" architectures. The work introduced in this thesis introduces new techniques which aim to reduce variations in task execution time that will in turn reduce difficulties in obtaining the task WCET. More importantly stabilising task execution time will increase the system predictability and determinism, for example the points at which each instance of the task starts, and finishes, can be known in advance.

1.8 Thesis contributions

The project described in this thesis made the following contributions to this research area:

First, problems facing developers of time-triggered architecture (particularly TTC and TTH), such as the need for stabilising the task execution time and carefully choosing task / scheduler parameters are identified. Effects of variations of task execution times, on violating task constraints and / or decreasing the system reliability, are discussed and analysed. Then a set of code-balancing techniques which helps to reduce variations in the task execution time to its WCET, while avoiding excessive increase in power consumption which faces other methods, are introduced.

Second, the need to appropriately choose the right scheduling strategy and configure the task and scheduler parameters is discussed. The effects of inappropriate choices of the scheduler and / or task parameters (such as task offset, task order and tick interval) on task schedulability and system power consumption are discussed and analysed.

Finally, a proposed TTSA1 algorithm which can be used to automate the process of scheduler selection and configuration, while reducing power consumption, for TT schedulers is presented. The proposed algorithm uses a novel two-stage search technique and is intended to support the configuration of time-triggered schedulers for use with resource-constrained embedded systems which employ a single processor. The overall goal is to identify a scheduler implementation which will ensure that: (i) all task constraints are met; (ii) CPU power consumption is "as low as possible"; (iii) a fully co-operative scheduler architecture is employed whenever possible.

The performance of the proposed TTSA1 algorithm is improved by developing the TTSA2 algorithm. This algorithm tries to increase the chance of finding a suitable schedule by dividing some tasks into two, or more, segments, in cases were a suitable schedule can not be found when scheduling each task as one segment. It assumes that the points at which a task can / cannot be pre-empted / divided into two or more tasks are known in advance.

1.9 Thesis outline

Following this introductory chapter which sets up the background and aims of the current work, Chapter 2 gives an overview of the task model, task parameters and constraints which are normally used in scheduler design. It then discusses various scheduling strategies introduced in the literature.

Chapter 3 reviews previous work in scheduler design and the need for automatic schedule generation in embedded systems.

Chapter 4 discusses the problems encountered form variations in task execution times and the challenges involved in the process of estimating accurate values for task WCET.

Chapter 5 introduces a new set of code-balancing techniques which help to reduce variations in the task execution time, and hence reduce jitter and increase predictability, while avoiding the unnecessary high increase in power consumption caused by other methods.

Chapter 6 discusses the need for choosing the appropriate scheduling strategy and configuring the task and scheduler parameters. The effects of inappropriate choices are also discussed and analysed.

Chapter 7 introduces and evaluates a new heuristic scheduling algorithm "TTSA1" which helps to automate the process of scheduler selection and configuration for single-processor embedded systems. It then discusses the scheduling overhead and its effects on task schedulability. It also introduces an easy way of measuring this overhead and taking its effects into account while designing the scheduler.

In Chapter 8 an improved scheduling algorithm, TTSA2, is presented. The TTSA2 algorithm assumes that points at which a task can be divided / pre-empted (such as critical sections boundaries) are predefined. The TTSA2 uses this information to increase the chance of finding a feasible scheduler for a given task set by dividing one, or more, long tasks into two, or more, segments in case where it is not possible to schedule each task as one segment.

Finally Chapter 9 discusses the work presented in the thesis, gives the conclusions and future work.

1.10 Conclusions

This chapter has presented a background about embedded systems design and emphasised on time-triggered approaches which is to be preferred in safety-related systems. Two main problems which face developers of such systems, scheduler fragility, and difficulties in having accurate estimates of WCET, have been discussed. Techniques to overcome these problems will form the main focus of this document.

Chapter 2
Scheduling strategies

This chapter gives an overview of the task model, task parameters and constraints which are normally used in scheduler design. Then it discusses various scheduling criteria[2] [3]introduced in the literature.

2.1 Task characteristics

Embedded applications are usually implemented as a collection of communicating tasks (Shaw, 2001). In the design phase of such applications each task is used to perform certain function(s). In embedded systems in general, and in safety-related applications in particular, it is not sufficient that the required function is implemented correctly to produce the right output but it has to produce this output at the right time as well (Stankovic, 1988; Cheng, 2002; Laplant, 2004). In order to achieve this each task is assigned a set of timing parameters and constraints, such as period and deadline.

Tasks in embedded application are classified into 3 categories: periodic, aperiodic, and sporadic.

Periodic tasks are repeated or activated frequently; the time between two activations is called the task period (P) (Liu and Layland, 1973). Typical examples of such tasks include sampling and processing of data (Jeffay *et al.*, 1991).

Sporadic tasks are those tasks which are not activated regularly at fixed time intervals but rather they have minimum inter-arrival times. Sporadic tasks can be represented as periodic tasks with periods equal to the minimum inter-arrival time of the equivalent sporadic tasks (Jeffay *et al.*, 1991). A typical example of such tasks includes an alarm or the emergency shutdown of a production robot (Zurawski, 2005).

[2] Scheduling criterion describes the basic features of the scheduler (such as TT or ET, online or offline, etc).

[3] Scheduling strategy describes the way in which specific property (such as the task period) is assigned to each task.

Aperiodic tasks are those tasks which are activated irregularly. Their invocation time is not known in advance as they can be activated at any time. Typical examples of such tasks include operator’s commands and exception handling subroutines (Lin and Tarng, 1991).

The work presented here is concerned with high reliability safety-related applications.

According to the IEC 61508 (IEC, 2005) "The term safety-related is used to describe systems that are required to perform a specific function or functions to ensure risks are kept at an accepted level.[4]". The IEC 61508 is an international standard concerned with safety-related electronic and/or programmable electronic systems.

In the worked presented in this thesis it is assumed that task characteristics are known in advance and all the tasks in the system are periodic tasks (and sporadic tasks which are replaced by their equivalent periodic tasks). The following parameters are usually used to characterise each task (Liu and Layland, 1973; Tindell, 1994; Buttazzo *et al.*, 2005), and are shown in Figure 2-1.

- Period (P_i): is the time interval after which task T_i should be repeated, in another word it is the length of time between every two invocations.
- Offset (O_i): is the time, measured from the start of the system power on, after which the first period of task T_i starts.
- Release time (r_i): is the time, measured from the start of the task period, after which task T_i becomes ready to run. In the rest of this document, it will be assumed that all tasks have release time equal to zero; otherwise release time will be explicitly stated.
- Worst-case execution time ($WCET_i$): is the longest time taken by the processor to execute task T_i without pre-emption.

[4] This text contains extracts from the IEC Functional Safety Zone. All such extracts are copyright of International Electrotechnical Commission © 2005, IEC, Geneva, Switzerland. All rights reserved. IEC has no responsibility for the placement and context in which the extracts are reproduced. This notice takes precedence over any general copyright statement.

- Best-case execution time ($BCET_i$): is the shortest time taken by the processor to execute task T_i.
- Deadline (D_i): is the time before which task T_i should be completed. Deadline can be measured from the start of the system power on, in which case it is called absolute deadline. Alternatively it can be measured from the start of the task period, in which case it is called relative deadline; this is illustrated in Figure 2-1.

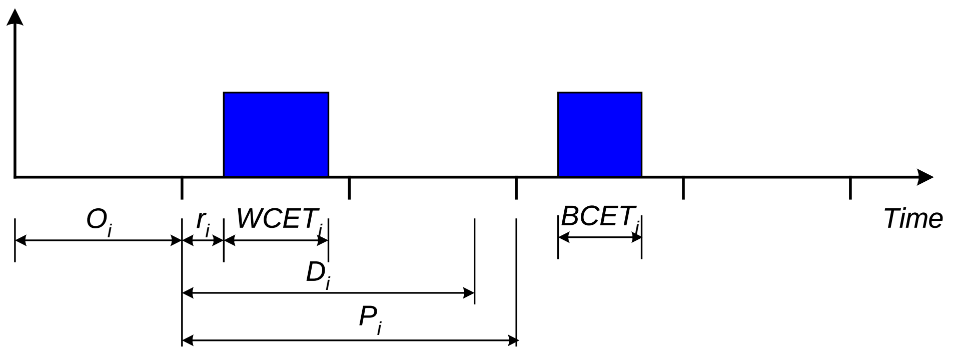


Figure 2-1 Typical task parameters.

2.2 Task constraints

As embedded systems become widespread and more complex, developers need ways to specify various application requirements. This is normally done by specifying some restrictions that govern the way in which each task runs and the way in which it interacts with other tasks. These restrictions are usually represented by specifying one or more of the task constraints described below.

2.2.1 Jitter

Many real-time applications require tasks to run at specific time instants, even small variations of these times may cause problems. For example, in multimedia applications, such as CD audio or video, data must be displayed/replayed under relative timing constraints: sample K+1 must be played no later than a fixed interval (e.g., 125 μs) once sample *K* is played (Han *et al.*, 1996). Variations in this time will affect the playback quality.

Another example is a chemical process control system. In such systems it is important that all necessary ingredients are added in at the right times (Han *et al.*, 1996). When scheduling a moving cart to ship the ingredients to the container, it is important to have the cart come in regular intervals so that one ingredient must be added into the container within a certain time after another has been put in (Han *et al.*, 1996).

A final example is the sampling task that used to measure the height of the aircraft. Variations in the interval of executing this task will results in inaccuracy of the aircraft estimated height.

"*Output* [or input] *jitter refers to the variation between the inter-completion* [or activation] *times of successive jobs of the same task"* (Baruah *et al.*, 1999). Jitter can be caused by either hardware or software factors or both. Hardware factors include a drift in the oscillator frequency, noise, or crosstalk caused by electromagnetic interference (EMI) along a circuit or a cable pair (a further discussion are given in Phatrapornnant (2007)). On the other hand software factors include variation in task execution time, task pre-emption, or inappropriate design of the scheduler (Ayavoo *et al.*, 2007; Short and Pont, 2007; Hughes and Pont, 2008). The current work considers only the jitter which results from software factors.

According to Baruah *et al* (1999) the absolute jitter of task T$_i$ "AbsJitter(T$_i$)" is defined as:

$$AbsJitter(T_i) \stackrel{\text{def}}{=} \max(P_i^{(max)} - P_i, P_i - P_i^{(min)}) \qquad \textbf{Equation 2-1}$$

where:

$$P_i^{(min)} \stackrel{\text{def}}{=} \min_{k\geq 0}\{C_i^{(k+1)} - C_i^{(k)}\} \qquad \textbf{Equation 2-2}$$

or

$$P_i^{(min)} \stackrel{\text{def}}{=} \min_{k\geq 0}\{S_i^{(k+1)} - S_i^{(k)}\}; \qquad \textbf{Equation 2-3}$$

and

$$P_i^{(max)} \overset{\text{def}}{=} \max_{k\geq 0}\{C_i^{(k+1)} - C_i^{(k)}\}$$ **Equation 2-4**

or

$$P_i^{(max)} \overset{\text{def}}{=} \max_{k\geq 0}\{S_i^{(k+1)} - S_i^{(k)}\};$$ **Equation 2-5**

$P_i^{(min)}$ denotes the minimum time intervals between successive completions (or invocation) of task T$_i$;

$P_i^{(max)}$ denotes the maximum time intervals between successive completions (or invocation) of task T$_i$;

$S_i^{(k)}$ denotes the start time of the kth invocation of task T$_i$;

$C_i^{(k)}$ denotes the completion time of the kth invocation of task T$_i$.

For simplicity of notation the "AbsJitter(T$_i$)" will be simply donated as "Jitter(T$_i$)" in the remainder of this document.

2.2.2 Precedence

Precedence constraints are used to specify the execution order between two tasks (Sandström and Norström, 2002). For example in control application it is required that the sampling process (sampling task) is done first and that it is followed by calculating the control algorithm (control task) and finally the actuation process (actuation task) is run at the end. So the sampling task should precede the control task which in turn should precede the actuation task.

If it is required that task T$_i$ precedes task T$_j$, then, in any tick, task T$_j$ is allowed to start its execution only after task T$_i$ completes its execution, i.e. for any *k*:

$$C_i^{(k)} \leq S_j^{(k)}$$ **Equation 2-6**

where: *k* is the *k* th invocation of task T$_i$ and task T$_j$.

2.2.3 Exclusion

Exclusion constraints are used to maintain data consistency and control access to shared resources (Buttazzo, 2005a). For example if two tasks, task T$_i$ and task T$_{j,}$, share one (or more) variable(s), in order to prevent simultaneous updating process of the shared variable(s) by the two tasks an exclusion relation is specified between them. This means that task T$_i$ is not allowed to pre-empt task T$_j$ and vice versa; i.e. if task T$_i$ starts its execution before task T$_j$ then task T$_j$ is not allowed to start is execution before task T$_i$ completes its execution (Xu and Parnas, 1990) and vice versa. This is if task T$_i$ excludes task T$_j$ this means that:

$$\text{if } S_i^{(k)} < S_j^{(k)} \Rightarrow C_i^{(k)} \le S_j^{(k)}$$ **Equation 2-7**

and

$$\text{if } S_j^{(k)} < S_i^{(k)} \Rightarrow C_j^{(k)} \le S_i^{(k)}$$ **Equation 2-8**

2.2.4 Distance

The distance constraint is defined as the minimum time interval between the completion of one task and the start of another task (Sandström and Norström, 2002). A reason for this could be the delays in the communication hardware between two communicating tasks, or the limitations in the processing speed of the environment that the tasks interact with (Ekelin and Jonsson, 1999).

If the distance between task T$_i$ and task T$_j$ is required to be Distance(i,j) this means that:

$$S_j^{(k)} - C_i^{(k)} \ge Distance(i,j)$$ **Equation 2-9**

2.2.5 Latency

The latency relation between any two tasks can be defined as the maximum duration of time between the start of one task and the completion of another task (Sandström and Norström, 2002). For example in control applications it may be required that the time interval between stating the sampling task to the completion of the actuation task does not exceed a predefined value.

If the latency between task T_i and task task T_j is required to be Latency(i,j) this means that:

$$C_j^{(k)} - S_i^{(k)} \leq Latency(i, j)$$ **Equation 2-10**

2.3 Scheduling criteria

As explained in the previous sections, embedded applications are normally developed as a collection of tasks which should run under certain constraints to ensure that the system generates the correct output within the required time interval.

The time and order in which each task should be activated can be configured through the use of an appropriate scheduler. A feasible schedule is that in which all task constraints are met, in which case the set of tasks is called a "schedulable" task set. The scheduling algorithm which is able to find a feasible schedule for any schedulable task set is called an "optimal scheduling algorithm" (Buttazzo, 2005a). In another words an optimal scheduling algorithm can find a feasible (not necessary the best) schedule for a given task set if any other scheduler can, i.e. this optimality is not mean related to the quality of the schedule produced by the optimal scheduler but it is related to its ability to find a feasible schedule.

The following subsections discuss various scheduling criteria.

2.3.1 Event-triggered and time-triggered scheduling

The event-triggered and time-triggered scheduling differs in the way in which tasks are called. If tasks are invoked as a response to the occurrence of events represented by external interrupts then the scheduler is event-triggered (ET), whereas tasks in time-triggered (TT) architectures are invoked in a predefined time intervals under the control of timer (Kopetz, 1993; Kopetz, 1997; Albert, 2004).

The question of whether to use the ET or TT architecture has been debated and discussed intensively in the literature (Kopetz, 1991; Kopetz, 1993; Kopetz, 1997;

Domaratsky and Perevozchikov, 2000; Pont, 2001; Albert, 2004; Scheler and Schröder-Preikschat, 2006).

According to Albert: "*In general, reality is neither black nor white but rather gray. Thus it depends on the application whether a time-triggered or event-triggered behavior is more suitable*" (Albert, 2004). The ET approach may prove cost effective in cases where the system must handle many aperiodic and sporadic events (Kopetz, 1993; Albert, 2004; Scheler and Schröder-Preikschat, 2006), since the conversion of such events to periodic events may reduce the system utilisation. On the other hand time-triggered systems are considered as the preferred choice for supporting safety-related applications (Kopetz, 1997; Domaratsky and Perevozchikov, 2000; Pont, 2001; Scheler and Schröder-Preikschat, 2006). The work in this document focuses on systems with a TT architecture.

2.3.2 Pre-emptive and non pre-emptive (co-operative) scheduling

In pre-emptive schedulers a high priority task can pre-empt a lower priority one if the higher priority task becomes ready to run while the lower priority task is still running (Liu and Layland, 1973; Pont, 2001). The context of the pre-empted task (the lower priority task) is saved to enable it later to continue its execution from the point at which it is pre-empted; this context is loaded when the pre-empting task (the higher priority task) finishes its execution. The disadvantages of the pre-emptive scheduler are the imposed overhead necessary for performing this context switching and the necessity for developing mechanisms to manage access to shared resources, so it is desirable to keep the context switching (or the number of pre-emptions) as fewer as possible (Jeffay *et al.*, 1991; Joseph, 1996). On the other hand the main advantage of the pre-emptive scheduling strategy is its responsiveness: when a high priority task becomes ready to run it will immediately gain control of the CPU by pre-empting the current running task (Labrosse, 2002).

By contrast, in non pre-emptive (also called co-operative) schedulers, task(s) is executed to completion without being pre-empted (Pont, 2001; Baruah, 2006). The main drawback of this strategy is its latency to responding to important events: a higher priority task will have to wait until the current running task finishes its execution (Labrosse, 2002). This problem can be solved by dividing long task(s) into two or more shorter tasks (Baker and Shaw, 1988; Locke, 1992; Pont, 2001). The non pre-emptive scheduling has the advantage of requiring low overhead and simple design (as it guarantees exclusive access to shared resources and it has low context switch overhead, the context switch only happen when the task finishes its execution) (Jeffay *et al.*, 1991; Labrosse, 2002). Another advantage is that the interrupt latency is typically low (Labrosse, 2002).

2.3.3 Static priority and dynamic priority scheduling

In priority scheduling each task is assigned a specific priority and the schedule is generated based on these priorities (Buttazzo, 2005b). Task priority can be either fixed or dynamic. In the case of static (or fixed) task priority scheduling the priority of each task is decided at design stage and does not change afterward. An example of this scheduler is the rate monotonic (RM) scheduler in which the priority of each task is assigned based on the task period; the shorter the task period the higher its priority (Liu and Layland, 1973). By contrast, in dynamic priority scheduling the priority of each task is dynamically assigned and can be changed at runtime (Buttazzo, 2005b). An example of such scheduler is the earliest deadline first (EDF) scheduler in which the priority of each task depends on its absolute deadline, the sooner the task deadline, with respect to other tasks' deadlines, the higher the task priority (Liu and Layland, 1973).

The main advantage of priority scheduling is its flexibility in handling dynamic situations such as creating a new task, or changing one or more of the task characteristics (such as task period) (Buttazzo, 2005b). On the other hand this flexibility comes at the price of increasing the system complexity, especially in the case of using dynamic priority scheduling. In such systems careful analysis of the scheduler and resource allocation techniques has to be made to avoid problems which can arise during run time. An example of such problems is the famous priority inversion problem (Sha *et al.*, 1990) which will be discussed later in this chapter.

A well known example of a situation in which such problem is encountered and caused multiple systems resets is the NASA Mars Pathfinder (Reeves, 1997).

2.3.4 Offline and online scheduling

In online schedulers, scheduling decisions are taken at run-time (Marwedel, 2006). This approach is normally used in systems in which it is likely that a task(s) can be added / removed during run time, such as a team of robots cleaning up a chemical spill (Stankovic *et al.*, 1995). These schedulers generate overhead at run-time, but are quite flexible (Marwedel, 2006).

By contrast, in offline schedulers, scheduling decisions are taken at design time (Xu, 1993; Marwedel, 2006). These schedulers are normally used in cases where a complete knowledge of task properties and constraints are known in advance (at design time), such as control applications which have well defined environment and processing requirements and uses fixed sets of sensors and actuators (Stankovic *et al.*, 1995). Offline schedulers are usually based on pre-run-time analysis of the system. The scheduler is often computed for a period of time equals to the least common multiple (LCM) of task periods, which is the period of time after which the same sequence of task calling is repeated over time (Xu and Parnas, 1993). The main advantage of pre-run-time scheduling is that satisfying all the deadlines and other constraints are easy to verify (Xu and Parnas, 2000).

Finally it remains the case that the choice of certain architecture depends on the application at hand. As noted by Laplant "*even after more than 30 years of research there is no methodology available that answers all of the challenges of real-time specification and design all the time and for all applications*." (Laplant, 2004).

A wide range of schedulers can be used in real-time systems based on the scheduling criteria discussed above. These schedulers have different behaviours. The following sections will discuss the most commonly used schedulers.

2.4 Cyclic executives

As its name implies, in cyclic executive based schedulers the sequence and pattern at which tasks (which are normally periodic) are executed takes the form of a cycle which is repeated over time. This cycle is usually called the "major cycle". The length of this major cycle is taken as the least common multiple (LCM) of all tasks' periods. The major cycle is divided into a number of smaller cycles called "minor cycles" (in TT architectures they are called "ticks"). Normally the length of the minor cycle is taken as the greatest common divisor (GCD) of the tasks' periods.

Although they have been used in many applications for a long time, the first formal description of cyclic executives is introduced by Baker and Shaw (1988). This work is followed by many studies and discussions over the years (Baker and Shaw, 1988; Locke, 1992; Burns, 1995; Kopetz, 1997; Zamorano *et al.*, 1997; Pont, 2001). Recent uses of such simple schedulers can be noted. For example Huang *et al*. (2003) used a fixed-polling binary tree for USB bandwidth scheduling using a cyclic-executive-based approach "*which has low run-time overhead*" (Huang *et al.*, 2003). They showed that this method helped to guarantee the quality of service requirements for the device. In another recent study, Gangoiti *et al.*(2005) used co-simulation of different tools employed in PLC programming and process modelling and simulation. They used a cyclic executive design to mark the time instants in which data exchange must be performed for each control loop, in order to facilitate the design of the simulation steps in both tools.

Cyclic executives based schedulers are usually used in offline scheduling, in which scheduling decisions are taken at the design time.

Figure 2-2 shows an example of a simple cyclic executive scheduler which has 3 tasks (Task A; Task B, Task C) with fixed durations of 3, 4, and 5 *ms* respectively; assuming that each has a period of 30 *ms*, as shown in Table 2-1 (Pont, 2001; Kurian and Pont, 2007).

Table 2-1 Task specifications for a system runs with simple cyclic executive scheduler.

Task	WCET (*ms*)	Deadline (*ms*)	Period (*ms*)
A	3	30	30
B	4	30	30
C	5	30	30

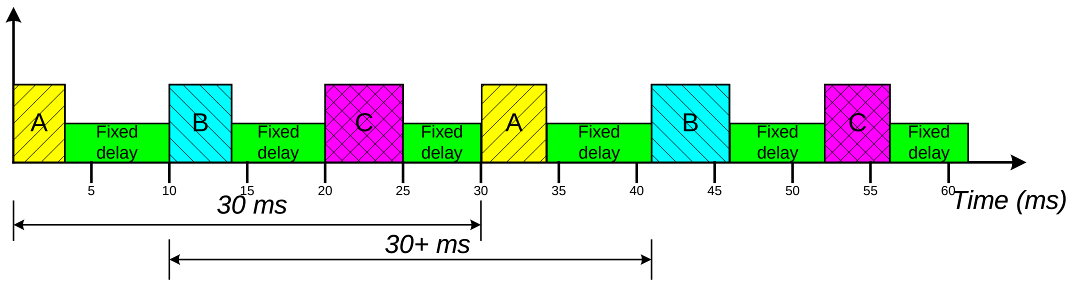


Figure 2-2 Cyclic executive scheduler for the task set shown in Table 2-1.

The major advantage of this scheduler is its simple implementation and small resource requirements (Pont, 2001; Kurian and Pont, 2007). On the other hand, the main disadvantage of this scheduler is that it is difficult to obtain a fixed period length of tasks if the tasks execution times are not fixed (Kurian and Pont, 2007). This will, in turn, have bad effects on task jitter and / or may cause some tasks to miss their deadlines. This problem can be solved by careful use of timers, such as that which is used in TTC and TTH schedulers, further details are given in Appendix A.

2.4.1 Common problems with cyclic executive schedulers

The main problems facing developers of cyclic executive based scheduler are mainly similar to those problems facing designers of TTC and TTH (the possibility of task overruns (at run time) and the schedule fragility (at design time)). These problems along with the proposed solutions were previously discussed in Section 1.6.

2.5 Priority schedulers

Priority-based schedulers are usually used in online scheduling, in which scheduling decisions are taken at run-time; when a running task finishes its execution or a new task enters the system (Cottet *et al.*, 2002). As described above task priority can either be fixed and never changed once it is assigned to the task (fixed priority scheduling) or can be changed during the run-time (dynamic priority scheduling).

The following subsections discuss the most common priority-based scheduling strategies.

2.5.1 Fixed priority schedulers

As discussed earlier, in fixed priority schedulers once a certain priority is assigned to a task it does not change at run-time. This priority is usually based on a fixed parameter which is assigned to the task before its activation (Liu and Layland, 1973; Cottet *et al.*, 2002). Examples of such schedulers are the rate monotonic and deadline monotonic algorithms.

2.5.1.1 Rate monotonic scheduler

Rate monotonic (RM) scheduler is said to be as the most commonly used priority assignment scheduling strategy used in real-time systems (Buttazzo, 2005b). The RM scheduler assigns fixed priority to tasks according to their rate; tasks with shorter periods are assigned higher priorities (Liu and Layland, 1973).

Bini *et al.* (2003) gave an example of possible use of the RM scheduler in a radar system which is used to track a number of moving targets. Assuming that each of these targets moves with a constant speed, which may be different from other targets' speeds, the RM can be used to assign priorities to tasks according to their speeds, high speed task gets high priority. As targets dynamically enter and exit the visual field of the radar system, corresponding tasks are activated and removed from the schedule. When a new target enters the visual field of the radar system an acceptance test (or utilisation test, described below) is done to check the schedulability of the new task and make sure that accepting it will not violate the task constraints of the tasks already run in the system. If the acceptance test fails an appropriate action can be taken, such as allocating additional computational resources (if available), or activating an alarm.

The interest of this scheduler was initiated by a significant study introduced in 1973 by Liu and Layland (1973). In this study tasks are assumed to be independent, periodic, can be pre-empted at any time, have relative deadlines equal to periods, and have fixed execution times. They showed that a set of *n* tasks can be scheduled with RM if the total processor utilisation (*u*) satisfies the following sufficient (but not necessary) condition:

$$u = \sum_{i=1}^{n} \frac{e_i}{P_i} \leq n(2^{\frac{1}{n}} - 1) \qquad \textbf{Equation 2-11}$$

where:

n is the number of tasks;

e_i is the execution time of task T_i;

P_i is the period of task T_i.

This gives a least upper bound of utilisation of 0.69 for large values of n (as n tends to infinity). This means that any task set is guaranteed to be scheduled by RM if $u \leq 0.69$, but not all task sets can be scheduled if $0.69 < u \leq 1$, for example when all pairs of periods in the task set are in harmonic relation the tasks can still be scheduled by RM (ignoring the other overheads) although the total utilisation of the tasks can be 1.00 (Buttazzo, 2005b). It has been shown that RM is optimal in the sense that if there exist a feasible priority assignment schedule for a given task set, then the RM is also feasible for that task set (Liu and Layland, 1973).

Since 1973 a great deal of work has been done by many researchers to improve the efficiency and relax the assumptions of the RM introduced by Liu and Layland. For example Bini *et al.*(2003) proposed a less pessimistic acceptance ratio, yet their formulation has the same complexity, which allows the acceptance of task sets that would be rejected by using the Liu and Layland original schedulability test. Sha *et al.* (1990) extended the original model, which assumes that all the tasks are independent, and propose novel protocols (the Priority Inheritance Protocol and the Priority Ceiling Protocol) which solve the problems that may arise in the existence of shared resources (such as task priority inversion and deadlock).

Figure 2-3 shows an example of three tasks run with RM. The specifications of these tasks are shown in Table 2-2. It can be noticed that the condition in Equation 2-11 is satisfied as the total processor utilisation is 0.467 which is less than the upper bound of utilisation for three tasks (0.78).

It should be noted that that the highest priority is indicated by the smallest number. Arrows pointing down (as those shown Figure 2-3) will be used to mark the task deadline. These notations will be used in the rest of this document.

Table 2-2 Task specifications for a system scheduled by RM scheduler.

Task	WCET (*ms*)	Deadline (*ms*)	Period (*ms*)	Priority
A	1	5	5	0
B	3	15	15	1
C	2	30	30	2

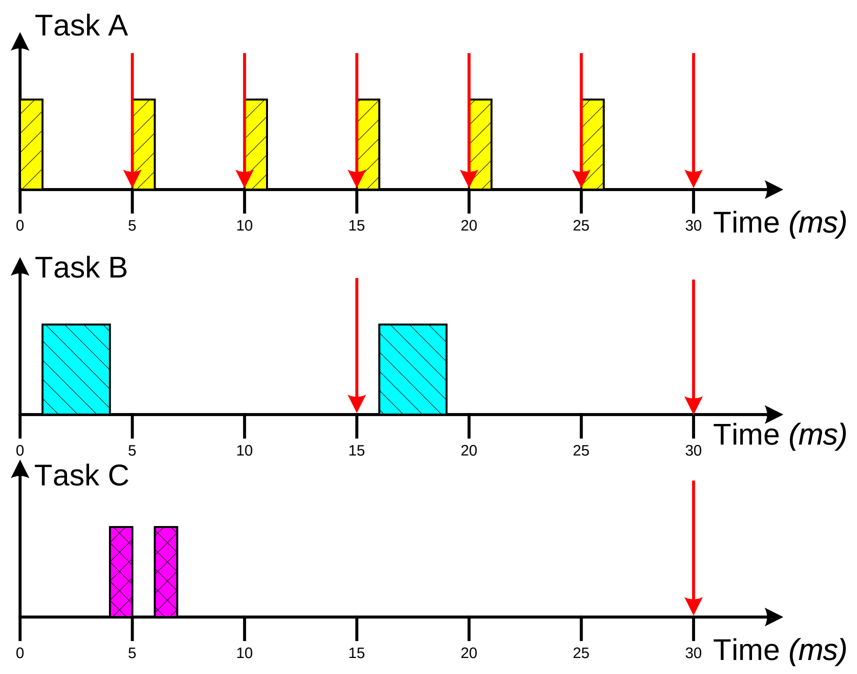


Figure 2-3 RM schedule of the task set shown in Table 2-2.

2.5.1.2 Deadline monotonic scheduler

The Deadline Monotonic (DM), or sometimes called inverse deadline, scheduler weakens the rate monotonic's "period equals deadline" constraint by assuming that tasks' deadlines can be less than their periods (Leung and Whitehead, 1982; Audsley *et al.*, 1991). DM scheduler was first introduced by Leung and Whitehead (1982). According to DM scheduling technique fixed priorities are assigned to tasks based on their relative deadlines; tasks with smaller relative deadlines are assigned higher priorities.

Audsley *et al.* (1991) showed that the DM scheduler provides the application designer with more flexible process model by discussing some motivating applications. For example they showed that DM can be used to easily include the effect of the inevitable communication delay in distributed systems which have precedence constraints among periodic processes. To maintain the precedence constraints while taking the effect of this delay into account, the deadline of these periodic processes must be set to a value less than the end of their periods, the deference between the period and the deadline is considered as a dead time represents the communication delay.

DM is optimal in the sense that if there exist a feasible fixed priority assignment schedule for some task set, where deadline less than periods, then the DM is also feasible for that task set (Leung and Merrill, 1980; Audsley *et al.*, 1991). It has been shown that a set of *n* tasks can be scheduled with DM if the processor unitization (*u*) satisfies the following sufficient condition (Cottet *et al.*, 2002):

$$u = \sum_{i=1}^{n} \frac{e_i}{D_i} \le n(2^{\frac{1}{n}} - 1)$$ **Equation 2-12**

where:

n is the number of tasks;

e_i is the execution time of task T_i;

D_i is the relative deadline of task T_i.

Figure 2-4 shows an example of three tasks run with DM. The specifications of these tasks are shown in Table 2-3. It can be noticed that the condition in Equation 2-12 is satisfied as the total processor utilisation is 0.75, which is less than the upper bound of utilisation for three tasks (0.78).

Table 2-3 Task specifications for a system scheduled by DM scheduler.

Task	WCET (*ms*)	Deadline (*ms*)	Period (*ms*)	Priority
A	1	5	5	0
B	2	6	30	1
C	3	14	15	2

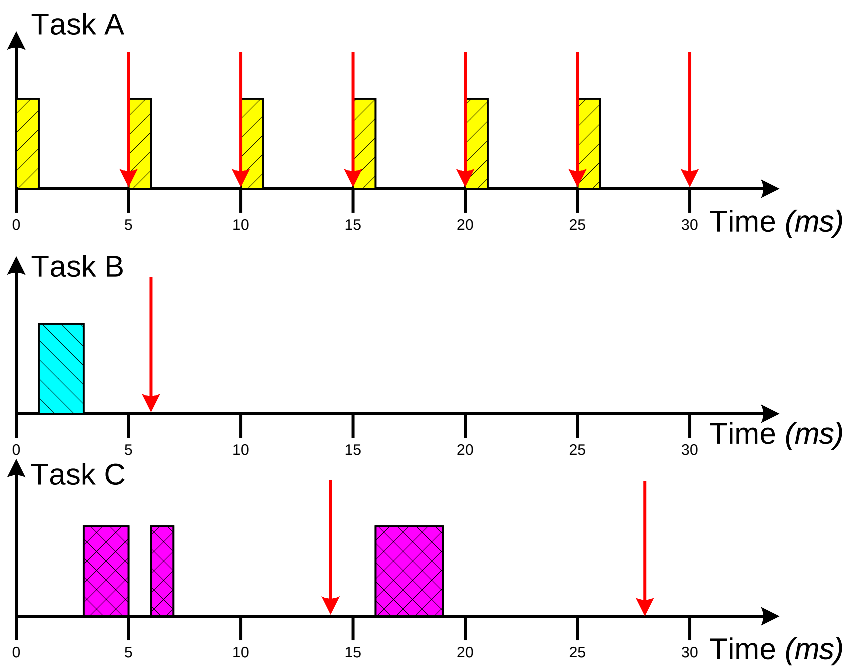


Figure 2-4 DM schedule for the task set shown in Table 2-3.

2.5.2 Dynamic priority schedulers

Unlike fixed priority schedulers in dynamic priority schedulers, the priority assigned to a given task can be changed at run-time. This priority is usually based on a dynamic parameter that may change during the system evolution (Cottet *et al.*, 2002). Examples of such schedulers include the earliest deadline first and least laxity first algorithms.

2.5.2.1 Earliest deadline first scheduler

While RM and DM statically assigns priorities to tasks, earliest deadline first (EDF) is said to be the most common scheduler that dynamically assigns priority to each task according to its current absolute deadline. The task which has the earliest deadline is assigned the highest priority (Liu and Layland, 1973). An example of using EDF in a real-time application is to maintain strict delay requirements of the multimedia communication involving digital audio and/or digital video (Ferrari and Verma, 1990). In which case the EDF is used, usually in conjunction with other scheduling strategies, to assign high priority to packets with earliest deadline to maintain the end to end delay requirements (Ferrari and Verma, 1990; Maina and Saidane, 2006).

Liu and Layland (1973) showed that using the same assumptions which are introduced in their RM analysis, for a set of *n* periodic tasks to be scheduled by EDF the necessary and sufficient condition is that the total processor utilisation should be less than or equals to 1, that is:

$$u = \sum_{i=1}^{n} \frac{e_i}{P_i} \leq 1$$ **Equation 2-13**

According to Cottet *et al.* (2002), for tasks which have deadlines less than periods the schedulability condition becomes:

$$u = \sum_{i=1}^{n} \frac{e_i}{D_i} \leq 1$$ **Equation 2-14**

It is shown also that the EDF is optimal in the sense that if there exists a feasible priority scheduler for a given task set, then the EDF is also feasible for that task set (Liu and Layland, 1973).

Figure 2-5 shows an example of two tasks scheduled by EDF scheduler. The specifications of the two tasks are shown in Table 2-4.

Table 2-4 Task specifications for a system scheduled by EDF scheduler.

Task	WCET (*ms*)	Deadline (*ms*)	Period (*ms*)
A	3	6	6
B	7	15	15

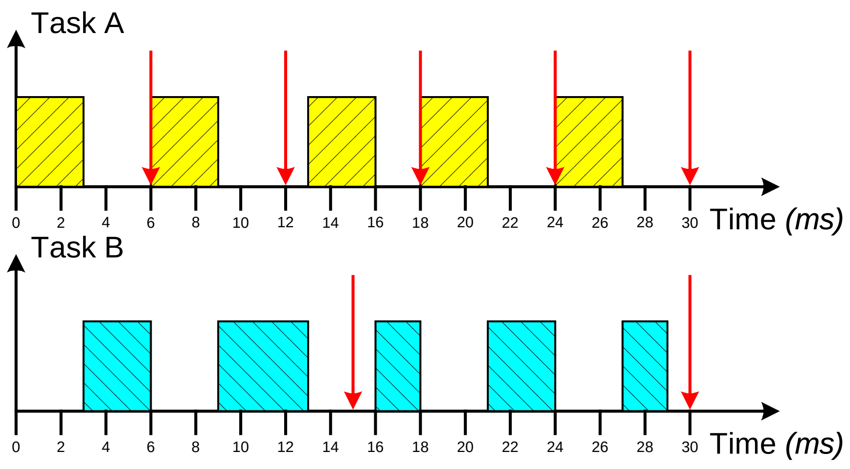


Figure 2-5 EDF schedule for the task set shown in Table 2-4.

2.5.2.2 Least laxity first

The laxity (or slack) of a task is defined as the maximum time the task can be delayed without missing its deadline (Cheng, 2002). According to Buttazzo (2005a), the laxity (L_i) of task T_i can be calculated as:

$$L_i = D_i - r_i - e_i \qquad \textbf{Equation 2-15}$$

where:

D_i is the relative deadline of task T_i

r_i is the arrival time of task T_i,

e_i is the execution time of task T_i.

The least laxity first (LLF) algorithm assigns priority to each task according to its laxity, the task that has the least laxity is assigned the highest priority (Leung, 1989). As shown in Oh and Yang (1998) using the LLF will result in poor system performance in case where there are two or more tasks have same, or close, laxity. This is because of the frequent context switches which takes place at every scheduling point until the tie breaks. They introduced a Modified Least-Laxity-First (MLLF) to solve this problem. The MLLF scheduling algorithm reduces the number of context switches by delaying them until necessary, even if the laxity-tie occurs.

LLF, like EDF, is optimal for pre-emptable tasks with no precedence, resource, or mutual exclusion constraints (Cottet *et al.*, 2002).

Figure 2-6 shows an example, given by Cottet *et al.* (2002), to describe the idea of LLF. Task specifications are shown in Table 2-5.

Table 2-5 Task specifications for a system scheduled by LLF scheduler.

Task	WCET (*ms*)	Deadline (*ms*)	Period (*ms*)
A	2	4	5
B	3	7	20
C	1	8	10

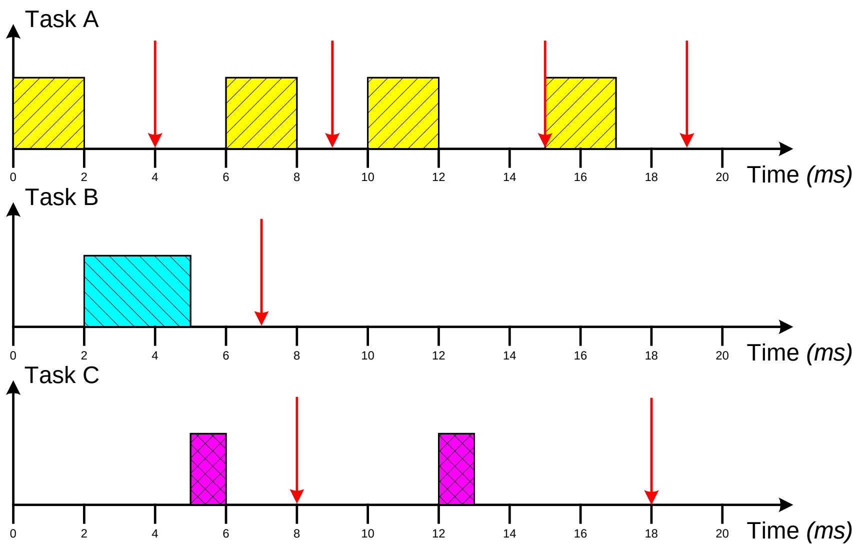


Figure 2-6 LLF schedule for the task set shown in Table 2-5, adapted from Cottet *et al.* (2002).

2.5.3 Common problems with priority schedulers

As shown above, priority-based schedulers usually allow task pre-emption to occur. In case where tasks share resources, specific techniques, such as using semaphores, can be used to prevent simultaneous access to shared resources. If not carefully designed, the use of these techniques may cause other problems, such as priority inversion or deadlock, which may results in violating task constraints, such as missing deadlines (Sha *et al.*, 1990; Silberschatz and Galven, 1998; Cottet *et al.*, 2002).

The above problems along with suggested solutions, which can be used to avoid their occurrence, or at least reduce their effects, are previously discussed by Sha *et al.* (1990). The following subsections will summarise this work.

2.5.3.1 Priority inversion

As defined by Sha *et al.*, "*priority inversion is the phenomenon where a higher priority job is blocked by lower priority job.*" (Sha *et al.*, 1990). They showed that this can happen in situations where both the high priority tasks and the low priority ones have a shared resource. In situations where the high priority task is activated and requested access to the shared resource while the lower priority task is still running and holding that shared resource, the high priority task will be blocked and wait until the low priority task completes its execution, or at least releases the shared resource. In this case it can be seen that the priorities of the two tasks are inverted, as the lower priority task enforced the high priority one to wait until it finishes. The situation may exaggerate in cases where the lower priority task is pre-empted by another task with intermediate priority, which does not need to access the shared resource. In this case the high priority task has to wait until both the two tasks finish their executions. This may cause the high priority task to miss its deadline.

Table 2-6 and Figure 2-7 show an example given by Sha *et al.* (1990) for a system that may suffer from the priority inversion phenomenon. The system has 3 tasks which have the priorities shown in Table 2-6. Task A and Task C share a common resource R1. The sequence of events listed in Table 2-6 can be described as follows:

- At time t_0 Task C is released and gains access to the microprocessor.
- At time t_1 it requires access to the shared resource R1. As this resource is not currently in use, Task C gains access to it and hence locks it.
- At time t_2 Task A becomes ready to run and pre-empts Task C as it has a higher priority.
- At time t_3 Task A requires access to the shared resource R1. As this resource is currently being locked by Task C, Task A is blocked and the control goes back to Task C.

- At time t_4 the task with the intermediate priority, Task B, becomes ready to run and pre-empts Task C as it has a higher priority.
- At time t_5 Task B finishes its execution and releases the processor for Task C to resume its execution.
- At time t_6 Task C releases the shared resources and is pre-empted by Task A which now can use the resource.
- At time t_7 Task A finishes its execution and releases the shared resource so Task C resumes its execution and eventually releases the processor when it finishes.

Table 2-6 Task specifications for a system which may encounter priority inversion

Task	Priority	Release time	Request R1 time
A	0	t_2	t_3
B	1	t_4	-
C	2	t_0	t_1

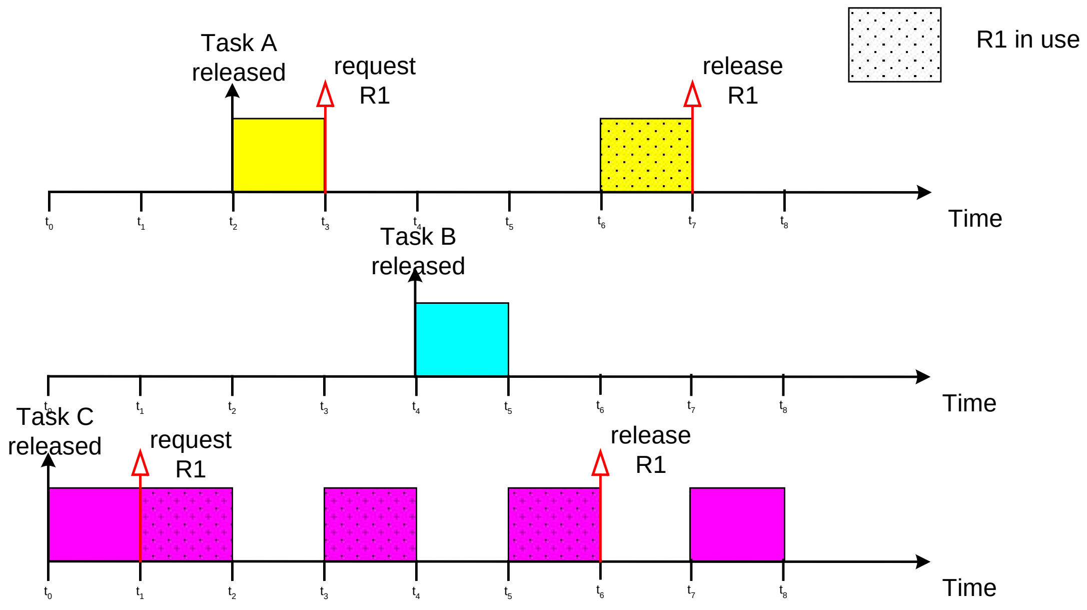


Figure 2-7 Illustration of the priority inversion phenomenon for tasks shown in Table 2-6, adapted from Sha *et al.* (1990).

As can be noticed from the above scenario the priority inversion problem occurred as the highest task, Task A, was forced to wait for a lower priority task, Task C.

Moreover, the situation was exaggerated when Task C had been pre-empted by Task B, an intermediate priority task, which in turn increased the blocking time of the highest priority task.

In order to reduce the effects of priority inversion, i.e. to shorten the task blocking time, the priority inheritance protocol introduced by Sha *et al.* (1990) can be used. The basic idea of the priority inheritance protocol is that any task executing in its critical section (the area of the code during which the tasks should not be pre-empted), or holding a shared resource, inherit the priority the highest priority task waiting for this resource, and return to its original priority after exiting its critical section or releasing the shared resource. This has been proven to reduce the blocking time of the high priority tasks.

Figure 2-8 shows how priority inheritance protocol helps to reduce the effects of priority inversion in the above example. This is can be explained as follows.

At time t_3 when Task A required access to the shared resource R1, which is currently blocked by Task C, Task A is blocked and Task C temporary inherited the priority of Task A and continued running with this priority. Later, at time t_4, when Task B became ready to run it could not pre-empt Task C as its priority is lower than the new priority of Task C. As a consequence Task C continued running with this priority until it released the shared resource, at which point it restored its original priority. This enabled Task A to pre-empt Task C and gain access to the shared resource. After Task A finished its execution control goes to Task B as it has had the highest priority amongst tasks which are waiting. Eventually, Task C resumed its execution after Task B finished.

As can be noticed from the above example, with the help of using the priority inheritance protocol the blocking time of the higher priority task can be reduced.

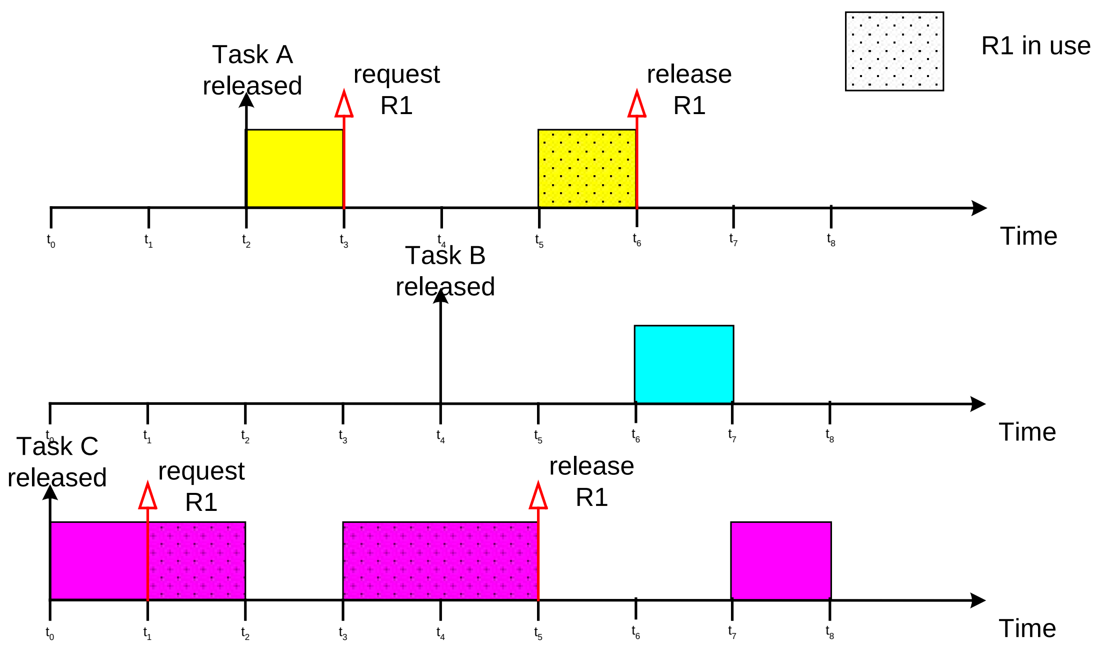


Figure 2-8 Illustration of using the priority inheritance protocol for tasks shown in Table 2-6, adapted from Sha *et al.* (1990).

2.5.3.2 Deadlock

A deadlock may occur if two, or more, tasks share two, or more, resources and each one of them hold one resource while it is waiting for another resource which is currently held by the other task. To explain this situation consider the following example which is given by Sha *et al.* (1990).

Consider the system of tasks indicated in Table 2-7 and

Figure 2-9. Assume that Task B and Task C share two common resources which can be locked by the semaphores S1 and S2. Only Task A has an access to a resource which can be locked by the semaphore S0. The deadlock problem can be explained by considering the following sequence of events as listed in Table 2-7.

- At time t_0 Task C is released and gaines access to the microprocessor.
- At time t_1 it requires access to the shared resource locked by semaphore S2. As this resource is now available Task C gaines access to it and lockes it by semaphore S2.
- At time t_2 Task B becomes ready to run and pre-empts Task C as it has a higher priority.

- At time t_3 Task B requires access to the shared resource locked by semaphore S1. As this resource is currently available Task B gains access to it and lockes it by semaphore S1.
- At time t_4 Task B requires access to the shared resource currently locked by semaphore S2, this causes Task B to be blocked and the control to go back to Task C.
- At time t_5 Task A becomes ready to run so it pre-empts Task C and gains access to the microprocessor.
- At time t_6 Task A requires access to the shared resource locked by semaphore S0. As this resource is now available Task A gains access to it and locks it.
- At t_7 Task A finishes its execution and the control goes back to Task C, as Task B is currently blocked and waiting for S2.
- At time t_8 Task C requires access to the shared resource currently locked by semaphore S1, this causes Task C to be blocked.

Table 2-7 Task specifications for a system which may suffer from deadlock.

Task	Release time	Priority	Request S0	Request S1	Request S2
A	t_5	0	t_6	-	-
B	t_2	1	-	t_3	t_4
C	t_0	2	-	t_5	t_1

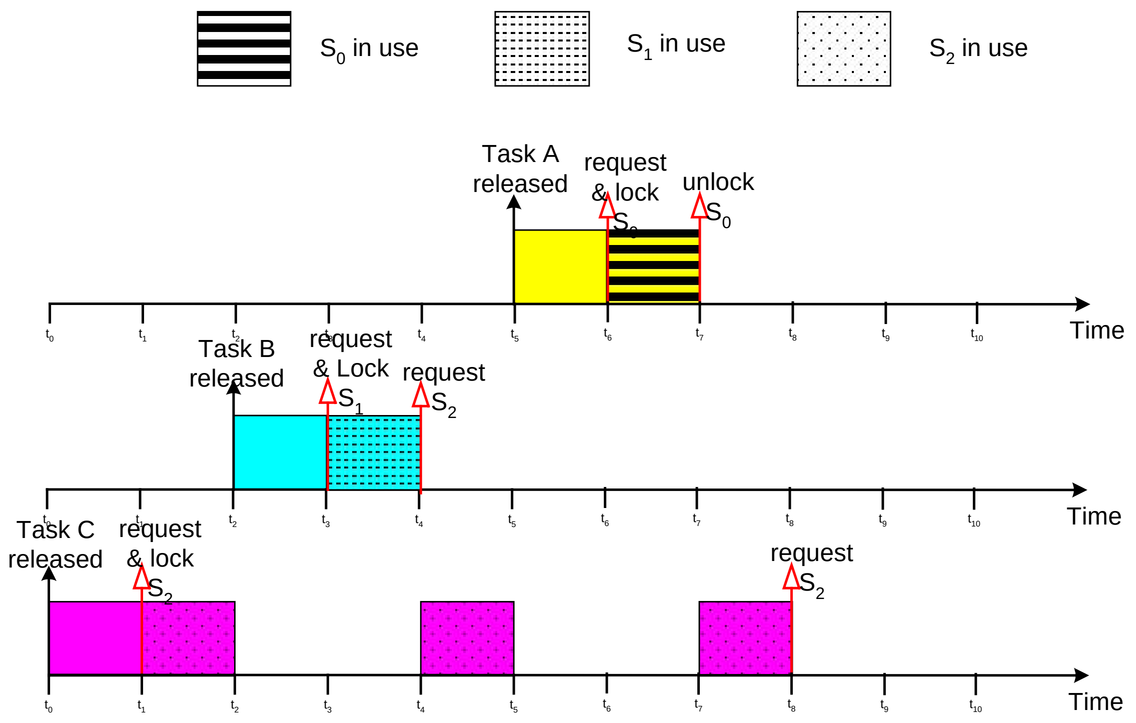


Figure 2-9 Illustration of the deadlock phenomenon for tasks shown in Table 2-7.

As can be noticed from the above scenario the system goes to the deadlock status as each one of Task B and Task C was holding one resource and waiting for the resource currently held by the other task to be released.

Techniques like priority ceiling protocol, introduced by Sha *et al.* (1990) can be used to prevent the occurrence of such deadlocks. The underlying idea of this protocol is to extend priority inheritance protocol by assigning each resource a priority (called the ceiling priority) equals to the priority of the highest priority task that may use this resource. A task is only allowed to gain access to a resource if the resource is free and the task priority is higher than all priority ceilings of all resources currently held by other tasks.

Applying this protocol to the above example helps to avoid the deadlock occurrence. This can be described as follows.

As the resource locked by semaphore S0 can be used by Task A it is allocated a priority ceiling equals to the priority of that task. In the same manner both semaphores S1 and S2 are allocated a priority ceiling equals to the priority of Task B, as this is the highest priority task amongst all tasks which may use the resources locked by these semaphores. Under these conditions the scenario of events will occur according to those shown in Figure 2-10 and can be described as follows.

The sequence of events will occur in the same manner as described above until time t_3. At this point of time Task B requires access to the shared resource locked by semaphore S1. As the priority of Task B is not higher than the ceiling priority of this semaphore, it will be blocked. This will cause Task C to inherit this ceiling priority and to resume its running until it is pre-empted by Task A which has the highest priority and which will continue to run in the same way as described above. Task C will resume its running after Task A finishes its execution. In which case it requires and gains access to the shared resource locked by semaphore S1, and continue running until it releases both S1 and S2. At this point of time (t_8) Task B resumes its execution, gains access to both S1 and S2, and eventually finishes its execution.

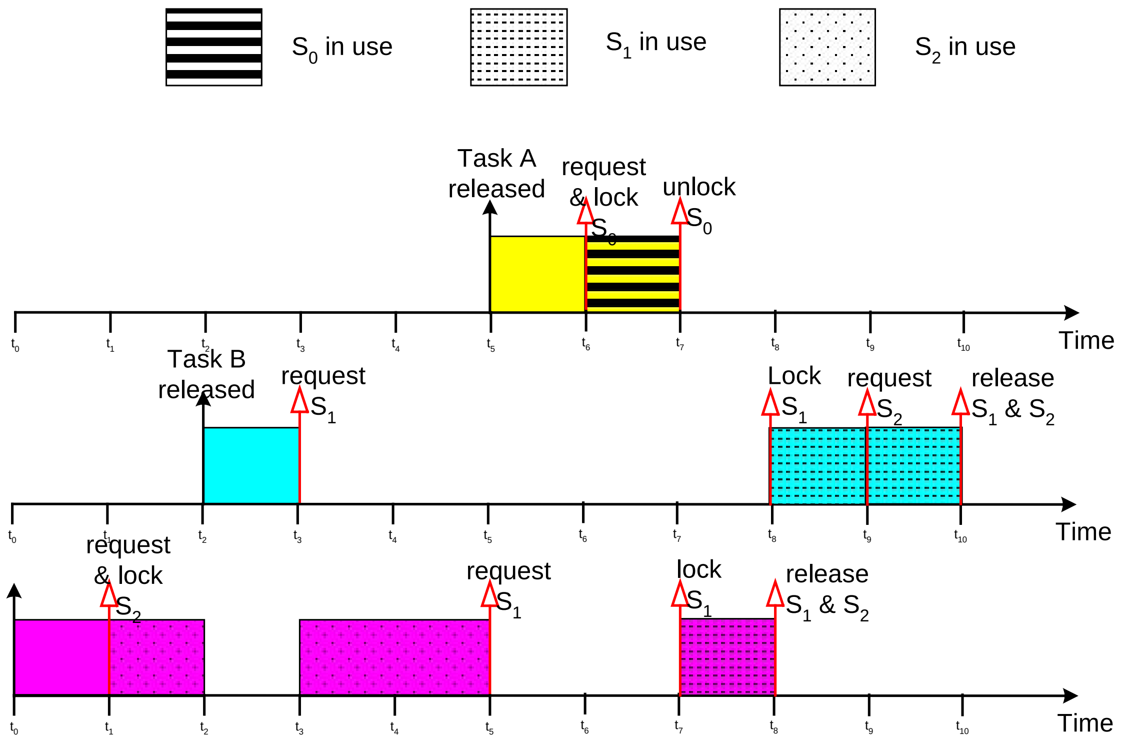


Figure 2-10 Illustration of using the priority ceiling protocol for tasks shown Table 2-7, adapted from Sha *et al.* (1990).

2.6 Discussion

In this chapter an overview of various scheduling criteria, along with the most common scheduling strategies introduced in the literature is discussed. However there are some misconceptions that exist between developers about the characteristics and behaviours of various schedulers which are sometimes lead to favouring one or another. The following subsections will consider these misconceptions.

2.6.1 Misconceptions about schedulers classification

It is noted in the literature that there are some confusions in the way various scheduling strategies are mapped to the appropriate scheduling criteria. Not least the confusion between different scheduling criteria themselves, for example fixed and dynamic priority scheduling are sometimes confused with offline (or static) and online scheduling (Stankovic *et al.*, 1995; Marwedel, 2006).

In order to avoid the above confusions it is necessary to emphasis that each scheduling strategy can be implemented using more than one criterion. For example the RM is always referred to as a fixed priority scheduling as the priority of each task does not change once it is assigned. However, RM can be implemented by allowing or disallowing task pre-emption, i.e. using pre-emptive or non pre-emptive criterion respectively. As a result one may have pre-emptive rate monotonic or non pre-emptive rate monotonic. In the same way it can be implemented using either time-triggered or event-triggered, hence one may have time-triggered rate monotonic or event-triggered rate monotonic. Consequently different forms of RM can be used such as (non) pre-emptive time-triggered rate monotonic or (non)-pre-emptive event-triggered rate monotonic.

Moreover, if the complete schedule of a given task set is constructed offline by assigning priorities according to RM - in cases where the number of tasks, along with all the task parameters, is known in advance and these parameters will not change at runtime- then a scheduler is referred to as a static (or offline) scheduler. On the other hand if some task parameters, such as the task execution time, or the number of tasks in the system may be changed at runtime, then the resulted schedule may not be the same in every major cycle, in which case the scheduler is referred to as an online (or dynamic) scheduler.

Consequently one can have (non pre-emptive)/(pre-emptive) (time-triggered)/(event-triggered) (offline)/(online) RM scheduling strategy.

In sum may it can be said that a given scheduler can be mapped to more than one criterion in the same time.

2.6.2 Misconceptions about offline and online schedulers

It is often argued that use of offline, or pre-run-time, scheduling produces a highly predictable systems (Xu and Parnas, 1992; Pont, 2001; Buttazzo, 2005a). But on the other hand there are some misconceptions about the limitations and performance of pre-run-time schedulers that favour run-time schedulers, especially those based on priority schedulers. These misconceptions have been addressed by Xu (1993; 2003) and Xu and Parnas (1993; 2000) and are summarised below.

First *it has been argued that pre-run-time schedulers are inadequate for an environment whose behaviour is not completely known in advance, especially for systems in which not all the tasks are periodic.*

However, it is impossible to make sure that the timing constraints will be met if the scheduler does not know the major characteristics of the system, especially in safety-related systems. In addition if the minimum inter-arrival times of the non-periodic tasks are known in advance it is possible to translate them into equivalent periodic tasks so that it will still be possible to schedule them using the pre-run-time schedulers.

Second, *it has been claimed that pre-run-time schedulers are less flexible, compared to run-time scheduler, for systems with changing application requirements.*

In reality it is not guaranteed that the run-time scheduler will be able to compute appropriate scheduler, especially for complex systems, as in most cases the amount of time assigned for making the scheduling decision is severely limited. This restriction in time usually forces the run-time scheduler to use rigid hierarchy of priorities without investigating different combinations of times and orders in which tasks can execute. By contrast this limitation does not exist for pre-run-time schedulers which are computed offline. In addition, the pre-run-time scheduler can cope with changing in the systems by computing more than one schedule in advance (Kopetz *et al.*, 1998). A code can easily be inserted in the scheduler and activated by external event to enable the processor to switch between the pre-computed schedulers (Xu and Parnas, 2000).

Third, *it has been argued that the scheduler computed by pre-run-time scheduling can be failing long as – in theory – the LCM of the task periods could be very long (particularly in large task sets with co-prime periods).*

However, in practice, there may be some flexibility in the choice of task periods (Xu and Parnas, 1993; Pont, 2001). As an example Gerber *et al.* (1994; 1995) introduced a design methodology in which the end-to-end timing constraints (which is initially defined in such a way like: the car dynamics, such as speed, must be updates, based on the input throttle position, within period of 5 *ms* (Ayavoo *et al.*, 2005; Ayavoo *et al.*, 2007; Short and Pont, 2008)) are transformed into a set of intermediate rate constraints. They introduce an algorithm that solves these constraints with minimising the CPU utilisation. They showed that a feasible solution for task constraints (like the period) can found by considering the relationship of period of that task with the periods of all its successors (for example if Task A precedes Task B, then the may have the same period or at least one of them divides the other). Kim *et al.* (1999) went further to improve and automate this period calibration method.

Fourth, *it has been claimed that translating all non-periodic tasks to periodic ones and adapting the period of all tasks to have a harmonic relation with minor cycle (usually the greatest common divisor "GCD" of task periods) results in wasting systems resources.*

Although these techniques affect the processor utilisation their effects can be less than the effects of other factors which adds to the cost and inefficiency of the run-time scheduler. For example it is difficult to reduce the number of pre-emptions, and consequently reduce the overhead cost of the of context switching, in run-time schedulers as it has limited time to take the scheduling decision. By contrast the number of pre-emptions can be minimised (Dobrin and Fohler, 2004) offline in the case of pre-run-time scheduler. Another factor that adds to the system cost in a run-time scheduler is the overhead encountered in executing the required mechanisms which are used to avoid deadlocks and control access to shared resources. In addition, using such mechanisms can reduce the processor utilisation and, even in some cases reduce the chance of finding a feasible scheduler; for example in some cases using a priority ceiling protocol may results in a task being blocked, and hence missed its deadline, to prevent a possible occurrence of deadlock even if in reality executing the task would not cause any problem. Again such mechanisms are not needed in pre-run-time scheduler.

Finally computing the scheduler offline gives the designer the opportunity to investigate, compare, test, and verify different schedulers and choose the best one which fits the application needs. In contrast it is impossible to find a totally online optimal run-time scheduler in some cases, such as when there are mutual exclusion constraints (Mok, 1983).

2.7 Conclusions

As the work in this thesis is concerned with automating the process of scheduling time-triggered architecture this chapter starts by defining task characteristics and constraints that are usually used to model tasks in real-time systems. Various scheduling criteria, along with the most commonly used scheduling strategies, are reviewed. These schedulers can be implemented either using ET or TT architecture. When using certain scheduling strategy task parameters along with scheduler parameters need to be appropriately chosen in order to ensure that all constraints will be met, whenever possible.

The next chapter will review previous work that has been done in automating the process of scheduler configuration.

Chapter 3
Scheduling algorithms

In this chapter previous work in automating the process of testing the schedulability of a set of tasks and creating the schedule, where possible, is reviewed.

3.1 The function of scheduling algorithms[5]

Although some strategies, such as RM, are known to be optimal, it should be emphasised that this optimality is only guaranteed under certain conditions (Audsley *et al.*, 1993), for example all tasks are assumed to be completely pre-emptable (Section 2.5.1.1 describes the conditions in which RM is optimal). Furthermore, as noted by Xu and Parnas (2000) "*Even if all processes are completely pre-emptable—an unlikely situation in a complex hard real-time system, scheduling processes according to priorities, is still not optimal.*"

Consequently the designer not only has to choose the appropriate scheduler for the application at hand but he/she also has to customise some additional parameters, like task starting times and task order. Inappropriate choices may lead to violating task constraints. Manually exploring these choices is a tedious and time consuming process, even for a small number of tasks. Hence an appropriate scheduling algorithm is often used to automate the process of scheduler selection and customisation.

The following sections discuss the effects on task schedulability of an inappropriate choice of scheduler or inappropriate task order and/or task starting time.

3.2 Choosing the right scheduling strategy

Inappropriate choice of the scheduling strategy may affect the task schedulability. For example although "*RM algorithm is probably the most used priority assignment in real-*

[5] Scheduling algorithm decides the order and the starting point of each task (by configuring a given strategy / criterion).

time applications" (Buttazzo, 2005b) it fails to find a workable schedule in some cases where there actually exists a scheduler which can successfully schedule the tasks, such as the EDF.

Table 3-1, Figure 3-1, and Figure 3-2 show an example given by Xu and Parnas (2000) to illustrate this fact.

Table 3-1 Task specifications for a system in which the scheduler strategy decision affects task schedulability.

Task	r (ms)	WCET (ms)	Deadline (ms)	Period (ms)
A	0	3	6	6
B	0	4	8	8

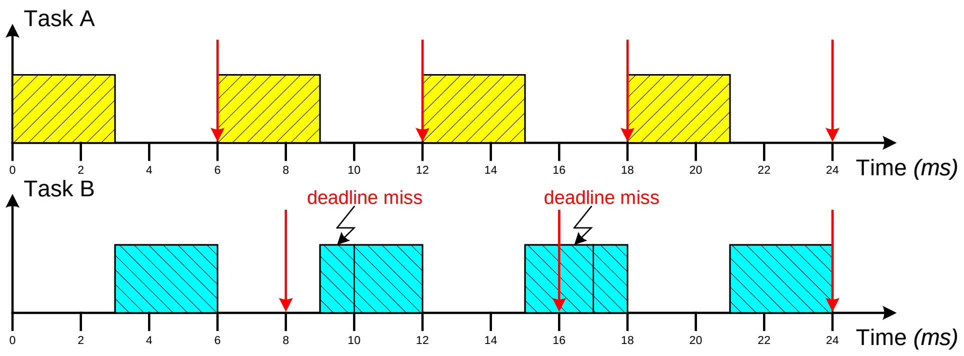


Figure 3-1 Infeasible RM scheduler for task shown in Table 3-1, adapted from Xu and Parnas (2000).

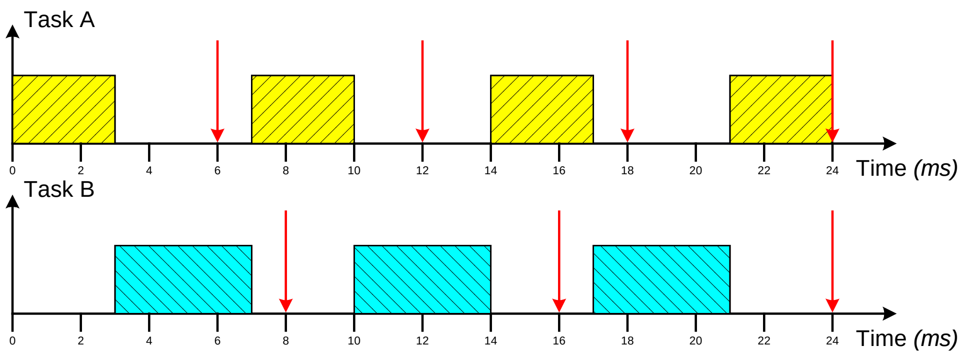


Figure 3-2 Feasible non pre-emptive EDF scheduler for task shown in Table 3-1, adapted from Xu and Parnas (2000).

3.3 Choosing the appropriate task order/starting time

As noted by Cheng (2002) there is no optimal priority based scheduler for non-pre-emptable tasks with arbitrary start times, computations times, and deadlines, even on a uniprocessor. In these cases assigning task priority without considering appropriate assignment of task starting times and task execution order may lead to missed deadlines.

Table 3-2, Figure 3-3 , and Figure 3-4 illustrate an example adapted from Xu and Parnas (2000) for a system in which inappropriate assigning of task stating time can lead to missed deadline if task pre-emption is not allowed.

In this example if Task A starts its execution as soon as it becomes ready (at t=0) this will cause Task B to miss its deadline (as it can not pre-empt Task A). On the other hand delaying the stating time of Task A by 1 *ms*, and leaving the processer idle during this period, will allow Task B to start its execution early, and eventually both the tasks will meet their deadlines.

Table 3-2 Task specifications for a system in which task order/starting times affects task schedulability.

Task	r (ms)	WCET (ms)	Deadline (ms)	Period (ms)
A	0	10	12	12
B	1	1	2	12

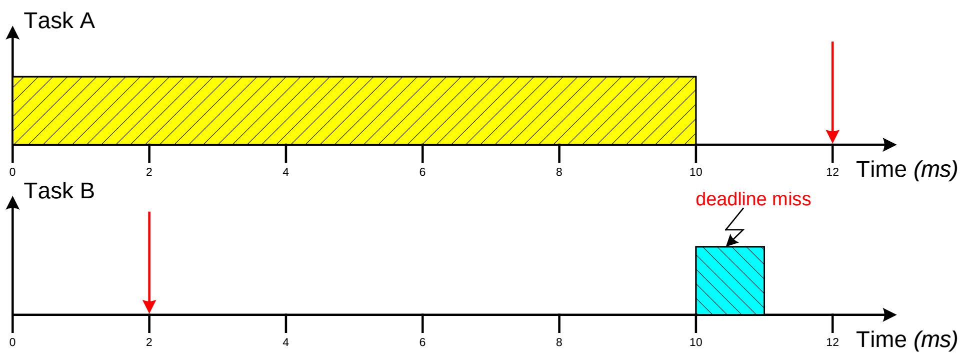


Figure 3-3 Infeasible fixed priority scheduler for task set shown in Table 3-2, adapted from Xu and Parnas (2000).

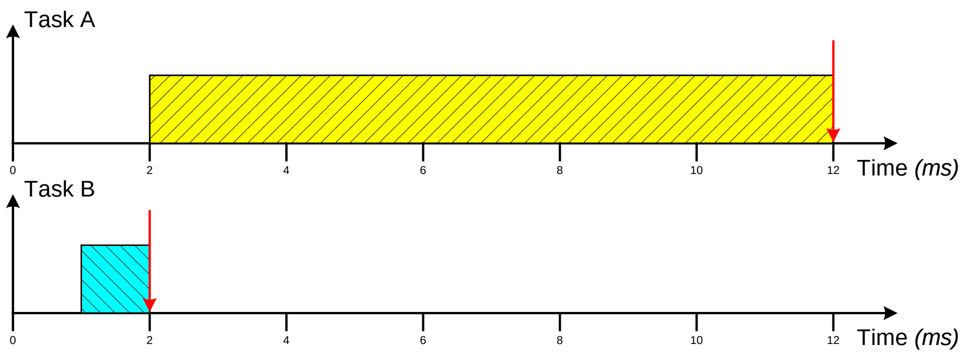


Figure 3-4 Feasible scheduler for task set shown in Table 3-2, adapted from Xu and Parnas (2000).

3.4 Automatic schedule generation in real-time systems

As explained above careful decisions have to be made in choosing and customising the appropriate scheduler for a specific application. Although testing the schedulability and customising the scheduler parameters can be done manually for systems with a small number of tasks, this process is challenging and time consuming. The situation becomes more complex and intractable as the number of tasks and the inter-task constraints in the systems grows. As explained in Chapter 1 this problem is NP-hard.

Considerable work has been described in the literature to overcome these difficulties. This work resulted in introducing schedulability tests via evaluating and checking the value of a specific expression which takes task parameters as input ((Jeffay *et al.*, 1991; Cheng, 2002; Bini *et al.*, 2003; Baruah, 2006) for example).

Unfortunately these tests are not sufficient as they do not normally take all task constraints into account, such as jitter, distance, latency, precedence and exclusion. As a result scheduling algorithms are introduced to help automate the process of scheduler selection and customisation.

The following subsections give a brief overview of commonly used scheduling algorithms introduced in the literature.

3.4.1 Brute-force search

Brute-force searching tests all possible combinations of settings until it finds a feasible solution. In the worst-case it will test all possible solutions. This can be used if the number of tasks and the inter-task constraints in the systems are small. However, as the problem size grows (for example: as the number of tasks and inter-task constraints increase), the brute force approach may take an extremely long time until it finds a feasible schedule, if one exists. As Burns (1995) noted "*For any reasonable number of processes this is clearly impracticable*".

For the reasons above, brute force search can be used for small size problems when there is no other way that leads to a possible feasible scheduler, otherwise one of the following algorithms is normally used.

3.4.2 Branch-and-bound (BaB)

As described by Ekelin (2000) the branch-and-bound (BaB) algorithm searches for a feasible solution in a tree-like pattern, branch by branch. An initial solution is calculated in the beginning (the tree root). At each step a new branch is created to improve the initial solution. A rule, which is calculated depending on the application at hand, is usually used to prune the branching (bounding).

For example Xu and Parnas (1990) presented a branch-and-bound algorithm that finds the optimal schedule (if one exists), for a set of processes (or tasks). The algorithm tries to find a valid schedule for the set of tasks by minimising the lateness of all tasks (lateness is defined as the difference between the completion time and the deadline). It starts by computing an initial schedule based on the EDF strategy. This initial schedule forms the root of the tree. The tree is expanded in two directions. The first direction is formed by adding additional precedence relation by allowing the latest task to run before a task in the parent branch. The second direction is formed by adding additional pre-emption relation by allowing the latest task to pre-empt a task in the parent branch. At each stage, the lower bound of lateness of all the new braches is calculated. The tree is expanded from the node which has the lower bound of lateness. The search continues until a feasible schedule is found or until there no improvement in the lateness. Xu (1993) extended the previous study to find a feasible co-operative scheduler (if one exists) in a multi processor environment. Kovalyov and Xu (2000) went on to further refine this algorithm.

3.4.3 Heuristic search

Unlike brute-force and BaB algorithms, which may end up testing all possible paths, in the worst-case that lead to a feasible solution, heuristic search only tests paths that are likely to lead to the solution (Burns, 1995). The advantage is that it takes less time, but on the other hand it is not guaranteed that to find a solution if one exists.

The following subsections will give an overview of heuristic searche examples.

3.4.3.1 Simulated annealing

Simulated annealing is an optimisation technique based on the annealing process used in solids to obtain a perfect configuration for a crystal, the process by which all the atoms are aligned in a lattice, this configuration having the least total energy (Kirkpatrick *et al.*, 1983; Radcliffe and Wilson, 1990). In the annealing process the temperature of the metal is increased, to allow the atoms to move so as to increase the chance that they line up with their neighbours. Then the temperature is slowly decreased to restrict the movement of the atoms to their new aligned positions.

Kirkpatrick *et al.* (1983) used an analogy of this method to solve several problems that arise in the manufacturing of computer components, such as subdividing each design into smaller circuits small enough to fit into single chip or group of chips, deciding the location of the circuits inside the chips, and deciding the best location of these chips on printed circuit boards that reduce the length of the paths between them as much as possible (Radcliffe and Wilson, 1990).

A similar approach is used by Tindell *et al.* (1992) to solve the problem of allocating a number of tasks to a number of processors in a distributed hard real-time architecture. The energy in this algorithm represents a measure of the suitability of a certain allocation of the tasks to the processors in certain manner. The algorithm tries to find the allocation with the lowest energy. In doing so it starts with a random point, which represents a certain allocation, and computes the energy function of this point, *Es*. Then the energy function, *En*, is calculated for a random point in the neighbouring space. This point can be represented by choosing a random task and moving it to a randomly chosen processor. The new point becomes the new starting point if its energy is less than that of the old point or if the probability that the system energy will be decreased, in subsequent steps, is higher than a certain value (to escape from a local minimum). This probability is computed based on the current value of a control variable, *C*, which is analogous to the temperature factor in a thermodynamic system. The control variable is slowly reduced (‘cooling’ the system) making higher energy jumps less likely during the annealing process. Deadline monotonic scheduling is chosen to schedule the tasks and a token protocol is used to control the communication between different processors.

3.4.3.2 Genetic algorithm

Genetic algorithms are inspired by Darwin's theory of evolution. The basic idea around genetic algorithms as described in Mitchell (1998) is to encode candidate (initial) solutions to the problem at hand to abstract chromosomes, which are represented as strings of ones and zeros. A fitness function is calculated for each chromosome. Based on their fitness values, a number of these chromosomes are chosen to form the following generation of solutions (next population). The new population is formed by performing two operations on some, or all, of the selected chromosomes, crossover and mutation. In crossover operation a number of bits are randomly selected from two different chromosomes and swapped together to form two new chromosomes. In the mutation operation one, or more, bits in specific chromosomes are randomly selected and flipped. The fitness function of each chromosome in the new generations is then calculated and the process is repeated again until a specified value of the fitness, or the maximum number of iterations, is reached.

Sandström and Norström (2002) used a genetic algorithm to assign attributes such as priorities and offsets to a set of tasks that have complex timing constraints in pre-emptive priority-based run-time systems (using off-the-shelf operating systems). The chromosomes were constructed by assigning each periodic task an offset and a priority value, and assigning each sporadic task a priority value. The objective function was to calculate the start times and completion times of each task so that the deviations from the required constraints (such as the distance between tasks) is minimised.

Oh and Wu (2004) used a genetic algorithm to decide the order at which tasks have to be scheduled and the number of the required processors in a multiprocessor environment. In order to achieve their objective they used 2-partitions chromosomes, the first partition was used to represent task order while the second partition was used to represent the processor at which each task should run. Their goal was to reduce the task finishing time and the number of needed processors for a set of tasks with precedence constraints.

3.4.4 Constraint programming

In constraint programming the problem at hand is expressed in terms of variables, which can take a range of values, and some constraints on these variables. A constraint solver is used to find suitable, or optimal, values for these variables. The constraint solver uses propagation in order to reduce the search space variables by removing values that cannot be part of a solution.

Schild and Würtz (1998; 2000) used constraint programming, with the help of constraint programming language Oz, to solve the problem of finding a feasible schedule and map tasks to different processor in time-triggered architecture. Ekelin and Jonsson (2000; 2001) extended this work by extending the problem to include more task constraints and system constraints using the SICStus Prolog and its associated constraint solver.

3.5 Previous work done in ESL for auto code generation for TT systems

All the above work is relevant to a discussion about tool support for scheduler design. However, none of this previous work relates directly to TTC / TTH architectures: instead, such previous studies have tended to focus on "conventional" RT operating systems (e.g. VxWorks: (Sandström and Norström, 2002)). Such operating systems exceed the resource requirements available in the types of processor considered in this study.

While previous studies on scheduler parameter selection do not relate directly to the work presented in this thesis, there has been considerable work carried out by researchers in Embedded Systems Laboratory (ESL) at the University of Leicester over recent years on the "automatic" creation of systems with a TTC architecture (e.g. (Mwelwa *et al.*, 2003; Mwelwa *et al.*, 2004; Mwelwa *et al.*, 2005; Mwelwa, 2006; Mwelwa *et al.*, 2006)). Such work supports the creation of code for complete TTC systems (including the system scheduler) using a collection of "design patterns" (Pont, 2001; Kurian and Pont, 2005a; Kurian and Pont, 2005b; Kurian and Pont, 2006a; Kurian and Pont, 2006b; Pont *et al.*, 2006; Kurian and Pont, 2007). The following subsections summarise the work done in this area.

3.5.1 Design patterns

In software engineering, design patterns describe solutions to commonly recurring software problems (Sommerville, 2007). Work on design patterns in general was initiated by Alexander and his colleagues (Alexander *et al.*, 1977; Alexander, 1979;

Kurian and Pont, 2007) who use it in architecture. This work inspired researchers working in software to use design patterns in software design (Cunningham and Beck, 1987; Gamma *et al.*, 1995; Kurian and Pont, 2007).

Since 1996, researchers in ESL lab at the University of Leicester, more specifically Michael J. Pont, focused on producing patterns used to support the development of embedded systems with TTC architectures. His work resulted in the creation of a collection of patterns (more than 70 patterns) which were referred to as the Pattern for Time-Triggered Embedded Systems (PTTES) collection (Pont, 2001).

In recent work, Kurian and Pont (2007) have revised the form of the PTTES patterns. In this revised structure each pattern consists of three layers:

(i) Abstract patterns: which are used to describe the pattern at the higher abstraction level – that of the system design

(ii) Patterns: describe ways in which a given abstract pattern can be implemented

(iii) Pattern Implementation examples (PIEs): intended to illustrate how a particular pattern can be implemented on different hardware platforms.

It has been shown that design patterns provide an effective means of developing software for resource-constrained embedded systems in which reliability is a key design consideration.

3.5.2 Auto code generation tool

Initially, the above work on patterns focused on a manual approach, it is assumed that the developer has to manually search a catalogue for the appropriate pattern and adapt it to fit his application needs (Mwelwa *et al.*, 2005). Despite the above mentioned advantages of using these patterns, this manual process has the potential to be error prone and time consuming thus resulting in unreliable systems (Mwelwa *et al.*, 2003; Mwelwa *et al.*, 2005; Mwelwa *et al.*, 2006)).

To avoid these problems a pattern-based code generation tool is developed. This tool supports the automatic code generation based on the PTTES collection (Mwelwa *et al.*, 2003; Mwelwa *et al.*, 2006)).

It has been demonstrated that using this tool helps reducing the time and effort required to develop TT embedded software and potentially improve its reliability (Mwelwa *et al.*, 2003; Mwelwa *et al.*, 2004; Mwelwa *et al.*, 2005; Mwelwa, 2006; Mwelwa *et al.*, 2006).

3.6 Discussion

The tool mentioned above supports the development of embedded software using the TT architectures. However, the user still needs to "hand tune" some task parameters (like the offset) and scheduler parameters (like the tick interval).

The work presented in this thesis seeks to address the problem of choosing between TTC and TTH schedulers and – for the chosen scheduler – determining an appropriate set of task, and scheduler, parameters.

While choosing the appropriate scheduling strategy and the suitable task and scheduler parameters do ensure that various task constraints are met, the resulting system is not fully predictable; for example the points of time at which each instant of the task will start or finish its execution cannot be predicted. This is due to the variation in task execution times. One possible way to increase system predictability is to minimise variations in the task execution times.

The following chapter will give an overview of the impacts of variations in task execution time and the goals that can be achieving by fixing task execution time.

3.7 Conclusions

In this chapter previous work on scheduling algorithms for real-time systems has been reviewed. It started by discussing the effects of choosing inappropriate scheduling strategy and/or inappropriate task parameters on task schedulability. Then the most commonly used scheduling algorithms are reviewed. Other scheduling algorithms are used to deal with soft real-time systems (such as Feedback scheduling (Lu *et al.*, 1999; Stankovic *et al.*, 1999) and group-EDF scheduling (Li *et al.*, 2007)), these schedulers are out of the scope of the current work.

Most scheduling algorithms require the upper bound of task execution time to be known at design time. The following chapter will highlight previous work done in this area and discuss the effects of variations in task execution times.

Chapter 4
Necessity of stabilising task execution time

This chapter discusses the need for stabilising the task execution time as an important requirement for increasing the predictability and satisfying task constraints in safety-related systems. Previous work that has been published in the literature in this area is reviewed.

4.1 Impacts of variations of task execution time

Variations of task execution time may affect the system predictability. Moreover, it may cause violation of task constraints (such as violating the distance constraints and / or introduction of unwanted high levels of jitter). This will be discussed in the following subsections.

4.1.1 Impact of variations of task execution time on system predictability

In safety-related systems it may required that knowledge about what the system will be doing in every moment of time during its execution (such as: which task is expected to be running at each point of time) to be known in advance. Variations in task execution time will affect the points at which each task finishes its execution and hence it will also affect the points at which subsequent tasks start their executions. Moreover it may affect the sequence at which the tasks run.

Table 4-1 describes the specifications of a set of tasks in which the system predictability may be affected by variations in tasks execution times. Predicting the task sequence and / or the points at which each task will start / finish its execution of such system, when scheduled by any scheduling strategy, may not be possible. For example using the non pre-emptive EDF strategy for scheduling these tasks will result in the schedule shown in Figure 4-1 if tasks ran with their BCETs while it will result in the schedule shown in Figure 4-2 if tasks ran with their WCETs.

In the first case, it can be noted that when Task A runs with its BCET it finishes its execution before the release time of Task C, and after the release time of Task B. In which case, Task B starts its execution directly after Task A finishes, as it is the only ready available task at that time. On the other hand, when Task A runs with its WCET it finishes its execution after the release time of both Task B and Task C. In which case, the non pre-emptive EDF strategy chooses to run Task C first as it has the earliest deadline amongst the ready tasks, and Task B will run at last.

Table 4-1 An example that show the effect of variations of execution time on task order.

Task	Release time (ms)	BCET (ms)	WCET (ms)	Deadline (ms)	Period (ms)
A	0	6	12	50	50
B	5	7	15	100	100
C	10	8	20	75	75

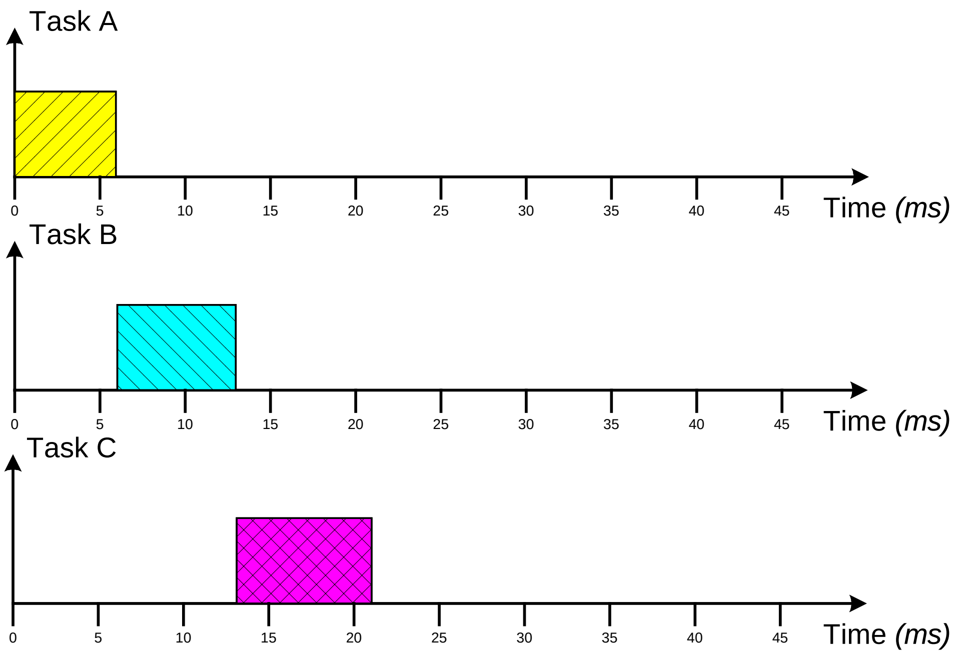


Figure 4-1 Task schedule when tasks shown in Table 4-1 run with their BCETs.

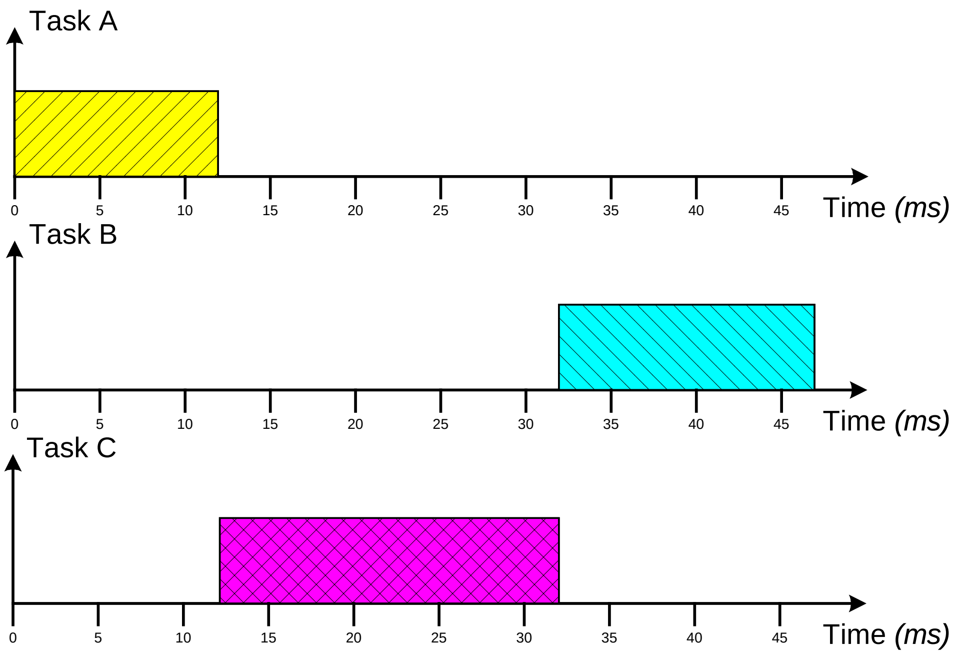


Figure 4-2 Task schedule when tasks shown in Table 4-1 run with their WCETs.

4.1.2 Impact of variations of task execution time on task constraints

Variations of task execution time may result in violating the distance constraints. For example, assuming that the WCET of Task C in a given task set equals the required distance between two other tasks, Task A and Task B. The required distance constraint between these two tasks can be achieved by scheduling Task C to run after Task A and before Task B. However, variations in Task C execution time will result in violating the distance constraint of the other two tasks.

Variation of task execution time can also affect the jitter level of the task(s) which follows this task which will in turn reduce the system predictability.

Table 4-2, along with Figure 4-3, shows an example for a set of tasks scheduled by TTC. It should be noted that using this scheduler Task C runs either directly after Task A finishes its execution, or after both Task A and Task B finish their executions. Assuming that the upper bound of jitter that can be tolerated by Task C is 3 *ms*, the schedule shown in Figure 4-3 can satisfy all task constraints as long as both Task A and Task B run always with their WCET. However variations of the execution times of these two tasks (especially that of Task B) will increase the jitter level of Task C.

Table 4-2 An example that show the effect of variations of task execution time on jitter.

Task	WCET (ms)	Deadline (ms)	Period (ms)	Offset (ticks)
A	2	10	10	0
B	3	20	20	0
C	4	30	30	0

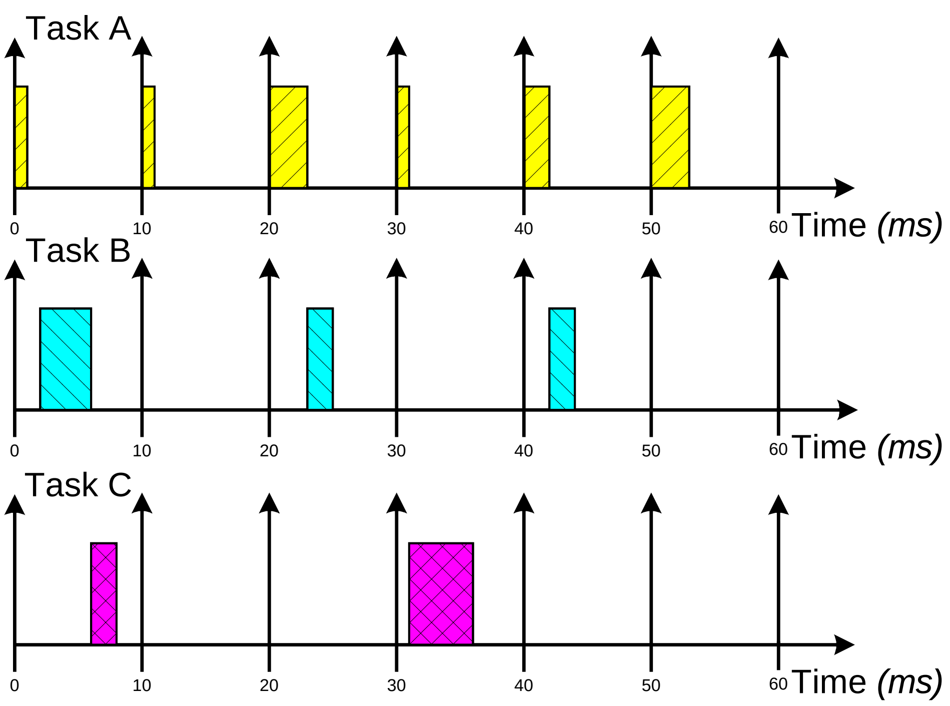


Figure 4-3 Task schedule for tasks shown in Table 4-2.

4.2 The need for stabilising the task execution time

In safety-related systems it is important for the scheduler to have the major characteristics of the system in order to ensure that the timing constraints of all tasks will be met (Xu and Parnas, 1993). The upper bound of task execution time is considered as a key requirement when determining the most appropriate scheduling strategy (and scheduler parameters) for use with embedded systems (Engblom and Ermedahl, 2000; Engblom and Jonsson, 2002). For example the initial schedulability test for any kind of scheduler is to check that the total processor utilisation is less than or equals to one, task utilisation is calculated as the ratio of its WCET to its period.

The following subsections discuss how stabilising the task execution times to a value equals to its WCET will help in choosing the appropriate scheduler and configuring scheduler / task parameters, focusing on the TT schedulers.

4.2.1 Choosing an appropriate scheduler

Fixing the task execution time to a value equals to its WCET will help in choosing the appropriate scheduling strategy as most scheduling algorithms rely on knowledge about task WCET. For example, by checking the tasks' WCETs, along with their periods, for a given task set one can find that this set may not be schedulable by RM if their utilisation exceeds the limit in Equation 2-11. But, on the other hand, the same task set may still satisfy the EDF utilisation limit given by Equation 2-13 and hence be schedulable by the EDF.

Moreover task execution time is considered as a key player in other scheduling strategies, for example the Shortest Job First (SJF) strategy (Stankovic and Ramamritham, 1987; Cottet *et al.*, 2002; Buttazzo, 2005a) assigns priorities to tasks according to the length of their execution time, the shorter the task execution time the higher its priority.

The same applies in the situation where it is required to choose between the TTC and TTH schedulers. As explained earlier TTC fails to find a workable scheduler for a system which has one, or more, long task with WCET longer than the time frame within which the system has to respond to an external event, assuming that the long task has to be run as one segment. In this case the TTH may be considered as an appropriate choice. Additionally the pre-emptive task in TTH is normally chosen to be the task with the smallest period, which usually has a short execution time (Pont, 2001).

4.2.2 Configuring scheduler/task parameters

Given that an appropriate scheduling strategy is used, knowledge about task execution time still plays a key role in configuring task / scheduler parameters such as task offsets and tick intervals. For example the designer of safety-related applications may need to employ error checking and recovery mechanisms (such as task guardians (Hughes and Pont, 2004), and/or watchdog timers (Pont and Ong, 2002)). In these cases the tasks should be scheduled in such a way that the summation of WCETs of tasks which run at any tick does not exceed the length of the tick interval. To achieve this, an appropriate tick interval must be chosen. Moreover, if such a tick interval cannot be found, offsets can be used to balance the load over different ticks (an example is discussed in Section 6.3).

4.3 Challenges with estimating the task execution time

There are two main challenges involved in the process of estimating the task WCET (i) the path problem; and (ii) the state problem (Engblom and Ermedahl, 2000; Kirner and Puschner, 2008).

The path problem was related to finding the longest execution path (the path which takes the longest time) amongst all feasible paths through the program. It has been shown that as the complexity of today's embedded applications increases the number of different paths in the program may grow exponentially with program size, which makes the process of obtaining this path a challenging task (Deverge and Puaut, 2005; Kirner and Puschner, 2008).

On the other hand it is shown that obtaining the upper bound of the time taken by the processor to execute the longest path is another challenging process, this is due to the fact that the time taken to execute a specific instruction is not constant; rather it depends on the state of the processor at the time of executing the instruction (Rochange and Sainrat, 2002; Kirner and Puschner, 2008). As today's embedded designs become more complex and make use of faster and smaller processors and "system on chip" architectures (Baynes *et al.*, 2003; Kirner and Puschner, 2003), which often incorporate features such as pipelines, caches, and branch predictors that help to increase the performance, these features, on the other hand, make it difficult to determine the internal state of the system (Deverge and Puaut, 2005)

As a consequence, analysing the timing of the whole system and estimating the WCET requires a significant effort even for systems with a simple software architecture, such as time-triggered systems which use a table-driven task schedule (Puschner and Kirner, 2006). In addition, use of fault-tolerance polices based on knowledge of WCET becomes more challenging with these modern processors (Nett *et al.*, 1996).

A number of previous studies have been conducted to address the above problems for accurately estimating WCET: this work was based on two main approaches (i) static analysis; and (ii) measurements based analysis (sometimes called dynamic method) (Deverge and Puaut, 2005; Kirner and Puschner, 2008).

In measurement-based methods the time taken by the processor to execute the program (or a set of program segments) is normally measured on real hardware (or an accurate simulator) (Engblom and Ermedahl, 2000; Deverge and Puaut, 2005). The WCET is obtained by taking the maximum measured time over a number of trials using a set of selected input data.

Although this method seems to be an easy solution for obtaining the task WCET, it still has a number of drawbacks. Firstly, it may require a custom hardware and complex system setup, which may be available only at late stages of development (Engblom and Ermedahl, 2000). Secondly, the upper bound of the task execution time may not be captured; as sets of the used input data may not guaranteed to catch the longest execution time (Deverge and Puaut, 2005). Finally, it is not guaranteed that the measured value will be the same as the real value as it is sometimes needed to modify the code (for example by adding some instructions to steer timers) and hence the measured value will be for code different from the original code (Thesing, 2004).

On the other hand in static analysis based methods the WCET is calculated by statically analysing the program structures with the help of a theoretical model of the hardware (Deverge and Puaut, 2005). As explained earlier, although obtaining the longest path of the program is not an easy job with today’s complex programs, it is still not sufficient since the time taken to execute each instruction depends on the execution history (to incorporate the effect modern processor features such as of cache, pipelines and branch predictions) (Rochange and Sainrat, 2002; Kirner and Puschner, 2008). As in the case of the measurement-based method, the estimated value of the WCET calculated here will depend on the used input data (Thesing, 2004).

A number of previous studies have been conducted to address the problem of accurately estimating WCET: this work was based on the above explained methods, static-analysis-based method and / or measurements-based method, (e.g. (Ferdinand *et al.*, 1997; Theiling *et al.*, 2000; Engblom, 2002; Engblom and Jonsson, 2002; Puschner and Burns, 2002b; Rochange and Sainrat, 2002; Deverge and Puaut, 2005; Aparicio *et al.*, 2008)).

Deverge and Puaut (2005) proposed generation of test data to measure the execution times of different program segments. The test data is initially applied to the whole program and the execution times of the program are measured over a number of runs. If the resulting execution times are widely spread then the program is divided to a number of small segments, with the help of the program syntax tree. This process is repeated until the segments become small enough so that their measured execution times do not vary. The authors assumed that the execution time of the same program path with different data values will always be the same. They assumed that this can be done by controlling advanced processors mechanisms (for example by disabling the cache).

Engblom and Jonsson (2002) examined the timing of instructions from the perspective of static WCET using a mathematical model of instruction execution on in-order single-issue pipelined processors. The particular concern in this study was to study the timing effects between non-adjacent instructions. It is shown that it is not sufficient to calculate the WCET of a set of instructions by simply subtracting the pipelines speedup effect of overlapping pairs of adjacent instructions from the total sum of individual execution time. Rather it is necessary to take the effect of executing certain instruction on non-adjacent instruction as well, this can be either positive or negative.

In another study, Rochange and Sainrat (2002) studied the effect of speculative execution on instruction scheduling and fetching time . They showed that branch misprediction can result in delaying the scheduling of instruction belonging to the correct path. Moreover it is shown that an instruction belonging to the wrong path that has its operands ready can be executed before a preceding instruction from the correct path that is waiting for one of its operands. It is also shown that branch misprediction can result in longer fetching times of instruction from the correct path in case where the later is replaced by an instruction from the wrong path.

A different approach has been proposed by Puschner and Burns (2002b; 2002a; 2003): this is called "the single-path programming paradigm". This approach was based on the idea of writing the program in a manner which ensures that there is only one execution path. They showed that this helps to produce a constant task execution time and hence makes estimating the task WCET an easy job, this method will be discussed in more detail in the following chapter.

4.4 Dealing with execution time errors

As explained earlier, problems may be caused by variations of task execution time and inaccuracy in estimating task WCET. One simple solution to these problems is to err on the side of caution when employing WCET estimates, thereby reducing the chances that an overrun will occur. Typical "safety margins" used in this way are said to be around 20% (Vallerio and Jha, 2003).

Such an approach is simple and can be effective, but it inevitably adds to costs. An alternative is to be slightly more conservative when estimating WCET values (e.g. by adding 5% to accurate estimates) and then to extend the scheduler (or add additional hardware) in such a way that (at run time) any overrunning tasks can be shut down, and / or the schedule can be adjusted. Such an approach may allow dealing with error-related overruns (for example, tasks which overrun because of a hardware-related error). In these circumstances, the problem can be addressed (at least in part) by employing some form of "watchdog timer" (e.g. Ganssle (1992)) in a "scheduler watchdog" design (e.g. Pont and Ong (2002)). As an example of such an approach Pont and Ong (2002) introduced different scenarios that can be implemented in case of scheduler error caused by problems such as task overrun detected by watchdog timer. The first introduced scenario was based on resetting the system "reset recovery". This can be used in cases where the overrun caused by a temporary error. However, if the task overrun was the result of a permanent error they introduced a "fail silent" mode. In such mode the systems is assumed to be put in a freezing known safe state. Moreover a "limp home recovery" scenario was introduced in which a simple control algorithm, for example, can be used to control the system.

Alternatively, greater control over the system behaviour can be obtained by using a "task guardian" presented by Hughes and Pont (2004; 2008). The simplest form of the presented task guardian was to shut-down the overrun task and return the system to its original status before running that task. A better solution is also provided in which the facility of running a backup task, if one exists, is added to the task guardian which can be useful for safety-related applications. Alternatively, if a backup task does not exist then the scheduler can lower the priority of the overrunning task to reduce its impact in case it overran again.

The effectiveness of the above techniques can be increased if all tasks in the system run with a fixed execution time. This will help in having prior knowledge about the points in time at which each task should start / finish its execution and the order at which the tasks in the system run, as indicated earlier in this chapter.

4.5 Discussion

As discussed above, variations in task execution time may lead to violation of task constraints and / or decreasing the system predictability.

On the other hand by stabilising the task execution time and using the appropriate scheduling algorithm, and choosing the suitable task and scheduler parameters, all task constraints can be met. Moreover stabilising the task execution times will increase the systems determinism and predictability as it will help in predicting not only the task starting time but also its finishing time. Finally, maintaining a fixed task execution time has the potential of helping safety agents to check the correct operation of the system and take appropriate actions in case of error.

4.6 Conclusions

In this chapter the impacts of variations in task execution time is discussed. It has been shown that variations in task execution time may affect the system reliability and may cause violations of task constraints. On the other hand it has been shown that stabilising the task execution time, to a value equals to its WCET, can help in choosing the appropriate scheduler and configuring the scheduler / task parameters.

The chapter then gave an overview of previous work that has been described in the literature in the area of estimating task execution time. Finally it concluded by discussing ways used to deal with execution time errors.

The following chapter will present a set of novel code-balancing techniques that are intended to help reducing variations in the task execution time.

Chapter 5
Code-balancing techniques

This chapter presents a set of novel code-balancing techniques which help in reducing the variations in task execution time with limited overhead in system power consumption[6].

5.1 Toward a fixed execution path

As explained in the previous chapter the upper bound of task execution time is considered as a key characteristic which should be known in advance when designing safety-related systems. This upper bound can be obtained by calculating, or measuring the time taken by the processor to execute the longest path in the program. As noted by Deverge and Puaut (2005) "*The analysis of complete paths of the whole program could be unachievable in practice. Moreover, number of paths could be exponential even for small program.* " Unfortunately, "*Any brute-force approach like executing or simulating the program with all possible input data is doomed to fail due to the high number of paths and typically even higher number of different values of input data*." (Kirner and Puschner, 2008).

As part of an effort to address this problem, Puschner and Burns (2002b) proposed the "*single path programming paradigm*". As its name implies, program code written according to this paradigm has only one execution path: this helps to ensure a constant execution time. Yet there are two problems with the techniques described by Puschner and Burns: (i) they are applicable only to hardware which supports "conditional move" or similar instructions; (ii) their balancing approach increases power consumption. The work presented in this chapter will address both of these problems with a set of novel code-balancing techniques. The effectiveness of these new techniques is explored by means of an empirical study.

---

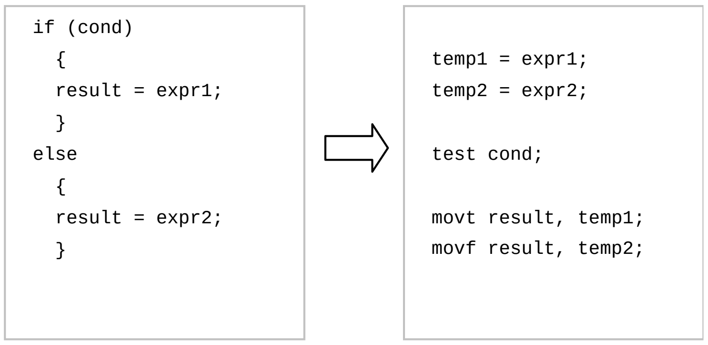
[6] Parts of this chapter have been published previously in Gendy and Pont (2007)

5.2 The single path programming paradigm

This section gives an overview of the single-path programming paradigm as described previously by Puschner and Burns (2002b; 2002a; 2003).

Programming code that complies with the single-path programming paradigm has only one execution path. This can be achieved by replacing input-data dependencies in the control flow by predicated (instead of branched) code. In predicated execution, instructions are associated with predicates: if the predicate evaluates to true the instruction is executed; otherwise the microprocessor internally replaces the instruction by a no-operation (NOP) instruction. It is assumed that a simple predicated execution model is used (such as the conditional move instruction in M-Core processor, in which conditional instructions have a constant, data-independent execution time).

As an example, Figure 5-1 shows pseudo code that indicates how a code branch using an *if-then-else* structure can be translated to the single path form. In this example the conditional move instruction "movt" copies the value of "temp1" to "result" if the result of the "test" instruction is true; otherwise the processor performs a NOP instruction. The same can be said for the "movf" instruction; it will copy the value of "temp2" to "result" if the result of the "test" instruction is false; otherwise the processor performs a NOP instruction. This code can be easily modified to be used with nested if statements (Allen *et al.*, 1983).



```
if (cond)
  {
  result = expr1;
  }
else
  {
  result = expr2;
  }
```

```
temp1 = expr1;
temp2 = expr2;

test cond;

movt result, temp1;
movf result, temp2;
```

Figure 5-1 Converting *if-then-else* structure to single path, adapted from (Puschner and Burns, 2003).

In a similar manner, a loop of variable length can be translated into a loop of constant length (provided that the maximum size of the loop is known). Less structured "*goto*" and "*exit*" statements are not considered in this approach.

It has been demonstrated by Puschner and Burns (2002b; 2002a; 2003) that using this method helps to produce a constant task execution time. However, this method has some drawbacks:

(i) Its usage is limited to hardware which supports "conditional move" or similar instructions

(ii) It is likely to increase power consumption because the CPU will always execute the single-path code for a fixed (maximum) period. During this time, the processor will be in "full power" mode.

5.3 The proposed CB1 techniques

This section addresses the drawbacks mentioned above by using a set of novel code-balancing techniques. For ease of reference, the approaches described here are referred to as the "CB1 techniques" in the remainder of this thesis.

5.3.1 Overview

The main idea behind the CB1 approach can be explained by considering an example. This example is intended to reduce variations in the time taken to complete a number of iterations in a given loop.

Assume that the time spent in performing "*x*" iterations of the loop is equal to Time(*x*). The microcontroller is set to enter a power-saving mode for the period of time required to perform (MAX - *x*) iterations, where MAX is the maximum number of iterations. This time can be approximated by Equation 5-1.

$$\text{Time}\,(\text{MAX} - x) = \frac{(\text{MAX} - x) * \text{Time}(x)}{x}$$ **Equation 5-1**

Hardware timers can be used to measure Time(x); the time spent in performing x iterations, by starting the timer directly before the start of the loop and stop it directly after the last statement of the loop. By substituting this value in Equation 5-1, Time(MAX - x) can be calculated. This time can then be used to set a timer interrupt to waken the microcontroller after the required time has elapsed.

It should be noted that it is assumed that the loop will be executed at least once for Equation 5-1 to give real results. It should also be noted that Time(MAX – x) given by Equation 5-1 is an approximation as it does not take into account the effects of the performance improvements features mentioned earlier. The reason for this approximation is to simplify the calculations and the implementations process so as to be suitable for any platform while avoiding complex analysis of each specific feature.

Based on this form of "sandwich delay"(Pont *et al.*, 2006), a set of balanced code can be used to reduce the variations in executing *for* and *while* loops, as explained in the following subsections.

5.3.2 Balanced *for* loop

Figure 5-2 shows pseudo code that can be used to reduce variations in the WCET of a *for*-loop for any number of iterations in the range of [1, MAX], where MAX is the maximum number of iterations.

A small "safety margin" was added to the time calculated in Equation 5-1 to assure that there is time to enter sleep mode before the interrupt occurs even at the maximum loop length.

```
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;
```

Figure 5-2 Pseudo code of a balanced *for*-loop.

5.3.3 Balanced *while* loop used for waiting for input

Figure 5-3 shows a pseudo code that can be used to reduce variations in the WCET of executing a *while*-loop which is usually used to wait, for a predefined maximum time (TMAX), for an input to be ready.

A small "safety margin" was (again) added to the time TMAX after the end of the while loop to ensure that there is time to enter sleep mode before the interrupt occurs, even in cases where the input becomes ready at time TMAX. The safety margin will typically be 1% of the value of TMAX.

```
Set timer interrupt to occur after Time = TMAX,
//(where TMAX equals to maximum time of waiting for input //
to be available)

Start timer;

while (1)
   {
   if input is available or timer
      count equals to TMAX then
      break;
   }
Stop timer;
Adjust timer interrupt to occur after time:
Time_till_next_int=(TMAX + “safety margin” – “timer value”);
Send the microcontroller to power saving mode;
```

Figure 5-3 Pseudo code of a balanced *while*-loop used for waiting for input.

5.3.4 Balanced *if-then-else* structure

Figure 5-4 shows a pseudo code that can be used to reduce variations in the WCET of a general *if-then-else* structure.

It should be noted that the number of the assignment instructions in the *if*-part must be equal to those in the *else*-part ("NOP" padding or similar approaches must be used, if necessary).

```
temp1 = expr1;
temp2 = expr2;

if (cond)
   {
   result = temp1;
   }
else
   {
   result = temp2;
   }
```

Figure 5-4 Pseudo code of a balanced *if-then-else* structure, adapted from (Puschner and Burns, 2003).

5.4 Performance of the CB1 techniques

An empirical test was carried out to explore the effectiveness of the CB1 techniques. The procedure employed and the results obtained are detailed in this section.

5.4.1 Experimental methodology

The following subsections describe the experimental methodology used to obtain the results shown in this chapter.

5.4.1.1 Hardware platform

The tests was carried out on an NXP (formerly Philips) LPC2106 microcontroller running on a small evaluation board (Philips, 2004b). The LPC2106 is based on an ARM7TDMI core and is typical of modern (low cost) embedded processors. Except where otherwise noted, the microcontroller used an oscillator frequency of 12 MHz, and run – under control of an on-chip PLL – a CPU frequency of 60 MHz. Because this microcontroller does not support the conditional move instruction the single path code is modified, while keeping the main structure described by (Puschner and Burns, 2002b), to cope with this limitation.

5.4.1.2 Software tool chain

The compiler used was the GCC ARM 4.1.1 operating in Windows by means of Cygwin (a Linux emulator for windows). The IDE and simulator used was the Keil ARM development kit (v3.12), with all compiler optimizations turned off.

5.4.1.3 Power measurements

To obtain representative values of power consumption, the input current and voltage to the LPC2106 CPU core were measured using the same method described in Phatrapornnant (2007) and Nahas (2009), as shown in Figure 5-5. The voltage measurements were obtained by using the National Instruments data acquisition card ‘NI PCI-6035E’ in conjunction with LabVIEW 7.1 software. Values for currents and voltages were then multiplied and then averaged out to give the power. The sampling rate was 100 KHz and a total of 800000 samples were recorded. Although this sampling rate was not high enough to capture all the voltages fluctuations that results from executing different instructions it gives a trend of the results. More accurate results can be obtained by using a data acquisition card with higher sampling rate.

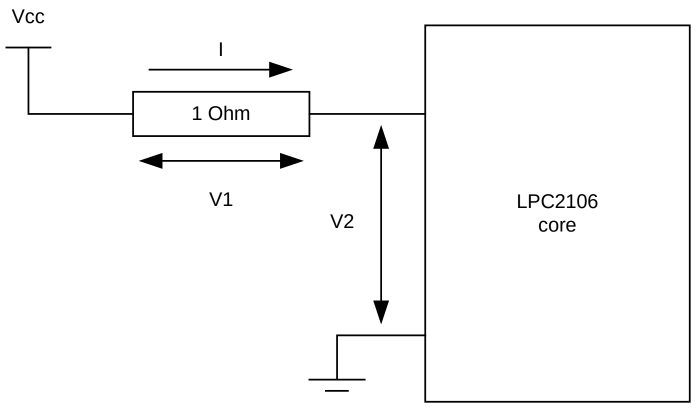


Figure 5-5 The circuit used to measure the system power consumption, copied from Nahas (2009)

5.4.1.4 Jitter and execution time measurements

In order to measure the task jitter experimentally, a pin was set high at the beginning of the task (for a short time) then the periods between every two successive rising edges were measured using the same data acquisition card mentioned above (this has 24 bit timer/counter works with 20 MHz). For every experiment a total of 2000 periods were recorded and the jitter was calculated according to Equation 2-1. Those were averaged to get the final jitter value.

A similar approach was adapted to measure the task execution time, a pin was set high at the beginning of the task and set low at the end of the task execution (this will produce a pulse of width equals to the task execution time) then the pulse width between the raising edge and failing edge was measured. For every experiment a total of 2000 pulse widths were recorded from which the maximum, minimum and average values were calculated.

5.4.2 Initial test

In this test an example which was used to assess the effectiveness of the single path programming paradigm (in Puschner and Burns (2002b)) is used. The original example explores different implementations of a "bubble sort" for arrays of 10 elements. The original example was used to sort all the elements of the array. The version used here sorts the first *x* elements of the array (where *x* is <= SIZE, the total number of the array elements). This modification was made in order to explore the impact of different implementations on the execution time, jitter, and power consumption.

The tests employed a TTC scheduler. The tick interval was set to 10 *ms*. The main (sorting) task was run every two ticks.

Three additional tasks were also scheduled:

i) A "jitter-test" task. This low-priority task was scheduled to execute in the same tick as the sorting task. It was used to measure the effect of the variations in the execution time of the sorting task on the jitter in the start times of other tasks in the system

ii) A "sort length" task used to increment the value of *x*, from 2 to 10 and then back to 2.

iii) A "sort complexity" task used to initialise the array in "completely sorted" or "completely unsorted" forms, in order to vary the time taken to carry out the sort process.

Figure 5-6, Figure 5-7 , and Figure 5-8 show different implementations of the bubble sort using the traditional, single path, and the proposed CB1 code respectively.

Table 5-1 and Table 5-2 show the measured minimum and maximum execution time, maximum jitter, and average power consumption resulted from each implementation. From these tables it can be noticed that:

- Both the single-path code and CB1 code demonstrated a reduction in both the variation of the execution time and in jitter levels. These improvements were at the expense of an increase in the average power consumption and execution time.

- The jitter and the variations in execution time obtained by using the CB1 code was less than that of the traditional code and higher than that of the single-path code.

- The average power consumption obtained by using the CB1 code was less than that of the single path code and higher than that of the traditional code.

```
void  traditional_bubble( )
   {
   int i,j,t;
   for (i=x-1; i>0; i--)
      {
      for (j=1;j<=i;j++)
         {
         if (a[j-1] > a[j])
            {
            //swap
            t = a[j];
            a[j] = a[j-1];
            a[j-1] = t;
            }
         }
      }
   }
```

Figure 5-6 Traditional implementation of bubble sort.

```
void single_path_bubble( )
   {
   int i,j,s,t, dummy[10];
   char  Finished_i,Finished_j;

   for (i=SIZE-1; i>0; i--)
      {
      if (i > (x-1))
         { Finished_i = 1; }

      if (!( i > (x-1)))
         { Finished_i = 0; }

      for (j=1;j<= SIZE-1;j++)
         {
         if (!(j <= i))
            { Finished_j = 1; }

         if (j <= i)
            { Finished_j = 0; }

         if (!Finished_i && !Finished_j)
            { //do real statements
            s = a[j-1];
            t =   a[j];

            if (s <= t)
               {
               a[j-1] = s;
               a[j] = t;
               }

            if (s > t)
               {
               a[j-1] = t;
               a[j] = s;
               }
             }

         if (!(!Finished_i &&
             !Finished_j))
            { //do dummy statements
            s = dummy[j-1];
            t = dummy[j];
```

Figure 5-7 Single path implementation of bubble sort (adapted to work without the support of the conditional move instruction) (Part 1/2).

```
                    if (s <= t)
                       {
                       dummy[j-1] = s;
                       dummy[j] = t;
                       }

                    if (s > t)
                       {
                       dummy[j-1] = t;
                       dummy[j] = s;
                       }
                    }
                 }
              }
           }
```

Figure 5-7 Single path implementation of bubble sort (adapted to work without the support of the conditional move instruction) (Part 2/2).

```
void balanced_code_bubble()
   {
   int i,j,s,t;
   unsigned long Temp1=0,Temp2=0;
   // Initialise and start T1
   T1MCR = 0x00;
   T1TCR = 0x2;    // Reset counter
   T1TCR = 0x01;   // Counter enable
   for (i=x-1;i>0;i--)
      {
      //Store T1 value before inner loop
      T1TCR = 0x00;
      Temp1 = Temp1 + T1TC;

      // Prepare T1 for inner loop
      T1MCR = 0x00;
      T1TCR = 0x2;
      T1TCR = 0x01;

      for (j=1;j<=i;j++)
         {
         s = a[j-1];
         t = a[j];
         if (s<=t)
            {
            a[j-1] = s;
            a[j] = t;
            }
         if (s > t)
            {
            a[j-1] = t;
            a[j] = s;
            }
         }
      // Store T1 value at end of inner
      // loop
      T1TCR = 0x00;
      Temp2 = T1TC;
```

Figure 5-8 CB1 implementation of bubble sort (Part 1/2).

```
        // Adjust MR for the remaining
        // time,
        // start T1 and go to sleep
        T1MR0 = (( SIZE- i)*Temp2 )/i;
        T1TCR = 0x2;
        T1MCR = 0x05;
        T1TCR = 0x01;
        PCON = 1;      // Go to sleep

        // Complete the outer loop
        // Initialise and start T1
        T1MCR = 0x00;
        // Add the time spent in inner
        // loop to the time spent so
        // far in the outer loop
        Temp1 = Temp1 + Temp2 + T1TC;
        T1TCR = 0x2;  // Reset counter
        T1TCR = 0x01;// Counter enable
        }

     // Stop T1 and calculate the remaining
     // time
     // in the outer loop
     T1TCR = 0x00;    // Stop counter
     Temp1 = Temp1 + T1TC;
     // Adjust MR for the remaining time,
     // start T1 and go to sleep
     T1MR0 = (( SIZE- (x-1))*Temp1)/(x-1);
     T1TCR = 0x2;    // Reset counter
     T1MCR = 0x05;
     T1TCR = 0x01;   // Counter enable
     PCON = 1;       // Go to sleep
     }
```

Figure 5-8 CB1 implementation of bubble sort (Part 2/2).

Table 5-1 Minimum, maximum, (maximum – minimum), and percentage of variations (w.r.t. the maximum) in task execution time resulted from different implementations of bubble sort.

	Min (ms)	*Max (ms)*	*(Max- Min) (ms)*	*% of variations*
Traditional	0.00815	0.33165	0.32350	97.54 %
CB1	0.90050	1.00265	0.10215	10.19 %
Single-path	0.80210	0.85305	0.05095	5.97 %

Table 5-2 Maximum jitter and average power consumption resulted from different implementations of bubble sort.

	Maximum jitter (*ms*)	Average power consumption (*mW*)
Traditional	0.3208	11.8120
CB1	0.0689	13.0185
Single path	0.0515	13.0193

5.4.3 Extended test

The experiment described in the previous section was repeated using two additional benchmark test cases which have been used in previous WCET studies (Engblom and Ermedahl, 2000; Engblom *et al.*, 2001):

i) The first test case implements a single nested loop used to calculate the Fibonacci series for up to 30 elements.

ii) The second test case implements a triple-nested loop used to calculate matrix multiplication of two 2-D arrays up to 20x20 elements in size.

The length of the tick interval of the TTC scheduler was set to 10 *ms* for the first test and 100 *ms* for the second test.

Table 5-3 through Table 5-6 show the measured minimum and maximum execution times, maximum jitter, and average power consumption resulted from each case. These results were in line with the results obtained from the test described in the previous section.

Table 5-3 Minimum, maximum, (maximum – minimum), and percentage of variations (w.r.t. the maximum) in task execution time resulted from different implementations of Fibonacci.

	Min (*ms*)	Max (*ms*)	(Max- Min) (*ms*)	% of variations
Traditional	0.00565	0.08810	0.08245	93.59 %
CB1	0.11345	0.17700	0.06355	35.90 %
Single path	0.16640	0.16635	0.00005	0.03 %

Table 5-4 Maximum jitter and average power consumption resulted from different implementations of Fibonacci.

	Maximum jitter (*ms*)	Average power consumption (*mW*)
Traditional	0.0829	11.2170
CB1	0.0631	11.2684
Single path	0.0005	11.4732

Table 5-5 Minimum, maximum, (maximum – minimum), and percentage of variations (w.r.t. the maximum) in task execution time resulted from different implementations of Matrix Multiplication.

	Min (ms)	*Max (ms)*	*(Max- Min) (ms)*	*% of variations*
Traditional	0.06760	41.62990	41.56230	99.84 %
CB1	52.39640	59.85655	7.46015	12.46 %
Single path	66.25970	66.49350	0.23380	0.35 %

Table 5-6 Maximum jitter and average power consumption resulted from different implementations of Matrix Multiplication.

	Maximum jitter (*ms*)	Average power consumption (*mW*)
Traditional	41.5651	15.8626
CB1	3.5232	18.2561
Single path	0.2366	39.0217

5.5 Discussion

As explained in this chapter obtaining accurate estimates of task WCET is a challenging process. Yet it is a key factor in designing the task schedule and ensuring the satisfaction of task constraints. Of equal importance is reducing the variations of task execution time.

In an effort to address these problems Puschner and Burns (2002b; 2002a; 2003) introduced what they called the "single path programming paradigm" which helps to fix task execution time to its maximum length so that it can be measured easily.

The work presented in this chapter highlighted two issues with the techniques described by Puschner and Burns: their limitation to be applicable to certain hardware, and their effects on slightly increasing the power consumption. In this chapter, both of these problems have been addressed with a novel set of code-balancing techniques.

The CB1 techniques presented in this chapter involved two stages:

i) using an interrupt-based sandwich delay to keep the execution time of tasks fixed without requiring much increases in system power consumption.

ii) calculating the maximum execution time (and required timer settings) for each form of branch / loop structure.

It has been demonstrated (using empirical studies) that the presented code-balancing techniques gave intermediate results (compared with those of the traditional code and the single-path programming paradigm) both in terms of reducing the variations in task execution time and slightly increasing the power consumptions.

5.6 Conclusions

This chapter gave an overview of the "single path programming paradigm" as introduced by (Puschner and Burns, 2002b). Then it presented some code-balancing techniques which address two issues that limit the usage of this technique. It has been shown that the use of these code-balancing techniques help to reduce variations in task execution time by adjusting the total time taken by the processor to execute the task with its WCET. This is done by sending the processor to power saving mode for a time period equals to the difference between the task execution time of any task instance and its WCET. The proposed code-balancing techniques are then empirically assessed.

In the following chapters a novel scheduling algorithm, which takes task WCET along with other task constraints, as an input, is presented. The algorithm tries to find a suitable scheduler and configure the task and scheduler parameters, whenever one is found. Before introducing the algorithm the effects of inappropriate choices of task and scheduler parameters are discussed and analysed in the following chapter.

Chapter 6

Analysis of scheduler/task configuration

This chapter analyses the effects of inappropriate configurations of the scheduler and / or task parameters on the task behaviour and system power consumption.

6.1 A close look at TTC and TTH architectures

Before starting to discuss the possible effects of scheduler / task parameters, this section begins by giving a detailed overview of the TT schedulers discussed in this project, as described by (Pont, 2001; Maaita and Pont, 2005b; Kurian and Pont, 2007).

6.1.1 TTC scheduler

As its name implies, the TTC schedules the tasks co-operatively according to the time-triggered architecture. This means that tasks are dispatched in predefined points in time and each task runs to completion without being pre-empted. The time duration between each two such points is usually fixed and called the "tick interval". At every tick the status of each task is checked / updated and the tasks which are ready to run, if any, are dispatched in order of their importance (priority). This order is specified offline before the scheduler starts its execution.

The length of the scheduler tick interval can be adjusted by using suitable delay functions. Generating the tick interval in this way is very simple and has small resource requirements (Pont, 2001; Kurian and Pont, 2007). However the main disadvantage of this method is that it is almost impossible to obtain a fixed length of tick interval if the tasks executions times are not fixed (Kurian and Pont, 2007). This will, in turn, have negative effects on task jitter and / or may cause some tasks to miss their deadlines.

A better way to have a fixed tick interval is to use a hardware timer. An interrupt can be set to occur whenever a timer overflows (or a timer count matches certain value). The timer can easily be adapted to generate the interrupt in periodic bases with a fixed period of time equals to the tick interval. Using this method will allow the scheduler to set the processor to an "idle" or power saving mode, to reduce the power consumption, after finishing the execution of all ready tasks, if any, in each tick. A possible implementation of such a scheduler is described in Appendix A.

Provided that an appropriate implementation is used, a time-triggered, cooperative (TTC) architecture is a good match for a wide range of low-cost, resource-constrained applications. TTC architectures also demonstrate very low levels of task jitter (Locke, 1992), and they can maintain their low-jitter characteristics even when techniques such as dynamic voltage scaling (DVS) are employed to reduce system power consumption (Phatrapornnant and Pont, 2006).

Table 6-1 lists the specifications of 3 tasks as an example of a system of tasks scheduled by TTC. It should be noted that the offsets of Task A and Task B are zero while the offset of Task C is 1 *ms*.

Figure 6-1 shows the timeline of the corresponding TTC scheduler which uses a tick interval of 1 *ms*.

Table 6-1 Example of task specifications scheduled by TTC.

Task	WCET (ms)	Deadline (ms)	Period (ms)	Offset (ticks)
A	0.3	1	2	0
B	0.4	1	2	0
C	0.5	1	2	1

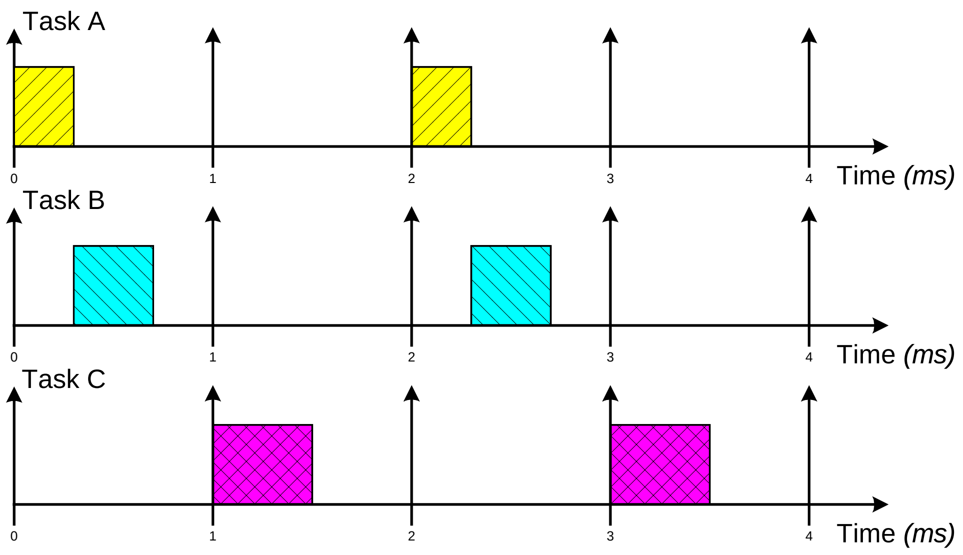


Figure 6-1 Illustrating the operation of a typical (interrupt-driven) TTC scheduler implementation.

6.1.2 TTH scheduler

The TTC scheduler described above is a non pre-emptive scheduler, in the sense that it is not possible that a low priority task to be pre-empted by a high priority task if the latter becomes ready to run while the former is still running, i.e. each task is guaranteed to run to completion without being pre-empted by another task. Hence, systems based on this kind of schedulers have the advantage of being highly predictable and having limited resource requirements (due to the limited number of context switching and the unnecessity of implementing complex techniques to control access to shared resources).

Unfortunately these schedulers can not be used in all situations. Allworth (1981) noted: "[The] *main drawback with this* [co-operative] *approach is that while the current process is running, the system is not responsive to changes in the environment. Therefore, system processes must be extremely brief if the real-time response* [of the] *system is not to be impaired*."

This concern can be expressed slightly more formally by noting that if a system is being designed which must execute one or more tasks of (worst-case) execution time e and also respond within an interval t to external events then, in situations where $t < (e +$

execution time of the task that handles the event), a pure co-operative scheduler will not generally be suitable, as shown in Table 6-2 and Figure 6-2.

Table 6-2 Example of task specifications which cannot be scheduled by TTC.

Task	WCET (ms)	Deadline (ms)	Period (ms)
A	0.2	0.2	1
B	1.3	3	3

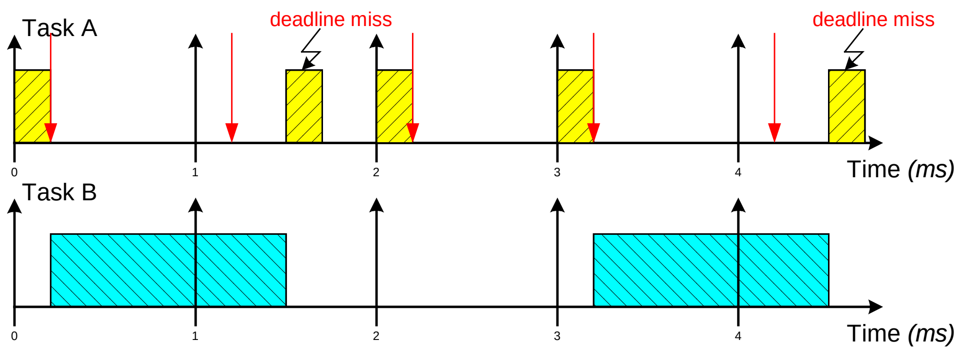


Figure 6-2 illustration of TTC schedule of task set shown in Table 6-2.

In such circumstances, it is tempting to opt immediately for a fully pre-emptive design. Indeed, some studies seem to suggest that this is the only viable alternative (e.g. Locke (1992) dna Bate (1998)). However, there are other design options available. For example, a single, time-triggered, pre-empting task can be added to a TTC architecture, to give what have been called a "time-triggered hybrid" (TTH) scheduler (Pont, 2001; Maaita and Pont, 2005b) or "multi-rate executive with interrupts" (Kalinsky, 2001). Use of a TTH scheduler allows the system designer to create a static schedule made up of (i) a collection of tasks which operate co-operatively and (ii) a single – short - pre-empting task, Figure 6-3 shows how the tasks shown in Table 6-2 can be scheduled with a TTH scheduler with a tick interval of 1 *ms*.

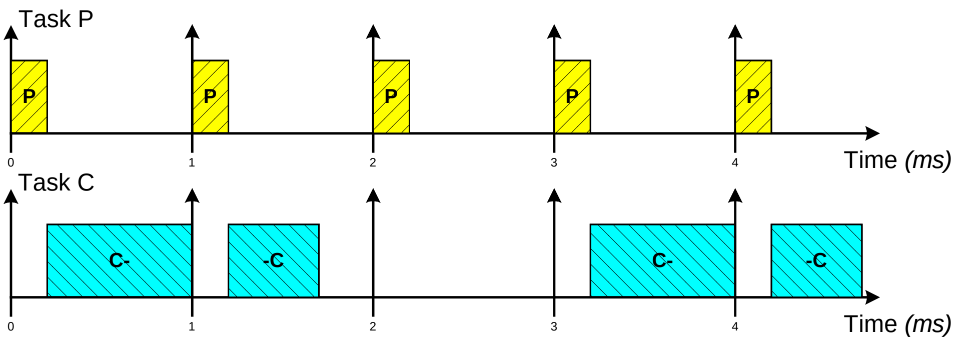


Figure 6-3 Illustrating the operation of a typical TTH scheduler implementation, adapted from Maaita and Pont (2005b), Figure 1.

It should be noted that in this schedule, all tasks are periodic. This is in contrast to architectures investigated in some previous studies (e.g. Sandström *et al.* (1998)) which have sought to integrate time-triggered task scheduling with the response to aperiodic (event related) interrupts.

6.2 The need for appropriate configuration of scheduler/task parameters

Whether a TTC or TTH implementation is used, a number of key scheduler/task parameters must be determined (including the tick interval, task order, and initial delay - or phase - of each task). Inappropriate choices may mean that a given task set cannot be scheduled at all. Where the parameter set does ensure that all tasks are scheduled, inappropriate decisions may still lead to unnecessarily high levels of task jitter and / or to increased system power consumption.

The following sections will discuss the effects of inappropriate choices of task and / or scheduler parameters. Some examples will be used to illustrate the effects of these parameters. However, to limit the thesis length, and the number illustrations, in cases where the same ideas applies in the same way when using either the TTC or TTH schedulers, the idea will be illustrated using the TTC scheduler for ease of clarification.

6.3 Effects of task offset

Inappropriate choices of task offset may render the task set unschedulable (for example some tasks may miss their deadlines). Moreover even if the chosen offsets do ensure that all tasks are scheduled, inappropriate choice of offsets may still lead – for example – to unnecessarily high levels of task jitter.

6.3.1 Effects of task offset on schedulability

As shown earlier, the task offset specifies when the task should start (more precisely it specifies the first tick at which the first instance of the task is ready to run). In some situations, tasks cannot be scheduled if they start execution at the same time. In these situations a task offset can be used to balance the processor load over various ticks so that each task meets its deadline.

Table 6-3 shows an example for a set of tasks which cannot all be scheduled when all tasks begin their executions simultaneously (all offsets equal to zero). This is because the sum of the WCETs means that Task C cannot meet its deadline as it has to wait until both Task A and Task B finish their executions before it can start its execution.

By using a suitable tick interval and adjusting the task offsets, a workable schedule can often be achieved (Goossens and Devillers, 1997; Xu and Parnas, 2000; Pont, 2001). For example, the tasks in Table 6-3 can be scheduled if a tick interval of 5 *ms* is used and the offset of Task C is adjusted to 1 tick (as shown in Table 6-4).

Table 6-3 Task specifications for a system in which task offsets are inappropriate (Task C missed its deadline).

Task	WCET (ms)	Deadline (ms)	Period (ms)	Offset (ticks)
A	1	5	5	0
B	1.5	5	10	0
C	3	5	10	0

Table 6-4 Task specifications for a system in which task offsets are appropriate. (All tasks met their deadlines).

Task	WCET (ms)	Deadline (ms)	Period (ms)	Offset (ticks)
A	1	5	5	0
B	1.5	5	10	0
C	3	5	10	1

6.3.2 Effects of task offset on jitter

Where the parameter set does maintain various task requirements and constraints (such as deadlines, precedence, distance, and latency) inappropriate setting of offsets may still increase the task jitter to a value beyond the permitted level.

Table 6-5, and the corresponding figure (Figure 6-5), shows an example for inappropriate choices of a set of offsets for a task set. By using the shown values of offsets it can be noticed that in some ticks Task C runs after both Task A and Task B while in the other ticks it runs directly after Task A. So Task C will suffer from jitter values approximately equal to the WCET of Task B which may not be a desired behaviour if Task C is a jitter sensitive task.

Table 6-5 Task specifications for a system in which task offsets cause high jitter.

Task	WCET (ms)	Deadline (ms)	Period (ms)	Offset (ticks)
A	2	10	10	0
B	3	40	40	0
C	2	20	20	0

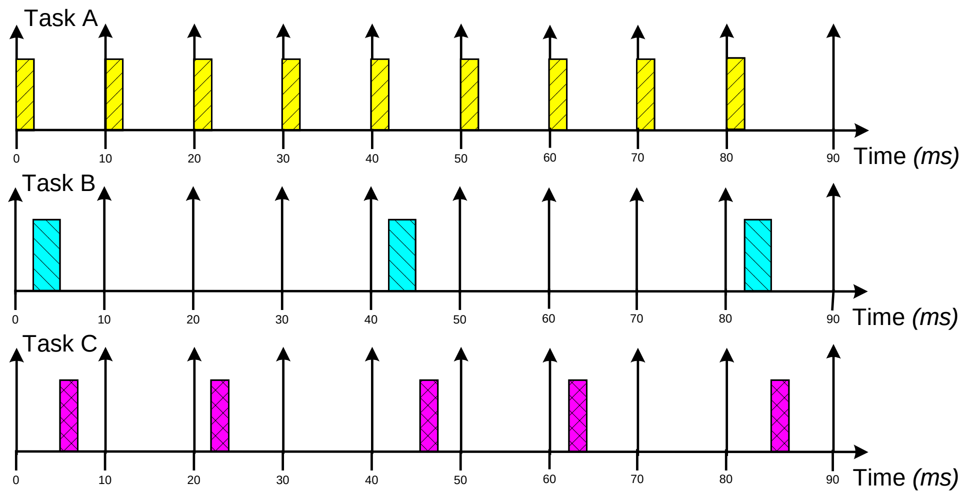


Figure 6-4 Task schedule for tasks shown in Table 6-5.

Alternatively if the offsets are adjusted as shown in Table 6-6, Task C will always run after Task A and hence will have a reduced jitter value (depending on the variations in Task A execution times), as shown in Figure 6-5.

Table 6-6 Task specifications for a system in which task offsets cause low jitter.

Task	WCET (ms)	Deadline (ms)	Period (ms)	Offset (ticks)
A	2	10	10	0
B	3	40	40	0
C	2	20	20	1

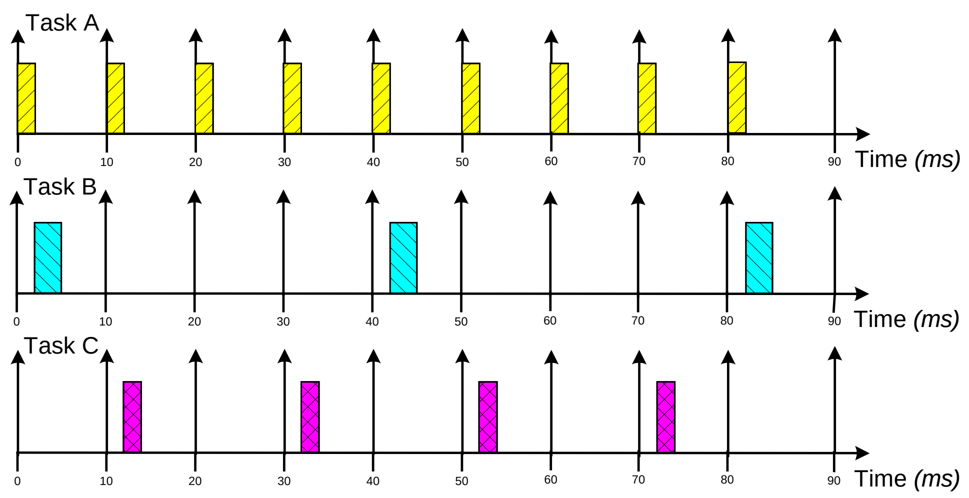


Figure 6-5 Task schedule for tasks shown in Table 6-6.

6.4 Effects of tick interval

Choosing the appropriate length of the tick interval plays an important role in TT architectures. The following subsections discuss the effects of various lengths of tick interval on task schedulability, jitter, and system power consumption.

6.4.1 Effects of tick interval on schedulability

The developer of TT architectures has to carefully choose a suitable value for the length of the tick interval so that it fits his application needs. An inappropriate value of tick interval may cause violations of task constraints (such as missed deadlines).

An example that illustrates the effect of inappropriate choice of tick interval in violating task deadlines is shown in Table 6-7 and Figure 6-6. It can be noticed that using a tick interval of 2 *ms* will cause missed deadline of Task B. On the other hand using a tick interval of 1 *ms* (with the appropriate offsets), as shown in Table 6-8 and Figure 6-7, will help to ensure that all the tasks meet their deadlines.

Table 6-7 Task specifications for a system in which tick interval is inappropriate (Task B missed its deadline).

Task	WCET (ms)	Deadline (ms)	Period (ms)	Offset (ticks)
A	3	5	20	0
B	4	5	20	0

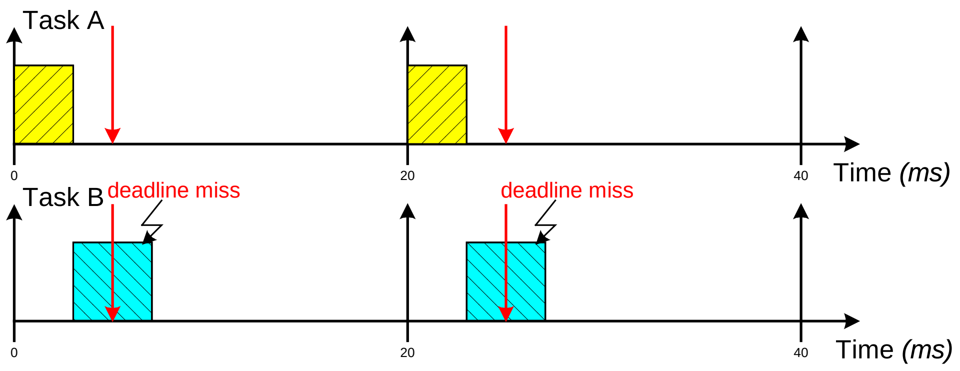


Figure 6-6 Task schedule for tasks shown in Table 6-7 (with tick interval = 2 *ms*).

Table 6-8 Task specifications for a system in which tick interval is appropriate (all tasks met their deadlines).

Task	WCET (ms)	Deadline (ms)	Period (ms)	Offset (ticks)
A	3	5	20	0
B	4	5	20	1

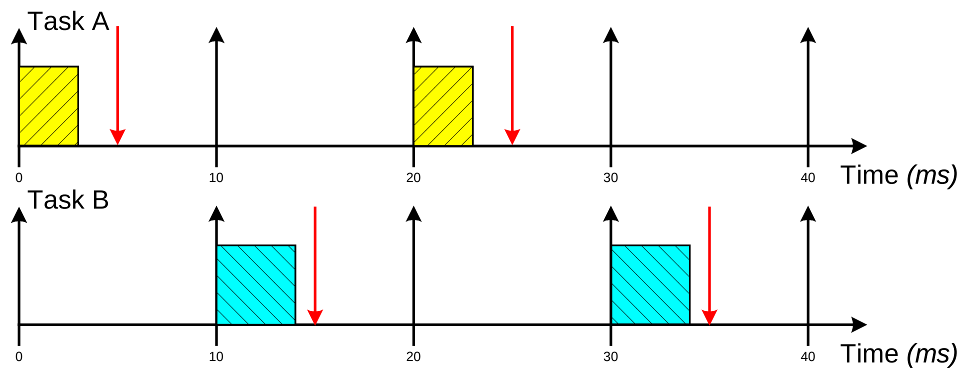


Figure 6-7 Task schedule for tasks shown in Table 6-8 (with tick interval = 1 *ms*).

6.4.2 Effects of tick interval on jitter

In cases where the application at hand contains one or more jitter-sensitive tasks, a suitable tick interval (along with appropriate offset) has to be chosen.

To illustrate the effect of tick interval on task jitter, assume that all the tasks specified in Table 6-9 are jitter-sensitive tasks. Using a tick interval of 2 *ms* will ensure that all tasks meet their deadline, but on the other hand it will cause Task B to suffer from jitter which is caused by the variations in Task A execution times as shown in Figure 6-8. One way to overcome this problem is to use a tick interval of 1 *ms* (with appropriate values for the offsets) as shown in Table 6-10 and Figure 6-9. This will enable every task to run in a separate tick and hence not be affected by the variations in the other task execution time, if any.

Table 6-9 Task specifications for a system with an inappropriate tick interval (Task B suffers from jitter).

Task	WCET (ms)	Deadline (ms)	Period (ms)	Offset (ticks)
A	0.3	2	2	0
B	0.4	2	2	0

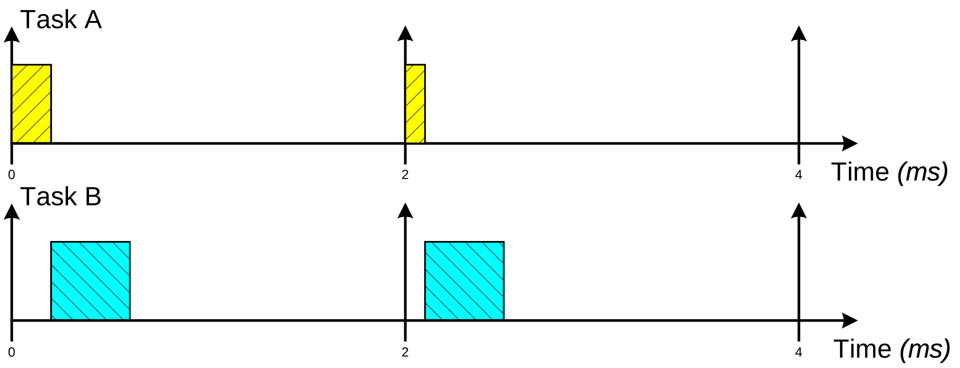


Figure 6-8 Task scheduler for tasks shown in Table 6-9 (with tick interval = 2 *ms*).

Table 6-10 Task specifications for a system with an appropriate tick interval (all tasks have low jitter; approximately zero).

Task	WCET (ms)	Deadline (ms)	Period (ms)	Offset (ticks)
A	0.3	2	2	0
B	0.4	2	2	1

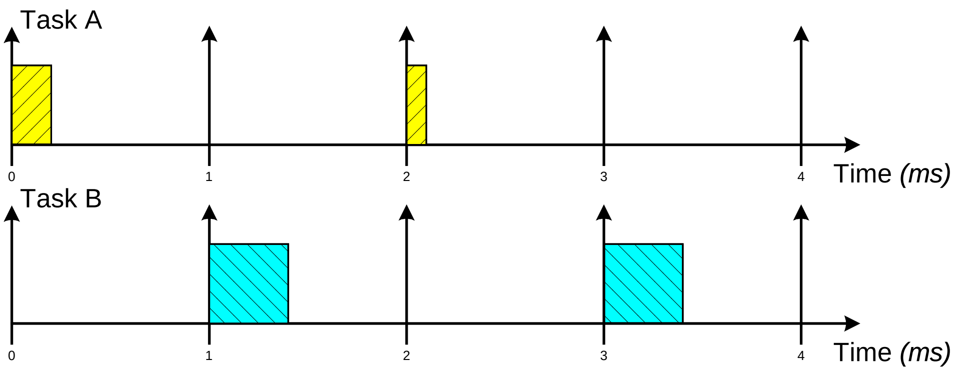


Figure 6-9 Task scheduler for tasks shown in Table 6-10 (with tick interval = 1 *ms*).

6.4.3 Effects of tick interval on power consumption

As explained earlier in this chapter, the TT architectures which are discussed in this thesis (TTC / TTH) are built on the idea of executing the tasks from a (dispatcher) function which is invoked at every scheduler tick. This function updates the state of each task, runs "ready" tasks, and then it places the processor into a power-saving mode until the following tick. If the tick interval employed is shorter than necessary, there may be some empty ticks (ticks in which there are no tasks ready to run, Figure B-3 shows an example) during which the system has to come out of the power saving mode to execute the dispatcher before the system goes back to the power-saving mode. This will, inevitably, increase power consumption when compared to a design with an "optimal" tick interval.

To illustrate the effect of tick interval on system power consumption an experiment was carried out. In this experiment a set of 3 dummy tasks was used and run on an NXP LPC2106 microcontroller (Philips, 2004b). The period of all the three tasks was set at 10 *ms*. The power consumption of the core microcontroller was measured using a range of different tick intervals (using the approach described in Section 5.4.1.3). In each case, the results of several runs were averaged using both TTC and TTH architectures (as shown in Table 6-11).

Table 6-11 Average power consumption (mW) using different tick intervals.

Tick interval (ms)	TTC	TTH
1	16.6725	17.5583
2	16.3807	16.5104
5	16.1999	16.2332
10	16.1262	16.1524

It can be seen from Table 6-11 that, for both TTC and TTH, choosing the largest possible tick interval reduces the power consumption.

It should be noted that a specific value of tick interval may help in reducing the total average power consumption but at the same time it may cause an increase in the jitter level of one or more tasks, and vice versa, i.e. another choice of tick interval may cause more power consumption but, if used with appropriate offsets, may reduce the jitter level of jitter-sensitive tasks in the system.

So choosing the length of the tick interval will depend on the requirements of the application at hand.

6.5 Effects of task order

The sequence at which TTC and TTH check and update the status of each task, and run the ready ones (in each tick) has to ensure that task precedence constrains are met. Even in situations where task order does maintain these precedence constraints, an inappropriate choice of task order can affect task schedulability and/or jitter.

Although all tasks run co-operatively in TTC, and there is a limited level of pre-emption in TTH, the sequence at which tasks are added to the schedule assigns different levels of priority of the co-operative tasks. For example the first task in each tick will normally have the lowest level of jitter as will be explained in the following subsections.

The following subsections give an overview on the effects of inappropriate task order in the task schedulability and jitter.

6.5.1 Effects of task order on schedulability

Some scheduler strategies such as EDF and RM, are optimal only under certain conditions (detailed in Section 2.5) and this optimality only applies to certain constraints (normally task deadline). Other constraints (such as jitter, distance, latency, precedence, and exclusion) are not usually considered.

Table 6-12 shows an example of a set of tasks that explain the above idea. It is assumed that Task A is required to precede Task B, and the minimum distance between them must be at least 0.1 *ms*. Looking only at the task deadlines and scheduling these tasks according to a schedule like EDF will result in scheduling Task B directly after Task A finishes its executions, as it has the nearest deadline. This will satisfy all task deadlines. But, on the other hand, this will violate the distance constraint, as shown in Figure 6-10. To meet this constraint, while at the same time satisfying the deadline constraints, Task C should follow Task A, and at last Task B should start, as shown in Figure 6-11.

Table 6-12 Task specifications for a system where an inappropriate task ordering affects schedulability.

Task	WCET (ms)	Deadline (ms)	Period (ms)	Offset (ticks)
A	0.4	0.5	2	0
B	0.3	1.0	2	0
C	0.2	2.0	2	0

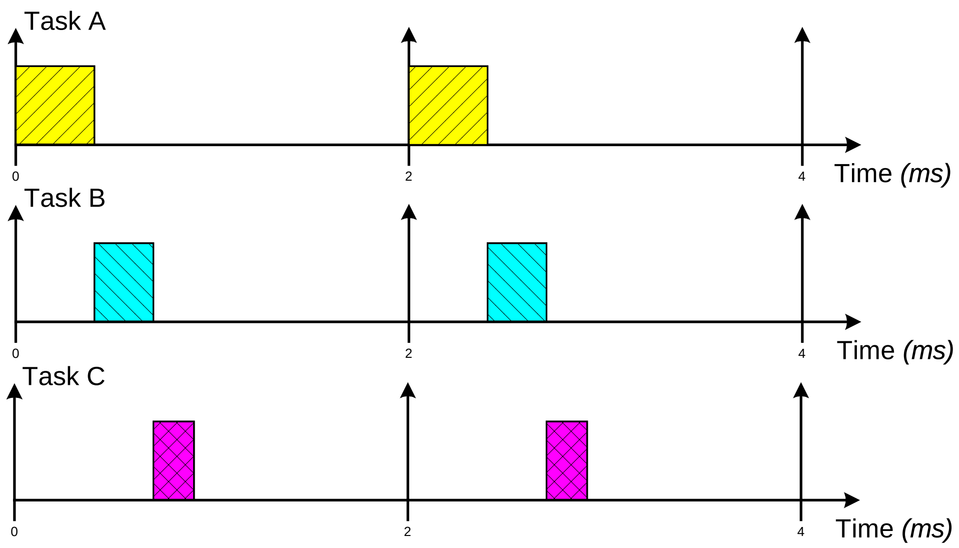


Figure 6-10 Inappropriate task ordering for tasks shown in Table 6-12.

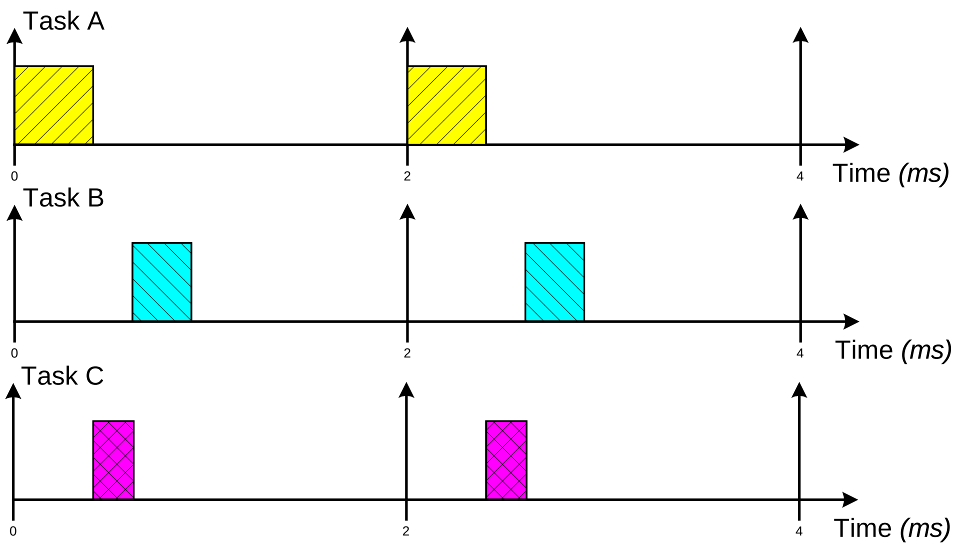


Figure 6-11 Appropriate task ordering for tasks shown in Table 6-12.

6.5.2 Effects of task order on jitter

If a given schedule does maintain task constraints (such as precedence, distance, and latency) it will still have to ensure that the resulting task jitter is within the permitted limits.

For an example it is assumed that, for the set of tasks shown in Table 6-13, Task C is a jitter-sensitive task. Scheduling these tasks with TTC and using a tick interval of 1 *ms* along with the offsets shown in the table will ensure that all deadlines will be met. On the other hand, using the shown task order may not satisfy the jitter constraints of the jitter-sensitive task, Task C, as shown in Figure 6-12. However scheduling the tasks according to the task order shown in Figure 6-13 will maintain the jitter level of the jitter-sensitive task.

Table 6-13 Task specifications for a system where an inappropriate task ordering affects jitter.

Task	WCET (ms)	Deadline (ms)	Period (ms)	Offset (ticks)
A	2	10	10	0
B	3	20	20	0
C	4	30	30	0

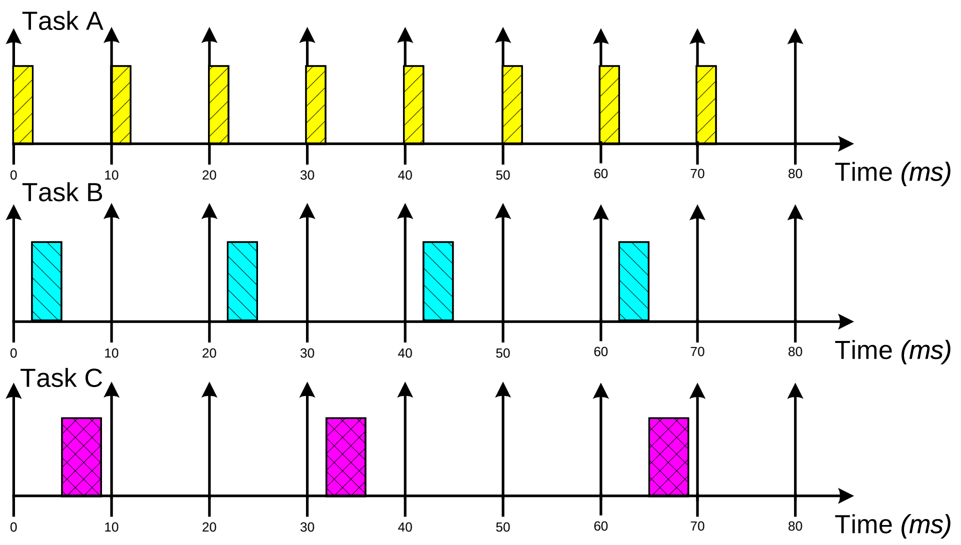


Figure 6-12 Possible schedule for tasks shown in Table 6-13.

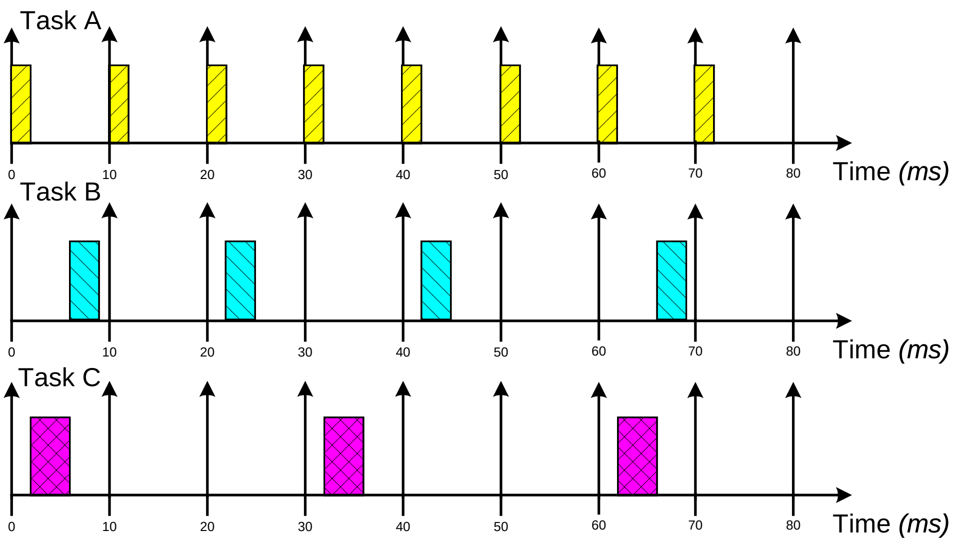


Figure 6-13 Jitter-aware schedule for tasks shown in Table 6-13.

6.6 Discussion

The work presented in this chapter explored the impact of inappropriate choice of task parameters and / or scheduler parameters (such as task offset, task order and tick interval).

It has been shown that inappropriate configurations of task offset may cause missed deadlines and/or increase the task jitter. Similar effects may be caused by inappropriate choice of task order. The analysis also concluded that choosing the scheduler tick interval has considerable effects on system power consumption in addition to the above mentioned effects caused by misconfiguration of task offset and task order. The analysis suggests that using the longest possible tick interval, whenever possible, can help in reducing the power consumption.

6.7 Conclusions

The work presented in this chapter began by reviewing the main functionality of TTC and TTH schedulers. Then it discussed the effect of inappropriately choosing task / scheduler parameters (mainly task offset, task order, and tick interval) when using such schedulers.

It has been shown in previous chapters that testing the task schedulability, and determining appropriate values of scheduler and task parameters, is a challenging and time consuming process.

The following chapter will introduce a new scheduling algorithm which helps to automate the process of selecting these parameters.

Chapter 7
TTSA1 algorithm

This chapter describes a novel two-stage search technique which is intended to support the selection and configuration of schedulers for use with resource-constrained embedded systems[7].

7.1 TTSA1 for automatically choosing/configuring scheduler/task parameters

The previous chapter considered the need for tools which will help to automate the process of developing TTC and TTH schedules. This chapter presents and assesses a novel algorithm which addresses this need. For ease of reference, this algorithm will be called "time-triggered scheduling algorithm 1" (TTSA1). TTSA1 is described in this section.

The overall goal of the TTSA1 algorithm is to identify a scheduler configuration which will ensure that: (i) all task constraints are met; (ii) CPU power consumption is "as low as possible"; (iii) a fully co-operative scheduler architecture is employed whenever possible.

7.1.1 Overview

The flow charts shown in Figure 7-1 and Figure 7-2 describe TTSA1. The input to TTSA1 is a list of task specifications and constraints. The algorithm tests the schedulability of the given task set and configures the scheduler and task parameter (task order, task offset, and tick interval), in case one found. This is done in two stages.

In the first stage the algorithm checks that the total task utilisation, u, does not exceeds the processor capacity ($u<1$). The algorithm takes the scheduling overhead into account in calculating u. A details description of an easy and efficient way of measuring this

[7] Parts of this chapter have been published previously in Gendy and Pont (2008a)

overhead is given in Section 7.1.7. Then the TTSA1 algorithm tries to find a feasible schedule for the task set using the TTC scheduler (which is non-pre-emptive) according to five different strategies as follows:

- According to their deadlines (shortest deadline first). This is "TTSA1-DM" and is based on a "deadline monotonic" scheduling algorithm (Leung and Whitehead, 1982).
- According to their slack – or laxity – time (least laxity first). This is "TTSA1-LLF" and is based on a "least laxity first" scheduling algorithm (Leung, 1989).
- According to their periods (shortest period first). This "TTSA1-RM" is related to a rate monotonic scheduling strategy (Liu and Layland, 1973).
- According to their WCET (shortest WCET first). This is referred here as "TTSA1-SJF" and is related to a "shortest job first" scheduling strategy (Stankovic and Ramamritham, 1987).
- According to their upper bound of jitter (shortest jitter first). This is referred here as "TTSA1-Jitter"

If the task set cannot be scheduled in the first stage the process is repeated using the TTH scheduler

To achieve this result, TTSA1 begins by sorting the tasks according to two criteria:
a) task precedence, b) scheduling criterion (such as TTSA1-DM), trying one at a time. It is then assumed that the first task will run with zero offset and the algorithm tries to find a suitable offset for the second task, using the longest possible tick interval. If such an offset is identified (and the constraints of both tasks are met), a third task is added to the system and the process is repeated. The algorithm continues in this way until all tasks have been scheduled (if this proves possible), as shown in Figure 7-2.

This search process is not exhaustive, and might be described as a "best characteristics first" approach: for example, the algorithm starts with a long tick interval (which is known to reduce power consumption: this is discussed in Section 6.4.3) and the tick interval is gradually reduced until the timing needs of the application are matched (if ever). The algorithm proceeds iteratively, stopping the search when it has identified the first workable solution. It assumes that - because it has begun the search with "best characteristics" - any schedule identified will represent a good (but not necessarily completely optimal) solution.

The output from the algorithm depends on the results of each schedulability test, as follows:

- If all the tasks are schedulable, a suitable tick interval is calculated, along with the task order and the required offset value for each task.
- If the tasks cannot all be scheduled, a list of the schedulable tasks is generated.

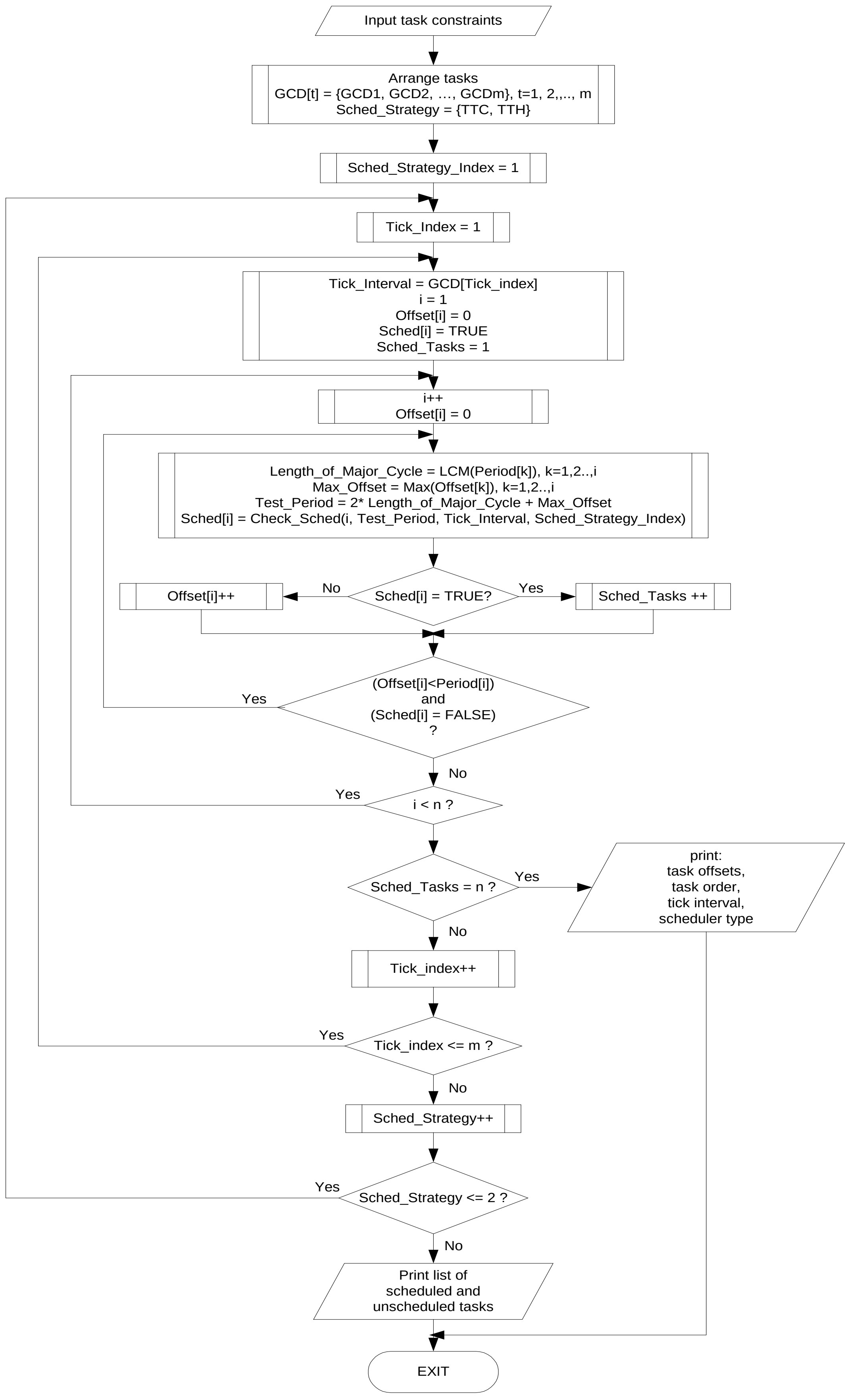

Input task constraints
Arrange tasks
GCD[t] = {GCD1, GCD2, …, GCDm}, t=1, 2,,.., m
Sched_Strategy = {TTC, TTH}
Sched_Strategy_Index = 1
Tick_Index = 1
Tick_Interval = GCD[Tick_index]
i = 1
Offset[i] = 0
Sched[i] = TRUE
Sched_Tasks = 1
i++
Offset[i] = 0
Length_of_Major_Cycle = LCM(Period[k]), k=1,2..,i
Max_Offset = Max(Offset[k]), k=1,2..,i
Test_Period = 2* Length_of_Major_Cycle + Max_Offset
Sched[i] = Check_Sched(i, Test_Period, Tick_Interval, Sched_Strategy_Index)
Offset[i]++
No
Sched[i] = TRUE?
Yes
Sched_Tasks ++
Yes
(Offset[i]<Period[i]) and (Sched[i] = FALSE) ?
No
Yes
i < n ?
Sched_Tasks = n ?
Yes
print: task offsets, task order, tick interval, scheduler type
No
Tick_index++
Yes
Tick_index <= m ?
No
Sched_Strategy++
Yes
Sched_Strategy <= 2 ?
No
Print list of scheduled and unscheduled tasks
EXIT

Figure 7-1 Flow chart for the TTSA1 algorithm.

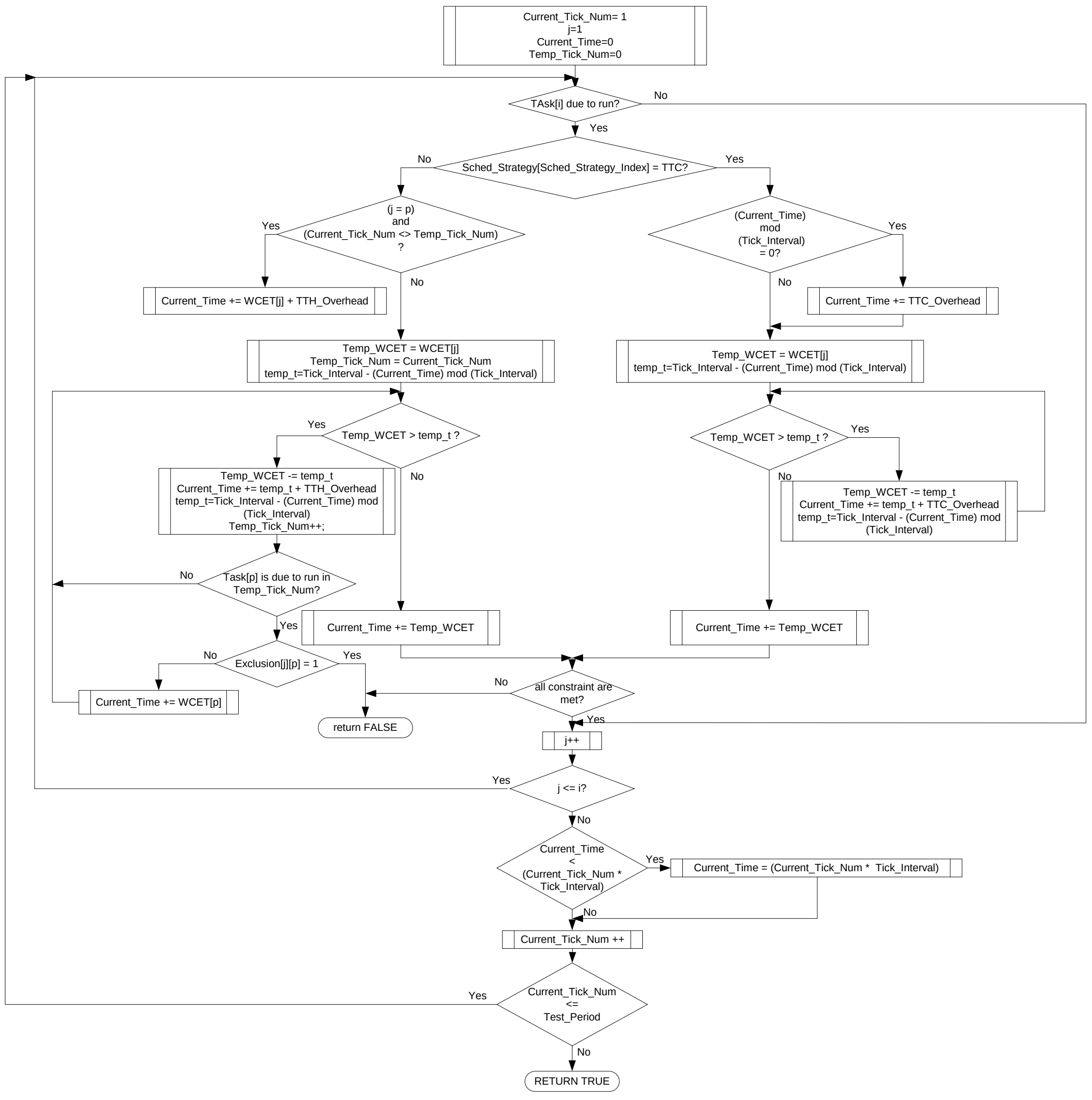

Current_Tick_Num= 1
j=1
Current_Time=0
Temp_Tick_Num=0
TAsk[i] due to run?
No
Yes
Sched_Strategy[Sched_Strategy_Index] = TTC?
(j = p) and (Current_Tick_Num <> Temp_Tick_Num) ?
(Current_Time) mod (Tick_Interval) = 0?
Current_Time += WCET[j] + TTH_Overhead
Current_Time += TTC_Overhead
Temp_WCET = WCET[j]
Temp_Tick_Num = Current_Tick_Num
temp_t=Tick_Interval - (Current_Time) mod (Tick_Interval)
Temp_WCET = WCET[j]
temp_t=Tick_Interval - (Current_Time) mod (Tick_Interval)
Temp_WCET > temp_t ?
Temp_WCET -= temp_t
Current_Time += temp_t + TTH_Overhead
temp_t=Tick_Interval - (Current_Time) mod (Tick_Interval)
Temp_Tick_Num++;
Temp_WCET -= temp_t
Current_Time += temp_t + TTC_Overhead
temp_t=Tick_Interval - (Current_Time) mod (Tick_Interval)
Task[p] is due to run in Temp_Tick_Num?
Current_Time += Temp_WCET
Exclusion[j][p] = 1
Current_Time += WCET[p]
return FALSE
all constraint are met?
j++
j <= i?
Current_Time < (Current_Tick_Num * Tick_Interval)
Current_Time = (Current_Tick_Num * Tick_Interval)
Current_Tick_Num ++
Current_Tick_Num <= Test_Period
RETURN TRUE

Figure 7-2 flow chart for the Check_Sched() function of TTSA1 algorithm.

7.1.2 Tick interval

It was previously shown that an inappropriate choice of tick interval may mean that a given task set cannot be scheduled at all. Also where the parameter set does ensure that all tasks are scheduled, inappropriate choice of tick interval may still lead – for example – to increased system power consumption.

To find the most suitable (that is, longest possible) tick interval, the algorithm checks the schedulability using all the common divisors of the task periods, starting with the Greatest Common Divisor (GCD) for the best results in power reduction. The algorithm stops at the largest possible tick interval with which all the tasks meet their temporal constraints (if such an interval exists).

7.1.3 Offset

As discussed in Section 6.3, the choice of offset can have a significant impact on both the task schedulability and the levels of task jitter in the system. The algorithm tries to choose an appropriate offset for a given set of tasks.

It is assumed (for the purposes of this work) that the offset of a task can take any value between zero and its period, assuming that values of offset and period are expressed in ticks.

7.1.4 Test period

Choosing a suitable offset for each task may require that the schedule is tested (using different offset combinations) over a period of time long enough to determine that all the tasks will meet their deadlines (or not). Since all tasks are periodic, the schedulability needed to be tested over the "major cycle" (a period of time equal to the Least Common Multiple – LCM – of the task periods: e.g. Section 2.4).

In addition, since each task may have a different offset, the full schedule will not necessarily begin immediately; instead, the algorithm must therefore test the schedule for one complete cycle, measured from the time that the last task to be added to the schedule is executed for the first time. Finally, it may be necessary to consider the task behaviour at the boundary between the end of one (major) cycle and the start of the next.

As a result, for a given tick interval and set of offsets, the testing period used in this work is represented by Equation 7-1; the units here are "ticks" (Leung and Merrill, 1980; Leung and Whitehead, 1982; Leung, 1989).

$$\text{Test\_period} = 2 * \text{Length\_of\_Major\_cycle} + \text{Maximum\_offset} \qquad \textbf{Equation 7-1}$$

As shown in the previous section, offset of a task is in the range zero and its period, so for a set of *n* tasks the longest test period can be calculated form Equation 7-2.

$$\text{Test\_period} = 2 * \text{LCM}(P[1], P[2], .., P[n]) + \text{Max}(P[1], P[2], \ldots, P[n]) \qquad \textbf{Equation 7-2}$$

7.1.5 Task starting time

At any time, task Task[i] is considered to be due to run at tick 'Tick_Num' if the condition represented by Equation 7-3 is true.

$$(\text{Tick\_Num} - \text{Offset}[i]) \bmod P[i] = 0 \qquad \textbf{Equation 7-3}$$

7.1.6 Deadline checking

Assuming that a specific instant of task Task[i], which has deadline D[i] that is less than or equal to P[i], begins its execution at time 'Starting_Time' and finishes its execution at time 'Finishing_Time' this task is considered to have met its deadline if the condition in Equation 7-4 is satisfied for all its segments:

$$(\text{Finishing\_Time} - \text{Starting\_Time}) <= D[i] \qquad \textbf{Equation 7-4}$$

7.1.7 Taking scheduler overheads into account

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

Previous work has been conducted in this area, for example Sandström et al (1998) handle the interrupt overhead in an efficient non-pessimistic way. The current work introduces an alternative way of representing the overall scheduler overhead for a given number of tasks. It assumes that the scheduler overhead can be represented by adding a dummy task to the set of tasks to be scheduled. This additional task is included in the schedule calculations at every tick and has a WCET equal to the actual scheduler overhead. This effect is shown in the *Check_Sched*() function (**Error! Reference source not found.**).

Of course, it is necessary to determine the WCET value for this "overhead" task. This value can not be predicted (without conducting an extensive – and expensive – modelling process). Therefore it is noted that the maximum scheduling overhead will occur when all the tasks run in the same tick (if ever).

Assuming that we have *n* tasks and that the scheduler enters "sleep" mode after running all the ready tasks in each tick (if there is time left), then the scheduling overhead is given by Equation 7-5.

$$\text{overhead} = \text{Tick\_Interval} - (\text{time\_spent\_in\_sleep\_mode} + \sum_{i=1}^{n} \text{WCET}[i]) \qquad \textbf{Equation 7-5}$$

The overhead can be determined empirically, using a scheduler with the same number of "dummy" (empty) tasks that will be employed in the final system. In this case, the last term in Equation 7-5, $\sum_{i=1}^{n} \text{WCET}[i]$, can be assumed to be 0, and a single set of measurements will be required for a given hardware platform, regardless of the particular system being implemented

Determining the overhead in this way may seem to be unduly pessimistic for a static schedule. However, this measure of the maximum scheduler load is easily obtained (one single measurement, rather than having to make numerous measurements as different schedules are tested). In addition making a precise measurement of this load is – in practice – not straightforward. Therefore it has been chosen to accept a slight risk that the scheduling decision made will be altered by the inaccuracy of this overhead measurement (indeed, it is assumed that any loss of accuracy that results from this approach is likely to be smaller than the error which results from WCET approximations for the tasks: as discussed in Chapter 4).

The value of *time_spent_in_sleep_mode* can be determined either through the use of a hardware simulator or by making direct measurements from the hardware.

The scheduling overhead for the experiments which are carried out in this thesis is measured as follows. The scheduler is adjusted so that it will set a spare I/O pin to "high" at the start of the interrupt service routine (ISR) of the tick interrupt. This pin is then set to "low" directly before the scheduler calls the sleep function, after all tasks are complete.

The duration of the time in which the pin is high in every tick is measured using appropriate external hardware.

By using three dummy tasks, the above measured time approximates the scheduling overhead (which is shown in Equation 7-5). The above measured time does not take into account the times taken to change out of the sleep mode, save registers values, or call the ISR. Also the code in the scheduler that does not run in all conditions may not have been taken into account. Therefore, an additional 20% is added to the measured value, a figure that is in line with previous studies (Vallerio and Jha, 2003).

7.2 Evaluating the TTSA1 algorithm

The TTSA1 algorithm is evaluated in this section. The "branch and bound" algorithm (BaB) was chosen previously as a benchmark to test the effectives of other heuristic algorithms (Cucu and Sorel, 2004). The same algorithm is adapted here to evaluate the effectiveness of the TTSA1 algorithm.

7.2.1 Algorithm complexity

The algorithm complexity can be calculated by considering a set of *n* independent tasks, Task[1], Task[2], …, Task[*n*], with periods P[1], P[2], .., P[*n*], respectively. As previously discussed, the offset O[*i*] of task Task[*i*] is assumed to take any value from zero to P[*i*]. Choosing a suitable set of offsets may require testing schedulability over the period defined by Equation 7-1.

Using the BaB search algorithm a partial schedule is constructed by adding tasks one by one to the system (trying all possible offsets of this task). A branch is terminated if the constraints of any added task, or the task under test, are violated. Ignoring the possible task offsets, in the worst-case this will require testing *n* paths each of length *n*!; this has a complexity of O(*n*.*n*!) which is "*computationally intractable and cannot be used in practical systems when the number of tasks is high*"(Buttazzo, 2005a). In this case the longest testing period will therefore be given by Equation 7-6 and Equation 7-7.

$$\text{longest testing period} = (\text{number of offsets combinations}) * (\text{number of possible execution orders}) * (\text{test period}) \qquad \textbf{Equation 7-6}$$

$$\sum_{i=1}^{n}\left\{\left(\prod_{i=1}^{n}\llbracket P[i]\right) * \ (i!) * [Max(O[1],O[2],\dots,O[i]) + \ 2 * LCM(P[1],P[2],\dots P[i])]\rrbracket\right\} \qquad \textbf{Equation 7-7}$$

This problem has an order of complexity O(t^n.n!), where t is the period (in ticks).

By contrast, the TTSA1 algorithm tries only a subset of the possible offset combinations. In this case, the longest testing period will be given by Equation 7-8.

$$\sum_{2}^{n}(P[i] * (Max(O[1],O[2],\dots,O[i]) + 2 * LCM(P[1],P[2],\dots P[i]))) \qquad \textbf{Equation 7-8}$$

The complexity of this algorithm is $O((n\text{-}1)t)$ or approximately $O(n.t)$.

It should be noted that summation in Equation 7-8 starts from index 2, (rather than index 1). This is because the TTSA1 algorithm assumes that, after sorting the set of tasks, the first task is added to the system with offset 0. The offsets of subsequent tasks are determined at the time they are added to the system (one by one). Once an offset for a given task is identified, this is "fixed".

It is also noteworthy that these calculations ignore the effort required to determine the scheduler overhead (for both the BaB and TTSA1 calculations).

7.2.2 Algorithm performance

An empirical test of the performance of the TTSA1 algorithm was carried out. The procedure and results obtained by applying the algorithm to a set of interdependent tasks are detailed in this section.

7.2.2.1 Method

The schedulability of the task sets was assessed using the BaB search. The results were then compared with those obtained using the TTSA1 algorithm.

The chosen hardware platform was an NXP (formerly Philips) LPC2129 microcontroller running on a small evaluation board (Philips, 2004a). The LPC2129 is based on an ARM7TDMI core and is typical of modern (low cost) embedded processors. The tests were conducted as follows:

- The measurements of scheduler load were carried out using the NXP board.
- The BaB and the TTSA1 algorithm schedulability tests were carried out using a simple (custom) schedule simulator, running on a desktop PC (making use of the load information obtained from the NXP board).

7.2.2.2 Dataset used

To explore the effectiveness of this algorithm, 1000 sets of tasks were pseudo randomly generated. Each set consisted of 3, 4 and 5 tasks. Task characteristics and constraints were generated according to the criteria specified in the following subsections.

7.2.2.3 Task characteristics

Task WCET, deadline and period were pseudo randomly generated according to the following inequalities:

$$0 < \text{WCET}(i) \leq 1000\ \mu\text{s} \qquad \textbf{Equation 7-9}$$

$$\text{WCET}(i) < P(i) \leq 10000\ \mu\text{s} \qquad \textbf{Equation 7-10}$$

$$\text{WCET}(i) \leq D(i) \leq P(i) \qquad \textbf{Equation 7-11}$$

In order to simplify the calculations, task periods were pseudo randomly generated at multiples of 1 *ms* (constrained by Equation 7-10).

7.2.2.4 Task constraints

Task constraints of precedence, exclusion, distance, latency, and upper bound of jitter were also pseudo randomly generated and were in line with the findings from previous studies (e.g. Xu (1993) and Sandström and Norström (2002)).

- Jitter:
  In the experiment discussed in the present chapter, the upper bound of task jitter is pseudo randomly generated according to Equation 7-12.

$$0 \le \text{Jitter} \le P(i)$$ **Equation 7-12**

- Precedence:

  If it is required that Task A precedes Task B, then, in any tick, Task B is allowed to start its execution only after Task A completes its execution (e.g. Xu and Parnas (1990)).

  In the current study, the precedence relation between any two tasks, Task A and Task B, is pseudo randomly generated if

$$P(A) = P(B)$$ **Equation 7-13**

and

$$P(A) \ge (\text{WCET}(A) + \text{WCET}(B))$$ **Equation 7-14**

- Distance:

  In the current study the precedence relation between two tasks, Task A and Task B, was pseudo randomly generated according to Equation 7-15.

$$0 \le \text{Distance}(A, B) \le P(A) - (\text{WCET}(A) + \text{WCET}(B))$$ **Equation 7-15**

- Latency

  In the current study the latency relation between two tasks, Task A and Task B, was pseudo randomly generated as follows:

  If there are no distances constraint between Task A and Task B then:

$$(\text{WCET}(A) + \text{WCET}(B)) \le \text{Latency}(A, B) \le \text{Max}(P(A), P(B))$$ **Equation 7-16**

  otherwise:

$$(\text{WCET}(A) + \text{WCET}(B) + \text{Distance}(A,B)) \le \text{Latency}(A, B) \le P(A)$$ **Equation 7-17**

Table 7-1 shows an example of a set of 3 tasks generated according to the above constraints.

Table 7-1 Sample of task specifications and constraints (set of 3 tasks)

Task	**WCET (*μs*)**	**Deadline (*μs*)**	**Period (*μs*)**	**Jitter (*μs*)**	**Exclusion**	**Precedence**	**Distance (*μs*)**	**Latency (*μs*)**
A	496	3964	4000	1618	Task A Excludes Task C	Task A Precedes Task C	Distance between Task A & Task C is 3335	Latency Between Task A & Task C is 3921
B	828	4711	10000	9488				
C	64	3673	4000	67				

7.2.2.5 Extending the basic algorithm

Variations on the original TTSA1 algorithm were also investigated in this trial. In the original algorithm (henceforth referred to as "TTSA1-DM"), the tasks are added to the schedule according to their deadline, the task with the shortest deadline is added first.

Variations on this algorithm are also explored so that tasks were added:

- According to their slack – or laxity – time (least laxity first). This is "TTSA1-LLF" and is based on a "least laxity first" scheduling algorithm (Leung, 1989).
- According to their periods (shortest period first). This "TTSA1-RM" is related to a rate monotonic scheduling strategy (Liu and Layland, 1973).
- According to their WCET (shortest WCET first). This is referred here as "TTSA1-SJF" and is related to a "shortest job first" scheduling strategy (Stankovic and Ramamritham, 1987).
- According to their upper bound of jitter (shortest jitter first). This is referred here as "TTSA1-Jitter"

7.2.2.6 Results (small task sets)

The numbers of identified task sets that were found to be scheduled using the TTSA1 algorithm and BaB are shown in Figure 7-3 to Figure 7-5. The results obtained by combining the (unique) results from TTSA1-DM, TTSA1-LLF, TTSA1-Jitter, TTSA1-

RM, and TTSA1-SJF are shown in these figures as TTSA1-ALL. The number of trials until each of the two algorithms identified the set of tasks as scheduled/unscheduled and the total time is also shown in Table 7-2.

From the results obtained it was noted that:

- For both the TTC and TTH schedulers the results obtained from TTSA1 (when overheads are taken into account) are found to be a subset of the complete list of valid schedules identified by the BaB search. In addition, although TTSA1 tests the schedulability using a subset of all the possible offset combinations, it produces results which are similar to those obtained with the BaB method.

- The criteria used for adding the tasks have an impact on the schedulability of the set (different criteria may give different results).

- Combining results from the variations of TTSA1 together gives results which are closer to those obtained from the BaB search while requiring a much lower number of trials, and hence less time (Section 7.2.1).

7.2.2.7 Results (large task set)

The results shown in Figure 7-3 to Figure 7-5 consider a maximum of 5 tasks. This is not an unrealistic number for the resource-constrained systems that are concerned with in this thesis. However, this task set does not fully test the algorithm. In order to explore the performance of TTSA1 on larger problems, 1000 new data sets were created. Each data set consisted of 50 tasks, each with a maximum execution time of 1 *ms* and maximum period of 100 *ms*. The task sets were randomly created according to the constraints described previously. To reduce the length of the major cycle, task periods were randomly generated as a multiple of 10 *ms*. The results from this test are shown in Figure 7-6. It took approximately 10 *s* to complete the schedulability test for one set of 50 tasks using TTSA1-DM, and a total of approximately 50 *s* to complete the test for TTSA1-All. It was not possible to complete this search using a BaB approach as this would have required performing a huge number of trials.

Table 7-2 Number of trial and the total time

	3-tasks sets				4-tasks sets				5-tasks sets			
	TTC		TTH		TTC		TTH		TTC		TTH	
	TTSA1	BaB	TTSA1	BaB	TTSA1	BaB	TTSA1	BaB	TTSA1	BaB	TTSA1	BaB
Minimum number of trials	2	2	2	2	3	3	3	3	4	4	4	4
Maximum number of trials	85	2966	75	2966	125	33571	64	35072	170	1585571	87	879901
Average number of trials	16.3	162.0	11.4	159.8	31.7	2561.7	17.4	2544.2	59.6	56283.7	23.7	46575.6
Total number of trials	16285	161962	11360	159823	31655	2561690	17360	2544241	59596	5.6E+07	23652	4.7E+07
Total time (s)	1	2	1.5	3	1.5	88	2	184	3	3091	3.5	4924

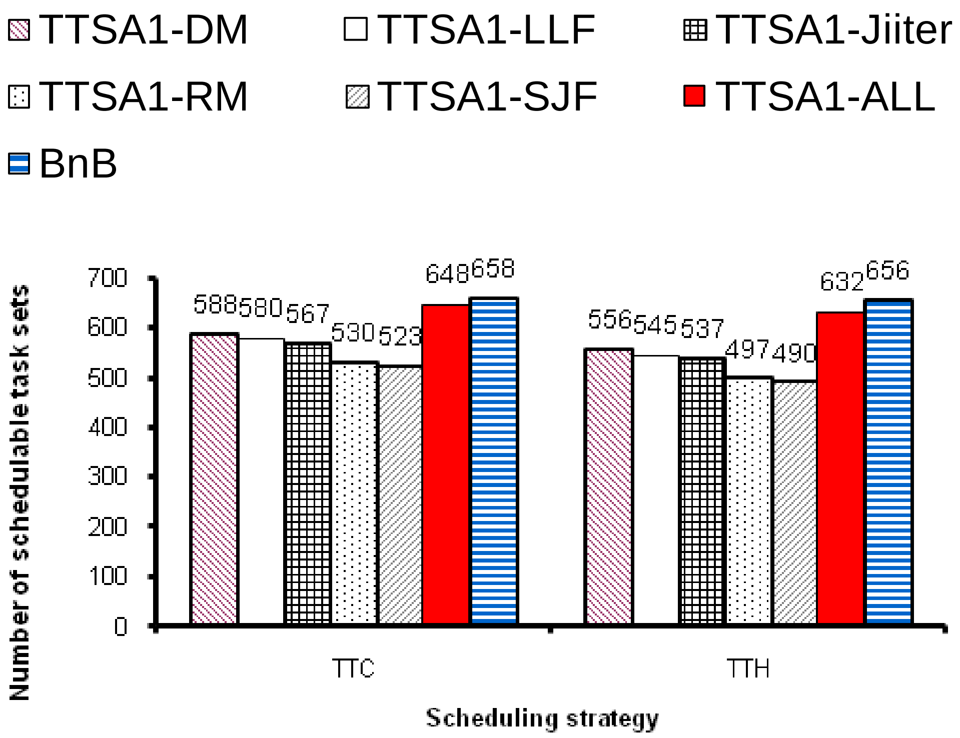


Figure 7-3 Number of scheduled task sets (3 interdependent tasks in each set).

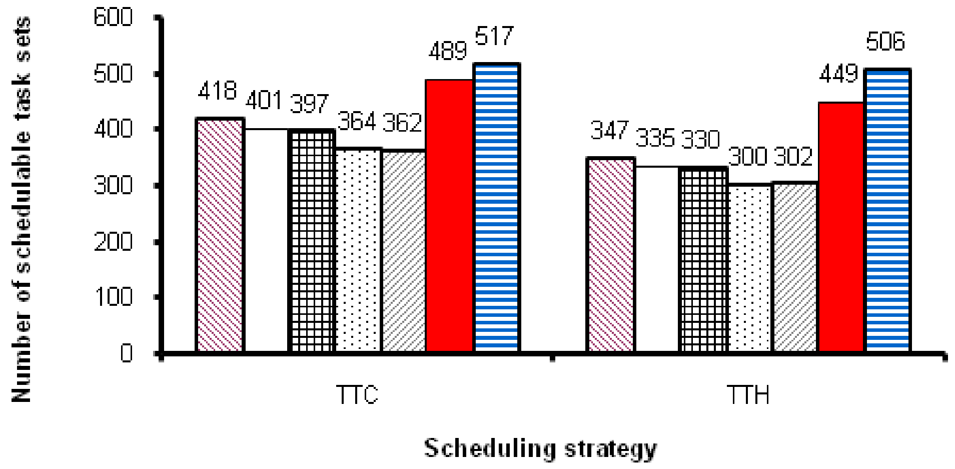


Figure 7-4 Number of scheduled task sets (4 interdependent tasks in each set).

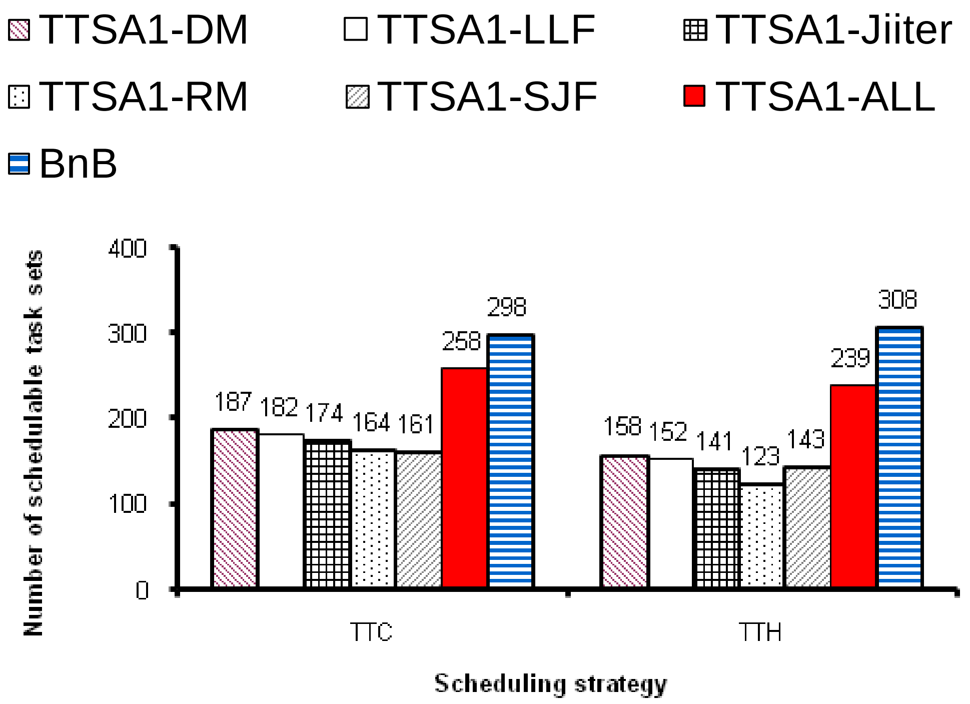


Figure 7-5 Number of scheduled task sets (5 interdependent tasks in each set).

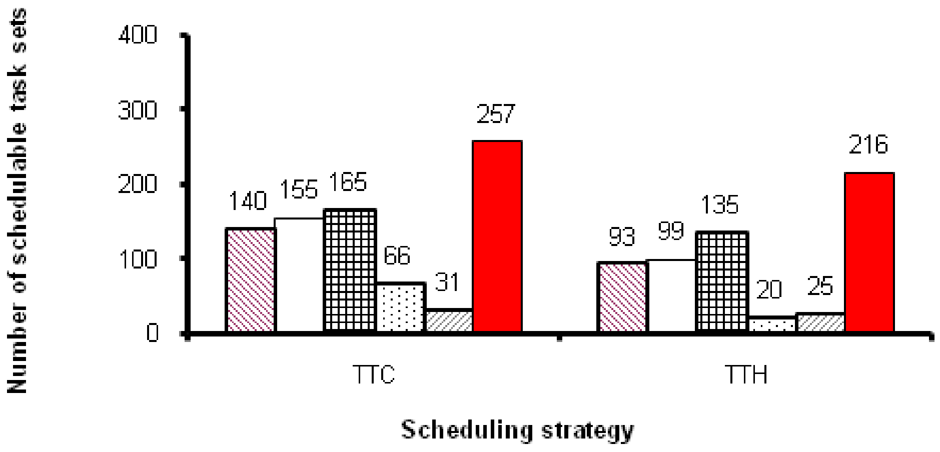


Figure 7-6 Number of scheduled task sets (50 interdependent tasks in each set).

7.3 Discussion

The work presented in this chapter introduced a scheduling algorithm (TTSA1) which helps to automate the process of determining the parameters required to schedule a given set of tasks in a resource-constrained embedded system employing a TTC or TTH architecture.

It has been shown in the literature that commonly used scheduling techniques (such as RM, DM, etc) are optimal in satisfying one of the task constraints (usually the task deadline). This can be guaranteed only in special cases, for example RM assumes independent tasks which have deadlines equal to their periods. On the other hand, finding just a workable schedule for dependent tasks with shared resources is known to be a challenging process.

Therefore the TTSA1 scheduling algorithm presented in this chapter tries to use heuristic search guided by the most commonly used scheduling techniques (such as DM, RM, LLF, and SJF), in addition to adding the jitter-based scheduling, to find a workable scheduler.

The effectiveness of the TTSA1 scheduling algorithm has been evaluated by applying it on 1000 sets of tasks. The results were compared with those obtained from applying a BaB search which has been taken previously as a benchmark by other researchers (Cucu and Sorel, 2004). It is believed that the aim has been achieved through the use of this algorithm which – while it does not perform an exhaustive search – does provide results close to those obtained in the BaB search, in a fraction of time. While searching for a workable scheduler the proposed scheduling algorithm ensures that the CPU power consumption is "as low as possible" (by choosing the longest possible tick interval), and that task constraints are met (by adjusting the tasks’ offsets, tick interval, and task orders).

Another empirical study has been done to evaluate the effectiveness of the above algorithm. In these experiments the algorithm was implemented and ported to a separate microcontroller called scheduler agent (SA). The aim of this agent was to continue measuring the execution times of the tasks running in the main target hardware, or main processor (MP), for a certain period of time at systems power up. These values were used by the algorithm to fine tune the task scheduler in the main processor. After the fine tuning process is completed the SA continue monitoring the MP and take appropriate action, such as resetting the system, in case of errors. More details about this case study are described in Appendix B.

7.4 Conclusions

This chapter presented a scheduling algorithm (TTSA1) which helps in overcoming the fragility problem encountered in TT designs (based on TTC or TTH architecture) by automating the process of selecting an appropriate scheduler and configuring the required parameters.

The effectiveness of the above mentioned algorithm is tested, and compared with that of the BaB algorithm, by empirical studies.

The following chapter will introduce an enhanced version of this algorithm.

Chapter 8
TTSA2 algorithm

In the previous chapter a novel scheduling algorithm ("TTSA1") was presented *that can be used to automate the process of selecting an appropriate scheduler and configuring the task and scheduler parameters. This algorithm focuses on time-triggered embedded systems (based on TTC and TTH) which employ a single processor.*

This chapter describes a modified version of the TTSA1 algorithm ('TTSA2"). TTSA2 employs task segmentation to increase the number of task sets which can be scheduled[8].

8.1 Problems with TTSA1 algorithm

Despite its attractive features, in some cases the TTSA1 algorithm fails to find a suitable schedule for a set of tasks. For example assume that for a given set all tasks have deadlines equal to their periods. Assume also that this set includes two short tasks (Task S1 and Tasks S2), and at least one long task (Task L).

The TTSA1 algorithm fails to find a suitable schedule for this set if:

$$\text{Deadline (S1)} < \text{WCET (S1)} + \text{WCET (L)} \qquad \textbf{Equation 8-1}$$

and

$$\text{Deadline (S2)} < \text{WCET (S2)} + \text{WCET (L)} \qquad \textbf{Equation 8-2}$$

For example Task B and / or Task C shown in Table 8-1 will miss their deadlines every time Task A runs if these three tasks are scheduled using TTC / TTH. To overcome this situation, while still using a TTC / TTH architecture, long task(s) can be divided into multiple short tasks (Baker and Shaw, 1988; Locke, 1992; Pont, 2001): for example Task A can be divided into two segments, Segment SA_1 and Segment SA_2, as shown in Table 8-2.

[8] Parts of this chapter have been published previously in Gendy and Pont (2008b)

Table 8-1 Task specifications for a task set that cannot be scheduled with TTC/TTH.

Task	WCET (ms)	Deadline (ms)	Period (ms)
A	10	50	50
B	1	10	10
C	1	10	10

Table 8-2 Task specifications for a task set that can be scheduled with TTC/TTH.

Task	WCET (ms)	Deadline (ms)	Period (ms)
SA_1	5	45	50
SA_2	5	50	50
SB_1	1	10	10
SC_1	1	10	10

8.2 TTSA2 algorithm

As previously indicated, testing the schedulability of a set of tasks and finding a suitable scheduler for them (if any) is an NP-hard problem. The problem becomes more complex if some parts of some tasks are required to exclude parts of other tasks in the set. For example, it may be that Segment SA_2 in Task A excludes Segment SB_3 and Segment SC_2 in Task B and Task C respectively, and Segment SB_1 in Task B excludes Segment SC_1 in Task C.

In the following subsections the TTSA2 scheduling algorithm, which takes inter-task and inter-segment constraints into account and supports task segmentation to increase schedulability, will be described and assessed.

8.2.1 Overview

It is assumed that the input to TTSA2 is a list of task specifications and constraints, including critical-section boundaries.

The TTSA2 algorithm tests the schedulability of the given task set, first using the TTC scheduler, if possible, otherwise it will try the TTH, considering each task as a single segment. If the task set still cannot be scheduled the process is repeated after dividing one or more long tasks into two or more shorter tasks. The algorithm calculates a suitable tick interval, the task order, the smallest number of task segments along with the required offset value for each task and task segment if all the tasks are schedulable; otherwise a list of the schedulable tasks and task segments is generated.

To achieve this result, TTSA2 begins (like TTSA1) by sorting the tasks according to two criteria: a) task precedence, b) task deadline, laxity, period, WCET, or jitter. It is then assumed that the first task will run as one segment with zero offset and the algorithm tries to find a suitable offset for the second task (in one segment), using the longest possible tick interval (the greatest common divisor of the task periods). If such an offset is identified (and the constraints of both tasks are met), a third task is added to the system and the process is repeated. This process continues until all tasks have been scheduled (if this proves possible). If a schedule cannot be found at any stage the last task added to the design is removed and divided into two segments. After adding the segmentation overhead and updating the segment deadlines (as explained in the following subsections) the search proceeds, (

Figure 8-1).

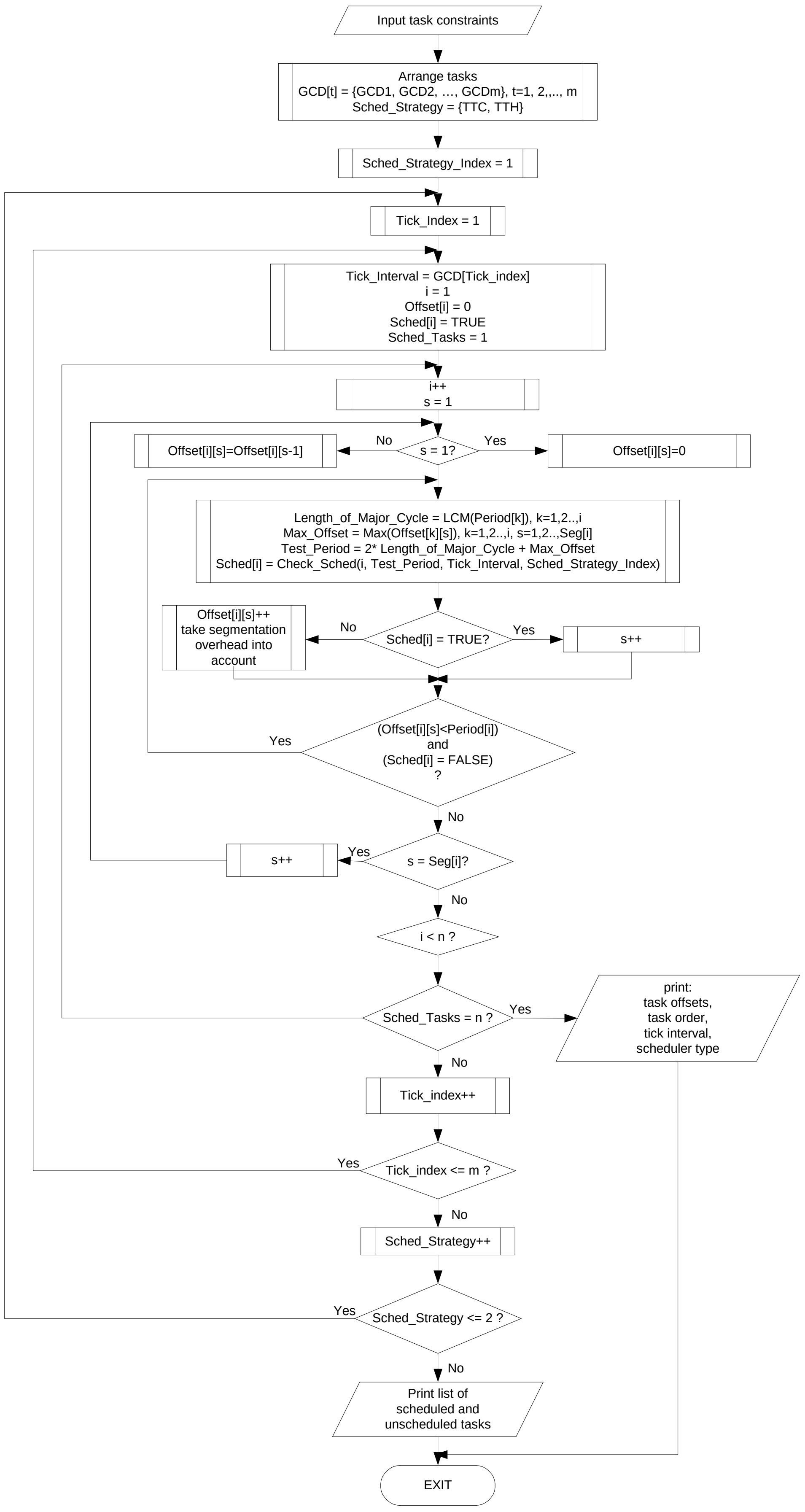

Input task constraints
Arrange tasks
GCD[t] = {GCD1, GCD2, …, GCDm}, t=1, 2,,.., m
Sched_Strategy = {TTC, TTH}
Sched_Strategy_Index = 1
Tick_Index = 1
Tick_Interval = GCD[Tick_index]
i = 1
Offset[i] = 0
Sched[i] = TRUE
Sched_Tasks = 1
i++
s = 1
s = 1?
No
Yes
Offset[i][s]=Offset[i][s-1]
Offset[i][s]=0
Length_of_Major_Cycle = LCM(Period[k]), k=1,2..,i
Max_Offset = Max(Offset[k][s]), k=1,2..,i, s=1,2..,Seg[i]
Test_Period = 2* Length_of_Major_Cycle + Max_Offset
Sched[i] = Check_Sched(i, Test_Period, Tick_Interval, Sched_Strategy_Index)
Sched[i] = TRUE?
No
Yes
Offset[i][s]++
take segmentation overhead into account
s++
(Offset[i][s]<Period[i]) and (Sched[i] = FALSE) ?
Yes
No
s = Seg[i]?
Yes
s++
No
i < n ?
Sched_Tasks = n ?
Yes
print: task offsets, task order, tick interval, scheduler type
No
Tick_index++
Tick_index <= m ?
Yes
No
Sched_Strategy++
Sched_Strategy <= 2 ?
Yes
No
Print list of scheduled and unscheduled tasks
EXIT

Figure 8-1 Pseudo code for the TTSA2 algorithm.

8.2.2 Adjusting the segment deadline

If Task T is divided into *n* segments, ST_1, ST_2.., ST_n, then the TTSA2 algorithm calculates the deadline of each segment as follows:

$$\text{Deadline }(ST_n) = \text{Deadline }(T) \qquad \textbf{Equation 8-3}$$

$$\text{Deadline }(ST_{i-1}) = \text{Deadline }(ST_i) - \text{WCET }(ST_i), \qquad \textbf{Equation 8-4}$$

where $i = n, n -1, n -2\ldots,2$.

The deadline of Segment SA_1 in Table 8-2 is an example of such deadline adjustment.

To be able to divide long tasks into multiple short tasks accurate information about the task WCET and the points at which the task can / cannot be pre-empted must be specified. This can be done using techniques such as the "single path programming paradigm" (Puschner and Burns, 2002b; Puschner and Burns, 2003) or code-balancing techniques presented in Chapter 5.

8.2.3 Adding the segmentation overhead

If a task is divided into two or more segments, the TTSA2 algorithm takes segmentation overhead into account. This overhead represents the time needed to save the context of the current segment and the time needed to restore this context when the following segment becomes ready to run. The time required for saving the context (*Context_Saving_overhead*) may not be the same as that required for loading the context (*Context_Loading_overhead*).

If Task T is divided into *n* segments, ST_1, ST_2.., ST_n, then the TTSA2 algorithm updates the segments WCETs to reflect this overhead, as follows:

$$\text{WCET }(ST_1) = \text{WCET }(ST_1) + \textit{Context\_Saving\_overhead}. \qquad \textbf{Equation 8-5}$$

$$\text{WCET }(ST_i) = \textit{Context\_Loading\_overhead} + \text{WCET }(ST_i) + \textit{Context\_Saving\_overhead}, \qquad \textbf{Equation 8-6}$$

where $i = 2, 3\ldots, n\text{-}1$.

$$\mathrm{WCET}\ (\mathrm{ST}_n) = \mathit{Context\_Loading\_overhead} + \mathrm{WCET}\ (\mathrm{ST}_n). \qquad \textbf{Equation 8-7}$$

8.3 Evaluating the TTSA2 algorithm

In this section the complexity and the effectiveness of the TTSA2 is evaluated. As in Chapter 7, the performance of the TTSA2 is compared with that of the "branch and bound" algorithm (BaB).

8.3.1 Algorithm complexity

Assume we have a set of *n* independent tasks and that each consists of *s* segments. Recalling from the previous chapter, investigating the schedulability of these tasks by means of a BaB algorithm has a complexity of $O(sn!\ (sn))$ which is computationally intractable (Buttazzo, 2005a) .

As noted previously, the offset of each task can be any value in the range [0, Period[, in ticks. In the worst-case the BaB algorithm will take all possible offset combinations (t^n), where *t* is the period, and considering each task as set of *s* segments, each may have a different offset, the complexity will increase to $O(t^{n.s}\ .sn!)$, as shown in Equation 8-8.

$$\sum_{i=1}^{s.n} \lll\!< \left(\prod_{i=1}^{s.n} P[i]\right) * (i!) * [Max(O[1], O[2], \ldots, O[i]) + 2 * LCM(P[1], P[2], \ldots P[i])]\} \qquad \textbf{Equation 8-8}$$

By contrast, the TTSA2 algorithm does not try all paths. In addition, it does not change the task or / and segment offset of a given task once it has been added successfully to the schedule (that is, added without causing violation of the constraints of any of the tasks and segments which have been included in the schedule previously). The complexity of this algorithm is $O(n.s.t)$, as shown in Equation 8-9.

$$\sum_{2}^{s.n} (P[i] * (Max(O[1], O[2], \ldots, O[i]) + 2 * LCM(P[1], P[2], \ldots P[i]))) \qquad \textbf{Equation 8-9}$$

8.3.2 Algorithm performance

An empirical test was carried out to explore the performance of the TTSA2 algorithm. The procedure and the results of this test are discussed in this section.

8.3.2.1 Method

The method which was used to asses the performance of TTSA1 is also used here. The measurements of scheduler overhead were carried out using the NXP (formerly Philips) LPC2129 microcontroller running on a small evaluation board (Philips, 2004a), in the same way previously described in Section 7.1.7 and Section 7.2.2.1. The segmentation overhead was measured simply by using break points on the simulator (the Keil ARM development kit (v3.2)).

8.3.2.2 Dataset used

The dataset used to explore the effectiveness of TTSA2 algorithm was pseudo randomly generated in the same way which is used to generate the dataset employed to test the effectiveness of TTSA1 algorithm. The only difference here is that longer tasks are added to some task sets to test the performance of TTSA2 in systems which include long tasks. To achieve this, WCETs were pseudo randomly generated according to the following criteria:

$$0 < \mathrm{WCET}(i) \leq 2000\ \mu\mathrm{s} \qquad \textbf{Equation 8-10}$$

8.3.2.3 Results (small task sets):

The effectiveness of the TTSA2 algorithm is tested when scheduling small sets of tasks (each containing 3, 4, or 5 tasks) and compared the results with those from the TTSA1 and the BaB algorithms. The results obtained from the BaB algorithm with / without using task segmentation are recorded as BaB1 and BaB2 respectively.

Figure 8-2 to Figure 8-4 show the number of task sets that was found to be schedulable using TTSA1, TTSA2, BaB1, and BaB2. The results obtained by combining the (unique) results from TTSAx-DM, TTSAx-LLF, TTSAx-Jitter, TTSAx-RM, and TTSAx-SJF are shown in these figures as TTSAx-ALL, where x equals 1 or 2 for TTSA1 and TTSA2. Table 8-3 also shows the number of trials until each algorithm identified the set of tasks as schedulable / unschedulable and the total time.

From the results obtained it was noted that:

- TTSA2 found a suitable scheduler for more sets than TTSA1.

- Because TTSA2 tries to find a suitable (TTC or TTH) scheduler using the lowest number of task segments, the results obtained from TTSA1 are found to be a subset of the complete list of valid schedules identified by TTSA2. This means that all the schedulers identified by both TTSA1 and TTSA2 have the same scheduling overhead and power consumption.

- The results obtained from TTSA1 and TTSA2 (when overheads are taken into account) are found to be a subset of the complete list of valid schedules identified by BaB1 and BaB2, respectively. In addition, although TTSA1 and TTSA2 test the schedulability using a subset of all the possible offset combinations, they produce results which are similar to those obtained with the BaB1 and BaB2 methods.

- The criteria used for adding the tasks to TTSA1 and TTSA2 have an impact on the schedulability of the set (different criteria may give different results).

- Combining results from the variations of TTSA1 and variations of TTSA2 together gives results which are closer to those obtained from the BaB1 and BaB2 respectively, while requiring a much lower number of trials, and hence less time (as shown in Table 8-3).

8.3.2.4 Results (large task set):

In order to explore the performance of TTSA2 on larger problems, 1000 new data sets were created, as in the case of evaluating the TTSA1 algorithm. Each data set consisted of 50 tasks, each with a maximum execution time of 2 ms and maximum period of 200 ms. The task sets were pseudo randomly created according to the constraints described previously. To reduce the length of the major cycle, task periods were pseudo randomly generated as a multiple of 20 ms.

The results from this test are shown in Figure 8-5. It took approximately 1 minute to complete the schedulability test for one set of 50 tasks using TTSA2-DM, and a total of approximately 6 minutes to complete the test for TTSA2-All.

Table 8-3 Number of trials and the total time.

3-task set	TTC				TTH			
	TTSA1	TTSA2	BaB1	BaB2	TTSA1	TTSA2	BaB1	BaB2
Minimum number of trials	2.00E+00	2.00E+00	2.00E+00	2.00E+00	2.00E+00	2.00E+00	2.00E+00	2.00E+00
Maximum number of trials	2.70E+02	5.80E+02	2.97E+03	4.66E+06	1.65E+02	3.30E+02	2.83E+03	1.99E+05
Average number of trials	2.46E+01	4.63E+01	3.51E+02	1.75E+04	1.59E+01	3.04E+01	1.97E+02	2.08E+03
Total number of trials	2.46E+04	4.63E+04	3.51E+05	1.75E+07	1.59E+04	3.04E+04	1.97E+05	2.08E+06
Total time (s)	4.00E+00	4.00E+00	3.90E+01	8.49E+02	2.50E+00	2.50E+00	2.50E+01	1.17E+02

4-task set	TTC				TTH			
	TTSA1	TTSA2	BaB1	BaB2	TTSA1	TTSA2	BaB1	BaB2
Minimum number of trials	3.00E+00	3.00E+00	3.00E+00	3.00E+00	3.00E+00	3.00E+00	3.00E+00	3.00E+00
Maximum number of trials	2.65E+02	5.30E+02	7.23E+04	1.60E+08	2.65E+02	5.30E+02	4.21E+04	2.78E+07
Average number of trials	3.49E+01	7.01E+01	5.45E+03	5.30E+05	2.49E+01	4.99E+01	3.24E+03	1.13E+05
Total number of trials	3.49E+04	7.01E+04	5.44E+06	5.29E+08	2.49E+04	4.99E+04	3.23E+06	1.13E+08
Total time (s)	3.50E+00	4.50E+00	2.22E+02	1.73E+04	2.50E+00	4.50E+00	1.11E+02	4.17E+03

5-task set	TTC				TTH			
	TTSA1	TTSA2	BaB1	BaB2	TTSA1	TTSA2	BaB1	BaB2
Minimum number of trials	4.00E+00	4.00E+00	4.00E+00	4.00E+00	4.00E+00	4.00E+00	4.00E+00	4.00E+00
Maximum number of trials	1.40E+02	3.14E+02	1.03E+06	1.21E+09	1.02E+02	2.26E+02	1.33E+06	1.98E+08
Average number of trials	4.04E+01	8.58E+01	8.84E+04	2.04E+07	2.93E+01	6.34E+01	5.41E+04	5.09E+06
Total number of trials	4.04E+04	8.58E+04	8.84E+07	2.04E+10	2.93E+04	6.34E+04	5.41E+07	5.09E+09
Total time (s)	2.50E+00	6.00E+00	3.69E+03	2.94E+06	2.50E+00	3.50E+00	5.26E+03	3.79E+05

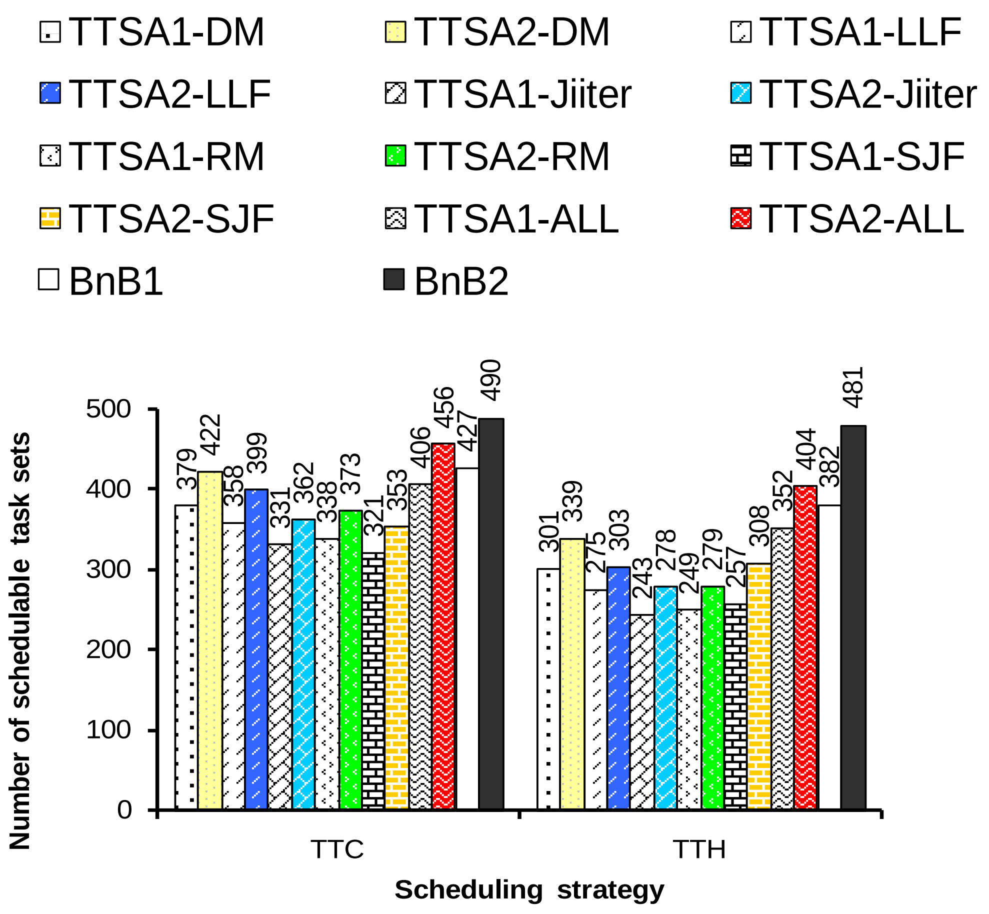


Figure 8-2 Number of scheduled task sets (3 interdependent tasks in each set).

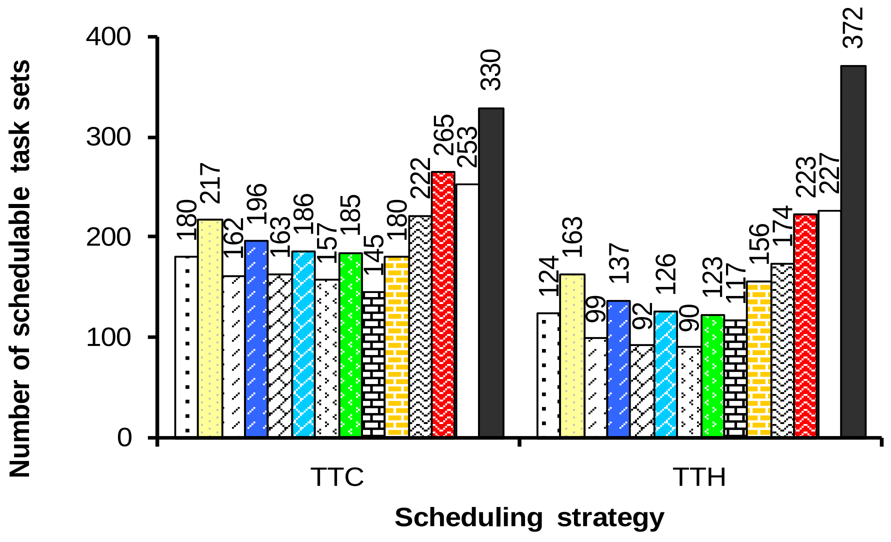


Figure 8-3 Number of scheduled task sets (4 interdependent tasks in each set).

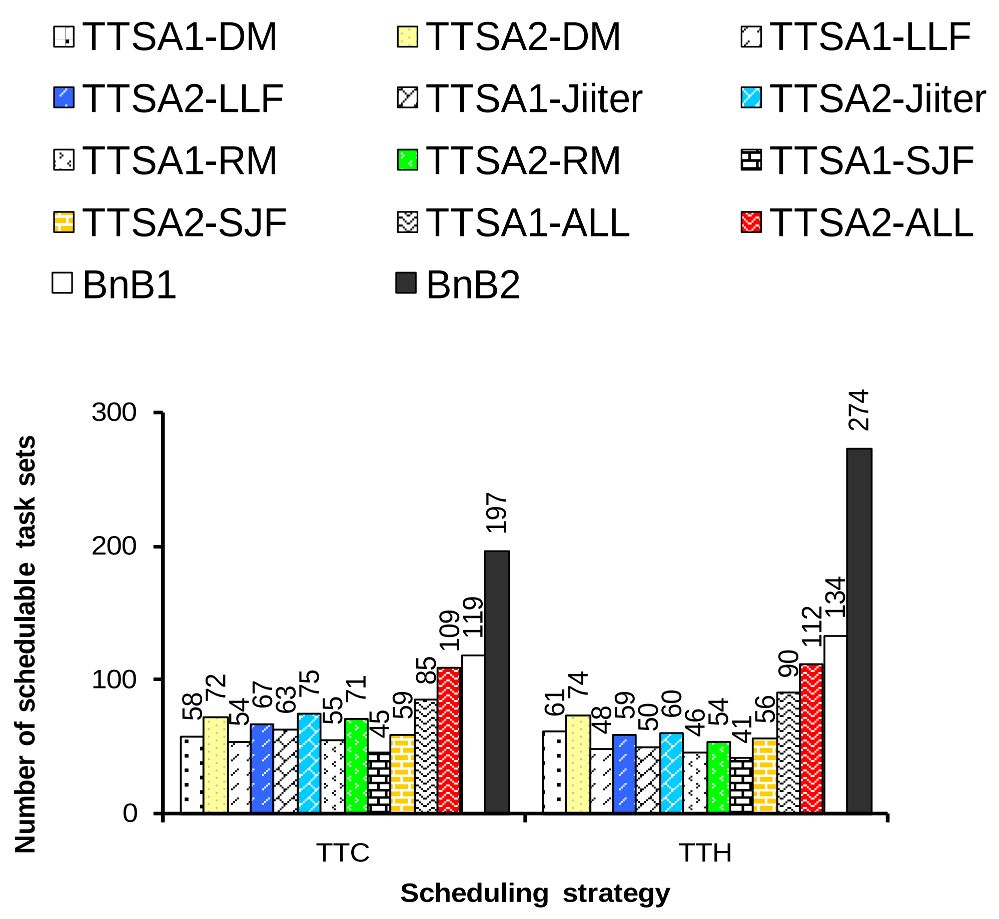


Figure 8-4 Number of scheduled task sets (5 interdependent tasks in each set).

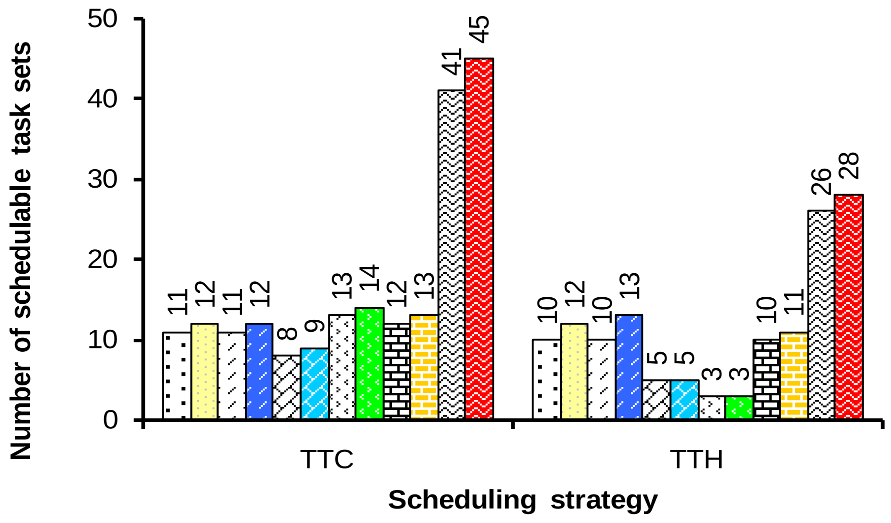


Figure 8-5 Number of scheduled task sets (50 interdependent tasks in each set).

8.4 Discussion

In this chapter a new offline scheduling algorithm, TTSA2, is introduced which helps to automate the process of determining the parameters required to schedule a given set of tasks in a resource-constrained embedded system employing a TTC or TTH architecture. The TTSA2 algorithm tries to find a suitable scheduler for the set of tasks by dividing each task into two or more segments.

Design patterns can be used to help identifying the points at which task may / may not be divided. For example these points can be chosen to mark the critical sections boundaries, where tasks gain access to a shared resource. Another way is to choose the points at which a condition is completed.

The effectiveness of the TTSA2 was tested in the same way which has been used to test the effectiveness of TTSA1. The results show that the TTSA2 algorithm improves on the performance of both a BaB search (with and without supporting task segmentation) and TTSA1.

To give an idea about the potential use of the TTSA2, a simple example has been discussed in Appendix C. The used set of tasks are intended to be representative of a case in which a single microcontroller is used to control three plants or to control a robot which has three degrees of freedom. More details are given in Appendix C.

8.5 Conclusions

The chapter has given an overview of the problems which may limit the use of the TTSA1 scheduling algorithm presented in the previous chapter. Then it introduced an enhanced version of the algorithm, called TTSA2, which tries to overcome these problems. The main concern was to increase the possibility of finding a feasible schedule (using TTC or TTH) by supporting task segmentation. The TTSA2 updates task parameters to take the segmentation overhead into account.

The effectiveness of TTSA2 was evaluated and compared with that of the TTSA1 and BaB algorithms using the same method which is used for evaluating the TTSA1 algorithm.

Chapter 9
Discussion and conclusions

This chapter discusses the work introduced in the thesis and analyses the thesis contributions. Limitation and future work are also discussed.

9.1 Reasons and motivation of the thesis work

The work described in this thesis is concerned with the development of reliable embedded systems using time-triggered architectures where limitted (or no) task pre-emption is permitted (TTC , TTH schedulers).

It has been argued that using offline schedulers, especially those based on time-triggered architecture, produce a highly predictable system. Building such architectures using approaches which support no (or limited) pre-emption helps to reduce the context switching overhead and increase the system determinism.

Despite the above features these schedulers have not been given much attention compared to that of online, or dynamic, schedulers which is based on full pre-emptive priority scheduling. It is claimed that the main reasons around this are the fragility of the schedule created by these schedulers and the necessity for having accurate knowledge of task parameters (like period and WCET) at the design stage. On the other hand, it is widely believed that priority scheduling, such as RM and EDF, are optimal (in the sense that if there exists a scheduler that can find a feasible schedule for a given set of tasks, then these schedulers can too) and it is easy to check the schedulability of a given task set using simple schedulability tests for these scheduler. This was taken as a reason to avoid using offline schedulers and support the use of full RTOSs which are normally based on fully pre-emptive priority schedulers, whatever the nature of the application at hand.

The work presented in this thesis helps to overcome the difficulties facing developers of the time-triggered approaches using TTC and TTH by automating the process of choosing an appropriate scheduler, and configuring the scheduler parameters, whenever one exists. Moreover the work discussed here introduced ways to reduce the variations in task execution times and hence increase the systems predictability. It is also shown that that priority driven schedulers, such as RM and EDF, are not always optimal as this optimality can only be achieved under certain conditions (such as assuming completely independent tasks which can be pre-empted at any point of time) which are seldom the case in real applications. Moreover the overhead, especially that which is caused by context switching, may affect the task schedulability. And finally it is shown that most of the schedulability tests for these schedulers normally test one constraint, such as the deadline, and do not take into consideration the maintainability of other constraints, such as precedence, distance, and jitter.

In sum the fundamental argument of the work presented here was to support and encourage the developers of embedded systems, especially those who work on safety-related applications, to first consider using time-triggered schedulers which have non pre-emption, or limited pre-emption, such as TTC and TTH.

9.2 A review of the contributions

This section reviews the key contributions of the studies presented in this document and discusses the extent to which the initial aims of the thesis were achieved.

9.2.1 Scheduling in real-time systems

The work presented in this thesis started in Chapter 1 by giving an introduction to the development of real-time systems and identifying the main problems facing developers of offline schedulers based on time-triggered architectures; in particular, the TTC and TTH. Two main problems are identified (i) the fragility of the schedulers; that is it may be necessary to redesign the whole schedule in case of making changes in the task set and (ii) the necessity for available accurate WCET estimates at design time.

Following this introduction, Chapter 2 gave an overview of the main task characteristics and constraints that are normally used to define tasks and inter-task relations in real-time systems. Then the chapter discussed various scheduling criteria and reviewed the most commonly used scheduling strategies. Some misconceptions about these schedulers, which tend to favour online-(or dynamic)-priority-based schedulers on offline schedulers, are then highlighted.

It has been shown that in order to guarantee that task constraints will be met it is necessary to choose an appropriate scheduling strategy, of equal importance is the configuration of the scheduler and task parameters; such as task order and task stating times. Schedulability tests are normally used to check the task schedulability under certain scheduling strategy.

Unfortunately these schedulability tests are not sufficient as they are normally used to check that only one constraint; such as task deadline, is met. Therefore Chapter 3 reviewed the most commonly used scheduling algorithms which are used to choose suitable values for task/scheduler parameters for a given scheduling strategy.

9.2.2 The need for stabilising task execution time

In order to ensure that task constraints will be maintained when using a specific scheduling strategy it is important that most task characteristics (such as BCET, WCET and period) to be known priori. Yet, as discussed in Chapter 4, variations in task execution time between its BCET and its WCET may have bad impacts in the systems predictability and it may also cause violations of task constraints. On the other hand, stabilising task execution time will help in increase system predictability, choosing the appropriate scheduler, and configuring the scheduler and task parameters. Moreover, with the use of TT schedulers (such as TTC and TTH) the complete task schedule can be known in advance and hence safety agents can easily be used to monitor the system while it is running and take appropriate actions in case of faults.

9.2.3 CB1 techniques

Chapter 5 started by reviewing previous work that has been done by Puschner and Burns (2002b; 2002a; 2003) to fix the task execution time. They introduced a technique called "single path programming paradigm". As its name implies, this technique is based on the idea of ensuring that the execution of any task will always follow one path. This is basically achieved by using a conditional move instruction, whenever a branch is needed. If the condition of the branching instruction is evaluated to be true the required set of instructions is executed; otherwise no operation instructions are executed. Hence the total execution time will be the same; this is obviously achieved at the expense of under-utilisation of the processor as tasks always run with their WCET even if this is not always the actual case.

The work presented in Chapter 5 highlighted two issues in the above technique: (i) it is limited to hardware that supports "conditional move" or similar instructions; (ii) it used no-operation instructions to balance the time which can increase power consumption. In an effort to tackle these problems the work presented in the chapter introduced a set of code-balancing techniques (CB1). The main idea around the CB1 techniques was to use a timer to measure the execution time of each form of branch / loop structure. These measurements are then used to (i) calculate the corresponding maximum execution time and (ii) use an interrupt-based sandwich delay, after sending the processor to idle mode to save power consumption, for a period of time equals to the difference between the current execution time and the maximum calculated execution time.

The effectiveness of the CB1 techniques was demonstrated using empirical studies. The results suggests that (i) the variation in task execution time, and hence the obtained jitter levels, achieved by using the CB1 techniques were less than those obtained by using the traditional coding techniques and higher than those of the single-path programming paradigm (ii) the average power consumption obtained by using the CB1 techniques was less than that of the single-path programming paradigm and higher than that of the traditional coding techniques (iii) both the results obtained by using single-path programming paradigm and CB1 techniques were achieved at the expense of an increase in the maximum task execution time; which can be seen as an acceptable price to pay for achieving this in safety-related systems.

9.2.4 Effects of task and scheduler parameters

After the attempts made to reduce variations in the task execution time, while avoiding excessive increase in power consumption, the focus of the work then shifted to tackle the problem of the scheduler fragility. As mentioned earlier the main cause of this problem is the need to recalculate task and scheduler parameters in case of updating the task set. The first step for addressing this problem was to study the effects of various task and scheduler parameters on task behaviour and average power consumption.

Chapter 6 presented an analysis of the effects of different values of task offsets, task order, and tick interval.

It was shown that inappropriate choices of task offsets may mean that the task set is not schedulable at all. Even in situation where task offsets do ensure that the set of tasks is schedulable, for example: task constraints such as deadline, precedence, distance, and latency are met, inappropriate choices may still lead to unnecessary high levels of jitter.

The analyses suggested that similar effects, as those which resulted from inappropriate offset values, may be caused by inappropriate configuration of task order and/or tick interval.

An empirical experiment was conducted to study the effects of choosing different values for the length of tick interval on the average power consumption. The results suggested the use of the longest possible tick length, as this will help in reducing the average power consumption.

9.2.5 TTSA1 algorithm

It was shown in the literature that testing the schedulability and choosing appropriate values for task and scheduler parameters is an NP-hard problem. Moreover, it is argued that schedulers based on TT architectures are seen to be fragile. In an attempt to address this problem, Chapter 7 introduced a novel scheduling algorithm (TTSA1 algorithm) which helps to automate the process of choosing an appropriate scheduler and configuring the scheduler and task parameters, whenever one exists, for time-triggered embedded systems, the focus was to choose between TTC and TTH schedulers.

It was shown that testing the schedulability of a given set of tasks using all the possible parameters combinations is a tedious and time consuming process even for a small number of tasks. Therefore the TTSA1 algorithm tries to choose a suitable scheduler, and suitable values for of task offsets, task order, and tick interval, whenever the set proved to be schedulable, using a heuristic approach that tests only a subset of all possible parameters combinations. In doing so the TTSA1 algorithm tries to test the task schedulability using the non pre-emptive scheduler first, the TTC scheduler. In order to keep the average power consumption as low as possible the TTSA1 algorithm tries to schedule the set of tasks using the longest possible tick interval.

As it is required to ensure that all task constraints, such as deadlines and jitter, are met, the TTSA1 algorithm tries to schedule the tasks using different strategies; TTSA1-DM, TTSA1-LLF, TTSA1-RM, TTSA1-Jitter, and TTSA1-SJF.

In reality any scheduler has an overhead, which is the time spent by the scheduler to check and update the status of each task in each tick, load the ready task(s) into memory, and perform the context switching. The work presented here suggested a simple method to empirically measure this overhead. This overhead was also taken into account by the TTSA1 algorithm. Moreover, with the help of using the CB1 techniques task WCET (which is required to be known a prior for almost all scheduling algorithms) can be easily obtained and fed to the TTSA1 algorithm.

The complexity of this algorithm has been evaluated and compared with that of the branch and bound algorithm which is used to test the effectiveness of other scheduling algorithms introduced in the literature. It has been found that the TTSA1 algorithm has considerably lower complexity than that of the BaB algorithm.

To test the effectiveness of the TTSA1 algorithm, different sets of tasks, sets of 3, 4, and 5 tasks, are randomly generated and fed to the algorithm. The number of sets which were found to be schedulable, the number of trails, and the total time taken by the TTSA1 algorithm has been recorded and compared to those of the BaB algorithm. The results showed that the number of identified sets that were found to be schedulable by the TTSA1 algorithm was close to those found by the BaB algorithm while the TTSA1 algorithm uses a smaller number of trials and shorter time to identify them. The experiment was repeated for sets of 50 tasks to test the effectiveness of the TTSA1 algorithm for scheduling sets of large numbers of tasks. In this experiment only TTSA1 algorithm was used as it has been found that the time taken by applying BaB algorithm was intractable.

Another set of empirical tests, described in Appendix B, were conducted to evaluate the effectiveness of the TTSA1 algorithm. In these experiments the TTSA1 algorithm was implemented in a separate microcontroller, called scheduler agent (SA), which is used to fine tune the schedule of a set of tasks running in the main target hardware, which was called the main processor (MP). After the fine tuning process is completed the SA continues to monitor the MP and takes appropriate action in case of faults.

9.2.6 TTSA2 algorithm

It was found that, despite its attractive features the TTSA1 algorithm cannot always find a workable scheduler, even if one exists. This happens if the task set has one or more long task(s) which has a WCET longer than the deadlines of other two or more shorter tasks.

To cope with this problem a modified version of the TTSA1 algorithm, called TTSA2 algorithm, was implemented. The TTSA2 algorithm tries to find a workable scheduler for a given task set by dividing one or more tasks into multiple segments, in case they cannot be scheduled as one segment. The points at which a task may / may not be divided into multiple segments may be chosen as the starting (or ending) points of critical sections, or the points at which a condition is completed.

The TTSA2 takes the segmentation overhead into account while checking the task schedulability and calculating the task and scheduler parameters, whenever one is found. The complexity and the effectiveness of the TTSA2 algorithm is calculated and compared to that of the BaB algorithm in the same way used for evaluating the TTSA1 algorithm.

A typical representative example that shows the effectiveness of the TTSA2 algorithm is given in Appendix C.

9.2.7 Potential appliaction

The algorithms (TTSA1 and TTSA2) can be used as part of a tool that can be used for automatic code generation for safety-related resource-constrained embedded systems. Using such a tool will not only reduce the time and effort required to develop such systems but it will also reduce the probability of the occurrence of scheduling errors, which may cause serious damage (an example of such damage is given in Reeves (1997)).

9.3 Limitations and future work

The work presented in this thesis was concerned with the development of time-triggered embedded systems which employ a single processor. This is a limitation as embedded applications are becoming more complex and hence tend to use multiple processors (Short and Pont, 2007; Short and Pont, 2008). Therefore future work needs to be done to extend the current work to support the use of multiple processors in cases where a workable scheduler can not be found using a single processor.

Another interesting extension of the code-balancing techniques presented here can be done at the compiler level to automatically balance the code for safety-related applications, which is currently worked on by another member of the ESL group.

It was shown that the number of tasks identified to be schedulable using TTSA1 and TTSA2 were close to (but still less than) those found by the BaB search. So in cases where a workable schedule can not be found the developer may still need to investigate the possibility of finding one using the BaB search. In which case someone may ask what is the benefit of using the TTSA1 or TTSA2 algorithms if it may be the case that BaB search may be needed at the end. The answer is that the time taken by either TTSA1 or TTSA2 was considerably shorter than that of the BaB search which was found to be intractable if the number of tasks in the set is large. So it is recommended to try the TTSA1 and TTSA2 algorithms at first and only use BaB if they cannot find a workable scheduler.

It was found that the results obtained by applying the TTSA1 and TTSA2 algorithms to find a workable schedule using the TTH scheduler were not as good as those obtained using the TTC scheduler compared to the results obtained by the corresponding BaB search. Thereby more work is needed to improve the performance in this case. This can be done by investigating ways to better choose the pre-emptive task as this can affect the task schedulability. Furthermore, the TTSA1 and TTSA2 algorithms can be extended to support other schedulers (such as the time triggered rate monotonic).

Finally it has been assumed that the points at which a task can be divided into more than one segment when applying the TTSA2 represent the critical section boundaries and they are defined in advance. Future work needs to be done to find a more efficient and flexible way of deciding these points of time to increase the efficiency of the TTSA2 algorithm.

9.4 Conclusions

The project described in this thesis has made three major contributions to the field of scheduling embedded systems using time-triggered architectures (TTC and TTH schedulers). Firstly, it introduced and assessed a set of code-balancing techniques which intend to reduce the variations in task execution time, and hence reduce the jitter and increase the systems predictability, while limiting the overhead in the average power consumption. This will, in turn, make the problem of estimating or measuring the WCET an easy job.

Secondly, it analysed the effects of inappropriate choices of a suitable scheduler and / or task and scheduler parameters on the task schedulability and power consumption.

Finally, it developed and assessed novel scheduling algorithms (TTSA1 and TTSA2) which help in automating the process of choosing an appropriate scheduler and configuring the scheduler parameters, in cases where one is found. Future work to extend and improve the efficiency of the introduced techniques is finally discussed.

Appendix A
TT architectures implemented in ESL

This appendix reviews the main implementation characteristics of TT schedulers used in this thesis (TTC and TTH schedulers). These implementations have been developed by researchers in the ESL research group at the University of Leicester.

A.1 TTC scheduler

One possible implementation of the TTC scheduler is described in the literature (Pont, 2001; Kurian and Pont, 2007). In this design, tasks are added to the scheduler in the initialisation stage, in the `main()` function, as shown in Figure A-1. A separate function, the `SCH_Dispatch_Tasks()` function in Figure A-2 , can be used to check / update the status of each task, in each tick, and send the system to the idle mode (to reduce the power consumption) after running the ready tasks, if any. One of the microcontroller's timers is set to overflow, causing an interrupt, every specified time interval (tick interval). In order to ensure having a fixed tick interval the timer interrupt service routine is kept as simple as possible. This can be achieved by allowing the tick ISR, the `SCH_Update()` function in Figure A-3 , to perform only its basic job of awaking the system from the idle mode and keeping track of the systems time through updating a global tick count.

```
void main(void)
   {
   // Set up the scheduler
   SCH_Init_T2();

   // Init tasks
   TaskA_Init();
   TaskB_Init();

   // Add tasks (10 ms ticks)
   // Parameters are <filename>, <offset in ticks>, <period
   // in ticks>
   SCH_Add_Task(TaskA, 0, 3);
   SCH_Add_Task(TaskB, 1, 3);
   SCH_Add_Task(TaskC, 2, 3);

   // Start the scheduler
   SCH_Start();

   while(1)
      {
      SCH_Dispatch_Tasks();
      }
   }
```

Figure A-1 The main function of a TTC scheduler which executes three periodic tasks, adapted from Kurian and Pont (2007).

```
void SCH_Dispatch_Tasks(void)
   {
   Update_required = 0;
    // Need to check for a timer interuppt since this
   // function was last executed (in case idle mode is not
   // being used)

   Disable_Timer_Interrupt();
   if (Tick_count_G > 0)
      {
      Tick_count_G--;
      Update_required = 1;
    }
   Enable_Timer_Interrupt();


   while (Update_required)
      {
     // Go through the task array
      for (Index = 0; Index < 3; Index++)
         {
         Update_Task_Status();
         if (Task[Index] due to run)
            {
            Run(Task[Index]);
            }
         }

      Disable_Timer_Interrupt();
      if (Tick_count_G > 0)
         {
         Tick_count_G--;
         Update_required = 1;
        }
      Enable_Timer_Interrupt();
      }  //end of while
      SCH_Go_To_Sleep();
   }
```

Figure A-2 The dispatch function of a TTC scheduler, adapted from Kurian and Pont (2007).

```
void SCH_Update(void)
   {
   // Note that an interrupt has occured
   Tick_count_G++;
   }
```

Figure A-3 The tick ISR function of a TTC scheduler, copied from Kurian and Pont (2007).

During the operation of the TTC scheduler, if a task temporary overran and its execution exceeded the tick interval, the tick ISR will pre-empt that task and update the system global time to keep track of the timing. In this case the pre-empted task will regain access to the CPU and continue its execution directly after the ISR ends. So there will be no chance for an uncontrolled access to a shared resource to occur.

Figure A-1, Figure A-2, and Figure A-3 show an example of three tasks run with TTC scheduler with a tick interval of 1 *ms*. It should be noted that the offsets (the time, measured from the start of the schedule, at which the task first starts execution) of Task A is zero, Task B is one, and Task C is 2 *ms*.

A.2 TTH scheduler

As explained earlier in Chapter 1 there are situations in which co-operative schedulers cannot satisfy task constraints, for example systems that have to react to some events within a period of time smaller than the execution time of a given task in the system. If these long tasks cannot be divided to multiple short tasks, then co-operative schedulers cannot be used. In such situations a modified version (called TTH) of the TTC scheduler discussed above may be used. As described in (Pont, 2001 and Maaita and Pont 2005) this scheduler has limited pre-emption capability, to reduce the scheduler overhead.

TTH supports multiple co-operative tasks and one short pre-empting task, which has priority higher than that of the co-operative tasks.

TTH can be implemented in the same way as TTC, with the exception of running the pre-emptive task from within the tick ISR to enable it to easily pre-empt other tasks whenever it becomes ready to run, as shown in Figure A-4.

```
void SCH_Update(void)
   {
   // Note that an interrupt has occured
   Tick_count_G++;
   Run pre-empting task;
   }
```

Figure A-4 The tick ISR function of a TTH, adapted from (Pont, 2001).

A.3 Conclusions

This appendix gave an overview of a possible implementation of the TTC and TTH schedulers as presented by researchers in ESL, University of Leicester.

Appendix B
TTSA1 Case study (Scheduler agent)

This appendix describes a case study which is developed during the course of the work in the thesis and is used to study the effectiveness of the TTSA1 algorithm[9].

B.1 The basic system description and functionality

In the proposed architecture introduced here, TTSA1 is used to fine tune the schedule of a set of tasks. For achieving this, TTSA1 is implemented and run in a separate hardware platform, in which case it is called the "scheduler agent" (SA).

The architecture is based on two components (i) the main processor (MP) platform, containing the time-triggered (co-operative) scheduler and task code, and (ii) a second processor, executing the "scheduler agent" (SA). In the experiments described in this appendix, the MP contains an instrumented scheduler and, during a "tuning" phase, the SA measures – on line – the execution time of each task as it runs. The measured values are then used by the TTSA1 algorithm to fine tune the task schedule in an attempt to ensure that (i) all task constraints - such as deadline and jitter - are met (ii) power consumption is reduced. After the tuning phase is completed the SA continues to monitor the MP and can take appropriate action (such as reseting the systems) in case of errors. The effectiveness of the proposed architecture is demonstrated empirically by applying it to a set of tasks that represent a typical embedded control system.

The monitoring approach in another TT architecture should also be mentioned here. The high-level Giotto language (Henzinger *et al.*, 2001) splits the system into two components, an "Embedded Machine" and a "Scheduling Machine". In the case of Giotto, the two components execute (as virtual machines) on the same CPU.

---

[9] Parts of this appendix have been published previously in Gendy *et al.*(2007)

The following section describes the proposed system architecture in detail.

B.2 THE MP-SA ARCHITECTURE

The remainder of this appendix presents and assesses the proposed system architecture.

B.2.1 Overview

An overview of the proposed system architecture is given in Figure B-1.

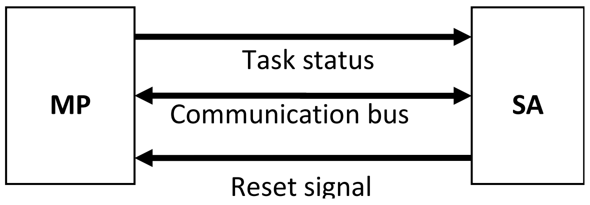


Figure B-1 An overview of the MP-SA architecture.

The proposed architecture employs an additional microcontroller (the Scheduling Agent, SA) to measure the BCET and WCET of each task while they are running on the main processor (MP) for a specified period of time. The SA also measures the scheduler overhead. The information gathered in this way is used in an attempt to fine tune task parameters (such as the task offsets, or "initial delay") and scheduler parameters (such as the tick interval) in order that all task constraints are met and the power consumption is reduced. After the fine-tuning phase is finished the SA continues to monitor the MP while running the tasks with the new parameters and take the appropriate action in case of errors.

For ease of reference, it will be referred to this two-processor arrangement as the "MP-SA architecture" throughout this appendix.

B.2.2 How does the MP-SA architecture operate?

The MP-SA architecture operates as follows:

i) At compile time, the task specifications (estimated BCET, estimated WCET, deadline, period, and upper bound of jitter) are provided to the MP.

ii) When the system starts, the MP sends task specifications to the SA.

iii) The SA asks the MP to schedule a number of dummy (empty) tasks, equal to the total number of the real tasks. The SA measures the scheduler overhead (by measuring the time spent by the scheduler out of "idle" mode as it executes the dummy tasks) for a pre-specified period of ("overhead") time.

iv) The SA calculates the initial task order, task offsets, and the scheduler tick interval based on the measured scheduler overhead and the given task parameters. In doing this the SA assumes that the overhead can be represented as an additional task that runs at every tick with BCET equal to zero and WCET equal to the scheduler overhead measured in Step iii. The SA sends these parameters to the MP and asks it to begin running the real tasks for a specified period of ("tuning") time.

v) While the MP is running the real tasks, the SA measures the actual BCET and WCET of each task.

vi) The SA repeats Step iv to fine tune the scheduler based on the actual measured BCET and WCET of each task.

vii) If a set of parameters is found so that all task constraints are met then the SA sends these parameters to the MP to restart the scheduler according to these new parameters (If a set of such parameters cannot be found then the SA tells the MP to "sail silently").

viii) If a suitable set of parameters have been found, the SA monitors the system during the normal program execution (functioning as a form of "task watchdog").

In Step i and Step ii it is assumed that the task specification will be (initially) stored in the MP. This allows us to create a generic SA (suitable for use for a wide range of different MPs).

B.2.3 Calculating task and scheduler parameters

Given a set of tasks each described by (BCET,WCET, deadline, period, jitter), the SA tries to calculate the appropriate task orders, task offsets, and tick interval so that task constraints (deadline and jitter in this study) are met and the power consumption is reduced. In doing this, the SA does not attempt to complete an exhaustive search. Instead, the SA employs the TTSA1 scheduling algorithm presented in Chapter 7.

B.3 THE MP-SA PERFORMANCE

An empirical test was carried out to study the performance of the MP-SA architecture[10].

The procedure and results obtained by using this architecture for a system with a set of 3 tasks are detailed in this section. A time-triggered co-operative (TTC) scheduler described previously was used to schedule the tasks.

B.3.1 Task set

To explore the effectiveness of the MP-SA architecture a set of 3 tasks (Task Sa, Task Co, Task Ac) was used: these were intend to be representative of those used in a typical embedded control system. In such a system, Task Sa would be the first task to run and would be used for data sampling. Task Co would then execute the control algorithm and – finally – Task Ac would control the actuator(s). In this study, it is assumed that the tasks run co-operatively, in this sequence.

The specifications of the tasks (BCET, WCET, deadline, period, and the maximum allowed jitter) are shown in Table B-1. There are two values for the BCET and another two for the WCET of each task indicated in this table. The first value is the value estimated by the designer and the second value is the value which measured by the SA while the tasks run on the target MP.

[10] It should be noted that for the purpose of simplifying the process of testing the proposed architecture various steps described in section B.2.2 have been carried out separately.

The chosen hardware platforms for both the SA and the MP were an NXP (formerly Philips) LPC2129 microcontroller running on a small evaluation board (Philips, 2004a). The communications between the SA and the MP was carried out via a CAN bus in this design (as shown in Figure B-2).

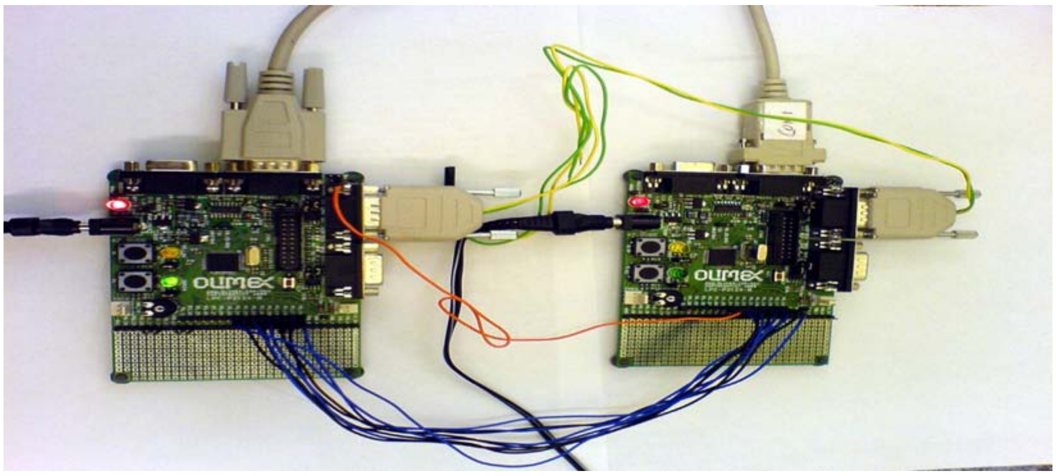


Figure B-2 The MP-SA hardware.

B.3.2 Task scheduling without the MP-SA architecture

One possible way to schedule the task set described Table B-1 is shown in Figure B-3. This schedule is based on the estimated values for the BCET and the WCET along with the other constraints for the periods, deadlines, and the upper bound of jitter[11]. The tick interval of this scheduler is chosen to be 100 *ms* length and the offsets of all the tasks are set to 0.

This schedule has several potential drawbacks which can be summarised as follows:

i) Estimated values of BCET and WCET may not be accurate (Table B-1). Building the scheduler based only on the estimated values may cause some tasks to miss their deadlines and / or encounter high level of jitter (Task Ac in this example: as shown in Figure B-4).

[11] Jitter is calculated as the difference between the maximum period and the minimum period.

ii) Ignoring the scheduler overhead can lead to similar effects (such as missed deadlines and / or increase the level of jitter): as shown in Figure B-5.

iii) Using a short tick interval, such as the 100 *ms* tick interval that was used here, instead of using the longest possible tick interval (which is 400 ms in the current case), will increase the system power consumption.

B.3.3 Task scheduling with the MP-SA architecture

The impact of the MP-SA architecture is illustrated in Figure B-6.

In this case, the SA has adapted the task schedule based on the measured values of the task BCET and WCET. One consequence is that the SA has identified a "compromise" tick interval (200 ms), which could be expected to reduce the power consumption (compared with the original value of 100 ms) while also ensuring low levels of jitter in both of the time-sensitive tasks (Task Sa and Task Ac). In this case, the jitter in the original schedule was 0.014 *ms* and 8.001 *ms* for Task Sa and Task Ac (respectively): after use of the SA, the jitter became 0.014 *ms* for both the tasks.

It should be noted that in addition to adjusting the tick interval to 200 *ms*, the SA adjusted the offset of Task Ac to 1 tick (rather than 0).

B.3.4 System behaviour in case of faults

The effectiveness of the MP-SA architecture on the system behaviour in the "normal" operating mode was also tested. During this test, a fault was injected in the system which changed the characteristics of Task Sa (the BCET became 37 *ms* and the WCET became 50 *ms* respectively). This error was assumed to represent the impact of a hardware fault.

Under these circumstances, without using the MP-SA architecture, the system ran without sensing the violated deadline and jitter constraints. In this test, the measured jitter of Task Co was 25.35 *ms*.

When the test was repeated in a system using the MP-SA architecture, the SA sensed the violated constraints and it forced the MP to restart (in an effort to recover from this fault). Other recovery behaviour (e.g. backup tasks) could also be implemented in response to the detected errors.

Table B-1 Task specifications.

Task	Execution time (ms)				Deadline (*ms*)	Period (*ms*)	Jitter (*ms*)
	Estimated		Measured				
	BCET	WCET	BCET	WCET			
Sa	37	38	37	40	50	400	1
Co	10	11	10	11	65	400	6.5
Ac	20	21	20	22	70	400	4.5

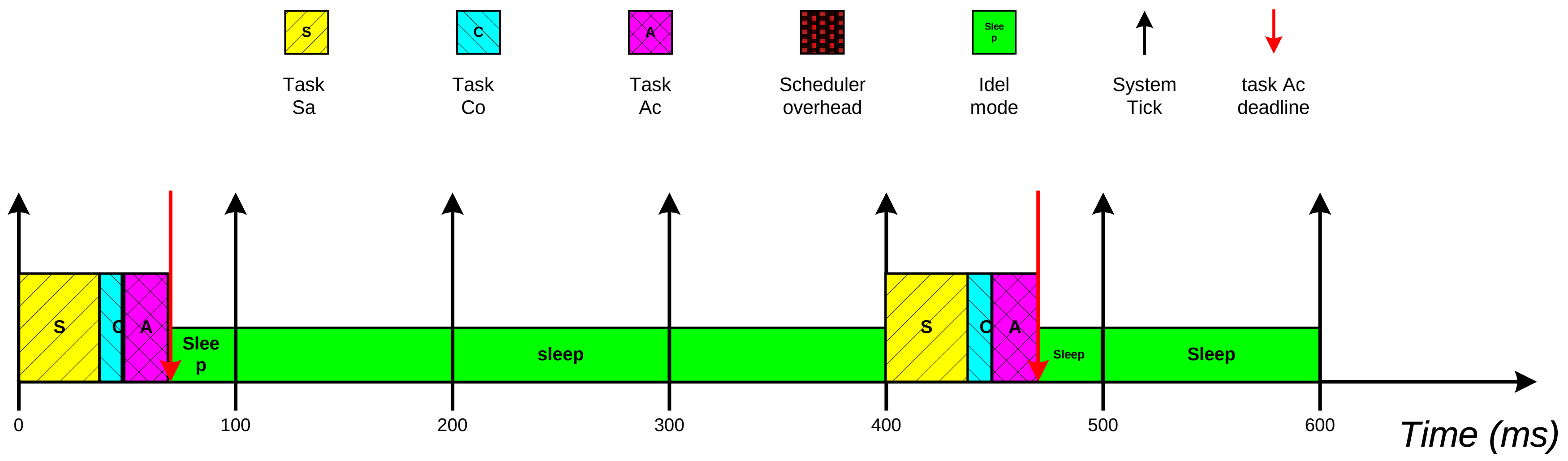


Figure B-3 A simple schedule based on the estimated BCET and WCET with 100 *ms* tick interval.

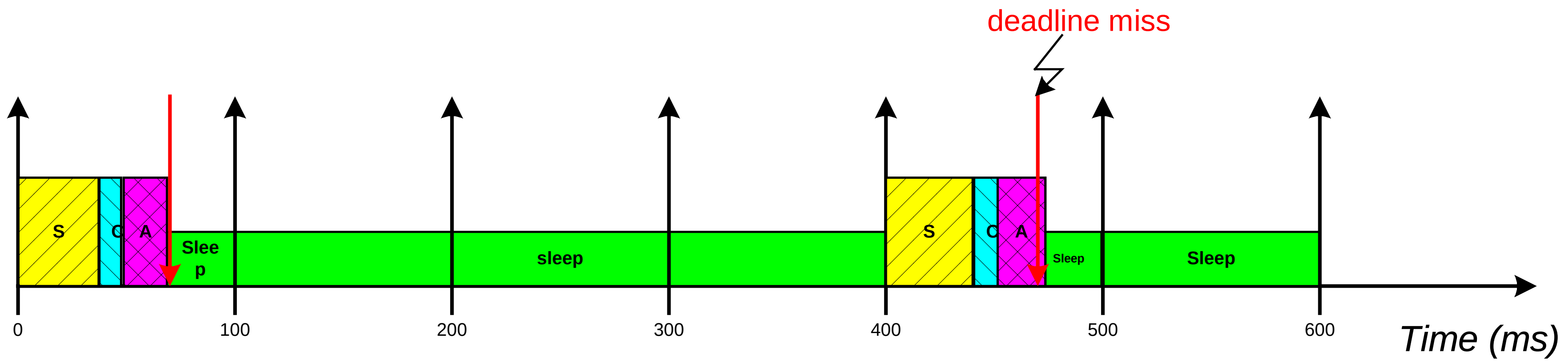


Figure B-4 Effect of inaccurate estimations of BCET and WCET on task behaviour.

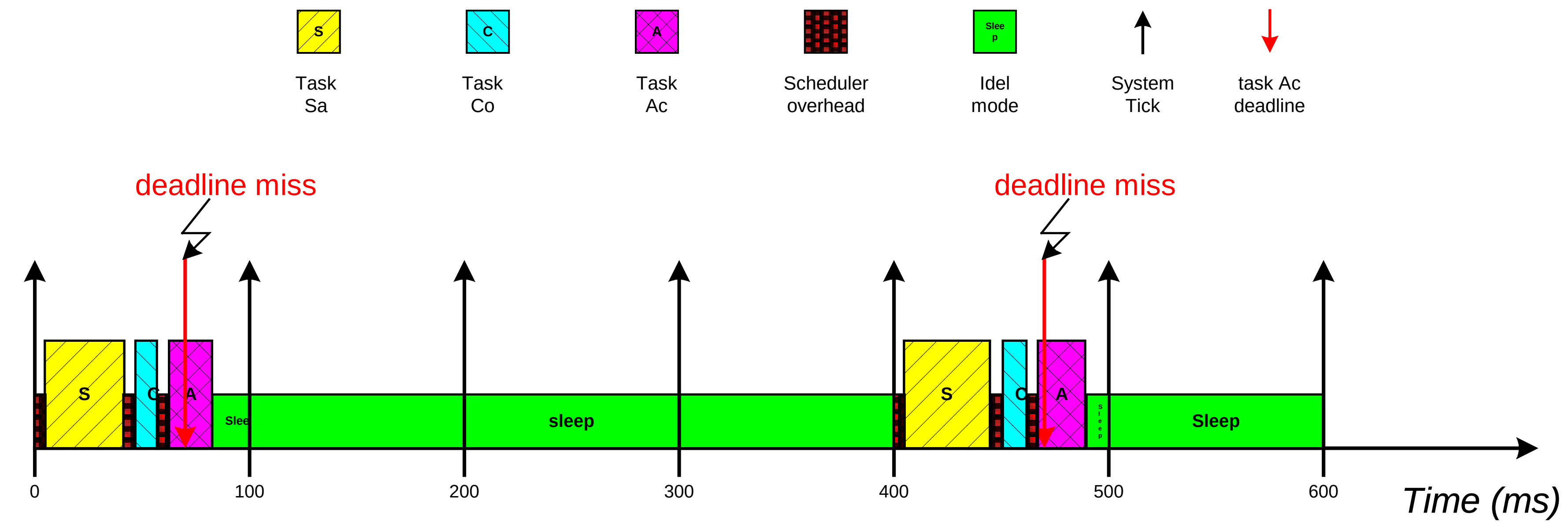


Figure B-5 Effect of scheduler overhead on task behaviour.

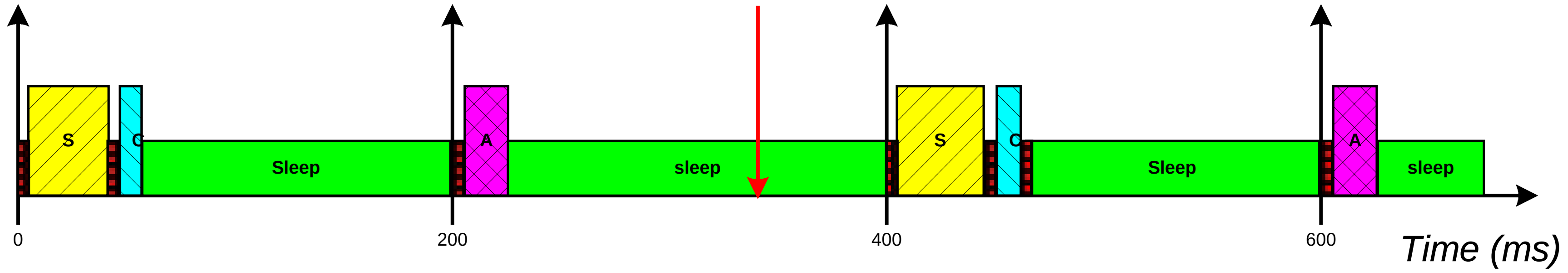


Figure B-6 Task behaviour with the scheduler produced by the MP-SA architecture

B.3.5 Extended task set

To test the effectiveness of the MP-SA architecture with a slightly more complex system, two additional tasks (Task EXT1 and Task EXT2) were added. Table B-2 shows the specifications of the additional tasks (BCET, WCET, deadline, period, and the maximum allowed jitter).

The same hardware platforms were used in this study.

In this case, the SA set the tick interval to 200 *ms* and the offsets of the Task Ac, Task EXT1 and Task EXT2 to 1 tick.

Using the SA, the measured values of the jitter from the Task Sa, Task Co, Task Ac, Task EXT1 and Task EXT2 was found to be 0.014 *ms*, 6.008 *ms*, 0.014 ms, 4.012 *ms* and 5.995 *ms* respectively. The jitter constraints for all 5 tasks were met.

Table B-2 Extended Tasks' specifications.

Task	Execution time (ms)				Deadline (*ms*)	Period (*ms*)	Jitter (ms)
	Estimated		Measured				
	BCET	WCET	BCET	WCET			
EXT1	3	4	4	5	80	400	5
EXT2	5	6	6	7	90	400	6.5

B.4 Conclusions

This appendix presented a novel architecture that can be used in time-triggered embedded systems to:

[1] Measure the BCET, WCET, and scheduler overhead during "normal" system operation.

[2] Fine tune the scheduler using the TTSA1 algorithm so that task constraints (such as deadline and jitter) are met and power consumption is reduced.

[3] Monitor the system and take the appropriate action in the event of faults.

These results were achieved at the expenses of using an additional microcontroller in the system.

The effectiveness of the proposed architecture was demonstrated using a small empirical study.

Appendix C
TTSA2 application example

This appendix describes a simple example that can be used to show the effectiveness of the TTSA2 algorithm.

C.1 The basic system description and functionality

This application example assumes having a system with a simple set of 3 tasks, each has 3 segments (Segment Sa, Segment Co, Segment Ac). In each task for such a system, Segment Sa would be the first segment to run and would be used for data sampling. Segment Co would then execute the control algorithm and – finally - Segment Ac would control the actuator(s). Although in reality each of these segments are usually implemented as a separate task, they are used here to in this way to explain a logical way of diving long tasks into multiple segments. These three tasks are intended to be representative of three plants that are controlled by a single microcontroller; such as controlling 3 inverted pendulums (Cervin *et al.*, 2004) or controlling a robot which has three degrees of freedom.

C.2 Task specifications

Table C-1 shows an example of the specification of the 3 tasks system. Table C-2 shows an example of the exclusion relation between various segments of different tasks. In this table a value of "T" in cell (*i*, *j*) means that segment *i* excludes segment *j* and vice versa. A value of "F" in cell (*i*, *j*) means that there is no exclusion relationship between segment *i* segment *j*. It should be noted that the shaded cells in that table are redundant entries so they do not need not to be filled.

Table C-1 Task specifications for the application example.

Task	No of Segments	WCET (*ms*)	Deadline (*ms*)	Period (*ms*)	Jitter (*ms*)	Latency (*ms*)	Distance (*ms*)	Precedence
1	3	Total:0.315 Sa_1:0.010 Co_1:0.300 Ac_1:0.005	1.0	1.0	0.01	Latency (A,B): 1.0	Distance (A,B): 0.0	Task A Precedes Task B
2	3	Total:0.340 Sa_2:0.013 Co_2:0.320 Ac_2:0.007	1.0	1.0	0.10			
3	3	Total:0.370 Sa_3:0.012 Co_3:0.350 Ac_3:0.008	5.0	5.0	1.00			

Table C-2 Task exclusion relations for the application example

Segment	Sa_1	Co_1	Ac_1	Sa_2	Co_2	Ac_2	Sa_3	Co_3	Ac_3
Sa_1	▨	▨	▨	T	T	T	T	T	T
Co_1	▨	▨	▨	T	F	F	T	F	F
Ac_1	▨	▨	▨	T	F	F	T	F	F
Sa_2	▨	▨	▨	▨	▨	▨	T	T	T
Co_2	▨	▨	▨	▨	▨	▨	T	F	F
Ac_2	▨	▨	▨	▨	▨	▨	T	F	F
Sa_3	▨	▨	▨	▨	▨	▨	▨	▨	▨
Co_3	▨	▨	▨	▨	▨	▨	▨	▨	▨
Ac_3	▨	▨	▨	▨	▨	▨	▨	▨	▨

C.3 Task scheduling according to one segment per task

Figure C-1 shows the schedule of this task set considering each task as only one segment. As can be noticed this will violate, at least, the jitter, and may be deadline, constraints of Task 1 every time Task 3 runs.

C.4 Task scheduling with considering multiple segments per task

By applying the TTSA2 scheduling algorithm to test the schedulability of this task set a workable schedule is found without violating any task constraints, as shown in Figure C-2. It should be noted that Task 1 and Task 2 each will run as one segment with offset 0, and Task 3 will as 2 segments, the first with offset 0 and the second with offset 1.

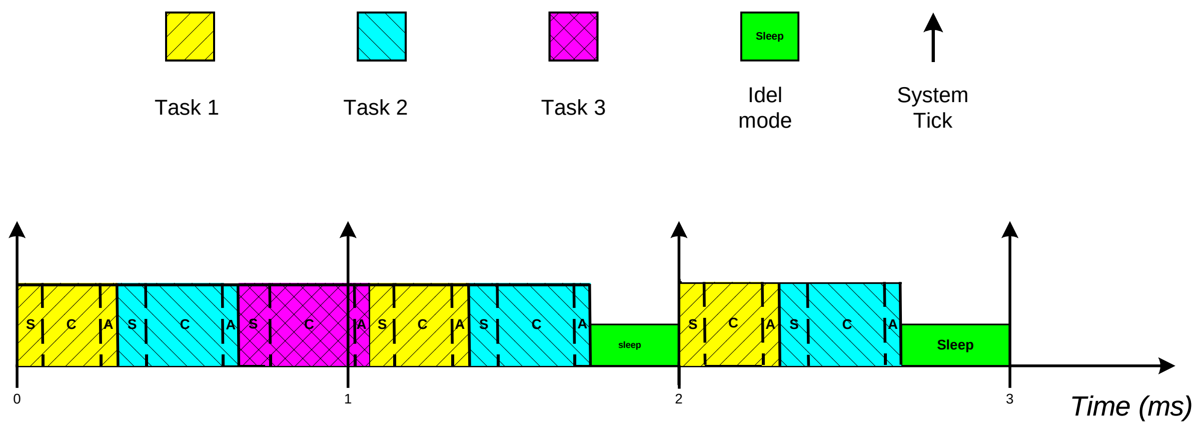


Figure C-1 Illustrating of the first 3 ticks for the tasks shown in Table C-1 and Table C-2 scheduled by TTC without considering task segmentation.

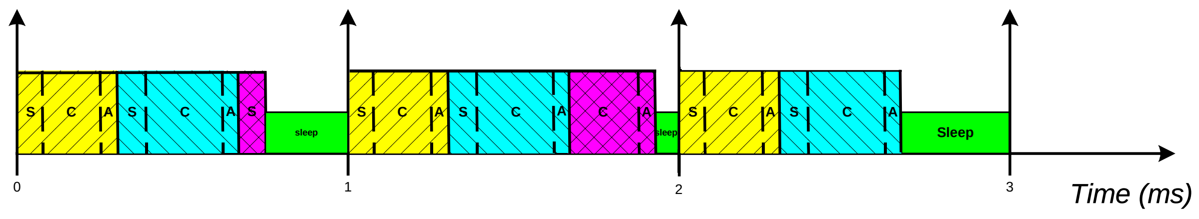


Figure C-2 Illustrating of the first 3 ticks for the tasks shown in Table C-1 and Table C-2 scheduled by TTC with considering task segmentation (TTSA2 algorithm).

C.5 Conclusions

In this appendix a simple example of a typical control system is used to show the effectiveness of the TTSA2 algorithm. It has been shown that allowing task segmentation may increase the chance of finding a feasible schedule for the task set using TTC; the same applies for TTH.

References

Albert, A. (2004). "*Comparison of event-triggered and time-triggered concepts with regard to distributed control systems*". Proceedings of Embedded World, Nurnberg, Germany.

Alexander, C. (1979). "*The timeless way of building*". NY, Oxford University Press.

Alexander, C., Ishikawa, S., Silverstein, M., Jacobson, M., Fisksdahl-King, I. and Angel, S. (1977). "*A pattern language*". NY, Oxford University Press.

Allen, J., Kennedy, K., Porterfield, C. and Warren, J. (1983). "*Conversion of control dependence to data dependence*". Proc. 10th ACM Symposium on Principles of Programming Languages, Austin, Texas, USA.

Allworth, S. T. (1981). "*An introduction to real-time software design*", Macmillan, London.

Aparicio, C., Segarra, J., Rodríguez, C., J. L. Villarroel and Viñals, V. (2008). "*Avoiding the WCET overestimation on LRU instruction cache*". Proceedings of the 2008 14th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications

Arnold, K. (2000). "*Embedded Controller Hardware Design*", Newnes.

Atkinson, C., C., B., Gross, H. and Peper, C. (2005). "*Component-based software development for embedded systems: An overview of current research trends (Lecture notes in computer science)* ", Springer-Verlag Berlin and Heidelberg GmbH & Co. K.

Audsley, N. C., Burns, A., Richardson, M. F. and Wellings, A. J. (1991). "*Hard real-time scheduling: the deadline-monotonic approach*". Proceedings 8th IEEE Workshop on Real-Time Operating Systems and Software, Atlanta, GA, USA.

Audsley, N. C., Tindell, K. and Burns, A. (1993). "*The end of line for static cyclic scheduling?*". Fifth Euromicro Workshop on Real-Time Systems.

Ayavoo, D. (2006). "*Development of a tool to support the design of real-time embedded control systems for X-By-Wire applications*". Embedded Systems Laboratory, University of Leicester. PhD thesis.

Ayavoo, D., Pont, M. J., Fang, J., Short, M. and Parker, S. (2005). "*A 'Hardware-in-the-Loop' testbed representing the operation of a cruise-control system in a passenger car*". In: Koelmans, A., Bystrov, A., Pont, M.J., Ong, R. and Brown, A. (Eds.), Proceedings of the Second UK Embedded Forum, Published by University of Newcastle upon Tyne, Birmingham, UK.

Ayavoo, D., Pont, M. J., Short, M. and Parker, S. (2007). "*Two novel shared-clock scheduling algorithms for use with 'Controller Area Network' and related protocols*." Journal of Microprocessors and Microsystems 31(5): 326-334.

Baker, T. P. and Shaw, A. (1988). "*The cyclic executive model and Ada*". Proceedings of the Real-Time Systems Symposium Huntsville, AL, USA.

Barr, M. (1999). "*Programming embedded systems in C and C ++*", O'Reilly.

Baruah, S. K. (2006). "*The non-preemptive scheduling of periodic tasks upon multiprocessors*." Real-Time Systems 32(1-2): 9-20.

Baruah, S. K., Buttazzo, G., Gorinsky, S. and Lipari, G. (1999). "*Scheduling periodic task systems to minimize output jitter*". Proceedings of the Sixth International Conference on Real-Time Computing Systems and Applications, RTCSA Hong Kong.

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

Baynes, K., Collins, C., Fiterman, E., Ganesh, B., Kohout, P., Smit, C., Zhang, T. and Jacob, B. (2003). "*The performance and energy consumption of embedded real-time operating systems*." Computers, IEEE Transactions on 52(11): 1454 - 1469.

Bini, E., Buttazzo, G. C. and Buttazzo, G. M. (2003). "*Rate monotonic scheduling: The hyperbolic bound*." IEEE Trans. Computers 52(7): 933 - 942.

Brucker, P., Garey, M. R. and Johnson, D. S. (1977). "*Scheduling equal-length tasks under treelike precedence constraints to minimize maximum lateness*." Mathematics of Operations Research 2(3): 275-284.

Burguiere, C. and Rochange, C. (2005). "*A contribution to branch prediction modeling in WCET analysis*". Proceedings of the Design, Automation and Test in Europe Conference and Exhibition (DATE'05).

Burns, A. (1995). "*Generating feasible cyclic schedules*." Control Engineering Practice 3(2): 151-162.

Buttazzo, G. C. (2005a). "*Hard real-time computing cystems: Predictable scheduling algorithms and applications*", Springer.

Buttazzo, G. C. (2005b). "*Rate monotonic vs. EDF: judgment day*." Real-Time Syst. 29(1): 5-26.

Buttazzo, G. C., Lipari, G., Abeni, L. and Caccamo, M. (2005). "*Soft real-time systems: predictability vs. efficiency*", Springer.

Cervin, A., Lincoln, B., Eker, J., Årzén, K. and Buttazzo, G. (2004). "*The jitter margin and its application in the design of real-time control systems*". Proceedings of the 10th International Conference on Real-Time and Embedded Computing Systems and Applications, Göteborg, Sweden.

Cheng, A. M. K. (2002). "*Real- time systems, scheduling, analysis, and verifications*", John Wiley & Sons.

Cottet, F., Delacroix, J., Kaiser, C. and Mammeri, Z. (2002). "*Scheduling in real-time systems*", Wiley.

Cucu, L. and Sorel, Y. (2004). "*Non-preemptive multiprocessor scheduling for strict periodic systems with precedence constraints*". Proc. 23rd Annual Workshop of the UK Planning and Scheduling Special Interest Group, PLANSIG'04, Cork, Ireland.

Cunningham, W. and Beck, K. (1987). "*Using pattern languages for object-oriented programs*". Proceedings of OOPSLA87', Orlando, Florida.

David, L. (2000). "*NASA report: too many failures with faster, better, cheaper*." from WWW Page: < http://www.space.com/businesstechnology/business/spear_report_000313.html >, (accessed May 2009).

Deverge, J. and Puaut, I. (2005). "*Safe measurement-based WCET estimation*". Proc. of the 5th Workshop on Worst-Case Execution Time Analysis, held in conjunction with the 17th Euromicro Conference on Real-Time Systems.

Dobrin, R. and Fohler, G. (2004). "*Reducing the number of pre-emptions in fixed priority scheduling*". Proc. 16th Euromicro Conference on Real-Time Systems.

Domaratsky, Y. and Perevozchikov, M. (2000). "*Highly dependable time-triggered operating system*". Dedicated Systems Magazine. 4: pp. 77-80.

Ekelin, C. and Jonsson, J. (1999). "*Real-time system constraints: Where do they come from and where do they go?*". Proceedings of the Int'l Workshop on Real-Time Constraints, Alexandria, Virginia, USA.

Ekelin, C. and Jonsson, J. (2000). "*Solving embedded system scheduling problems using constraint programming*". Tech. Rep. 00-12. S-412 96 Goteborg, Sweden, Dept. of Computer Engineering, Chalmers University of Technology.

Ekelin, C. and Jonsson, J. (2001). "*Evaluation of search heuristics for embedded system scheduling problems*". Proceedings of the international conference on Principles and Practice of Constraint Programming, Paphos, Cyprus.

Engblom, J. (2002). "*Processor pipelines and static worst-case execution time analysis*". Dept. of Information Technology. Acta Universitatis Upsaliensis, Uppsala University. PhD thesis.

Engblom, J. and Ermedahl, A. (2000). "*Validating a worst-case execution-time analysis method for an embedded processor*". Proc. 21st IEEE Real-time Systems Symposium (RTSS'00), Orlando, Florida, USA.

Engblom, J., Ermedahl, A., Sjoedin, M., Gustafsson, J. and Hansson, H. (2001). "*Worst-case execution-time analysis for embedded real-time systems*." Journal of Software Tools for Technology Transfer (STTT) 4(4): pp. 437-455.

Engblom, J. and Jonsson, B. (2002). "*Processor pipelines and their properties for static WCET analysis*". Proceedings of the Second International Conference on Embedded Software, London, UK, Springer-Verlag.

Ferdinand, C., Martin, F. and Wilhelm, R. (1997). "*Applying compiler techniques to cache behavior prediction*". Proc. ACM SIGPLAN Workshop on Language, Compiler and Tool Support for Real-Time Systems, Las Vegas, Nevada, USA.

Ferrari, D. and Verma, D. C. (1990). "*A scheme for real-time channel establishment in wide-area networks*." IEEE Journal on Selected Areas in Communications 8(3): 368 - 379.

Gamma, E., Helm, R., Johnson, R. and Vlissides, J. (1995). "*Design patterns: Elements of reusable object-oriented software*". Reading, MA, Addison-Wesley.

Gangoiti, U., Marcos, M. and Estévez, E. (2005). "*Using cyclic executives for achieving closed loop co-simulation*". Proc. of the Joint 44th IEEE Control and Decision Conference and European Control Conference CDC-ECC'2005, ISSN: 0-7803-9568-9, 4785-3790 (2005), Sevilla, Spain.

Ganssle, J. (1992). "*The art of programming embedded systems*", Academic Press, San Diego, USA.

Ganssle, J. and Barr, M. (2003). "*Embedded systems dictionary*", CMP Books.

Gendy, A., Dong, L. and Pont, M. J. (2007). "*Improving the performance of time-triggered embedded systems by means of a scheduler agent*". Proc. of the ASME 2007 International Design Engineering Technical Conferences & Computers and Information in Engineering Conference (IDETC/CIE 2007), Las Vegas, Nevada, USA.

Gendy, A. and Pont, M. J. (2007). "*Towards a generic 'single-path programming' solution with reduced power consumption*". Proc. of the ASME 2007 International Design Engineering Technical Conferences & Computers and Information in Engineering Conference (IDETC/CIE 2007), Las Vegas, Nevada, USA

Gendy, A. and Pont, M. J. (2008a). "*Automatically configuring time-triggered schedulers for use with resource-constrained, single-processor embedded systems*." IEEE Transactions on Industrial Informatics 4(1): 37 - 46.

Gendy, A. and Pont, M. J. (2008b). "*Automating the processes of selecting an appropriate scheduling algorithm and configuring the scheduler implementation for time-triggered embedded systems*". Lecture Notes in Computer Science, Computer Safety, Reliability, and Security, Volume 5219/2008, Springer Berlin / Heidelberg, 27th International Conference on Computer Safety, Reliability and Security, SAFECOMP08, 22-25 September 2008, Newcastle upon Tyne, UK.

Gerber, R., Hong, S. and Saksena, M. (1994). "*Guaranteeing end-to-end timing constraints by calibrating intermediate processes*". Proc. IEEE Real-Time Systems Symposium, IEEE Computer Society Press.

Gerber, R., Hong, S., Saksena, M. (1995). "*Guaranteeing real-time requirements with resource-based calibration of periodic processes*." IEEE Transactions on Software Engineering 21(7): 579-592.

Gergeleit, M. and Nett, E. (2002). "*Scheduling transient overload with the TAFT scheduler*". GI/ITG specialized group of operating systems. Berlin.

Goossens, J. and Devillers, R. (1997). "*The non-optimality of the monotonic priority assignments for hard real-time offset free systems*." Real-Time Systems 13: 107–126.

Gregory, F. D. (1996). "*Safety and mission assurance in a better, faster, cheaper environment*." Acta Astronautica 39(6): 465-469.

Han, C., Lin, K. and Hou, C. (1996). "*Distance-constrained scheduling and its applications to real-time systems*." IEEE Transactions on Computers 45 (7): 814 - 826.

Henzinger, A. T., Horowitz, B. and Kirsch, C. M. (2001). "*Giotto: A time-triggered language for embedded programming”*". Proceedings 1st International Workshop on Embedded Software (EMSOFT), Lecture Notes in Computer Science 2211, Springer-Verlag.

Huang, C., Chang, L. and Kuo, T. (2003). "*A cyclic-executive-based QoS guarantee over USB*". IEEE 9th Real-Time and Embedded Technology and Applications Symposium, Toronto, Canada.

Hughes, Z. M. and Pont, M. J. (2004). "*Design and test of a task guardian for use in TTCS embedded systems*". In : Koelmans, A., Bystrov, A. and Pont, M.J. (Eds.) Proceedings of the UK Embedded Forum 2004 (Birmingham, UK, October 2004), Published by University of Newcastle upon Tyne [ISBN: 0-7017-0180-3]. .

Hughes, Z. M. and Pont, M. J. (2008). "*Reducing the impact of task overruns in resource-constrained embedded systems in which a time-triggered software architecture is employed*." Transactions of the Institute of Measurement and Control 30(5): 427-450.

IEC. (2005). "*Functional safety of electrical/electronic/programmable electronic safety-related systems – Part 0: Functional safety and IEC 61508*." from WWW Page: < http://www.iec.ch/functionalsafety >, (accessed August 2009).

Intel. (2009). "*Intel Museum*." from WWW Page: < http://www.intel.com/museum/archives/4004.htm >, (accessed May 2009).

Jeffay, K., Stanat, D. F. and Martel, C. U. (1991). "*On non-preemptive scheduling of periodic and sporadic tasks*". the 12 th IEEE Symposium on Real-Time Systems.

Joch, A. and Sharp, O. (1995). "*How software doesn't work*." Byte 20(12): 49–60.

John, L. H. and David, A. P. (2007). "*Computer architecture: a quantitative approach*", Morgan Kaufmann.

Joseph, M. (1996). "*Real-time systems: specification, verification and analysis*", Prentice Hall.

Kalinsky, D. (2001). "*Context switch*." Embedded Systems Programming 14(1): 94-105.

Kalinsky, D. (2005). "*New directions in real-time operating system kernels*." Embedded Control Europe.

Kim, N., Ryu, M., Hong, S. and Shin, H. (1999). "*Experimental assessment of the period calibration method: A case study*." Real-Time Systems 17(1): 41 - 64.

Kirkpatrick, S., Gelatt, C. D. and Vecchi, M. P. (1983). "*Optimisation by simulated annealing*." Science 220: 671-680.

Kirner, R. and Puschner, P. (2003). "*Discussion of misconceptions about WCET analysis*". 3rd Euromicro International Workshop on WCET Analysis.

Kirner, R. and Puschner, P. (2008). "*Obstacles in worst-case execution time analysis*". 11th IEEE International Symposium on Object Oriented Real-Time Distributed Computing (ISORC), Florida, USA.

Kopetz, H. (1991). "*Event-triggered versus time-triggered real-time systems*". The International Workshop on Operating Systems of the 90s and Beyond, Springer-Verlag

Kopetz, H. (1993). "*Should responsive systems be event-triggered or time-triggered?*" IEICE Transactions on Information and Systems: 1325--1332.

Kopetz, H. (1997). "*Real-time systems: design principles for distributed embedded applications*", Kluwer Academic.

Kopetz, H., Nossala, R., Hexela, R., Krügera, A., Millingera, D., Pallierera, R., Templea, C. and Krugb, M. (1998). "*Mode handling in the time-triggered architecture*." Control Engineering Practice 6(1): 61-66.

Kovalyov, M. and Xu, J. (2000). "*Uniform processor scheduling with release times, deadlines, precedence and exclusion relations International*". Workshop Discrete optimization methods in scheduling and computer-aided design, Minsk, Belarus.

Kurian, S. and Pont, M. J. (2005a). "*Building reliable embedded systems using Abstract Patterns, Patterns, and Pattern Implementation Examples*". In: Koelmans, A., Bystrov, A., Pont, M.J., Ong, R. and Brown, A. (Eds.), Proceedings of the Second UK Embedded Forum, Published by University of Newcastle upon Tyne [ISBN: 0-7017-0191-9], Birmingham, UK.

Kurian, S. and Pont, M. J. (2005b). "*Mining for pattern implementation examples*". In: Koelmans, A., Bystrov, A., Pont, M.J., Ong, R. and Brown, A. (Eds.), Proceedings of the Second UK Embedded Forum , Published by University of Newcastle upon Tyne, Birmingham, UK,.

Kurian, S. and Pont, M. J. (2006a). "*Evaluating and improving pattern-based software designs for resource-constrained embedded systems*". In: C. Guedes Soares & E. Zio (Eds), "Safety and Reliability for Managing Risk": Proceedings of the 15th European Safety and Reliability Conference (ESREL 2006), Estoril, Portugal.

Kurian, S. and Pont, M. J. (2006b). "*Restructuring a pattern language which supports time- triggered co-operative software architectures in resource-constrained embedded systems*". 11th European Conference on Pattern Languages of Programs (EuroPLoP 2006), Germany.

Kurian, S. and Pont, M. J. (2007). "*The maintenance and evolution of resource-constrained embedded systems created using design patterns*." Journal of systems and software 8(1): 32 - 41

Labrosse, J. (2002). "*MicroC OS II: The real time kernel*", Cmp Books.

Laplant, P. A. (2004). "*Real-time systems design and analysis*", IEEE press / John Wiley & sons, Inc., publication.

Leung, J. (1989). "*A new algorithm for scheduling periodic - real-time tasks*." Algorithmica 4: 209 - 219.

Leung, J. and Merrill, M. (1980). "*A note on preemptive scheduling of periodic real-time tasks*." Information Processing Letters 11(3): 115-118.

Leung, J. and Whitehead, J. W. (1982). "*On the complexity of fixed priority scheduling of periodic real-time tasks*." Performance Evaluation 2(4): 237-250.

Leventhal, L. A. (1979). "*Introduction to microprocessors*", Prentice-Hall.

Li, W., Kavi, K. and Akl, R. (2007). "*A non-preemptive scheduling algorithm for soft real-time systems*." Computers and Electrical Engineering 33(1): 12-29.

Lin, T. and Tarng, W. (1991). "*Scheduling periodic and aperiodic tasks in hard real-time computing systems*." ACM SIGMETRICS Performance Evaluation Review 19(1): 31 - 38

Liu, C. L. and Layland, J. W. (1973). "*Scheduling algorithms for multiprogramming in a hard real-time environment*." Journal of the ACM 20(1): 40-61.

Locke, C. D. (1992). "*Software architecture for hard real-time applications: cyclic executives vs. fixed priority executives*." Real-Time Syst. 4(1): 37-53.

Lu, C., Stankovic, J., Tao, G. and Son, S. H. (1999). "*Design and evaluation of a feedback control EDF scheduling algorithm*". Proceedings of the 20th IEEE Real-Time Systems Symposium.

Ludemann, C. A. (1983). "*A microprocessor multi-task monitor*." IEEE Transactions on Nuclear Science 30(5): 3858 - 3863.

Maaita, A. and Pont, M. J. (2005a). "*Using 'planned pre-emption' to reduce levels of task jitter in a time-triggered hybrid*". In: Koelmans, A., Bystrov, A., Pont, M.J., Ong, R. and Brown, A. (Eds.), Proceedings of the Second UK Embedded Forum (Birmingham, UK, October 2005), Published by University of Newcastle upon Tyne [ISBN: 0-7017-0191-9]. .

Maaita, A. and Pont, M. J. (2005b). "*Using 'planned pre-emption' to reduce levels of task jitter in a time-triggered hybrid*". In: Koelmans, A., Bystrov, A., Pont, M.J., Ong, R. and Brown, A. (Eds.), Proceedings of the Second UK Embedded Forum (Birmingham, UK, October 2005), Published by University of Newcastle upon Tyne [ISBN: 0-7017-0191-9].

Maina, F. K. and Saidane, L. A. (2006). "*Probabilistic QoS guarantees with FP/EDF scheduling and packet discard in a real time context : A comparative study of local deadline assignment techniques*". proceedings of the 6th IEEE International workshop on IP operations and management, IPOM 2006, Dublin, Ireland.

Marwedel, P. (2006). "*Embedded system design*", Springer.

Mitchell, M. (1998). "*An introduction to genetic algorithms* ", MIT Press.

Mok, A. K. (1983). "*Fundamental design problems of distributed systems for the hard real-time environment*". Department of Electrical Engineering and Computer Science. Cambridge, MA, USA, MIT. PhD thesis.

Moore, S. and Simon, J. L. (2000). "*It's getting better all the time: 100 greatest trends of the 20th century* ", Cato Institute, U.S.

Musser, M. S. (1995). "*Faster, better, cheaper, how?: An interview with Domenick J. Tenerelli*." Mercury Magazine 24(4): 12-16.

Mwelwa, C. (2006). "*Development and assessment of a CASE tool to support the design and implementation of time-triggered embedded systems*,". Embedded Systems Laboratory, University of Leicester. PhD thesis.

Mwelwa, C., Athaide, K., Mearns, D., Pont, M. J. and Ward, D. (2006). "*Rapid software development for reliable embedded systems using a pattern-based code generation tool*". The Society of Automotive Engineers (SAE) World Congress, SAE document number: 2006-01-1457. Appears in: Society of Automotive Engineers (Ed.) "In-vehicle software and hardware systems", Published by Society of Automotive Engineers. [ISBN: 0-7680-1763-7], Detroit, Michigan, USA.

Mwelwa, C., Pont, M. J. and Ward, D. (2003). "*Towards a CASE tool to support the development of reliable embedded systems using design patterns*". In: Bruel, J-M [Ed.] Proceedings of the 1st International Workshop on Quality of Service in Component-Based Software Engineering, Published by Cepadues-Editions, Toulouse. ISBN: 2-85428-617-0, Toulouse, France.

Mwelwa, C., Pont, M. J. and Ward, D. (2004). "*Code generation supported by a pattern-based design methodology*". In: Koelmans, A., Bystrov, A. and Pont, M.J. (Eds.) Proceedings of the UK Embedded Forum 2004, Published by University of Newcastle upon Tyne, Birmingham, UK.

Mwelwa, C., Pont, M. J. and Ward, D. (2005). "*Developing reliable embedded systems using a pattern-based code generation tool: A case study*". In: Koelmans, A., Bystrov, A., Pont, M.J., Ong, R. and Brown, A. (Eds.), Proceedings of the Second UK Embedded Forum, Published by University of Newcastle upon Tyne, Birmingham, UK.

Nahas, M. (2009). "*Bridging the gap between scheduling algorithms and scheduler implementations in time-triggered embedded systems*". Department of Engineering. Leicester, Uinted Kingdom, University of Leicester. PhD thesis.

Nett, E., Streich, H., Bizzarri, P., Bondavalli, A. and Tarini, F. (1996). "*Adaptive software fault tolerance policies with dynamic real-time guarantees*". WORDS 96, IEEE Second Int. Workshop on Object-oriented Real-time Dependable Systems, Laguna Beach, California, U.S.A.

Oh, S. H. and Wu, C. (2004). "*Genetic-algorithm-based real-time task scheduling with multiple goals*." Journal of Systems and Software 71(3): 245 - 258.

Oh, S. H. and Yang, S. M. (1998). "*A modified least-laxity-first scheduling algorithm for real-time tasks*". the 5th International Workshop on Real-Time Computing Systems and Applications (RTCSA '98),, Hiroshima, Japan.

Phatrapornnant, T. (2007). "*Reducing jitter in embedded systems employing a time-triggered software architecture and dynamic voltage scaling*". Department of Engineering. Leicester, United Kingdom, University of Leicester. PhD thesis.

Phatrapornnant, T. and Pont, M. J. (2006). "*Reducing jitter in embedded systems employing a time-triggered software architecture and dynamic voltage scaling*." IEEE Transactions on Computers 55(2): 113-124.

Philips (2004a). "*LPC2119/2129/2194/2292/2294; Single-chip 32-bit microcontrollers user manual*".

Philips (2004b). "*NXP LPC2104, LPC2105, LPC2106 data sheet*", Data sheet, Philips Semiconductors.

Pont, M. J. (2001). "*Patterns for time-triggered embedded systems*", Addison-Wesley.

Pont, M. J., Kurian, S. and Bautista-Quintero, R. (2006). "*Meeting real-time constraints using 'Sandwich Delays'*". 11th European Conference on Pattern Languages of Programs (EuroPLoP 2006), Germany.

Pont, M. J. and Ong, R. H. L. (2002). "*Using watchdog timers to improve the reliability of single-processor embedded systems: Seven new patterns and a case study”*". Hruby, P. and Soressen, K. E. [Eds.] Proceedings of the First Nordic Conference on Pattern Languages of Programs.

Puschner, P. and Burns, A. (2002a). "*Transforming execution-time boundable code into temporally predictable code*". Bernd Kleinjohann, K.H. (Kane) Kim, Lisa Kleinjohann, and Achim Rettberg, editors, Design and Analysis of Distributed Embedded Systems,. Kluwer Academic Publishers, 2002. IFIP 17th World Computer Congress - TC10 Stream on Distributed and Parallel Embedded Systems (DIPES 2002).

Puschner, P. and Burns, A. (2002b). "*Writing temporally predictable code*". Proceedings of the 7th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems.

Puschner, P. and Burns, A. (2003). "*The single-path approach towards WCET-analysable software*". Proc. IEEE International Conference on Industrial Technology Maribor, Slovenia.

Puschner, P. and Kirner, R. (2006). "*From time-triggered to time-deterministic real-time systems*". 5th IFIP Working Conference on Distributed and Parallel Embedded Systems (DIPES 2006).

Radcliffe, N. and Wilson, G. (1990). "*Natural solutions give their best.*" New Scientist: 47-50.

Reeves, G. (1997). "*What really happened on Mars?, Authoritative Account, research.microsoft.com.*"

Rochange, C. and Sainrat, P. (2002). "*Difficulties in computing the WCET for processors with speculative execution*". 2nd International Workshop on Worst Case Execution Time Analysis, Vienna.

Sandström, K. and Norström, C. (2002). "*Managing complex temporal requirements in real-time control systems*". 9th IEEE Conf. Engineering of Computer-Based Systems, Sweden.

Sandström, K., Norström, C. and Fohler, G. (1998). "*Handling interrupts with static scheduling in an automotive vehicle control system*". Proc. 5th Int. Conf. on Real-Time Computing Systems and Applications IEEE Computer Society.

Scheler, F. and Schröder-Preikschat, W. (2006). "*Time-triggered vs. event-triggered: A matter of configuration?*". Dulz, Winfried; Schröder-Preikschat, Wolfgang : MMB Workshop Proceedings (GI/ITG Workshop on Non-Functional Properties of Embedded Systems Nuremberg, Berlin: VDE Verlag, 107—112, ISBN 978-3-8007-2956-2.

Schild, M. and Würtz, J. (1998). "*Off-line scheduling of a real-time system*". Proceedings of the 1998 ACM symposium on Applied Computing, Atlanta, Georgia, United States.

Schild, M. and Würtz, J. (2000). "*Scheduling of time-triggered real-time systems*." Constraints 5(4): 335-357.

Schlindwein, F. S., Smith, M. J. and Evans, D. H. (1988). "*Spectral analysis of doppler signals and computation of the normalized first moment in real time using a digital signal processor,* ." Medical & Biological Engineering & Computing, 26: 228-232.

Sha, L., Rajkumar, R. and Lehoczky, J. P. (1990). "*Priority inheritance protocols: an approach to real-time synchronization*." IEEE Transactions on Computers 39(9): 1175 - 1185.

Shaw, A. C. (2001). "*Real-time systems and software*". New York, John Wiley & Sons Inc.

Short, M. J. and Pont, M. J. (2007). "*Fault-tolerant time-triggered communication using CAN*." IEEE Transactions on industrial informatics 3(2): 131-142.

Short, M. J. and Pont, M. J. (2008). "*Assessment of high-integrity embedded automotive control systems using hardware in the loop simulation*." Journal of Systems and Software 81(7): 1163-1183.

Silberschatz, A. and Galven, P. B. (1998). "*Operating systems concepts*", Addison-Wesley, Boston, MA, USA.

Sommerville, I. (2007). "*Software engineering*". Essex, England, Pearson Education.

Stankovic, J. A. (1988). "*Misconceptions about real-time computing: a serious problem for next-generation systems*." IEEE Computers 21(10): 10 - 19.

Stankovic, J. A., Lu, C., Son, S. H. and Tao, G. (1999). "*The case for feedback control real-time scheduling*". Proceedings of the 11th Euromicro Conference on Real-Time Systems, York, UK.

Stankovic, J. A. and Ramamritham, K. (1987). "*The design of the spring kernel*". Proc. of the IEEE Real-Time Systems Symposium, Los Alamitos, CA, USA.

Stankovic, J. A., Spuri, M., Natale, M. D. and Buttazzo, G. (1995). "*Implications of classical scheduling results for real-time systems*." IEEE Computer 28(6): 16-25.

Theiling, H., Ferdinand, C. and R., W. (2000). "*Fast and precise WCET prediction by separated cache and path analyses*." Real-Time Systems 18(2-3): 157 - 179.

Thesing, S. (2004). "*Safe and Precise WCET Determination by Abstract Interpretation of Pipeline Models.* .", Universitat des Saarlandes. PhD thesis.

Tindell, K. W. (1994). "*Adding time-offsets to schedulability analysis*". York, England, Real-Time Systems Research Group; Department of Computer Science; University of York: 94 - 221.

Tindell, K. W., Burns, A. and Wellings, A. J. (1992). "*Allocating hard real-time tasks: an NP-hard problem made easy*." Real-Time Systems 4(2): 145 - 165.

Turley, J. (1999). "*Embedded processors by the numbers*." Embedded Systems Programming 12(5).

Vallerio, K. S. and Jha, N. K. (2003). "*Task graph extraction for embedded system synthesis*". Proc. 16th Int. Conference on VLSI Design concurrently with the 2nd International Conference on Embedded Systems Design, New Delhi, India, IEEE Computer Society, Washington, DC.

Volz, R. A. and Mudge, T. N. (1987). "*Instruction level timing mechanisms for accurate real-time task scheduling*." IEEE Transactions on Computers C-36(8): 988 - 993.

Ward, N. J. (1991). "*The static analysis of a safety-critical avionics control system*". in Corbyn, D.E. and Bray, N. P. (Eds.) "Air Transport Safety: Proceedings of the Safety and Reliability Society Spring Conference, 1991" Published by SaRS, Ltd.

Xu, J. (1993). "*Multiprocessor scheduling of processes with release times, deadlines, precedence, and exclusion relations*." IEEE Transactions on Software Engineering 19(2): 139-154.

Xu, J. (2003). "*Making software timing properties easier to inspect and verify*." IEEE Software 20(4): 34-41.

Xu, J. and Parnas, D. L. (1990). "*Scheduling processes with release times, deadlines, precedence and exclusion relations*." IEEE Transactions on Software Engineering 16(3): 360-369.

Xu, J. and Parnas, D. L. (1992). "*Pre-run time scheduling processes with exclusion relations on nested or overlapping critical sections*". 11th IEEE Int. Phoenix Conf. Computers and Communications, Scottsdale, AZ, USA.

Xu, J. and Parnas, D. L. (1993). "*On satisfying timing constraints in hard real-time systems*." IEEE Transactions on Software Engineering 19(1): 70 - 84.

Xu, J. and Parnas, D. L. (2000). "*Priority scheduling versus pre-run-time scheduling*." Real-Time Systems 18(1): 7-23.

Zamorano, J., Alonso, A. and de la Puente, J. A. (1997). "*Building safety critical real-time systems with reusable cyclic executives*." Control Engineering Practice 5(7): 999 -1005.

Zurawski, T. (2005). "*Embedded systems handbook*", CRC Press, Boca Raton, FL, USA.