Reducing jitter in embedded systems employing a time-triggered software architecture and dynamic voltage scaling

Thesis submitted for the degree of
Doctor of Philosophy
at the University of Leicester

by

Teera Phatrapornnant

B.Eng., M.Eng. (Bangkok)

Department of Engineering
University of Leicester
Leicester, United Kingdom

May 2007

UMI Number: U227747

All rights reserved

INFORMATION TO ALL USERS
The quality of this reproduction is dependent upon the quality of the copy submitted.

In the unlikely event that the author did not send a complete manuscript
and there are missing pages, these will be noted. Also, if material had to be removed,
a note will indicate the deletion.

UMI U227747
Published by ProQuest LLC 2013. Copyright in the Dissertation held by the Author.
Microform Edition © ProQuest LLC.
All rights reserved. This work is protected against
unauthorized copying under Title 17, United States Code.

ProQuest LLC
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106-1346

Reducing jitter in embedded systems employing a time-triggered software architecture and dynamic voltage scaling

Teera Phatrapornnant

Abstract

This thesis is concerned with the development of single-processor embedded systems in which there are requirements for both low CPU energy consumption and low levels of task jitter. The focus of the work is on ways in which dynamic voltage scaling (DVS) techniques can be incorporated in simple time-triggered scheduling algorithms in order to meet these constraints.

Following a review of previous work in this area, a presentation is made which illustrates the impact of a naive application of DVS in a system incorporating a time-triggered co-operative (TTC) scheduler. Novel algorithms (TTC-jDVS, TTC-jDVS2) are then introduced which more successfully integrate TTC and DVS techniques. These algorithms involve: (i) changes to system timer settings when the frequency is altered; (ii) use of a form of "sandwich delay" to reduce the impact of changes to the scheduler overhead which arise as a result of frequency changes, and (iii) execution of jitter-sensitive tasks at a fixed operating frequency.

The impact of these algorithms on both jitter and energy consumption is illustrated empirically on a representative hardware platform, using both "dummy" task sets and a more realistic case study.

In designs for which low jitter is an important consideration, at least a limited degree of task pre-emption may be required. A simple time-triggered hybrid (TTH) scheduler can be used to achieve such behaviour. A novel TTH scheduling algorithm (TTH-jDVS) is presented and evaluated, again through use of dummy task sets and a case study.

The third piece of experimental work presented in this thesis illustrates that – in situations where minimal jitter is required – hardware support is required. To illustrate the potential of such an approach a final case study is employed.

The thesis concludes by making suggestions for further work in this important area.

I dedicate this thesis to my parents

Samarn & Duangporn Phatrapornnant

Acknowledgements

First of all, I would like to thank my supervisor Dr. Michael J. Pont who has guided and encouraged me over the last four years. I have enjoyed working with him and am very grateful for all his wonderful support. Without him, this thesis would not have been possible.

I would like to express my gratitude to National Electronics and Computer Technology Center (NECTEC), particularly to Dr. Pansak Siriruchatapong and Dr. Kwan Sitathani, for their support and for awarding me the Royal Thai Government Scholarship.

Thanks to my colleagues in the Embedded Systems Laboratory for supporting me, especially Zemian Hughes who taught me to use the PH-processor. Also, many thanks to the Thai students with whom I shared many wonderful and enjoyable times.

Finally, I would like to specially thank my parents, sister, brother, and friends in Thailand for unflagging support from the beginning and throughout the final stages of thesis-writing.

Contents

LIST OF FIGURES IV

LIST OF TABLES VII

LIST OF LISTINGS VIII

LIST OF PUBILCATIONS IX

LIST OF ABBREVIATIONS X

LIST OF SYMBOLS XII

CHAPTER 1 INTRODUCTION 1

1.1 WHAT IS AN EMBEDDED SYSTEM? 1

1.2 THE GROWING CONCERN OF POWER CONSUMPTION IN PROCESSORS 3

1.3 THE IMPACT OF JITTER ON EMBEDDED SYSTEMS 6

1.4 AIMS OF THE THESIS 7

1.5 RESEARCH CONTRIBUTIONS 7

1.6 THESIS ORGANISATION 8

1.7 CONCLUSIONS 9

CHAPTER 2 ENERGY-EFFICIENT EMBEDDED SYSTEMS 10

2.1 INTRODUCTION 10

2.2 POWER DISSIPATION IN CMOS CIRCUITS 11

2.3 APPROACHES TO MINIMISE POWER IN CMOS PROCESSORS 14

2.4 ALGORITHMS FOR REDUCING ENERGY CONSUMPTION IN PROCESSORS 17

2.5 SYSTEM-LEVEL POWER REDUCTION 26

2.6 EXTENDING THE SERVICE LIFE OF MOBILE EMBEDDED SYSTEMS 29

2.7 METRIC FOR ENERGY EFFICIENCY 32

2.8 CONCLUSIONS 34

CHAPTER 3 LOW-JITTER SCHEDULING ALGORITHMS 35

3.1 INTRODUCTION 35

3.2 SOURCES OF JITTER 36

3.3 JITTER IN SCHEDULING SYSTEMS 41

3.4 LOW-JITTER SCHEDULING ALGORITHMS 45

3.5 CONCLUSIONS 53

CHAPTER 4 IMPLEMENTATION OF DVS IN A TTC SCHEDULER 54

4.1 THE HARDWARE PLATFORM 54

4.2 A TTC SCHEDULER 57

4.3 APPLYING DVS IN A TTC SCHEDULER 60

4.4 DVS ALGORITHMS 61

4.5 Assessing and comparing the DVS algorithms 68
4.6 The impact of DVS on system timing 76
4.7 The knock-on impact of frequency scaling 80
4.8 Conclusions 82

Chapter 5 Design and evaluation of a reduced-jitter TTC/DVS scheduler 83

5.1 Minimising jitter caused by DVS 83
5.2 The TTC-jDVS algorithm 84
5.3 Implementing the TTC-jDVS scheduler 88
5.4 Evaluating the TTC-jDVS algorithm 88
5.5 Impact of jitter on sampled data systems 95
5.6 Wireless ECG: A case study 96
5.7 Discussion 101
5.8 Conclusion 104

Chapter 6 Working with a hybrid scheduler 105

6.1 From TTC to TTH 105
6.2 A TTH Scheduler 106
6.3 Determining speed-settings in TTH designs 107
6.4 Implementing DVS in TTH systems 109
6.5 The TTH-jDVS algorithm 117
6.6 Evaluating the TTH-jDVS algorithm 119
6.7 Case study: Encrypted wireless ECG monitoring 124
6.8 Discussion 127
6.9 Conclusions 127

Chapter 7 Further reductions of jitter in TTC/DVS scheduler 129

7.1 The need for additional hardware 129
7.2 Previous work on hardware support for DVS 130
7.3 Including an independent timer in a TTC-jDVS2 design 130
7.4 Assessing the modified algorithms 132
7.5 Conclusions 137

Chapter 8 Discussion and conclusions 138

8.1 Introduction 138
8.2 Knock-on impact of DVS implementation 139
8.3 Minimising jitter in DVS systems 139
8.4 Limitations of the work 141
8.5 Potential applications of the work 142
8.6 Future work 142

8.7 CONCLUSIONS 144

APPENDIX A JITTER MODEL A-1

APPENDIX B REAL-TIME SCHEDULING ARCHITECTURES B-1

APPENDIX C THE TTC-JDVS2 ALGORITHM C-1

APPENDIX D INCORPORATING AN INDEPENDENT TIMER IN AN FPGA-BASED SOC DESIGN D-1

REFERENCES E-1

LIST OF FIGURES

FIGURE 1.1: A MOBILE 12-LEAD DIGITAL ECG. THIS IMAGE HAS BEEN USED WITH AUTHORISATION FROM DEL MAR REYNOLDS (DELMAR-REYNOLDS, 2002) 2
FIGURE 1.2: MICROPROCESSOR CURRENT AND VOLTAGE TRENDS IN INTEL MICROPROCESSORS. THIS IMAGE HAS BEEN USED WITH AUTHORISATION FROM INTEL TECHNOLOGY JOURNAL ("POWER DELIVERY FOR HIGH-PERFORMANCE MICROPROCESSORS", VOLUME 9, ISSUE 4, PAGE 274, 275) .. 4
FIGURE 2.1: DELAY, POWER VS V_{DD}. (REDRAWN FROM BURD AND BRODERSEN, 1995) 14
FIGURE 3.1: A SCHEMATIC REPRESENTATION OF JITTER. SEE TEXT FOR DETAILS........ 36
FIGURE 3.2: PRIMARY PARAMETERS OF A TASK 41
FIGURE 3.3: TASK PERIOD JITTERS (ADAPTED FROM MARTI, 2002, FIGURE 4.1)........ 44
FIGURE 3.4: GENERAL STRUCTURE OF TTC SCHEDULER: MINOR CYCLE = 10, MAJOR CYCLE = 40 46
FIGURE 3.5: STRUCTURE OF TTC SCHEDULER: MINOR CYCLE = 5, MAJOR CYCLE = 40 (ADAPTED FROM LOCKE, 1992, FIGURE 1). 47
FIGURE 3.6: STRUCTURE OF RATE MONOTONIC SCHEDULING (ADAPTED FROM LOCKE, 1992, FIGURE 3).49
FIGURE 3.7: ILLUSTRATING THE OPERATION OF A TTH SCHEDULER. SEE TEXT FOR DETAILS 52
FIGURE 4.1: SCHEMATIC CIRCUIT OF DVS POWER SUPPLY 55
FIGURE 4.2: SETTING THE CPU SUPPLY VOLTAGE (LPC 2106)........ 56
FIGURE 4.3: TTC SCHEDULING DIAGRAM 58
FIGURE 4.4: EXAMPLE ILLUSTRATING THE POSSIBILITY OF TASK STRETCHING 60
FIGURE 4.5: ILLUSTRATING A MORE REALISTIC DVS IMPLEMENTATION 61
FIGURE 4.6: CONSIDERING INTRA-TASK UTILISATION........ 64
FIGURE 4.7: THE BOUNDARY OF UTILISATION BASED ON 60 MHZ 64
FIGURE 4.8: A SCHEMATIC REPRESENTATION OF THE CIRCULAR ARRAY ALGORITHM........ 66
FIGURE 4.9: SCHEDULING OVERHEAD OF DVS ALGORITHMS ESTIMATION 69
FIGURE 4.10: MAXIMUM EXECUTION TIME OF TASK CONSIDERED AT MAXIMUM FREQUENCY 60MHZ.... 70
FIGURE 4.11: AVAILABLE EXECUTION TIME OF TASK – CD ALGORITHM 71
FIGURE 4.12: AVAILABLE EXECUTION TIME OF TASK – LT ALGORITHM 72
FIGURE 4.13: AVAILABLE EXECUTION TIME OF TASK – CA ALGORITHM 72
FIGURE 4.14: AVAILABLE EXECUTION TIME OF TASK – CS ALGORITHM – EXCLUDED VOLTAGE/FREQUENCY SCALING 73
FIGURE 4.15: AVAILABLE EXECUTION TIME OF TASK – CS ALGORITHM – PRACTICAL 73
FIGURE 4.16: POWER CONSUMPTION OF CPU CORE ON DIFFERENT SCHEDULING ALGORITHMS........ 75
FIGURE 4.17: HISTOGRAM OF TICK JITTER IN TTC-DVS RUN WITH RANDOM FREQUENCY 77
FIGURE 4.18: HISTOGRAM OF TICK JITTER IN TTC........ 77
FIGURE 4.19: HISTOGRAM OF TASK JITTER IN TTC-DVS RUN WITH RANDOM FREQUENCY 78
FIGURE 4.20: HISTOGRAM OF TASK JITTER IN TTC........ 78
FIGURE 4.21: TOTAL JITTER OF TASK IN TTC-DVS AND TTC SYSTEMS 79

Figure 4.22: Tick drift in DVS systems. (Phatrapornnant and Pont 2006, IEEE Transactions on Computer, Vol.55(2), © 2006 IEEE) 80
Figure 4.23: Sampling jitter caused by frequency scaling. (Phatrapornnant and Pont 2006, IEEE Transactions on Computer, Vol.55(2), © 2006 IEEE) 81
Figure 4.24: Release jitter caused by variation of scheduling overhead. (Phatrapornnant and Pont 2006, IEEE Transactions on Computer, Vol.55(2), © 2006 IEEE) 81
Figure 5.1: Voltage and frequency scaling steps. (Phatrapornnant and Pont 2006, IEEE Transactions on Computer, Vol.55(2), © 2006 IEEE) 85
Figure 5.2: Minimising release jitter through use of a "jitter guardian". (Phatrapornnant and Pont 2006, IEEE Transactions on Computer, Vol.55(2), © 2006 IEEE) 86
Figure 5.3: Minimising sampling jitter by fixed running speed. (Phatrapornnant and Pont 2006, IEEE Transactions on Computer, Vol.55(2), © 2006 IEEE) 87
Figure 5.4: Tick jitter in TTC-DVS. (Phatrapornnant and Pont 2006, IEEE Transactions on Computer, Vol.55(2), © 2006 IEEE) 90
Figure 5.5: Tick jitter in TTC-jDVS. (Phatrapornnant and Pont 2006, IEEE Transactions on Computer, Vol.55(2), © 2006 IEEE) 90
Figure 5.6: Histogram of tick jitter in TTC-DVS. (Phatrapornnant and Pont 2006, IEEE Transactions on Computer, Vol.55(2), © 2006 IEEE) 91
Figure 5.7: Histogram of tick jitter in TTC-jDVS. (Phatrapornnant and Pont 2006, IEEE Transactions on Computer, Vol.55(2), © 2006 IEEE) 91
Figure 5.8: Minimum and maximum jitter level of RJT at speed 10-60 MHz run by TTC-jDVS, TTC-DVS and TTC. (Phatrapornnant and Pont 2006, IEEE Transactions on Computer, Vol.55(2), © 2006 IEEE) 92
Figure 5.9 : CPU power consumption comparison of scheduling algorithms at different load. (Phatrapornnant and Pont 2006, IEEE Transactions on Computer, Vol.55(2), © 2006 IEEE) 94
Figure 5.10: A schematic representation of a system for ECG monitoring. 97
Figure 5.11: Histogram of period jitter for ECG study (TTC). (Phatrapornnant and Pont 2006, IEEE Transactions on Computer, Vol.55(2), © 2006 IEEE) 98
Figure 5.12: Histogram of period jitter for ECG study (TTC-DVS). (Phatrapornnant and Pont 2006, IEEE Transactions on Computer, Vol.55(2), © 2006 IEEE) 99
Figure 5.13: Histogram of period jitter for ECG study (TTC-jDVS). (Phatrapornnant and Pont 2006, IEEE Transactions on Computer, Vol.55(2), © 2006 IEEE) 99
Figure 5.14: The comparison of CPU core power consumption of TTC, TTC-DVS and TTC-jDVS. (Phatrapornnant and Pont 2006, IEEE Transactions on Computer, Vol.55(2), © 2006 IEEE) 100
Figure 6.1: TTH scheduling diagram 106
Figure 6.2: Example illustrating the use task stretching in a TTH scheduler 108
Figure 6.3: Function call tree for the TTH scheduler (normal operation) 109
Figure 6.4: Realistic DVS implementation 111

FIGURE 6.5: LOAD CHARACTERISTICS OF A PHILIPS LPC2106 PROCESSOR (ARM7TDMI CORE) WITH 6 SPEED STEPS 113
FIGURE 6.6: UTILISATION OF TASKS IN ONE TASK SLOT 114
FIGURE 6.7: SLACK TIME PRESENTED AFTER APPLYING RUNNING SPEEDS OF AN EXAMPLE: E_P = 0.1MS, E_C = 0.3MS, T_{SLOT} = 2MS, AND R_{INT} = 1MS 114
FIGURE 6.8: CPU POWER CONSUMPTION: (A) PREDICTED BY USING POWER MODEL, (B) MEASURED OF AN EXAMPLE: E_P = 0.1MS, E_C = 0.3MS, T_{SLOT} = 2MS, AND R_{INT} = 1MS 115
FIGURE 6.9: REDUCING RELEASE TASK JITTER BY JITTER GUARDIAN 118
FIGURE 6.10: REDUCING SAMPLING PERIOD JITTER BY FIXED RUNNING SPEED 119
FIGURE 6.11: HISTOGRAM OF TICK JITTER IN TTH-DVS 121
FIGURE 6.12: HISTOGRAM OF TICK JITTER IN TTH-JDVS 121
FIGURE 6.13: MINIMUM AND MAXIMUM RELEASE JITTER LEVEL OF PRE-EMPTIVE TASK AT SPEED 10-60 MHZ RUN BY TTH-JDVS, TTH-DVS AND TTH 122
FIGURE 6.14: POWER CONSUMPTION COMPARISON OF SCHEDULING ALGORITHMS AT DIFFERENT LOAD 124
FIGURE 7.1: USING A COTS TIMER 131
FIGURE 7.2: MINIMUM AND MAXIMUM JITTER LEVEL (FROM RJTS) AT SPEEDS 10-60 MHZ, FROM TTC-JDVS, TTC-JDVS2, TTC-JTDVS2 AND TTC (AT 60MHZ) ALGORITHMS 133
FIGURE 7.3: POWER CONSUMPTION COMPARISON OF SCHEDULING ALGORITHMS AT DIFFERENT LOAD.. 134

List of tables

TABLE 4.1: TASK SET PARAMETERS FOR ASSESSING POWER CONSUMPTION 74
TABLE 4.2: TICK JITTER AND TASK JITTER IN TTC-DVS AND TTC SYSTEMS 79
TABLE 5.1: COMPARING TICK JITTER RUN BY TTC-DVS, TTC-JDVS, AND TTC ALGORITHMS 89
TABLE 5.2: TASK SET PARAMETERS 92
TABLE 5.3: TASK SET PARAMETERS FOR ASSESSING POWER CONSUMPTION 93
TABLE 5.4: ECG TASK SET PARAMETERS 97
TABLE 6.1: POWER CONSUMPTION PREDICTIONS, MEASUREMENTS AND PREDICTION ERRORS 116
TABLE 6.2: COMPARING TICK JITTER RUN BY TTH-DVS, TTH-JDVS, AND TTH ALGORITHMS 120
TABLE 6.3: TASK SET PARAMETERS 122
TABLE 6.4: TASK SET PARAMETERS FOR ASSESSING POWER CONSUMPTION 123
TABLE 6.5: ECG TASK SET PARAMETERS 125
TABLE 6.6: TICK AND RELEASE JITTER MEASURED 126
TABLE 7.1: COMPARING TICK JITTER FROM THE TTC, TTC-JDVS, TTC-JDVS2 AND TTC-JTDVS2 ALGORITHMS 133

List of listings

LISTING 4.1: AN OVERVIEW OF POSSIBLE TTC SCHEDULER IMPLEMENTATION 58
LISTING 4.2: PSEUDO CODE OF THE UPDATE FUNCTION IN THE TTC DESIGN 59
LISTING 4.3: TASK DISPATCHER OF TTC ALGORITHM 59
LISTING 4.4: COMPUTE-DIRECT ALGORITHM...63
LISTING 4.5: LOOKUP TABLE ALGORITHM...65
LISTING 4.6: CIRCULAR ARRAY ALGORITHM...67
LISTING 4.7: CIRCULAR SKIP ALGORITHM 68
LISTING 5.1: SETTING THE EXECUTION SPEED OF "REDUCED-JITTER" TASKS (RJTS) 87
LISTING 5.2: DISPATCHING TASKS IN TTC-JDVS 88
LISTING 6.1: PSEUDO CODE OF THE UPDATE FUNCTION IN THE TTH SCHEDULER..................................107
LISTING 7.1: PSEUDO CODE OF FREQUENCY SCALING FUNCTION OF TTC-JTDVS2 135
LISTING 7.2: PSEUDO CODE OF FREQUENCY SCALING FUNCTION OF TTC-JDVS2 WITH TICK COMPENSATION..136

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.

Phatrapornnant, T. and Pont, M.J. (submitted a) "Reducing task jitter in resource-constrained embedded systems in which limited pre-emption is required and DVS is employed", Journal of Systems Architecture.

Phatrapornnant, T. and Pont, M.J. (submitted b) "Enhancements to an algorithm which reduces jitter in embedded systems employing a time-triggered software architecture and dynamic voltage scaling" Journal of Systems Architecture.

Pont, M.J., Kurian, S., Wang, H. and Phatrapornnant, T. (in press) "TTC Scheduler: An 'abstract' pattern for use with resource-constrained embedded systems" Paper to be presented at EuroPLoP 2007, Germany, July 2007.

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 (Special Issue on Design and Test of Systems-On-a-Chip), 55 (2), pp.113-124.

Phatrapornnant, T. and Pont, M.J. (2004a) "The application of dynamic voltage scaling in embedded systems employing a TTCS software architecture: A case study", Proceedings of the IEE / ACM Postgraduate Seminar on "System-On-Chip Design, Test and Technology", Loughborough, UK, 15 September 2004. Published by IEE. ISBN: 0 86341 460 5 (ISSN: 0537-9989), pp.3-8.

Phatrapornnant, T. and Pont, M.J. (2004b) "The application of dynamic voltage scaling in embedded systems employing a TTCS software architecture". In: Koelmans, A., Bystrov, A. and Pont, M.J. (Eds.) Proceedings of the UK Embedded Forum 2004 (Birmingham, UK, October 2004), pp.127-143. Published by University of Newcastle upon Tyne [ISBN: 0-7017-0180-3].

List of abbreviations

General terms

ABB	Adaptive Reverse Body Biasing
ACPI	Advanced Configuration and Power Interface
ADC	Analogue-to-Digital Converter
ALU	Arithmetic Logic Unit
ATM	Asynchronous Transfer Mode
BEE	Battery Energy Efficient
BER	Bit Error Rate
CAN	Computer Area Network
CMOS	Complementary Metal–Oxide–Semiconductor
CPU	Central Processing Unit
DAC	Digital-to-Analogue Converter
DC	Direct Current
DPM	Dynamic Power Management
DTMDP	Discrete-Time MarKov Decision Process
DVS	Dynamic Voltage Scaling
ECG	Electrocardiogram
EMI	Electromagnetic Interference
ET	Event-Triggered
FPGA	Field Programmable Gate Array
IF	Intermediate Frequency
ISR	Interrupt Service Routine
ITU	International Telecommunication Union
LF	Loop Filter
MIPJ	Millions Instructions Per Joule
MIPS	Millions Instructions Per Second
MPEG	Moving Picture Experts Group
MTCMOS	Multi-Threshold CMOS
NMOS	N-type Metal–Oxide–Semiconductor
PADWC	Programmable Active Datapath Width Control
PCR	Program Clock Reference
PD	Phase Detector
PLL	Phase-Locked Loop
PMOS	P-type Metal–Oxide–Semiconductor
QAM	Quadrature Amplitude Modulation
QPSK	Quadrature Phase-Shift Keying
RJT	Reduced-Jitter Task
SMDP	Continuous-time Semi-Markov Decision Process
SoC	System-on-Chip
SPI	Serial-Peripheral-Interface
TT	Time-Triggered
UART	Universal Asynchronous Receiver/Transmitter
VCO	Voltage-Controlled Oscillator
VLSI	Very-Large-Scale Integration
WCET	Worst-Case Execution Time

Scheduling algorithms

CA	Circular Array
CD	Compute-Direct
CS	Circular Skip
EDF	Earliest-Deadline-First
E-LEDF	Extended Low-energy Earliest-Deadline-First
HRR	Hierarchical Round Robin
Jitter-EDD	Jitter-Earliest-Due-Date
LC-EDF	Leakage Control EDF
LEDF	Low-energy Earliest-Deadline-First
LEDES	Low-Energy Device Scheduler
LT	Lookup Table
MUSCLES	Multi-State Constrained Low-Energy Scheduler
PACE	Processor Acceleration to Conserve Energy
RCSP	Rate-Controlled Static Priority
RM	Rate Monotonic
TTC	Time-Triggered Co-operative
TTC-DVS	TTC with Dynamic Voltage Scaling
TTC-jDVS	TTC with Dynamic Voltage Scaling (Reduced-jitter implementation)
TTH	Time-Triggered Hybrid
TTH-DVS	TTH with Dynamic Voltage Scaling
TTH-jDVS	TTH with Dynamic Voltage Scaling (Reduced-jitter implementation)

List of symbols

α_i	Switching activity of each node
β	Measure of velocity saturation
ms	Millisecond
µs	Microsecond
C_L	Node capacitance
C_{eff}	Average switched capacitance per cycle
d_i	Deadline
e_i	Execution time
e_C	Duration of the co-operative task
e_P	Duration of the pre-emptive task
E_{op}	Energy-per-operation
f	Clock frequency
f_{max}	Maximum clock frequency
f_{min}	Minimum clock frequency
f_C	Clock frequency for the running co-operative task
f_P	Clock frequency for the running pre-emptive task
f_i	Finishing time
I_j	Source and drain body junction leakage current
I_{subn}	Subthreshold leakage current
K	Technology constant
L_d	Logic depth of the path
MHz	Megahertz
n_{int}	Number of interruption in the task slot
n_{sw}	Number of voltage/frequency switching per interruption
P_{AC}	Dynamic power consumption
P_{DC}	Static power consumption
P_{freq}	Power function of CPU which running at clock frequency *freq*
P_{total}	Total power consumption of the circuit
r_i	Release time (arrival time)
R_{int}	Interruption rate
s_i	Start time
sch_C	Scheduling overhead to schedule co-operative tasks
sch_P	Scheduling overhead to schedule pre-emptive tasks
T_i	Period
T_{slot}	Duration of a given task slot
t_{inv}	Cycle time
t_{sw}	Voltage/frequency switching time
u_P	Utilisation of pre-emptive tasks in the given task slot
u_C	Utilisation of co-operative tasks in the given task slot
ΔV	Voltage change
V_{bs}	Body bias voltage
V_{DD}	Supply voltage
V_T	Thermal voltage
V_{th}	Threshold voltage
$WCET_C$	Worst case execution time of the co-operative task
$WCET_P$	Worst case execution time of the pre-emptive task

Chapter 1

Introduction

In this introductory chapter, an overview of the work undertaken in this thesis is presented and the importance of this area is discussed.

1.1 What is an embedded system?

A general-purpose computer system typically has a keyboard, mouse, disk, and graphical display, and can be programmed for a wide variety of purposes (for example, word processing, electronic mail, business accounting, scientific computing, and database systems). The user of a such computer is able to add or remove application software in the system and decide which application to launch or terminate (Valvano, 2000). By contrast, an embedded system will often use a microprocessor like the Motorola 6805, Intel 8051, Microchip PIC16F77 or Intel SA-1100, with simple I/O interface, such as switches, small keypads, LEDs and so on, to interact with the user. Such an embedded system is often configured to perform a specifically dedicated application and its software, which typically solves only a limited range of problems, is fixed into some form of ROM (Read Only Memory) that is not usually accessible to the user (Ball, 1996; Valvano, 2000).

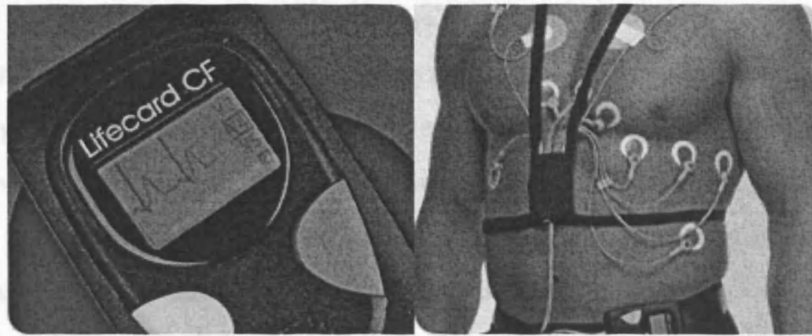

Figure 1.1: A mobile 12-lead digital ECG. This image has been used with authorisation from Del Mar Reynolds (Delmar-Reynolds, 2002)

Embedded systems are encountered regularly in everyday life. One example is a simple household appliance, such as a washing machine that uses a single microcontroller containing the different washing programs to control various motors and pumps, and even display the progress of washing operation (Heath, 1997; Agrawal and Bhatt, 2001). Other – rather more complex – systems also include embedded processors. For example a modern car that may have over 100 microprocessors linked as a network for controlling functions: engine through an electronic controller unit (ECU), brakes with electronic anti-lock brakes, transmission with traction control and electronically controlled gearboxes, safety with airbag systems, electric windows, and even entertainment systems (Heath, 1997; Leen *et al.*, 1999; Pop *et al.*, 2004). Furthermore, embedded processors can be found in other application areas, such as aircraft electronics, consumer electronics, aerospace, telecommunication, medical systems, military applications, smart buildings and so on (Marwedel, 2003).

The era of microprocessor-embedded system began in 1971 when Intel launched the world's first commercial single-chip microprocessor, a 4-bit CPU Intel 4004 (Intel, 2006b). The Intel 4004 was originally developed for the electronic calculators to replace many hundreds of discrete ICs. By using a microprocessor, functions can be changed by altering the program code, rather than analysing the system at gate level and modifying the hardware (Heath, 1997). This enabled the construction of calculators at a very low cost compared to the original systems.

In the past, an embedded system was normally based on a single-chip microcontroller (self-contained system with a processor, memory and peripherals) which had limited-

resources: for example, an original 8051 has 4 kbytes of ROM and 128 bytes of RAM (Intel, 1994). Today, modern embedded systems may be implemented using a wider range of processor platforms. For example, a TV "set-top box" may now contain a high-performance PowerPC processor with 32 MB of memory and 16 MB of Flash memory, that runs a multitasking operating system which simultaneously runs multiple applications, such as a video controller and a digital VCR (Hollabaugh, 2002). The advances in technology that increase hardware performance and the lower cost of that hardware have changed the form of embedded systems. In addition, a wide range of embedded applications are now implemented on system-on-chip (SoC), multiple devices (e.g. processors, memory, analog and mixed-signal circuitry for I/O) integrated into a single chip, in order to reduce costs (Jerraya and Wolf, 2005).

The success of embedded systems does not depend only on the hardware but also on software. The designers of software for embedded processors are often particularly concerned about system timing. For example, in an aircraft autopilot system, there is a need to process input and generate output on a time scale measured in milliseconds, and a slightly delay in making changes to the rudder setting (for example) may cause the plane to oscillate very unpleasantly or have more severe problems (Pont, 2001).

1.2 The growing concern of power consumption in processors

There is increasing interest in the development of mobile devices based on embedded processors. Examples include mobile phones, PDAs and MP3 players. Worldwide sales of mobile phones alone were close to 520 million units in 2003 (Gartner, 2004).

The demand for such mobile devices is not only for growing numbers of devices, but also for an increased range of features on existing devices. For example, mobile phones are – in many cases – also operating as personal digital assistants (PDAs) and / or providing links to email, chat services and general WWW access. Entertainment applications are now also an important feature of advanced mobile phones: the user can download and play MP3 music, watch video clips, or play games.

Growth in mobile embedded processors is by no means restricted to mobile phones. Even in medical applications, mobility is preferred in many cases. For example,

modern mobile-ECG equipment (see Figure 1.1) is required to continuously record heartbeat signals of the patient for up to 7 days, in order to allow non-symptomatic events to be captured over extended monitoring periods. A week of such "full disclosure" ECG data enables early diagnosis of patients with infrequent arrhythmias, including the probability of apnoea, before they become an everyday occurrence (Delmar-Reynolds, 2002).

With the latest 65 nm process technologies, processors now run at clock frequencies greater than 3 GHz. Having a billion transistors will shortly become possible (Bai *et al.*, 2004). Such high speed processors can match the performance demands of modern applications. However, power and thermal management become significant issues. The 1.6 GHz Intel Itanium 2 (Intel, 2006a), the high performing processor for server system that has 410 million transistors, dissipates 130 watts from its small die. In Figure 1.2 the current and voltage graphs illustrate the power trends in microprocessors.

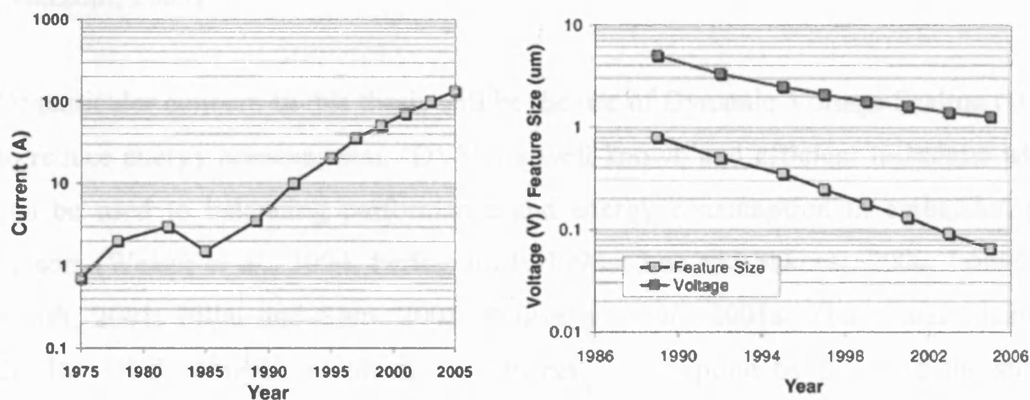

Figure 1.2: Microprocessor current and voltage trends in Intel microprocessors. This image has been used with authorisation from Intel Technology Journal ("Power Delivery for High-Performance Microprocessors", Volume 9, Issue 4, Page 274, 275)

With any such mobile device, power consumption becomes a crucial factor. While the power demand is growing from high computation (e.g. multimedia-features in mobile phones) and / or long-term use (e.g. mobile ECG recording), the size of batteries is becoming smaller, due to demands for smaller devices.

If he or she aims is to extend the service life of mobile systems, the designer has two (non-exclusive) options. The first is to find ways of increasing the storage capacity of batteries. The second is to reduce the energy consumption in the embedded application. Both areas present significant challenges.

The focus in this thesis will be on reduction in system energy consumption. Such energy reductions can be attempted at various levels. At the application level, energy consumption can be minimised by providing alternative functions that do a similar job but require less energy. For example, the user can be provided with the option of lower-quality pictures and audio quality on a multi-media player. One level below this, at a system level, designers can investigate both hardware and software design approaches dedicated to the specific-application. One level lower still (at a behavioural-level), designers can select the proper algorithm, energy or speed optimised, to the right job. At the physical level (the lowest level is), energy-saving options are limited to VLSI floorplanning and place-and-route techniques, and it has been argued that opportunities for energy saving at higher levels are 10 to 100 times greater (Mazzoni, 2003).

Of particular concern in this thesis will be the use of Dynamic Voltage Scaling (DVS) to reduce energy consumption. DVS is a well-known and efficient technique which can be used to balancing performance and energy consumption in embedded processors (Weiser *et al.*, 1994; Pering *et al.*, 1998a; Lee and Sakurai, 2000; Lorch and Smith, 2001; Pillai and Shin, 2001; Pouwelse *et al.*, 2001a; Zhang and Chanson, 2003). DVS achieves a reduction in energy consumption by lowering the supply voltage of the CPU when the full performance is not required. Technically, in a Complementary Metal Oxide Semiconductor (CMOS) circuit, if supply voltage is lowered, the delay in the circuit will increase (Burd and Brodersen, 1995). In the other word, the maximum frequency of the circuit decreases. This means that the throughput or performance of the circuit also decreases. When using DVS, the functions that do not require full speed will be run at a lower clock speed (and also a lower voltage) in order to save energy, while the functions which need to be completed more quickly will be run at a higher speed. With proper processor clock speeds which are selected to ensure that they (just) complete their workloads in line

with the system requirements: DVS techniques can then save energy without degrading performance.

1.3 The impact of jitter on embedded systems

As previously noted (Section 1.1), energy consumption is not the only concern in embedded systems. In particular, embedded applications involving data sampling or data output (e.g. medical systems, general data-acquisition systems, numerous control systems) require precise control of timing (Buttazzo, 2004). In other words, if the system is delayed or does not respond to the event within their time constraints, it may cause system to fail or malfunction (Bennett, 1994). This delay can arise from many factors, such as process-computational time or communication between nodes.

One key timing factor which can have a serious impact on system behaviour is "jitter". In general, jitter may be defined as the deviation from the ideal timing of an event (Wavecrest, 2001; Ou *et al.*, 2004).

Jitter can have serious implications, not least in applications involving control or data acquisition. Cottet and David (1999) show that – during data acquisition tasks – jitter rates of 10% or more can introduce errors which are so significant that any subsequent interpretation of the sampled signal may be rendered meaningless. Similarly Jerri (1997) discusses the serious impact of jitter on applications such as spectrum analysis and filtering. Also, in control systems, jitter can greatly degrade the performance by varying the sampling period (Torngren, 1998; Marti *et al.*, 2001b).

Jitter can arise from many factors, including oscillator hardware (Schossmaier and Weiss, 1999), electromagnetic interference in a printed-circuit board (Bogatin and Garat, 2004), or slightly different calculation times for a control algorithm (Kopetz, 1997).

In embedded systems employing real-time schedulers, jitter can arise from the scheduler implementation. In general, a real-time scheduler is used to manage the time allocation for a set of tasks which are to be executed on a processor (or processors). In such a scheduler, jitter can – for example – arise when a task is blocked

while awaiting access to a shared resource (Audsley *et al.*, 1993a) or when a lower-priority task is interrupted by a higher-priority task.[1]

As noted previously, a key concern in this thesis will be with the use of DVS in embedded systems. Use of DVS requires variations in the CPU clock frequency. Clearly – unless care is taken – it would be reasonable to expect that levels of jitter would be likely to increase in embedded systems in which DVS is employed. The focus of the thesis will be on the impact of DVS on jitter in the starting times of periodic tasks.

1.4 Aims of the thesis

This research project was established with goal of developing and evaluating DVS-based scheduling algorithms which would serve to reduce CPU energy consumption without adding significantly to jitter levels.

The specific aims of the research project described in this thesis were as follows:

- To explore the knock-on impact of the application of DVS in single-processor embedded systems employing real-time scheduling architectures.
- To develop algorithms for reducing jitter in such embedded systems.
- To evaluate the above algorithms.

1.5 Research contributions

This thesis is concerned with the development of single-processor embedded systems in which there are requirements for both low CPU energy consumption and low levels of task jitter.

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

First, novel algorithms (TTC-jDVS, TTC-jDVS2) for reducing jitter in time-triggered, co-operatively scheduled embedded systems were developed. These algorithms

[1] Links between scheduler designs and jitter will be considered in more detail in Chapter 3.

involve: (i) changes to system timer settings when the frequency is altered; (ii) use of a form of "sandwich delay" to reduce the impact of changes to the scheduler overhead which arise as a result of frequency changes, and (iii) execution of jitter-sensitive tasks at a fixed operating frequency.

Second, a novel algorithm (TTH-jDVS) for reducing jitter in time-triggered "hybrid" scheduling architectures was developed. This algorithm includes a novel approach to determine processor speed setting by using power model in situations where task pre-emption takes place.

Third, it was demonstrated that – in situations where minimal jitter is required – hardware support is required. A design for suitable hardware was developed, along with an appropriate scheduling algorithm (TTC-jtDVS2).

1.6 Thesis Organisation

The remainder of this thesis is organised as follows:

Chapter 2 reviews background and previous works of energy-efficient embedded system designs. The power dissipation in CMOS circuits and approaches to minimise power consumption are described. The techniques for reducing energy consumption at processor and system level are reviewed, including the techniques for extending battery life. The energy metrics for indicating energy efficiency are briefly described.

Chapter 3 reviews about jitter and its impact on embedded applications and others related. Specifically, the time-varying effect resulting from scheduling algorithms and task jitter are discussed. The examples of low-jitter scheduling algorithms are presented.

Chapter 4 describes DVS implementation and its impact. The study is initiated by applying DVS in a time-triggered co-operative (TTC) scheduler. The possible approaches to implement DVS are explored. A number of DVS algorithms are described and evaluated for comparing their performances. The problems which arise from the DVS employing into a TTC scheduler are explored and described.

Chapter 5 describes approaches to minimise jitter caused by dynamic voltage scaling in the TTC-DVS algorithm. A new algorithm TTC-jDVS is demonstrated and then evaluated using an artificial task set and in a realistic case study.

Chapter 6 presents the alternative solutions where the TTC-jDVS design is not appropriate. This study explores ways to implement DVS in TTH design in order to minimise both energy consumption and jitter level. Experiments are then carried out to evaluate TTH-jDVS algorithm using both an artificial task set and in a realistic case study.

Chapter 7 presents the further reduction jitter techniques. The study demonstrates an approach of using hardware timer incorporated with TTC-jDVS algorithm to further reduce jitter.

Chapter 8 presents conclusions of the thesis and its contributions.

The appendices present: an overview of the jitter model (Appendix A); the advantages and disadvantages of various common real-time scheduling architectures (Appendix B); the TTC-jDVS2 algorithm (Appendix C); and techniques for incorporating an independent timer in an FPGA-based SoC design (Appendix D).

1.7 Conclusions

This introductory chapter has presented an overview of embedded systems and emphasised that the energy consumption of such systems has become an important issue for designers. It has also been noted that use of Dynamic Voltage Scaling (DVS) to reduce energy consumption in processors is likely (as a side effect) to increase levels of "jitter" and – thereby – degrade the performance of some systems. Techniques to allow use of DVS in low-jitter systems will form the focus of the remainder of this document.

Chapter 2

Energy-efficient embedded systems

In the introductory chapter, it was argued that the issue of power dissipation is of growing importance in the design of advanced processors for use in mobile embedded systems. This chapter reviews previous work in the area of energy-efficient embedded systems. The focus is on minimising energy consumption in processors, including approaches that may help to extend the service life of mobile devices.

2.1 Introduction

When considering research in energy-efficient embedded systems, it can be helpful to split the work into two categories: that of *power consumers* and that of *power sources* (Martin, 1999).

- When considering power consumers, the top level of the hierarchy is low-power software: this involves use of low-power algorithms and / or re-compiling code to use low power instructions (Ishihara and Yasuura, 1998a; Lorch and Smith, 1998; Pering and Brodersen, 1998; Lee and Sakurai, 2000; Simunic *et al.*, 2000; Sinha and Chandrakasan, 2001; Mouw *et al.*, 2002; Swaminathan and Chakrabarty, 2003; Lin and Hsueh, 2006). The bottom level of the consumer hierarchy involves the development of novel circuit structure or devices that are specifically designed to reduce power consumption (Chandrakasan *et al.*, 1992; Burd and Brodersen, 1995; Assaderaghi *et al.*, 1997; Dancy and Chandrakasan, 1997; ARM, 2003).

- When considering power sources, the top level of the hierarchy involves battery modelling, in order to predict discharge times (Hageman, 1993; Doyle, 1995; Chiasserini and Rao, 1999b; Martin and Siewiorek, 1999a; Panigrahi *et al.*, 2001). The bottom level of this area involves work in electrochemistry which is aimed at increasing battery life (Fuller *et al.*, 1994; Doyle, 1995).

In this review chapter, the focus is on techniques for reducing power consumption in processors, including total system power management and battery life extension techniques.

2.2 Power dissipation in CMOS circuits

Power consumption in CMOS circuits is mainly categorised into two types: dynamic power consumption (which arises due to switching activity in a circuit) and static power consumption (which is present even when no switching activity is performed). Total power consumption of the circuit can be described by Equation 2.1.

$$P_{total} = P_{AC} + P_{DC} \tag{2.1}$$

where

P_{total} is total power consumption of the circuit,
P_{AC} is dynamic power consumption and
P_{DC} is static power consumption.

The detail of dynamic and static power consumption are described in the following sub-sections.

2.2.1 Dynamic power consumption

The dynamic power in a CMOS circuit arises from the charging and discharging of circuit node capacitances which can be found on the output of every logic gate (Burd and Brodersen, 1995; Martin *et al.*, 2002; Nguyen *et al.*, 2003; Jha, 2006).

Whenever logic switches in the circuit, node capacitance C_L incurs a voltage change ΔV from the supply voltage at potential V_{DD}. Power consumption of that node is $C_L \Delta V . V_{DD}$. Therefore, the total dynamic power is summation over all N nodes in the circuit, as shown in Equation 2.2 (see, for example: Burd and Brodersen, 1995; Martin *et al.*, 2002).

$$P_{AC} = V_{DD} . f_{clk} . \sum_{i=1}^{N} \alpha_i . C_{Li} . \Delta V_i \quad (2.2)$$

where α is switching activity of each node which is a fraction of the clock frequency f_{clk}.

The voltage of most nodes in CMOS circuits swings from ground to V_{DD}, thus, the ΔV is approximately V_{DD}, whereas, the product of the node capacitance C_L and the activity weighting factor α can be expressed as an average switched capacitance per cycle, C_{eff}, over all the N nodes. The dynamic power equation can be simplified as shown in Equation 2.3.

$$P_{AC} \approx V^2_{DD} . f_{clk} . C_{eff} \quad (2.3)$$

The dynamic power depends on how frequently the gate is switched. If there is no switching activity in the circuit, the dynamic power is zero.

2.2.2 Static power consumption

Static power in the CMOS circuit mainly arises from the subthreshold leakage and the reverse bias junction currents. The static power consumption, P_{DC}, is given by Equation 2.4 (see, for example: Jejurikar *et al.*, 2004)

$$P_{DC} \approx V_{DD} . I_{subn} + |V_{bs}| . I_j \quad (2.4)$$

where

I_{subn} is the subthreshold leakage current,

I_j is the source and drain body junction leakage current,

V_{bs} is the body bias voltage.

The threshold voltage V_{th} as a function of supply voltage V_{DD} and the body bias voltage V_{bs} is given below.

$$V_{th} = V_{th1} - K_1 . V_{DD} - K_2 . V_{bs} \tag{2.5}$$

where K_1, K_2 and V_{th1} are technology constants.

The subthreshold current I_{subn} can be described by Equation 2.6, (see, for example: Martin *et al.*, 2002).

$$I_{subn} = \left(\frac{W}{L}\right) I_s \left[1 - e^{\frac{-V_{DD}}{V_T}}\right] e^{\frac{-(V_{th}+V_{off})}{nV_T}} \tag{2.6}$$

where W and L are device geometries, I_s, n and V_{off} are empirically determined constants for a given process, and V_T is the thermal voltage.

In general, V_{off} is small and $1 - e^{\frac{-V_{DD}}{V_T}}$ is nearly 1 for all V_{DD}. By approximating and substituting Equation 2.5 into Equation 2.6, I_{subn} is simplified as shown in Equation 2.7.

$$I_{subn} = K_3 \, e^{K_4 V_{DD}} e^{K_5 V_{bs}} \tag{2.7}$$

where K_3, K_4 and K_5 are constant fitting parameters.

Thus, the static power as a function of supply voltage V_{DD} and the body bias voltage V_{bs} can be represented as shown in Equation 2.8.

$$P_{DC} \approx V_{DD} . K_3 \, e^{K_4 V_{DD}} e^{K_5 V_{bs}} + |V_{bs}| . I_j \tag{2.8}$$

2.3 Approaches to minimise power in CMOS processors

Models of dynamic and static power in CMOS circuits have been described in the previous section. In this section, approaches to reduce power consumption in CMOS processors will be explored, based on the CMOS circuit's power model.

2.3.1 Reducing supply voltage

Reducing supply voltage is a very effective method for reducing dynamic power consumption. As shown in Equation 2.3, if the supply voltage V_{DD} is lowered, the dynamic power consumption of CMOS circuit drops in line with the voltage squared. However, this power reduction must be traded with changes in delay.

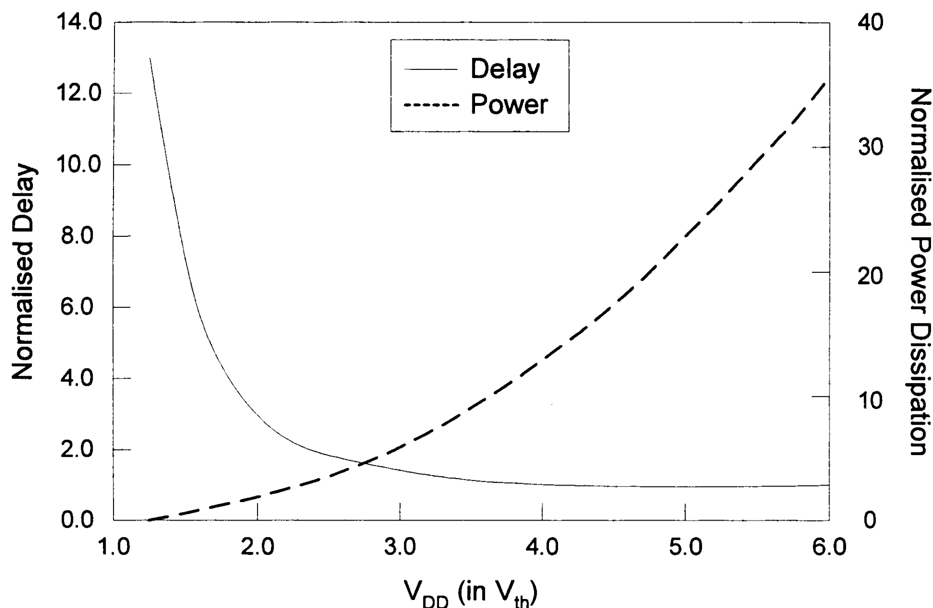

Figure 2.1: Delay, Power vs V_{DD}. (Redrawn from Burd and Brodersen, 1995)

The circuit delay, t_{inv}, is a function of both the power supply and threshold voltage of the internal resistors (Equation 2.9 – see, for example: Martin *et al.*, 2002).

$$t_{inv} = \frac{L_d K_6}{(V_{dd} - V_{th})^{\beta}} \tag{2.9}$$

where

L_d is the logic depth of the path

K_6 is a constant for a given process technology

β is a measure of velocity saturation, $1 < \beta \leq 2$

If the supply voltage is reduced, the delay increases dramatically as the supply voltage approaches threshold voltage (Burd and Brodersen, 1995). In CMOS processors, the increase in circuit delays means that the maximum clock frequency is decreased: in other words, the throughput of the processor will be reduced when the supply voltage is reduced.

The dynamic voltage scaling (DVS) technique was proposed as a means of lowering the supply voltage of a processor when the maximum clock frequency is not required. This technique can be implemented using software to determine and scale the supply voltage to match load requirements. Overall, DVS has proved to be an effective way to balance power and performance for CMOS processors. Further information about DVS techniques will be presented in later sections.

2.3.2 Reducing switching activity

Reducing switching activity in order to reduce dynamic power consumption can be achieved using two approaches. The first is to reduce the number of unnecessary gates from charging and discharging. The second is slowing down the charge and discharge rate of the gates. These approaches reduce amount of current drawn into the gates in CMOS processor.

Minimising the number of switching gates in the circuit can be achieved by gating the clock of unused parts. This clock-gating technique (Abdollahi and Pedram, 2006) has been applied in a novel processor, PowerPro architecture (Programmable Power Management Architecture) which supports dynamic adjustments of the active datapath width (Ishihara and Yasuura, 1998b). The processor obtains a PADWC (Programmable Active Datapath Width Control) function which provides the instructions that specify operation and control the variable datapath width.

The technique was applied in the registers, ALU, data-bus, off-chip driver and data memory. For example, two pieces of data which are assumed to be 32-bit and 7-bit in size are passed into the register. After the first cycle that loads the 32-bit data, only 7

bits of the second data element are allowed to transit while the rest of the bits are disabled by stopping the clock supplied to them. Thus, these 25 bits are unchanging while the transition of the 7-bit data is dependent on the previous 32-bit data value. Stopping the clock supply to unused bits of the datapath register can decrease power consumption significantly. However, the bit-width of the data-path is different for each operation. Thus, more sophisticated techniques to manage the power for datapath circuits are usually required (Ishihara and Yasuura, 1998b).

By contrast, decreasing the clock frequency only slows the switching activity, it does not reduce the number of switching events required to complete a task. While this approach reduces dynamic power consumption (see Equation 2.3) it is not an energy-efficient method (Chandrakasan *et al.*, 1992; Burd and Brodersen, 1995). More specifically, a task which is executed at a lower clock frequency (with constant supply voltage) will consume the same amount of energy (over the period required to complete a given task) than the same task executed at a higher frequency.

2.3.3 Reducing capacitance

A third approach for addressing power-consumption concerns involves reducing the capacitance of the circuit. Basically, the capacitance can be expressed as two parts (Burd and Brodersen, 1995). The first part is the device width which is composed of the subsequent gates' input capacitance and part of the diffusion capacitance on the gate output (Burd and Brodersen, 1995; Kumar, 2006). This can be controlled by the designer. The second part arises from the diffusion capacitance: this is dependent on technology only, and the capacitance of the wire interconnecting these gates, which may be optimised by efficient layout (Burd and Brodersen, 1995; Huang *et al.*, 2005). This approach optimizes both power and delay in CMOS circuits.

2.3.4 Reducing leakage current

As shown in Equation 2.8, turning off the supply voltage can reduce the leakage power dissipation in VLSI circuit. This approach can be implemented by using sleep transistors (one PMOS and one NMOS) in series with the transistors of each logic block, to make a virtual power supply and ground: this provides a "power gate" for reducing power in idle mode (Abdollahi and Pedram, 2006). This method can be applied for a large area or a group of logic gates by using one sleep transistor. Multi-

threshold CMOS (MTCMOS) is used for different modes (Kao and Chandrakasan, 2000). Transistors with a low-threshold voltage are used to implement the logic, while those with high-threshold voltage are implemented as sleep transistors.

In active mode, power gating cannot save power. Leakage power in active mode may be saved by using slower transistors which have a high threshold. Typically, leakage current can be reduced if the supply voltage is lowered. However, this approach degrades performance because of increases in gate delay (as discussed in Section 2.3.1).

Partitioning techniques enable parts of a device to be associated with a particular clock domain (Butts and Sohi, 2000). This allows functional units which do not require a high clock speed to be implemented separately, to save leakage power. Dual threshold / supply voltage is a technique presented to minimise both dynamic and static power (Srivastava and Sylvester, 2004).

The other factors for reducing leakage power are dependent on empirical parameters of technology and design.

2.4 Algorithms for reducing energy consumption in processors

Having considered techniques for power reduction at the VLSI level, the discussions in this section will centre on software-based techniques which have been proposed to achieve energy savings. More specifically, this section will describe dynamic voltage scaling (DVS) techniques – used to reduce dynamic power consumption – and also adaptive body bias (ABB) techniques, which are used to reduce static power consumption.

2.4.1 Dynamic voltage scaling

The key idea of dynamic voltage scaling to achieve energy saving in processors is to exploit the idle[2] and slack[3] times by running tasks for longer periods, at lower voltages (Weiser *et al.*, 1994; Schmitz *et al.*, 2004; Zhu and Mueller, 2004; Ge *et al.*,

---

[2] A period during which the processor does not execute any task (Buttazzo, 2004).

[3] The period for which a task can be delayed after activation and still complete by its deadline (Liu, 2000; Buttazzo, 2004).

2005). In general, the processor will have more performance than an application requires (indeed, designing the application to ensure this would seem to be common sense). Accordingly, by exploiting the slack, the processor can run slower in order that tasks – just – complete in time to satisfy their deadline requirements. As the clock frequency is reduced, the supply voltage can also be lowered, offering the potential for a significant energy reduction. In practice, appropriate (and sometimes complex) algorithms are required in order to determine the required clock frequency.

Dynamic voltage scaling was initially implemented as a set of algorithms for reducing the CPU energy consumption on desktop computer (Weiser *et al.*, 1994). The Weiser algorithms involve dividing the workload into intervals of various lengths: each interval may then be run at a different speed. The algorithms proposed by Weiser (OPT, FUTURE and PAST) aimed to lengthen the runtime of each scheduled segment, in order to minimise the task idle times, while retaining reasonable interactive response times. Basically, the processor clock is stopped during the idle time. However, without voltage reduction, it is not the most efficient way to save energy (see further information in Section 2.7.1). From Weiser *et al.* work, PAST was the only practical algorithm because it did not require future information: it used information of the previous interval to set the speed of the next interval. Govil et al. (1995) proposed an extension to this approach which involved making load prediction and speed-setting policies independently. The Govil algorithms (FLAT, LONG_SHORT, AGED_AVERAGES, CYCLE, PAT-TERN and PEAK), have been compared with the PAST. FLAT, which was the simplest policy and tried to keep the CPU's utilisation as flat as possible, proved to be the most effective. Lorch and Smith (2001) subsequently presented PACE (Processor Acceleration to Conserve Energy), an algorithm intended to reduce energy consumption in DVS algorithms without affecting their performance. PACE was not a complete DVS algorithm by itself but was a method for estimating the workload based on a probability distribution. On the basis of simulation results (Lorch and Smith, 2001), it was claimed that PACE reduced CPU energy consumption of the previous algorithms, such as PAST, FLAT, LONG_SHORT, by up to 49.5% (with an average of 20.6%).

After being applied on desktop computers, the dynamic voltage scaling technique was adapted for use in real-time systems (Pering and Brodersen, 1998). The fundamental parameters of real-time scheduling, such as start time, computation time and deadline, are used for scheduling tasks and determining running speed of CPU. The scheduler algorithm sets the CPU speed by determining computational resource in order to complete real-time tasks by their deadlines.

For real-time systems, the worst-case execution time (WCET) is normally used to determine the running speed of tasks in conventional voltage-scheduling methods (Ishihara and Yasuura, 1998a). Since tasks do not always execute at their WCET, Shin and Choi (1999) proposed a method to save energy when the task's execution time was less than the predicted WCET. The run-time voltage hopping method (Lee and Sakurai, 2000) proposed workload-variation slack time which partitions a task into several pieces (called timeslots) to dynamically vary the voltage inside a task. This scheme tries to fully exploit slack time and it was claimed that it could improve energy saving more effectively than techniques which varied the supply voltage on a task-by-task basis.

Some researchers have raised questions about the real-time nature of DVS algorithms. For example, Pillai and Shin (2001) suggest that previous DVS algorithms (Weiser *et al.*, 1994; Govil *et al.*, 1995; Pering *et al.*, 1998a) do not address real-time constraints because they aim to tackle the energy problem first. They go on to propose what they call real-time DVS (RT-DVS) algorithms which involve an adapted real-time scheduler and task management service which is intended to minimise energy consumption while retaining real-time deadline guarantees. The algorithms they propose include Statically-scaled RM (Rate Monotonic), Statically-scaled EDF (Earliest Deadline First), Cycle-conserving RM, Cycle-conserving EDF and Look-ahead EDF[4]. In Pillai and Shin study, Look-ahead EDF was claimed to be the best of those algorithms for saving energy. The Cycle-conserving policy can dynamically recompute voltage setting to run tasks but it is based on WCETs. The Look-ahead policy improves energy saving by lowering the frequency to stretch task for as long as possible, on the

---

4 These are based on rate monotonic (RM) – (Liu and Layland, 1973; Locke, 1992) – and earliest-deadline-first (EDF) – (Liu, 2000; Buttazzo, 2005) – algorithms

assumption that most tasks complete earlier than their WCETs. Of course, this may require running tasks at high frequencies later in order to complete all of the delayed tasks in time. In a similar vein, Swaminathan and Chakrabarty (Swaminathan and Chakrabarty, 2001) proposed the online scheduling algorithms, low-energy earliest-deadline-first (LEDF) and extended low-energy earliest-deadline-first (E-LEDF). These algorithms were designed for periodic non-preemptable task sets. E-LEDF was a modified (and more practical) version of LEDF.

In general, DVS algorithms can be broadly classified into "offline" and "online" approaches which depend on when the voltage settings are determined (Swaminathan and Chakrabarty, 2001; Bini *et al.*, 2005; Cai *et al.*, 2006). Offline (also called "statics" or "pre-runtime") approaches calculate all voltage (and frequency) settings at compile time: the voltage schedule will not be changed at runtime. The offline voltage scheduling algorithms compute voltage setting based on priori task parameters, such as periods and WCET, to guarantee satisfaction of time constraints. In real applications, actual execution times of tasks during operation are usually smaller than their WCETs but the offline algorithms cannot change scheduled voltages to exploit the runtime slack. However, the advantage of offline voltage schedules is that they avoid (voltage calculation) overheads during run time. Therefore, complicated techniques can be employed in order to increase the accuracy of the voltage schedule.

While the offline voltage scheduling techniques are usually based on WCET, Shin *et al.* (2001) analyse execution time variations among different execution paths of program code in order to explore actual execution times. It was found that, to improve energy savings, the average-case execution path (ACEP), rather than the worst-case execution path (WCEP), could be used to schedule the voltage of tasks by inserting voltage scaling point at branch or loop nodes inside a task. Incorporated with the intra-task voltage scheduling techniques, such as (Shin and Choi, 1999; Lee and Sakurai, 2000) which allowed to run each partition of a task with different voltages, meant that a task could fully exploit all slack time. Also, in order to improve a WCET-oriented schedule, Gruian (2001) used stochastic data (based on the probability distribution of the execution pattern) to build multiple voltage schedules,

in order to minimise the average case energy consumption. In particular, the voltage schedules at task level are correlated with the task execution length probability distribution. This offline decision (Gruian, 2001) was employed with an online scheduling policy in order to achieve further energy reductions.

In distributed systems, the voltage scheduling is further complicated by the presence of dependent tasks where the co-design flow (allocation, mapping, and scheduling) influences the ability to exploit DVS (Luo and Jha, 2000; Bambha *et al.*, 2001; Schmitz *et al.*, 2004). High-load search techniques (based on a genetic algorithm) were applied for exploring voltage-schedule solutions (Bambha *et al.*, 2001; Schmitz *et al.*, 2004) in order to maximise slack-time exploitation among multi-processing nodes and – thereby – optimise both the system (timing) schedule and the degree of DVS utilisation.

By contrast to these (predominately) offline techniques, online DVS approaches (e.g. (Zhu and Mueller, 2004; Zhuo and Chakrabarti, 2005; Cai *et al.*, 2006)) compute voltage settings during runtime to take advantage of early completion of tasks (i.e. slack time) in order to increase energy savings. Such online techniques clearly have the potential to achieve higher energy savings. However, the runtime overhead of online DVS algorithms must clearly be taken into account too.

Generally, online DVS algorithms are integrated into real-time task scheduling schemes. These schemes may be based on dynamic task priorities, e.g. EDF: see (Swaminathan and Chakrabarty, 2001; Kim *et al.*, 2002)), or fixed task priorities, e.g. RM: see (Kim *et al.*, 2003; Wei *et al.*, 2006). Zhu and Mueller (2004) applied an adaptive technique from control theory (PID-controlled feedback) to dynamically exploit slack time in the presence of varying workloads. This closed-loop DVS algorithm incorporated the PID controller into an online EDF scheduler.

Although general offline and online DVS techniques, e.g. (Gruian, 2001; Shin *et al.*, 2001; Swaminathan and Chakrabarty, 2001; Kim *et al.*, 2002; Zhu and Mueller, 2004; Wei *et al.*, 2006), are effective in reducing energy consumption, they are not necessarily efficient when it comes to extending battery life because they do not

consider non-linear battery characteristics (Cai *et al.*, 2006). Zhuo and Chakrabarti (2005) therefore proposed a battery-aware dynamic task scheduling algorithm, "dynamic average rate EDF" *(darEDF),* that combines the concepts of a slack forwarding algorithm (Shin and Choi, 1999) – which tries to make the power profile decrease with time, to improve battery efficiency – and a slack look-ahead algorithm (Shin *et al.*, 2000), which tries to increase system idle time, to increase opportunities for battery recovery. The *darEDF* algorithm updated the task density (WCET/period) whenever the run queue was changed: it tried to exploit the slack time as much as possible, in order to lower the battery-load current and also set the voltage for the tasks in a queue, in order to decrease power profile. Cai *et al.* (2006) have also introduced an online DVS algorithm for multiprocessor systems that takes into account heterogeneous tasks which draw different current from battery and require different WCETs, whilst maintaining low online complexity.

In practice, DVS overheads (both voltage calculation and voltage transition) are significant in term of execution time and power consumption. Previous studies, e.g. (Swaminathan and Chakrabarty, 2001; Andrei *et al.*, 2005), have reported that the overheads are large even if low-complexity online algorithms are used, and that such algorithms have a significant impact on energy consumption. For example, the online PID-feedback DVS algorithm (Zhu and Mueller, 2005) can improve energy saving, by employing dynamic slack. The specific improvement obtained is about 20-30%, compared to the static algorithm. However, the overhead of the dynamic algorithm is around 1600% higher than the static algorithm. Zhu and Mueller (2005) suggest that the trade-off between overhead and performance always needs to be examined carefully. Aware of the overhead / performance trade-off, Zhu *et al.* (2003) proposed a scheme to incorporate time overhead and energy overhead in an algorithm which took into account the voltage/frequency adjustment. The algorithm takes into account the DVS calculation overheads when determining whether the system timing constraints can be met. As a consequence, it may decide not to run at a lower voltage (if the energy overhead is larger than the energy saved by the voltage change).

The distinction between "online" and "offline" approaches need not be absolute and – to reduce the complexity of online DVS algorithms – some parts of the voltage

calculations have been carried out during the offline phase. For example, Andrei *et al.* (2005) proposed quasi-static voltage scaling algorithm that it is able to exploit the dynamic slack and also keeps the online overhead very low. To reduce the online overheads, the necessary voltage calculations are computed offline and stored in lookup tables within memory. Simulation results suggest that the quasi-static approach (Andrei *et al.*, 2005) is the best (in terms of energy saving and voltage calculation overhead) among various algorithms (greedy heuristic (Aydin *et al.*, 2001) and linear time heuristic (Gruian, 2002)). The overhead of the quasi-static algorithm is less than that of the greedy heuristic, which has the smallest overhead among comparable algorithms, about 34 times lower overhead when executed on an SA-1100 processor. Furthermore, Andrei *et al.* (2007) also reduce (voltage transition) overhead by reordering the voltage mode (a set of supply and body bias voltages) within a task, in order to eliminate voltage-mode switch between tasks. For example, if the voltage modes of the last partition of the current task and the first partition of the next task can be reordered to be the same, the voltage transition between tasks can then be removed while the numbers of intra-voltage transition in each task are the same.

It is also important to appreciate that the use of DVS may have an impact on system reliability (Hazucha and Svensson, 2000; Shivakumar *et al.*, 2002; Ernst *et al.*, 2004; Zhu, 2006). When the supply voltage is reduced, the systems are more easily affected by lower energy particles (i.e. gamma rays which arrive from space and alpha particles which are created when atomic impurities decay (Seifert *et al.*, 2001; Austin *et al.*, 2004)) which can lead to increased numbers of transient faults (Zhu, 2006). In safety-related applications, the use of DVS potentially impacts on levels of tolerance to transient faults (Ejlali *et al.*, 2006). In general, time-redundancy technique (i.e. rollback-recovery) also exploits slack time to improve transient-fault tolerance by performing recovery executions whenever a faulty run occurs: the number of recovery executions which can be carried out depends on the available slack time. Since DVS techniques aim to eliminate slack time, there is the clear potential for conflict between time-redundancy techniques and DVS. As a potential solution to such conflicts, Ejlali *et al.* propose information redundancy hardware that is used to correct faults during execution (i.e. without requiring a re-execution) in order to increase exploitable slack time for lowering voltage. Simulation results (Ejlali *et al.*, 2006) suggest that, as the

transient faults (single event upsets) rate increases, the proposed system has greater potential (in term of energy saving) when comparing to the conventional rollback recovery system. Overall, it is clear that voltage scaling technique should be carefully evaluated before they are applied in mission-critical embedded applications (e.g. military and aerospace) or systems deployed in vulnerable environments (e.g. satellite systems).

Use of DVS also has an impact on system timing. In the study (Mochocki *et al.*, 2005), Mochocki *et al.* considered meeting the completion jitter constraints of a set of independent periodic tasks scheduled by fixed-priority (RM) scheduling. They report that DVS has large potential to increase jitter when used aggressively. On the other hand, if DVS is used carefully, it is possible to meet jitter requirements. The *Jitter Aware DVS (JADVS)* (Mochocki *et al.*, 2005) algorithm is proposed for scheduling tasks to meet jitter constraints. The goal of the algorithm will be achieved if a valid "island" (the interval between release and deadline of a job but excluding all pre-emption and execution time) solution set can be identified for a task. The simulation results showed that, at utilisation of 0.3, *JADVS* can schedule 90% and 95% of task sets (20 tasks) to meet 10% of absolute and inter-completion jitters respectively, while the RM algorithm cannot schedule any tasks at the same situation. However, at utilisation of 0.5 and 0.6, all 20 tasks scheduled by *JADVS* suffer more than 10% absolute and inter-completion jitters respectively. At tighter jitter constraints (2%), no scheduled task can satisfy the requirements at utilisation of 0.4. In this study (Mochocki *et al.*, 2005), the transition overhead and discrete voltage levels are not taken into account.

Another impact of DVS on timing can be found in video decoding applications. In the study (Lee *et al.*, 2005), it was found that DVS caused video frame deadline misses (due to inaccuracy in decoding-time prediction) which then led the video quality degraded. Lee *et al.* (2005) focused on technique for predicting the decoding time of high variability video frames, in order to maximise energy saving under discrete voltage/frequency setting conditions. The *Frame-Data Computation Aware (FDCA)* (Lee *et al.*, 2005) technique achieved the deadline misses within 10-20% of the playback interval (which has a trivial impact on video quality) that were better

than the *GOP* (Son *et al.*, 2001) method but in the same range as the *Direct* (Pouwelse *et al.*, 2001c) and *Dynamic* (Nurvitadhi *et al.*, 2003) algorithms. Although the *FDCA* consumed more energy than the *Dynamic*, it was claimed that it supported real-time video applications, not only for stored video (as is the case with the other algorithms).

2.4.2 Adaptive body biasing

In the past, power dissipation in CMOS processors has been dominated by dynamic power (Marculescu *et al.*, 1994). However, as VLSI technology continues to scale, leakage power becomes of greater concern (Chandrakasan *et al.*, 2000). Jha (2006) shows that the ratio between leakage and dynamic power are about 22% and 78% of total power consumption in 0.07um CMOS technology but, in 0.035um technology, the leakage power becomes 67% of total power consumption. The dominance of leakage power is more apparent when applying voltage scaling (which results in significant reductions in dynamic power consumption while leaving leakage power unchanged).

As the supply voltage for modern high-performance processors is reduced, the threshold voltage is also proportionately reduced, in order to decrease delays and – hence – maximise the clock frequency. The reduction in threshold voltage results in a large increase in subthreshold leakage current which in turn leads to a larger standby current (Butts and Sohi, 2000; Jejurikar *et al.*, 2004). By contrast, the leakage current can be reduced significantly by increasing the threshold voltage, but the device will have a lower maximum operating frequency.

The threshold voltage may also be adjusted by applying a voltage to the body node of a gate to reverse bias the source body junction (Butts and Sohi, 2000). Such "adaptive reverse body biasing" (ABB) has been proposed as a way of minimising leakage current in standby mode (Keshavarzi *et al.*, 2001). Recent studies (Martin *et al.*, 2002; Wu *et al.*, 2004) have involved both scaling the threshold voltage and (simultaneously) applying DVS. Leakage Control EDF (LC-EDF) scheduling algorithms (Lee *et al.*, 2003) have also been proposed as a means of reducing the leakage power in real-time systems while Jejurikar *et al.* (2004) have proposed a procrastination scheduling scheme which attempts to maximise the duration of the idle interval in order to minimise both dynamic and static power consumption.

2.5 System-Level Power Reduction

Power consumption does not arise solely from a processor: other parts of the system hardware must also be considered. In small systems, such as PDAs, the processor is the most energy-hungry component of the system (Viredaz and Wallach, 2003). In larger systems, such as portable computers, the average power consumption in processors is typically only 14% of all components used and video and backlighting take more than one-third of the system power (Lorch and Smith, 1998).

In this section, power management policies for dealing with system components are reviewed.

2.5.1 System-level Power Management

Dynamic power management (DPM) (Lu *et al.*, 1999; Lu *et al.*, 2000; Simunic *et al.*, 2000; Simunic *et al.*, 2001; Cheng and Goddard, 2005; Zhang and Chakrabarty, 2006) has proved to be one of the most successful techniques for minimising power requirements at the system level. This technique reduces energy consumption by selectively turning off system components when they are idle. DMP operates by observing request patterns for component activation and predicting the length of "idle" periods. It may turn off components as soon as it is idle (an "aggressive" approach). However, turning on and off a component has associated time and energy overheads. Incorrect predictions may therefore have an impact on energy consumption and performance: constructing a complex system which supports DMP is a difficult procedure. To support DPM in complex systems the *Advanced Configuration and Power Interface* (ACPI) initiative was established to standardize interfaces between power-manageable hardware components and the power manager (Paleologo *et al.*, 1998).

Lorch and Smith (1997) have proposed techniques for improving the (naive) power-management strategy in the MacOS. This original approach simply turned off the processor whenever no activity had occurred in the last 2 seconds and no I/O activity had occurred in the last 15 seconds. The *greediness* technique (Lorch and Smith, 1997) identified and blocked processes doing unnecessary work in order to avoid wasting energy and degrading performance. Lorch and Smith also proposed software

strategies which were classified into three categories: transition, load change, and adaptation, for managing energy of hardware components in portable computers (Lorch and Smith, 1998).

Paleologo *et al.* (1998) represented a power-managed system (devices and workloads) by means of a stochastic model based on discrete-time Markov decision process (DTMDP). Using his approach, policy optimisation can be transformed into a linear programming problem and proven by a mathematical framework. Following this work, the continuous-time semi-Markov decision process (SMDP) model was developed: this allows event-based policy implementation instead of discrete time (Simunic *et al.*, 1999). It was claimed that event-driven power managements were more energy efficient than clock-driven approaches because the decisions were made only in response to the changes of the system without creating additional activity on each clock cycle when the system is idle.

Lu *et al.* (2000) built a framework to evaluate power management algorithms and explore their impact on user perceptions about system performance. The results they obtained suggested that (with a total wait period kept fixed), users preferred to wait for many short consecutive activities than a single long activity. In this case, the system with multiple short activities was felt to be more responsive.

Approach for reducing I/O device energy consumption is not limited to shutdown-based scheme. As digital circuits can gain benefit from DVS technique, in wireless communications, dynamic modulation scaling (DMS) (Schurgers *et al.*, 2001; Yu *et al.*, 2004; Yang *et al.*, 2005; Yuan *et al.*, 2005), was introduced to minimise energy consumption of radio devices as part of power management technique (Schurgers *et al.*, 2001; Schurgers *et al.*, 2003). Similar to DVS, DMS is a scaling technique which changes its modulation on the fly using for balancing between energy and packet throughput. The advanced energy reduction method must be incorporated in packet scheduling schemes in order to manage energy consumption of the wireless communication subsystem.

2.5.2 DPM and Real-Time Systems

Dynamic power management via I/O device scheduling for real-time systems was initially proposed by Swaminathan and Chakrabarty (2003). A device scheduler for hard real-time systems – called a low-energy device scheduler (LEDES) – was introduced to reduce the energy consumption of I/O devices. Its predictive scheme was based on an adaptive learning tree and the aim was to shut down devices by predicting the length of the next idle period, based on past observation of requests. A more practical I/O device scheduler – multi-state constrained low-energy scheduler (MUSCLES) – was also presented to reduce energy consumption for hard real-time systems (Swaminathan and Chakrabarty, 2003). Both approaches guarantee that real-time constraints are not violated.

However, Cheng and Goddard (2005) reported that the algorithms for minimising I/O device energy consumption for real-time systems in (Swaminathan and Chakrabarty, 2003; Swaminathan and Chakrabarty, 2005) did not consider the energy consumption of processors in order to maximise the overall system energy saving. They then proposed an online system-wide energy-efficient scheduling algorithm, *System-wide Energy-Aware EDF (SYS-EDF)* (Cheng and Goddard, 2005), which regarded the sum of I/O device and processor energy consumption, supporting periodic task sets with non-preemptive shared resources. The simulation results showed that the *SYS-EDF* improved system energy saving up to 37% when comparing with algorithm using I/O device power management or processor voltage scaling solely, and up to 10% when comparing with algorithm that integrated both techniques regardless system-wide energy-efficiency.

Dynamic power management is also explored for integrating in fault tolerance real-time embedded systems (Melhem *et al.*, 2004; Zhang and Chakrabarty, 2006; Zhu, 2006). Beyond energy-saving issue, power management is also significant in order to assist in reducing transient faults caused by high die temperature or lower processor voltages that are likely to lead to lower noise margins (Zhu *et al.*, 2004; Zhang and Chakrabarty, 2006). Concerning both energy and reliability issues, studies in (Melhem *et al.*, 2004; Zhang and Chakrabarty, 2006; Zhu, 2006) adopted checkpointing and rollback recovery (a fault detection and recovery techniques (Ziv

and Bruck, 1997; Kwak *et al.*, 2001)) and then combined with DVS to reduce energy consumption and improve the run time reliability of the system.

2.6 Extending the service life of mobile embedded systems

The battery becomes a crucial factor when designing mobile devices. Due to the shrinking size of modern handheld devices, designers have to consider the development of smaller and lighter batteries. Such batteries are the main factors which limit the energy source and service life of mobile systems. Previous studies on batteries have had the intention of improving understanding and – thereby – predicting their behaviour (Martin, 1999; Martin and Siewiorek, 1999a; Rakhmatov *et al.*, 2002; Rakhmatov and Vrudhula, 2003; Lahiri *et al.*, 2004), including steering their discharge (Luo and Jha, 2001; Benini *et al.*, 2002; Choi *et al.*, 2006).

This section reviews the factors which affect battery performance and then considers the results from previous studies which have been aimed at reducing battery discharge rates in order to maximise the service life of mobile embedded systems.

2.6.1 Factors affecting battery performance

Beyond environmental factors, such as low temperature which shorten the battery service life by reducing chemical activity and increasing internal resistance of battery (Linden and Reddy, 1995; Martin, 1999; Rakhmatov and Vrudhula, 2003; Lahiri *et al.*, 2004), other factors affecting performance of batteries are de-scribed below.

High (continuous) discharge rate can quickly shorten the service life of a battery (Linden and Reddy, 1995; Martin, 1999). If large current is drawn from a battery, electro-chemical reactions occur only at the outer surface of the cathode and, consequently, the active cathode sites remain un-utilised: this decreases the total capacity of the battery (Lahiri *et al.*, 2002) In general, the capacity of a battery, C, decreases with increasing discharge current but this relationship is not linear. For instance, a battery rated at 5 Ah at the $C/5$ rate (1 A) will operate for 5 hours. If the battery is discharged at a lower rate, for example the $C/10$ rate (or 0.5 A), its service life is more than 10 hours and it delivers more than 5Ah of capacity. On the other

hand, when discharged at C rate (or 5 A), the battery will run for less than 1 hour and deliver less than 5Ah of capacity.

Pulse discharge is one of factors affecting battery lifetime (Linden and Reddy, 1995; Martin, 1999; Rakhmatov and Vrudhula, 2003; Lahiri *et al.*, 2004). A high pulse discharge can dramatically drop the battery voltage within a short period: this may then mean that a battery-powered system may suffer a sudden power shortage (which may, for example, cause a system reset). Accordingly, Rakhmatov and Vrudhula (2003) reported that, under various load profiles being tested, the load profile consisting of lower peak current could have longer lifetime.

In addition, different types of battery have different responses to pulse discharges which may have an impact on battery lifetime (Linden and Reddy, 1995). For example, with a zinc-carbon battery, the output voltage drops sharply initially and then recovers. By contrast, the response of a zinc/alkaline/manganese dioxide battery is characterised by the voltage initially falling and then either remaining at this lower level or dropping slowly.

Intermittent discharge can extend battery lifetime (Chiasserini and Rao, 1999a; Martin, 1999; Rakhmatov and Vrudhula, 2003). When a battery stands idle after a discharge, a chemical reaction takes place that raises the battery voltage: this is called a "recovery effect" (Linden and Reddy, 1995; Chiasserini and Rao, 1999a; Chiasserini and Rao, 1999b; Panigrahi *et al.*, 2001; Lahiri *et al.*, 2002). After a (high) discharge, a battery should be relaxed (idle or lower discharge) to gain benefit from recovery effect, in order to maximise battery capacity. Rao and Vrudhula (2005) showed that the recovery time for a battery increased if the duration of load discharge was longer: in other words, the more the battery had been discharged, the more rest time was required. Appropriate rest periods can extend the battery lifetime but the voltage recovery rate of the battery depends on many factors, such as the particular battery system and constructional features, length of recovery period and discharge rate (Linden and Reddy, 1995).

The shape of the battery discharge curve, which depends on the electrochemical system and constructional features of batteries, can affect the service life of mobile systems (Linden and Reddy, 1995). For example, suppose Battery A gives a flat output voltage and then sharply drops while Battery B's output voltage gradually drops throughout its service life. In applications where the device is restricted to (say) -15% of battery voltage level, at time X, Battery A (with the flat discharge curve) may still have an output voltage above the restricted voltage while Battery B may have dropped below this threshold. Battery A then has longer service life. By contrast, if the device can tolerate a wider voltage spread, the output voltage from Battery B may exceed the threshold for longer.

2.6.2 Controlling battery discharge

In order to extend system life, various studies have been aimed at understanding battery characteristics and employing this knowledge to control the discharge. Martin and Siewiorek (1999a) studied the non-ideal properties of batteries with intermittent discharge arrangements. Their results showed that the intermittent discharge can increase battery lifetime when compared with continuous discharge. They also suggested that, with the same average power load, the system which discharges with a low power peak can have a longer service life than that with high power peak. By being aware of discharge profile shape, Luo and Jha (2001) proposed a battery-aware static scheduling scheme which employed voltage scaling in order to flatten the discharge power profile as well as reduce the average discharge power consumption. The battery-aware scheduling algorithm increased battery lifetime by up to 76% over fixed-voltage scheduling algorithm and by up to 56% over a general voltage-scaling scheduling algorithm.

To gain benefit from the recovery effect, the BEE (Battery Energy Efficient) network routing protocol has also been developed to balance the battery consumption among all nodes (Chiasserini and Rao, 2000). The scheme tried to route the data packet through the nodes which have a high battery capacity and allow the other nodes to rest for recovery. Similarly, in the work of Rakhmatov and Vrudhula (2003) the battery-aware scheduling was used to insert an idle period or lower-voltage task after the high-load task was executed. As an alternative, Choi *et al.* (2006) divided battery runtime into three regions, (i) the recovery effect, (ii) the rate capacity and (iii) the

alarm. They then used these periods to apply different policies for discharging a battery. A DVS algorithm that makes use of recovery effect can be applied in the first region (the battery residual remains steady constant). In the rate capacity region (the battery residual decreases at a constant rate), a DVS algorithm (with quality of service (QoS) guaranteed) is applied to control the discharge in small-amount. In the alarm region (the battery residual drops dramatically), the QoS requirement is possibly ignored and the battery-lifetime maximisation algorithm could be applied. The results (Choi *et al.*, 2006) showed that the policy significantly improved battery lifetime with a trivial QoS degradation.

2.7 Metric for energy efficiency

In order to maximise energy efficiency, an energy metric can be used to identify the optimal trade-off point. In this section, energy efficiency metrics for processor and mobile embedded systems are explored.

2.7.1 The processor's energy metric

Millions of instructions per joule (MIPJ) was first introduced by Weiser (1994) to index energy performance of CPUs. It is different from MIPS (millions instructions per second) which is a metric for performance of CPUs. The aim with MIPJ is to specify how many instructions are executed for a given amount of energy (MIPS/W). For example, a 200-MIPS Alpha chip consumes 40.0 watts, so it has a rating of 5 MIPJ. The Motolora 68349 consumes 300 mW and is rated at 6 MIPS: this translates as 20 MIPJ. Indexing by *MIPJ*, the low performance Motorola 68349 has an energy efficiency greater than a high-speed Alpha chip.

Similar to *MIPJ*, energy-per-operation is a metric for indexing energy efficiency of processors (Pering and Brodersen, 1998; Pering *et al.*, 1998b; Min *et al.*, 2002; Mouw *et al.*, 2002). For CMOS designs, the energy-per-operation (E_{op}) is energy / clock frequency (Equation 2.10).

$$E_{op} \; \alpha \; C_{eff} V^2_{DD} \tag{2.10}$$

where C_{eff} is effective switching capacitance and V_{DD} is the supply voltage.

From the Equation 2.10, illustrates (again) that changing a processor's clock frequency (alone) is not an effective technique for minimising energy. Reducing the clock frequency reduces the power consumed by processor but it does not reduce the energy required to perform a given task. The MIPJ rating has also been used to demonstrate that energy performance was unchanged if frequency scaling (alone) is employed (Mouw *et al.*, 2002).

2.7.2 Computation per discharge

Energy-per-operation measures are suitable to comparing energy efficiency for general use(Pering and Brodersen, 1998; Pering *et al.*, 1998b; Min *et al.*, 2002; Mouw *et al.*, 2002). For systems powered by a battery, a different energy efficiency metric has been proposed: called "computation per discharge" (Martin and Siewiorek, 1999b), this relates performance to the discharge rate of the battery. The computation per discharge measure is calculated as shown in Equation 2.11.

$$Computations\ per\ Discharge = \frac{Capacity(SystemPower(f))}{SystemPower(f)} \cdot Performance(f) \quad (2.11)$$

The computation per discharge is concerned with (i) battery capacity as a function of system power; (ii) system power as a function of CPU clock frequency; and (iii) application performance as a function of CPU clock frequency (f). To extend battery life, the major factors which are considered in order to maximise computation per discharge are described below.

The first concern is the system's power and CPU clock frequency. Generally, systems which run at a lower CPU clock speed can gain a longer battery discharge. However, the slowest CPU frequency setting, in case of variable-voltage CPU, may not minimize the total power of the system (Martin, 1999; Cheng and Goddard, 2005; Aydin *et al.*, 2006) (because the rest of system is still supplied with a fixed voltage and a slow running speed may increase the energy per operation).

The next concern is battery capacity. If battery capacity is assumed to be constant, maximizing computations per discharge is the same way as minimizing the energy per

operation. In practice, the battery capacity is not constant but it is a function of system power which (in turn) depends on CPU speed setting (Martin, 1999). When power consumption rates are high, the battery capacity drops dramatically (Linden and Reddy, 1995; Chiasserini and Rao, 1999b; Benini *et al.*, 2000; Lahiri *et al.*, 2004).

The final factor to be considered is application performance. This too is a function of CPU speed; however the performance may not linearly increase as CPU frequency increases because other components will also be involved in the system operation (Martin and Siewiorek, 1999b; Pouwelse *et al.*, 2001b; Hsu and Kremer, 2002). If an application becomes limited by "external" factors such as memory bandwidth, increasing the CPU frequency may have little impact on performance but will increase power consumption. Therefore, setting CPU speed for battery-powered designs must consider the whole system (Martin and Siewiorek, 1999b).

2.8 Conclusions

In this chapter, previous work on energy-efficient techniques for use with embedded systems (especially those involving CMOS processors), has been reviewed. In this review, DVS has been highlighted as a successful technique which has been employed in procedures which have been intended to reduce both dynamic and static power consumption. In practical designs, DVS will usually be incorporated in a real-time scheduler implementation, in order that the system can schedule tasks and (simultaneously) set appropriate voltage and clock frequency for such tasks. In Chapter 3, the literature review therefore continues with a focus on scheduling algorithms.

Chapter 3

Low-jitter scheduling algorithms

In Chapter 2, key techniques used to develop energy-efficient embedded systems were reviewed. The present chapter continues the literature review, with a focus on jitter and its impact on embedded systems.

3.1 Introduction

Jitter can be viewed as the deviation from the ideal timing of an event (Wavecrest, 2001; Ou *et al.*, 2004), as shown in Figure 3.1. In the IEEE dictionary, jitter is defined as "time-related, abrupt, spurious variations in duration of any specified, related interval" (Gannett *et al.*, 1972) while an Oxford dictionary of computing defines jitter as the "variation in the arrival time of a supposedly synchronous signal". It goes on to say that "Jitter may be caused by the fact that the original source of the signals has variations from its ideal periodic time, or by variations in the transit time from the signal source to the point at which the arrival times are actually observed, caused either by variations in the pathlength or in the speed at which the signal travels. In practice it is likely that all three effects contribute, and jitter is nearly always present." (Daintith, 2004). Jitter is a term used to refer to timing variations that occur rapidly while other timing variations that occur more slowly are called wander (Mills, 1995; Agilent, 2002; Avoim, 2002; Tektronix, 2002). The International Telecommunication Union (ITU) has defined the threshold between wander and jitter as follows: if the phase variation of a signal (which is an oscillating movement

with an amplitude and a frequency) is more than 10 Hz, it is known as jitter, and when the phase variation is less than 10 Hz, it is called wander (Agilent, 2002; Tektronix, 2002).

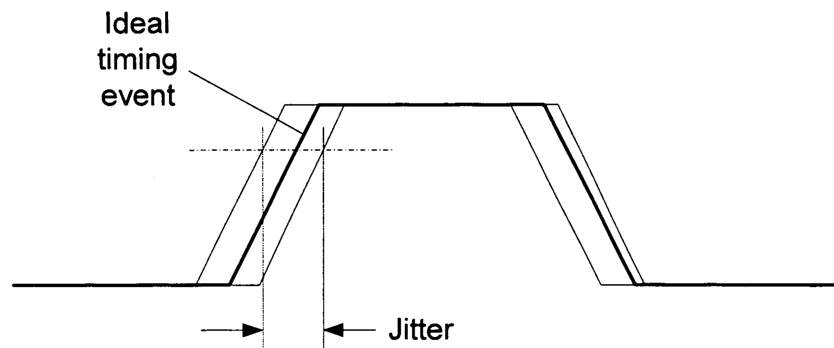

Figure 3.1: A schematic representation of jitter. See text for details.

In the past – when systems have relatively low signalling rates – jitter might have less impact on timing (because the errors induced by jitter were small compared with the time interval that they corrupted). However, the timing margins associated with modern high-speed signals means that strict jitter control throughout the system design is required (Tektronix, 2002). In modern processors, the demand for greater computation has forced the processor to the radio frequency (gigahertz) range and system performance is limited by parallel-bus data transfers: alternative architectures are therefore being explored (e.g. Infiniband, 3GIO, SONET, SATA). To achieve reliable operation for high-speed systems (such as video-in-demand and other information-hungry applications which run on Gigabit Ethernet networks), substantial understanding of timing jitter characteristics is required (Ong *et al.*, 2004).

3.2 Sources of jitter

Jitter can arise from many sources. In digital wireless communication systems, jitter can be found in form of phase noise of local oscillator (Hajimiri and Lee, 1998), a device used to generate a signal which is beat against the signal of interest to mix it to a different frequency (e.g. Hartley, Colpitts and Wien Bridge oscillators). In practice, noise may come from the power supply lines or interference from other nearby signals (Schilling and Belove, 1985). This phase noise, in the guise of clock jitter, may have a direct impact on timing margins and, consequently, limit system performance (Hajimiri and Lee, 1998). Timing jitter caused by noise can also be found in phase-locked loops (PLLs). Technically, a PLL is a circuit, primarily consisting of phase

detector (PD), loop filter (LF) and voltage-controlled oscillator (VCO), that synchronizes (or locks) both frequency and phase of output signal (generated by an oscillator) with an input signal, by means of a feedback control system (Tektronix, 2004). As a result of increasing speed in digital systems, PLLs are commonly required to generate low-jitter on-chip clocks but device and supply/substrate noise affect the PLLs operation which may itself result in jitter (Sidiropoulos and Horowitz, 1997; Mansuri and Yang, 2002). In order to design low-jitter PLLs, for example, the work in (Chuang and Liu, 2006) proposed a compensation circuit to reduce the jitter caused by the leakage currents in a PLL. Another proposed solution involves implementing a low-pass filter at the DC control voltage input terminal of the VCO to suppress noise component contained in the power supply (Agilent, 2000).

In high-speed-digital systems, jitter can arise from crosstalk, caused by electromagnetic interference (EMI) along a circuit or a cable pair. The data signal sent from one chip to another must propagate within one clock cycle. As the operating clock frequency of the system increases, the margin of the interchip propagation delay becomes smaller because the clock period is shorter. With careless design of a printed-circuit boards, crosstalk-induce jitter can occur when the simultaneous signal on two adjacent "aggressor" lines surround and interfere with the victim line. In some circumstances, this can delay the arrival time of the signal at the receiver and may cause system failure (Bogatin and Garat, 2004). Another example affected by EMI is a high-speed optical transmitter, which converts data from electrical to optical form at speeds of 10 Gbps. EMI can cause excessive clock jitter that then leads to data errors. Enclosing the transmitter oscillator in a metal shield can reduce clock jitter on a high speed optical transmitter (Oen and Schultz, 2006).

Jitter has an impact in the application of both analogue-to-digital converters (ADCs) and digital-to-analogue converters (DACs). In digital audio systems, sampling clock jitter degrades the performance, in term of the sound quality, in the ADCs and DACs themselves. The jitter distorts the signal waveform from the original shape (phase distortion rather than amplitude distortion). This small timing error has more impact, especially, when signal is slewing, rising or falling (Fourre, 1993; Story, 1998). The error caused from jitter at the output of a DAC results in unpleasant artefacts added to

the music (e.g. jitter with a frequency of 1 kHz affecting a DAC reproducing 7 kHz sine wave will also generate spurious output tone at 6 kHz and 8 kHz. If the jitter frequency is 2 kHz, the artefacts will be present at 5 kHz and 9 kHz). When playing music, constantly changing spectral content (which may be jitter at several frequencies), produces a highly complex spectrum which will be mixed into the music (Atkinson, 1990; Fourre, 1993). Moreover, if the jitter is random (a random spectral distribution rather than discrete frequency components), white noise (a random signal with a flat power spectral density) will be added to the DAC's output signal (Atkinson, 1990; Fourre, 1993; Story, 1998).

On 1-bit converters (which form the core of sigma-delta or noise-shaping converters: (Aziz *et al.*, 1996; Maxim-Dallas, 2003)), the impact of jitter is different to that observed with conventional (multi-bit) converters. Technically, 1-bit converters sample signal at a rate significantly higher than the Nyquist frequency, a process called oversampling (Macadie, 2004). This means that 1-bit converters are more sensitive to the sampling-clock jitter than conventional converters. To show how much more sensitive, in case of random jitter with a 200 ps RMS amplitude applied to the clock of both converters, the jitter leads to white noise at the 1-bit and the conventional DAC's output with an approximate amplitude of -75dB and -95dB in the audio range respectively (Fourre, 1993). However, 1-bit converters are popular used in such hi-fi digital audio systems (Aziz *et al.*, 1996; Macadie, 2004) because the oversampling in 1-bit converters reduces the quantization noise power in the signal band by spreading a fixed quantization noise power over a bandwidth much larger than the signal band (Jarman, 1995; Maxim-Dallas, 2003): in other words, the noise is pushed to frequencies that the human cannot hear.

In digital radio systems, ADCs are also employed to digitize the down-converted IF (Intermediate Frequency) signal in digital down-conversion receivers. Sampling jitter, mostly caused by sampling clock instability rather than the ADC's aperture jitter[5], results in phase error and has more impact when the input signal frequency

---

[5] The short interval required for sample-and-hold, connecting switch to charge input signal in capacitor and disconnecting quickly the hold capacitor from the input buffer amplifier, action is called *aperture time (or sampling aperture)*. Aperture jitter is the sample-to-sample-variation timing of the ADC's input switch (Kester, 2005).

increases. Accordingly, the jitter degrades performance of digital demodulation (bit error rate (BER) increasing or IF input frequency limited) such as Quadrature Phase-shift Keying (QPSK), Quadrature amplitude modulation (16QAM) (Kim *et al.*, 2005).

As use of the internet continues to grow, the demands for distributed multi-media applications (e.g. audio telephony, video conferencing and video-on-demand) over the network is also rising. However, most packet-switching networks, such as the Internet, do not provide guaranteed performance services and delay and jitter in a packet can degrade the application performance (Clark *et al.*, 1992). Various packet scheduling policies have been proposed in order to guarantee services in packet-switching networks, including delay and jitter control, for example, Jitter-Earliest-Due-Date (Jitter-EDD), Stop-and-Go, Hierarchical Round Robin (HRR), Rate-Controlled Static Priority (RCSP) (Zhang, 1995). Even in the ATM[6] broadband network, jitter present in the MPEG-2[7] stream, in a form of cell delay variation, has a significant impact on the video quality as seen at the receiver. Cell delay variation is unpleasant since it introduces synchronization problems between source and the decoder. When using a PLL to recover the clock from the program clock reference (PCR) timestamps transmitted within the stream, it is easy to have jitter impact on the quality of the reconstructed clock. A common practice is to employ a dejittering buffer at the receiver that absorbs the jitter introduced by the network. An advantage of implementing the dejittering buffer is that the network is transparent to the decoder phase-locked loop, however, this method requires a priori amount of the maximum delay variation to prevent overflowing/underflowing the buffer (Tryfonas and Varma, 1999).

Jitter is a common problem in real-time implementations of control systems. In computer control systems, the main processes (sampling, control computation, and actuation) of a control loop can experience delay in their operation that causes degradation in control performance and may lead to instability (Marti *et al.*, 2001a;

---

[6] Asynchronous Transfer Mode (ATM) is a standard for broadband networks that allows a wide range of traffic types (e.g. from real-time video to best-effort data) to be multiplexed in a single physical connection-oriented network (Tryfonas and Varma, 1999).

[7] MPEG-2 is compression standards for Audio and Video, agreed upon by MPEG (Moving Picture Experts Group), and typically used to encode audio and video for broadcast signals, including direct broadcast satellite and Cable TV (Bilas *et al.*, 1997).

Cervin *et al.*, 2004; Kao and Lincoln, 2004). Such causes of delay, in single processor, mainly arise from scheduling policy. For example, dynamic scheduling which is flexible and gives efficient resource utilisation may lead to unpredictable time delays (Cervin *et al.*, 2003). Specifically, the scheduling itself requires amount of time to execute some operations (e.g. context switching, interrupt handling) which may lead to a control process or task delay (Lin and Herkert, 1996), or the scheduling policy which manages a task queue for execution may let a high priority task execute first and then causes another task is blocked and delayed (Audsley *et al.*, 1993a). In addition, because the three main processes (i.e. sampling, control computation, and actuation) in a control loop perform sequentially and introduce various forms of jitter, these jitters are characterised as: sampling jitter, sampling-actuation delays, and sampling jitter and sampling-actuation delay (Marti *et al.*, 2001a).

In distributed real-time systems that consist of various nodes and a communication network connecting the various systems, jitter problem can arise from, such as, delay in network caused by a route consisting of several hops (Baruah *et al.*, 1999), network protocol (e.g. Ethernet, Computer Area Network (CAN)) (Tindell and Burns, 1994), and the variations in message transmission times (Nolte *et al.*, 2002). In a CAN bus, bit-stuffing mechanism will insert an additional bit of opposite polarity when five consecutive bits are transmitted on the bus with the same polarity (e.g. 11111 or 00000). This is done in order to provide edges to allow receivers to re-synchronise internal timing. However, this then causes one problem of the variation in frame length. The work in (Nolte *et al.*, 2002) aims to reduce the number of stuff-bits by transforming the message data using an XOR operation on the data together with a bit-mask and avoiding the occurrences of stuff-bits in the CAN header. Whereas, the work in (Barreiros *et al.*, 2000; Coutinho *et al.*, 2000) apply ability of genetic algorithms which is a search technique to manage the schedule of message transmission in order to minimise jitter. However, the technique requires a relatively high computational overhead.

Typically, jitter can be divided into two categorises which are random jitter and deterministic jitter. The deterministic jitter is further divided into three categories: periodic jitter, data-dependent jitter, and bounded uncorrelated jitter (Wavecrest,

2001) (Ou *et al.*, 2004). The detailed characteristics of these jitter types are described in Appendix A.

3.3 Jitter in scheduling systems

As noted in Chapter 1, a real-time scheduler plays an important role in embedded systems in order to manage and control system activities. Selecting an appropriate scheduling algorithm for the application is important. An inappropriate choice can lead to unwanted behaviour (e.g. jitter) that may cause system performance to be degraded. The advantages and disadvantages of various common scheduling architectures for real-time systems are presented and discussed in Appendix B.

An embedded system performs various activities which are usually identified as tasks. Ideally, real-time scheduler should schedule and execute these tasks precisely. However, in practice, the activity execution is not ideally accurate even using such real-time scheduling systems because of the use of imperfect scheduler implementations, improper scheduling algorithm selection, design and implementation decisions, constraints in application requirements and so on (Audsley *et al.*, 1993a; Marti, 2002). These factors may result in task jitter with a corresponding impact on system performance (Marti *et al.*, 2001b; Proctor and Shackleford, 2001; Cervin *et al.*, 2003).

3.3.1 Task jitter (Overview)

In real-time scheduling systems, task execution may deviate from the expected time as a result of scheduling mechanism or other reasons: this then causes task jitter.

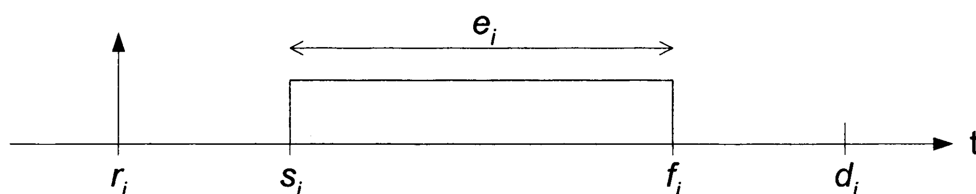

Figure 3.2: Primary parameters of a task

In general, timing parameters of a task can be described as follows.

- Release time (also known as arrival time) is the instant of time at which a task becomes ready for execution, denoted by r_i,.
- Start time is the instant of time at which a task starts its process, denoted by s_i.

- Execution time is the amount of time for which the processor executes a task from the beginning to the completion (without interruption), it is denoted by e_i.
- Finishing time is the instant of time at which a task completes its process, it is denoted by f_i.
- Deadline is the instant of time by which the task should be completed, it is denoted by d_i.

In this section, the definition of task jitter is characterised as release jitter, execution jitter and finishing jitter. An overview of each of these forms of jitter is presented in the sections which follow.

3.3.2 Task jitter - Release jitter

Release jitter is the deviation of the actual start time of tasks. Theoretically, every task should be executed immediately at the time at which it is released, called the release time, but the actual start time of task can be delayed which mainly arises from task scheduling method, or blocking (Audsley *et al.*, 1993a; Liu, 2000). For example, if task A is released at time X while there is another task executing, this results in task A has to be delayed and executed at time $X+\Delta t$. In practice, it is difficult to define the start time of each task because there are many reasons to vary the start time in real situation. It can only be defined the range of delay time respected to the release time (i.e. the earliest start time and the latest start time). This range of the release time is called as *release-time jitter* or *release jitter*. The *release jitters* can be divided as *absolute release jitter* and *relative release jitter*, as shown in Equation 3.1 and Equation 3.2 respectively (Buttazzo, 2004). The *absolute release jitter* describes the maximum deviation of the start time among all task instances and the *relative release jitter* describes the maximum deviation of the start time of two consecutive instances.

$$\text{Absolute release jitter} = \max_k (s_{i,k} - r_{i,k}) - \min_k (s_{i,k} - r_{i,k}) \quad (3.1)$$

$$\text{Relative release jitter} = \max_k \left| (s_{i,k} - r_{i,k}) - (s_{i,k-1} - r_{i,k-1}) \right| \quad (3.2)$$

3.3.3 Task jitter - Execution jitter

Execution jitter is the deviation of the execution interval of tasks. The actual amount of time taken by a task to complete its process may vary for many reasons. For

example, program code of a task may contain conditional branches, and these conditional branches may take different amounts of time to complete (Liu, 2000). The branches taking during the execution time depend on the input data and, moreover, the amount of execution time may vary if the performance enhancing features, such as cache memory and pipeline, are employed – even there are no condition branches in the process (Eisenbeis and Windheiser, 1993; Li and Malik, 1995). In a real application, the actual amount of time to transmit each compressed video frame of a MPEG always varies because the numbers of data in each frame are different. Although the actual execution time of the task is unknown, the execution time of the task can be determined as a range of the minimum execution time and the maximum execution time (Liu, 2000). The range of execution time is called as *execution jitter*, and also categorised as *absolute execution jitter* and *relative execution jitter,* see Equation 3.3 and Equation 3.4 (Buttazzo, 2004).

$$\text{Absolute execution jitter} = \max_k (f_{i,k} - s_{i,k}) - \min_k (f_{i,k} - s_{i,k}) \qquad (3.3)$$

$$\text{Relative execution jitter} = \max_k | (f_{i,k} - s_{i,k}) - (f_{i,k-1} - s_{i,k-1}) | \qquad (3.4)$$

3.3.4 Task jitter - Finishing jitter

Finishing jitter is the deviation of the finishing time of tasks. The finishing time can be varied by inherent timing such as *release jitter* or *execution jitter*. Moreover, the finishing time can be interrupted by the execution of other high priority task or can be blocked when trying to access shared resources, shared data memory, transmission links, or other I/Os (Audsley *et al.*, 1993a; Tindell and Burns, 1994). For these reasons, they cause finishing jitter of tasks in the real-time systems. The finishing times can be predicted the occurrence within the range of the earliest finishing time and the latest finish time. The *finishing jitters* are also divided as *absolute finishing jitter* and *relative finishing jitter*, see Equation 3.5 and Equation 3.6 (Buttazzo, 2004).

$$\text{Absolute finishing jitter} = \max_k (f_{i,k} - r_{i,k}) - \min_k (f_{i,k} - r_{i,k}) \qquad (3.5)$$

$$\text{Relative finishing jitter} = \max_k | (f_{i,k} - r_{i,k}) - (f_{i,k-1} - r_{i,k-1}) | \qquad (3.6)$$

3.3.5 Jitter in a periodic tasks (Overview)

In many real-time applications, most activities are periodic. Periodic activities normally arise from sensory signal acquisition, data encoding / decoding, data display, system monitoring, or DC motor controlling (Pont, 2001). Such activities require tasks to be executed periodically at rates dictated by the application requirements.

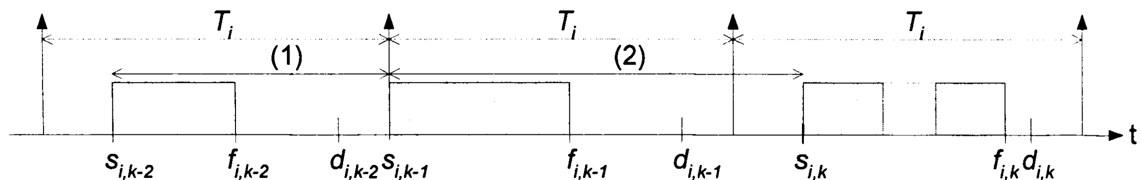

Figure 3.3: Task period jitters (adapted from Marti, 2002, Figure 4.1).

3.3.6 Jitter in a periodic tasks - Task period jitter

Ideally, tasks in a periodic task set are expected to be executed repeatedly at their own periods. Figure 3.3 illustrates a periodic task is assigned to run every T_i period. The *(k-2)*th task instance starts executing later than the *(k-1)*th task instance starts when comparing with their release times. This results in the time interval (1) is shorter than T_i whereas the time interval (2) is greater than T_i. The inconsistent interval of the start times causes task period jitters and this also affect to finishing time which can cause worse finishing jitter.

3.3.7 Jitter in a periodic tasks - Sampling and sampling-actuation jitter

In computer control applications, jitter, caused by the variation of task execution, can impose to the timing of closed-loop control activities (Marti *et al.*, 2001b). Typically, a closed-loop control consists of the three main activities (i.e. sampling, control computation and actuation) and they are implemented in a task (or possibly in multiple tasks). The sampling takes place at the beginning of each closed-loop task while the actuation takes place at the end of each closed-loop task, and the closed-loop task run periodically. As closed-loop control tasks are run by a real-time scheduler, time intervals between consecutive sampling are not constant due to inconsistency of start times: this causes *sampling jitter*. Moreover, the elapsed time between the start and finishing time of task can also be varied by scheduling algorithm. This results in a variable time interval between sampling and actuation which results in *sampling-*

actuation jitter. The level of these jitters which can impact on a control system is dependent on scheduling algorithm and design (Marti, 2002).

3.4 Low-jitter scheduling algorithms

In this section, key scheduling algorithms which have the potential to provide low levels of jitter are illustrated and reviewed.

3.4.1 Time-triggered scheduling

In real-time scheduling systems, the triggering mechanisms can be divided into two different approaches: event-triggered and time-triggered (Kopetz, 1991; Bennett, 1994; Pont, 2001). The focus throughout this thesis will be on time-triggered architectures and periodic task sets. Time-triggered and event-triggered architectures are compared in this section.

In a time-triggered (TT) schedule, all tasks are activated at specific time instants which are known before the system starts execution. To implement such behaviour in a single-processor design, a hardware timer will usually be used to generate periodic interrupts (Bennett, 1994). In distributed (multi-processor) systems, all nodes are synchronised by means of some form of global clock (Kopetz, 1997; Pont, 2001). In a (pure) event-triggered (ET) system, all tasks are activated in response to significant external incidents (e.g. the depressing of a push button by a user, the activation of a limit switch, or the arrival of a new message at a node): such events can occur at any time (Kopetz, 1997). In practice, an event-related signal might be obtained from a sensor for detecting environmental activities and presented to the system in the form of an interrupt request.

Both ET and TT approaches have strengths and weaknesses (and many systems will employ both ET and TT tasks). The main advantage of (pure) TT scheduling is the deterministic temporal behaviour of the system that results: this makes it easier to validate, test and certify the system (Liu, 2000; Marti, 2002). TT architectures require careful planning during the design phase: without this, predictable behaviour will not be achieved. In general, ET systems are more responsive and flexible (Kopetz, 1991).

However, ET can fail under heavy load conditions and also require more overhead for responding to unpredictable loads (Marti, 2002).

3.4.2 The Time-Triggered Co-operative Scheduler (Cyclic executive)

A time-triggered co-operative (TTC) scheduler - also known as a cyclic executive or a time-line scheduler – is the simplest form of practical TT architecture. A TTC is a real-time scheduler that divides the application into a set of tasks which are assumed to be non-preemptible (Locke, 1992). The TTC scheduler schedules these tasks by activating them at a time instant according to their predefined sequence. In many TTC designs, the sequence of task activations is computed at the design phase and then stored as fixed schedule, usually in some form of table. These tasks are repeatedly executed in this fixed sequence throughout the period of system operation (Bate, 1998; Liu, 2000; Buttazzo, 2004).

The operation of a TTC scheduler is often viewed in terms of a "major cycle" and a "minor cycle" (Baker and Shaw, 1989; Locke, 1992; Burns *et al.*, 1995). The major cycle is the amount of time which elapses before the sequence of tasks is repeated. When scheduling individual tasks, the major cycle may be partitioned into small segments known as minor cycles (or frames) (Liu, 2000).

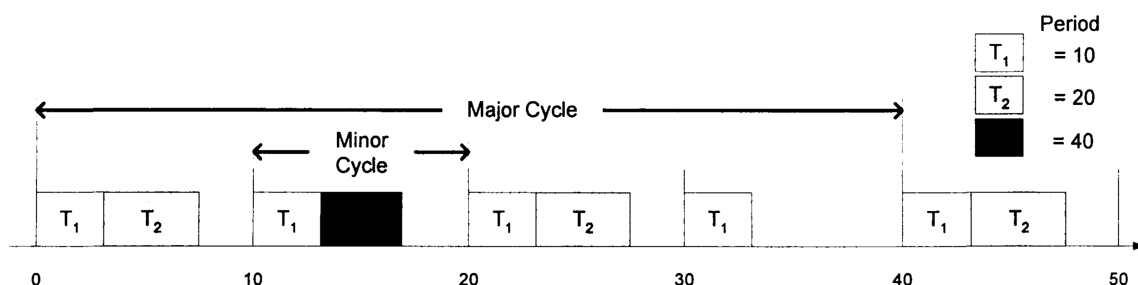

Figure 3.4: General structure of TTC scheduler: minor cycle = 10, major cycle = 40

The length of the major cycle can be determined by using the least common multiple of the period of each periodic task (Locke, 1992; Buttazzo, 2004). For example, in Figure 3.4, the periods of the tasks are 10, 20, and 40 ms. The least common multiple of these periods is 40, therefore the length of the major cycle in which all tasks will be executed periodically is 40 ms. The minor cycle of the task set can be set equal to or less than the greatest common divisor value of all task periods. Figure 3.4 illustrates

that the tasks are scheduled on the minor cycle which has been set as 10 ms. In practice, the minor cycle is implemented by using a periodic timer interrupt or by some other periodic external hardware interrupt in order to synchronise the system with its external environment (Locke, 1992; Pont, 2001).

In order to verify task schedulability, the worst-case execution times of all task must be known (Buttazzo, 2005). If the total execution time of each minor cycle is less than or equal to the minor cycle, a schedule feasibility is guaranteed. In normal situations, the tasks are run and completed by the next minor cycle – as seen in Figure 3.4, there is usually "slack time" between the end of task(s) and the start of the next minor cycle. In case that there is a failing task running in system and it cannot be completed within its minor cycle, this failing task can cause a domino effect on the other tasks which this situation can break the entire schedule (Buttazzo, 2004). To detect the task overrun or a processor overload, it can be determined by observing that the interrupt has occurred when the system is not run as background process or in idle state. This means that if the system is executing a task when timer indicates that the end of the minor cycle has been reached, the scheduler can detect that a task has overrun (Bate, 1998; Hughes and Pont, 2004).

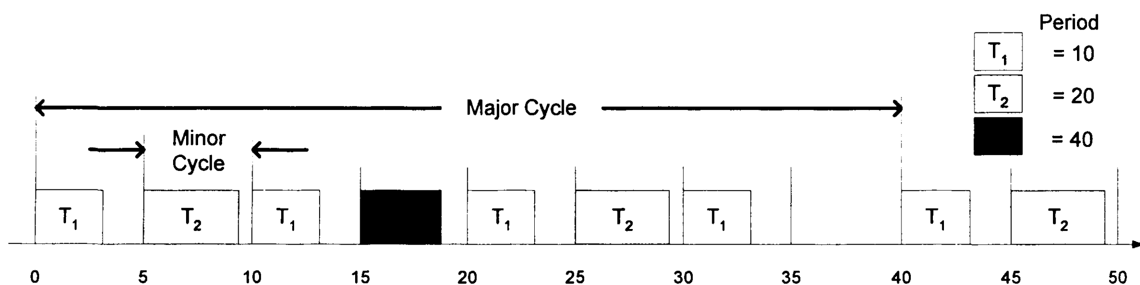

Figure 3.5: Structure of TTC scheduler: minor cycle = 5, major cycle = 40 (adapted from Locke, 1992, Figure 1).

Alternatively, when run with an offset or phase (Locke, 1992; Audsley *et al.*, 1993b), it is possible to have minor cycle which is less than the greatest common divisor, 5 ms, as shown in Figure 3.5, if the worst-case execution times are less than the minor cycle. This can result in very low (both release and output) jitter in the execution of each task's activation in this scenario. All tasks executes at start times that are trig-

gered by a timer interrupt, therefore, these tasks will suffer from less jitter in their start times (when compared to the previous example: for example, see Figure 3.4, in which task T_2 and T_3 may suffer jitter from the variation of the execution time of task T_1). However, the design that has faster minor cycle, as illustrated in Figure 3.5, will have a higher scheduling overhead due to the more frequent interrupts (Locke, 1992).

There are several advantages of a TTC scheduler. The major attractiveness is its simplicity (Liu, 2000; Pont, 2001). By programming a timer to interrupt at the same rate as minor cycle, this timer interrupt can be used as a trigger to call each task sequentially in order to synchronize with the task timing. There is no need to be concerned about the integrity of shared data structures or shared resources during the development of systems based on such a scheduler (Locke, 1992). The runtime overhead of a TTC scheduler is also very low because there are no unexpected context switches as can be found in pre-emptive schedulers (Locke, 1992; Buttazzo, 2004). The TTC scheduler also presents very low jitter because all tasks run regularly according to their queues in a deterministic manner (Locke, 1992; Bate, 1998). Thus, there is no unexpected interruption or blocking to make large variations affecting on task schedule.

However, fragility is the main drawback of TTC scheduler (Locke, 1992; Bate, 1998). If there are any changes to the system during development, such as changes in its requirements or due to software errors, or when maintaining the system, such as adding new functions, it may result in significant changes to the schedule and / or to existing task implementations. Making such changes manually can be time consuming. Note, however, that recent tools for "automatic code generation" may help to reduce the work involved in developing and maintaining such systems: see, for example, (Mwelwa *et al.*, 2006; Kurian and Pont, in press).

Another problem with TTC scheduler is when the system has a task that its execution time is long compared to the period of the highest rate periodic task (Locke, 1992). Because pre-emption is not allowed in TTC scheduler, the solution is to break the long task into multiple short parts that do not exceed minor cycle arbitrated by a high rate task (Pont, 2001). Two alternatives are to use a rate monotonic scheduler (as

discussed in Section 3.4.3), or to use a "hybrid" scheduler (as discussion in Section 3.4.4).

3.4.3 Rate monotonic

Where a TTC scheduler is not found to be suitable for use in particular low-cost embedded systems, fixed-priority scheduling has been proposed as the most attractive alternative (Audsley *et al.*, 1993b; Bate, 1998).

Rate monotonic (RM) is a well-known fixed-priority scheduling algorithm that was introduced by (Liu and Layland, 1973) in 1973. Technically, rate monotonic is a dynamic pre-emptive algorithm which is based on a fixed-priority assignment (Kopetz, 1997). In particular, the priorities assigned to periodic tasks accord to their occurrence rate or, in other words, priorities are inversely proportional to their period, and they do not change through out of the operation (because their periods are constant). The RM algorithm was proved by (Liu and Layland, 1973) to be optimal amongst all fixed-priority algorithms: that is, Liu and Layland demonstrated that – if it is possible to schedule a task set using a fixed-priority algorithm and meet all of its timing constraints – then a rate-monotonic algorithm can achieve this. Theoretically, every task can meet its deadline if the total utilization is equal or less than 69% and under these assumptions: all tasks are periodic and independent of each other, deadline of every task is equal to its period and the worst-case execution time of all tasks is known, and context switching time can be ignore (Liu and Layland, 1973; Locke, 1992; Bate, 1998; Buttazzo, 2004). Thus, it may be an attractive option in systems where a fully predictable hard real-time is required.

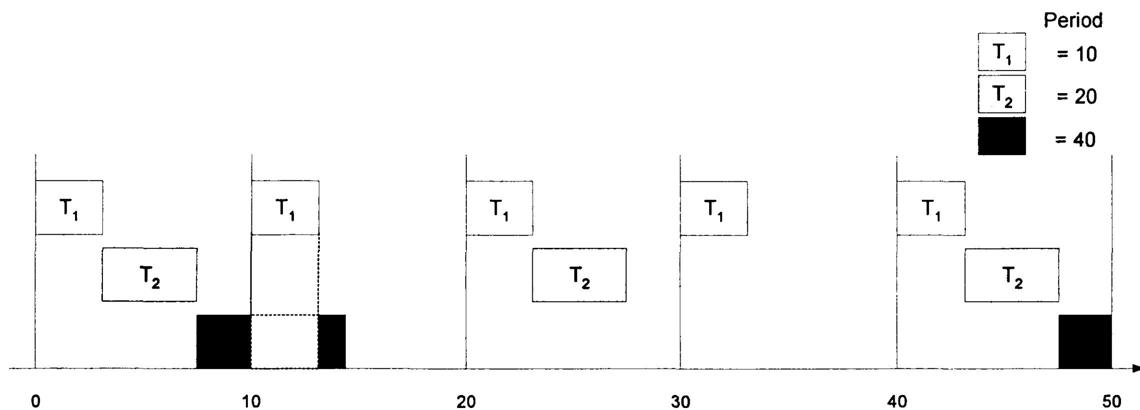

Figure 3.6: Structure of rate monotonic scheduling (adapted from Locke, 1992, Figure 3).

To illustrate this process, Figure 3.6 shows the set of periodic tasks is scheduled by the RM algorithm. Task T_1 is executed periodically at the fastest rate, every 10 ms, and is determined to be the highest priority in this scheduling policy, while task T_2 and T_3, which are run every 20 and 40 ms respectively, have lower priority levels according to their rates. A task scheduled by the RM algorithm can be pre-empted by a higher priority task. As illustrated in Figure 3.6, task T_3 - which is running - is pre-empted by task T_1 is at time 10: it carries on after the completion of task T_1. Generally, the deadline of a task in RM scheduling is defined as the period. By comparison, the completion times of tasks in a TTC scheduler are usually limited to a frame or minor cycle: this can be said to make the TTC scheduler more “rigid” in design.

It has been claimed that the main advantage of rate monotonic scheduling is flexibility during design or maintenance phases, and that such flexibility can reduce the total life cost of the system (Locke, 1992; Bate, 1998). The schedulability of the system can be determined based on the total CPU utilization of the task set: as a result - when new functionalities are added to the system – it is only necessary to recalculate the new utilization values. In addition, unlike a TTC design, there is no need to break up long individual tasks in order to meet the length limitations of the minor cycle. The need to employ harmonic frequency relationships among periodic tasks is also avoided. Finally, the scheduling behaviour can be predicted and analysed using a task model proposed by Liu and Layland (1973).

However, the scheduling overheads of rate monotonic schedulers tend to be larger than those of TTC schedulers because of the additional complexity associated with the context switches when saving and restoring task state (Locke, 1992). This is a concern in embedded systems with limited resources. Indeed, it has been demonstrated that another popular pre-emptive scheduler (EDF) has a lower runtime overhead than RM approaches (Buttazzo, 2005). Even though EDF always needs to update task deadlines this increased load may be offset by a reduction in the number of pre-emptions that occur under EDF (with a consequent reduction in context-switching time). Overall, Buttazzo (2005) suggests that the real advantage of rate monotonic is its simpler implementation.

Of greater concern in this thesis is that RM scheduling seems likely to have more jitter than TTC scheduling, because the pre-emption from higher priority tasks may interrupt or block the lower priority tasks. These interferences may delay the release time of tasks, or interrupt running tasks and then prolong the output of a process residing at the end of a task: this may which result in jitter (Buttazzo, 2004). For example, in Figure 3.6, the output jitter can take place when task T_3 is pre-empted by task T_1.

Overall, in the type of "low jitter" application with which this thesis is concerned, use of an RM algorithm presents two main challenges.

The first challenge is that the RM algorithm is based on the assumption that task deadlines are equal to periods: this means that use of RM guarantees only that a given task will complete its execution before it is due to run again. For short tasks, this means that jitter rates may be in the region of 90% (of the sample period), and the schedule will still be "correct". As noted previously, even jitter levels of 10% can render sampled data meaningless. Note that use of high task priorities will tend to reduce jitter levels: however – even if the tasks are wholly independent – the only safe assumption is that the highest-priority task will be guaranteed to demonstrate very low jitter levels (Locke, 1992).

The second challenge is that tasks are unlikely to be independent and that more than one task may require access to a mutually-exclusive resource (e.g. serial port, ADC and etc.). Where such critical sections are accessed through semaphores, even the highest-priority task may be blocked by a lower priority task – a process known as priority inversion – and then experience jitter or delay (Buttazzo, 2005). The priority inversion problem can be "solved" by using concurrency control protocols (e.g. Priority Inheritance Protocol or Priority Ceiling Protocol, developed by Sha et al. (1990)) to access shared resources: however, such techniques were developed to address problems of deadlock and their impact on jitter is not always easy to predict.

3.4.4 The "hybrid" scheduler

A simplified pre-emptive scheduler with characteristics that lie between those of TTC and rate-monotonic designs has also been described in previous studies (Pont, 2001; Maaita and Pont, 2005). This time-triggered hybrid (TTH) scheduler is a code-derived algorithm: that is, it originated from an observation that – by invoking a task from the interrupt service routine in a TTC implementation – a scheduler supporting limited degrees of pre-emption can be created (Pont, 2001).

Use of a TTH scheduler allows the system designer to create a static (fixed-priority) schedule made up of (i) a collection of tasks which operate co-operatively and (ii) a single – short – pre-emptive task. In many designs, the pre-emptive task will be used for periodic data acquisition, typically through an analogue-to-digital converter or similar device: such requirements are common in, for example, a wide range of control systems (Buttazzo, 2005).

The operation of the TTH scheduler is illustrated schematically in Figure 3.7. This figure shows the situation where a short pre-emptive task is executed every millisecond, while a co-operative task (with a duration greater than 1 ms) is "simultaneously" executed every 3 milliseconds.

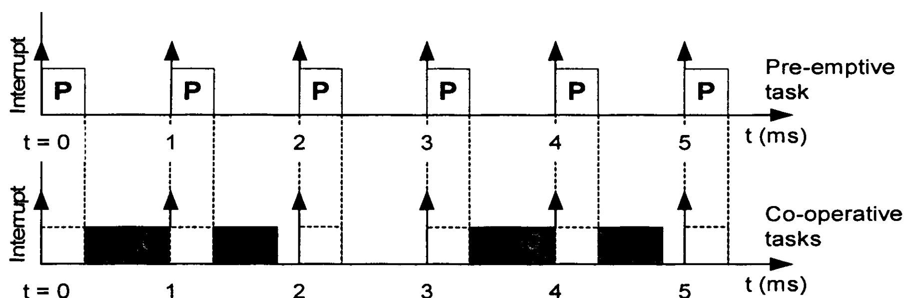

Figure 3.7: Illustrating the operation of a TTH scheduler. See text for details

In many cases, a TTH (Pont, 2001) implementation will be used to implement an RM (or similar) schedule. It should be reiterated that only a single pre-emptive task is supported in this architecture. As a consequence, in terms of a theoretical analysis, this type of scheduler is of limited interest. However, TTH algorithm implementation is simpler than rate monotonic and its context switch is very low – as supported only one pre-emption. For a jitter issue, the pre-emptive task can be expected to have low

jitter as the highest priority task of rate monotonic whereas execution time of the pre-emptive task directly impacts on jitter of co-operative tasks.

Thus, in a low-cost embedded system, TTH is a very attractive proposition because it allows us to create a scheduler with minimal resource requirements which is *precisely* matched to the needs of many applications that are interesting in developing.

3.5 Conclusions

This chapter has presented a review of some of the key sources of jitter in embedded systems. In keeping with the requirements introduced at the start of this thesis, the aim is to explore software architectures with minimal resource requirements, low task jitter and low energy consumption. The particular focus will be on systems employing TTC and TTH schedulers.

In Chapter 4, in order to reduce energy consumption, the implementation of a combined TTC / DVS technique is explored. The study considers both the impact on system timing and on energy consumption.

Chapter 4

Implementation of DVS in a TTC scheduler

As discussed in Chapter 2, Dynamic Voltage Scaling (DVS), has been developed as a means of balancing performance and energy consumption in scheduled systems. As discussed in Chapter 3, TT architectures (and in particular TTC schedulers) can result in systems with very low levels of jitter. The aim of the work in this chapter is to consider the impact (on both jitter and energy consumption) of integrating DVS and TTC techniques in a single scheduler. The basic TTC-DVS scheduler implementation developed and assessed in this chapter will then form the basis for studies described in subsequent chapters of this thesis.[8].

4.1 The hardware platform

The hardware platform used in the studies presented in this chapter is described in this section.

4.1.1 Choice of processor

In order to apply DVS, the processor must provide separate power supply pins for the CPU core and I/O[9]. Generally, DVS can be applied with a wide range of existing

---

8 Part of this chapter has previously been published in different forms in Phatrapornnant, T. and Pont, M.J.(2004b) and Phatrapornnant, T. and Pont, M.J. (2006).

9 If CPU power and I/O are not separated in this way, then any changes to the CPU voltage will also have an impact on the I/O circuitry. The consequences of changes to the I/O voltage are very hard to predict (and may involve changes to the state of I/O pins due to the application of DVS). Clearly, such consequences must be avoided.

microcontrollers and microprocessors. For example, Intel StrongARM SA11x0, ST-Microelectronics STR71x, Philips LPC 2000 series, which are based on ARM architecture. Other processors which support DVS (and which are based on different architectures) include the Transmeta Crusoe, Intel XScale, UltraSPARC-III, and x-86 architecture, such as AMD: K6-2, K6-3, Duron, Athlon, Intel: Pentium III, Pentium 4, Pentium M.

The studies reported in this chapter (and throughout this thesis) used a Philips LPC-2106 processor. The LPC2106 is 32-bit microcontroller with an ARM7-core which can run – under control of an on-chip PLL – at frequencies from 10 MHz to 60 MHz (Philips, 2003). It is a low-cost device (well under $10.00 per unit) which is felt to be a good example of a modern resource-constrained processor which is suitable for use in a wide range of embedded systems.

4.1.2 A variable voltage power supply

A dynamic voltage scaling technique often employs a DC-DC converter and frequency synthesizer to control the supply voltage and clock frequency.

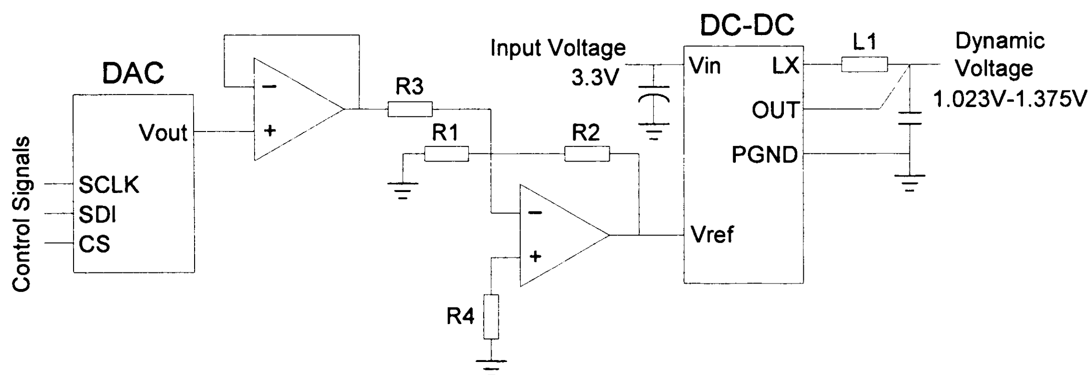

Figure 4.1: Schematic circuit of DVS power supply

To vary core voltage of processor, it is required power supply which can adjust output voltage dynamically. In this study, a digital to analog converter, DAC, receives input commands to select the requested core voltage from a processor via serial-peripheral-interface, SPI (see Figure 4.1). The DAC output voltage modifies reference voltage of the DC-DC for producing the required dynamic voltage. Operational amplifiers (op-amp) and resister network are employed to set the initial reference voltage of DC-DC converter when the system is turn on which there is no control signals from a proces-

sor. After a processor starting and running, the output voltage will be controlled via SPI as normal.

4.1.3 Voltage and frequency steps

The main factor which determines how many DVS voltage / frequency steps can be employed in a DVS implementation is hardware support on the target processor, such as the range of operating frequencies or the programmable frequency synthesizer.

After defining the frequency steps, the minimum voltage levels of each frequency step that allow a processor run in the right manner must then be determined. If there are a large number of possible frequency steps – for example the SA-1110 allows its PLL to synthesize frequency 12 steps which are in range 59 – 221 MHz (Intel, 2001) - it may be necessary to group some frequency steps so that they employ the same voltage level. Overall, having a large number of steps may provide an opportunity to save more power. However, increased numbers of steps will also increase memory requirements for the DVS algorithm.

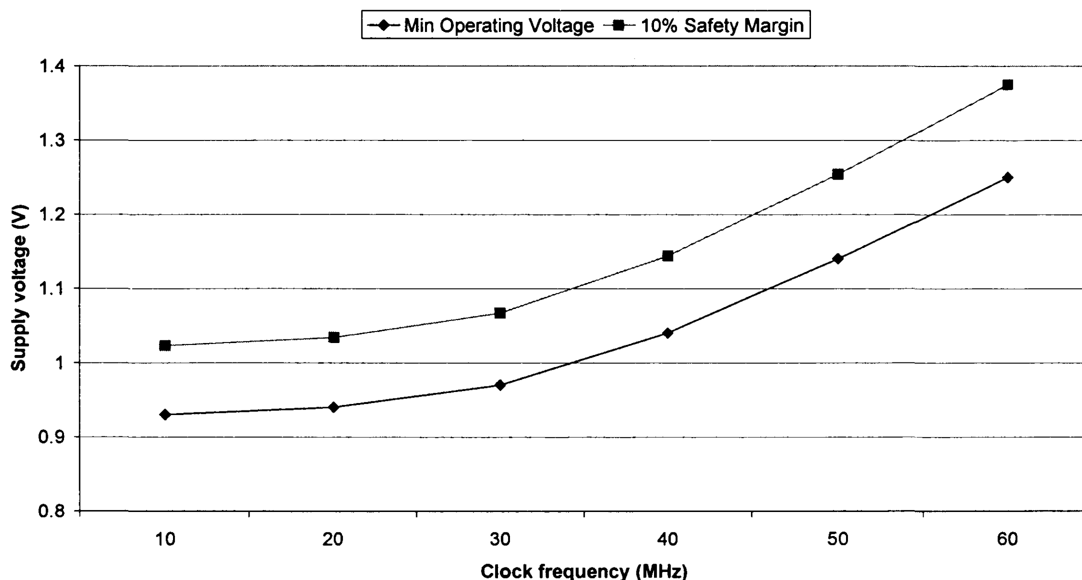

Figure 4.2: Setting the CPU supply voltage (LPC 2106)

In the studies described in this thesis, the CPU speed was divided into 6 steps. An external crystal oscillator of 10 MHz was linked to a PLL multiplier (1x to 6x) to generate 10, 20, 30, 40, 50 and 60 MHz system clocks. The supply voltage also had 6 levels corresponding to the 6 speed steps: in other words, there are 6 frequency/

voltage pairs. The required voltage level at each speed was determined empirically, by measuring the lowest voltage at which processor still worked properly and then adding a 10% safety margin (Figure 4.2).

4.1.4 System integration

For the purposes of these studies, the LPC2106 was mounted on an Ashling LPC2000 evaluation board (Ashling, 2003). The board provided separate 1.8V (CPU core) and 3.3V (I/O) supply jumpers, making it easy to implement a variable-voltage CPU supply. An external digital to analogue converter (DAC) was controlled by the processor via an SPI interface. This DAC was used to vary the reference voltage on a DC-DC converter, thereby producing the required CPU voltage.

4.2 A TTC scheduler

In order to study the potential impact of DVS on jitter, a TTC scheduler (as discussed in Section 3.4.2) was employed. However, the TTC scheduler is only an algorithm: in practice, there are many possible ways to implement such a scheduler.

A simple TTC scheduler implementation can be created by using interrupt service routine (ISR) linked to the overflow of hardware timer. The timer can be set to generate an interrupt at the same or higher rate than that of the task that runs at the highest frequency. When the timer expires (overflows) and an interrupt occurs, tasks will be activated from the ISR directly. This method can be easily used to implement a simple TTC scheduler (e.g. see Pont, 2001). Such a scheduler will have little or no release jitter (Kurian and Pont, in press). However, in use, such a scheduler requires a significant amount of hand coding to control the task timing and the code of scheduler also becomes combined with the code of the application (Kurian and Pont, in press).

4.2.1 Scheduler structure

In this study, the TTC scheduler was based on that described in (Pont, 2001). This is a more complete scheduler implementation. The structure of the scheduler is illustrated in Listing 4.1).

```
begin MAIN:

    Initialise the Scheduler and Timer ;
    Initialise all tasks ;
    for 1 to N tasks
        ADD_TASK (Task, Delay, Period) ;
    end for
    SCHEDULER_START () ;
    while (1)
        DISPATCH_TASKS () ;
    end while

end MAIN
```

Listing 4.1: An overview of possible TTC scheduler implementation

TTC scheduler contains *Update* and *Dispatcher* functions which are employed to manage task execution. The *Update* function (see Listing 4.2) is the scheduler ISR which is invoked by the overflow of timer (the "tick interrupt"). This function is very short and intends to use for updating a flag to notify the *Dispatcher* when it is due to determine tasks to run. The *Update* function does not execute any tasks directly, as run for supervision the system, but the tasks that are due to run are invoked through the *Dispatcher* function. The split between the *Update* and *Dispatcher* operation is aimed to maximize the reliability of the scheduler in the presence of task which is longer than a tick interval (Pont, 2001). If the *Update* invokes the tasks directly while the long task is being executed, the ISR interrupts are effectively ignored. Furthermore, the TTC scheduler also supports dynamic schedule that can add or remove tasks from the schedule at any time during the program execution (Kurian and Pont, in press). The diagram of the TTC scheduler operation is illustrated in Figure 4.3.

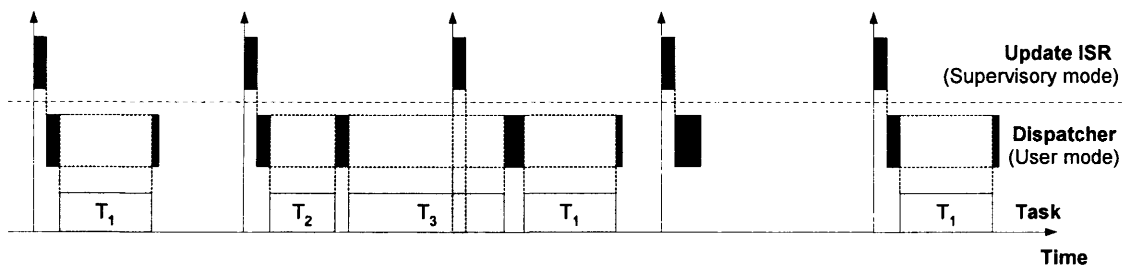

Figure 4.3: TTC scheduling diagram

```
begin UPDATE ISR:

     Tick_Count ++ ;
     reset timer_interrupt_FLAG ;

end  UPDATE
```

Listing 4.2: Pseudo code of the Update function in the TTC design

4.2.2 TTC Dispatcher

In general, the Dispatcher is a function which gives control of the CPU to process task switching. At run time, the Dispatcher is invoked after a clock tick (scheduling tick) occurs. It will test all tasks from its list to find which tasks are due to run. The Dispatcher functions then releases these tasks in order. The task priority is normally solved by Dispatcher by selecting the highest priority task to execute first.

```
begin DISPATCH_TASKS:

     while Tick_Count > 0
          for loop all tasks in tasks array
              if -- Delay == 0 then
                 release Task ;
                 if it is periodic task then
                    reload Delay = Period ;
                 else
                    delete Task from array ;
                 end if
              end if
          end for
          Tick_Count -- ;
     end while
     sleep ;

end DISPATCH_TASKS
```

Listing 4.3: Task Dispatcher of TTC algorithm

In the TTC scheduler, the task priority is fixed and the task at the top of the list is automatically defined as highest priority because it will be released first if it is due to run. In most real-time schedulers, latency of dispatch process which takes form the time of a previous task stop to the time of another task start is quite short. In the TTC scheduler, the detail of Dispatcher function is shown in Listing 4.3.

To implement DVS, voltage and frequency scaling must be performed before tasks are dispatched. In this study, the focus was on the Dispatcher function in order to incorporate a DVS algorithm in the scheduler.

4.3 Applying DVS in a TTC scheduler

The key to applying DVS in a TTC application is the presence of slack time. Under DVS, tasks – which normally run at the same, fixed, CPU speed – will be stretched to fill the available slack time (see Figure 4.4). Therefore, the speed-setting policy is determined by the available slack time for each task.

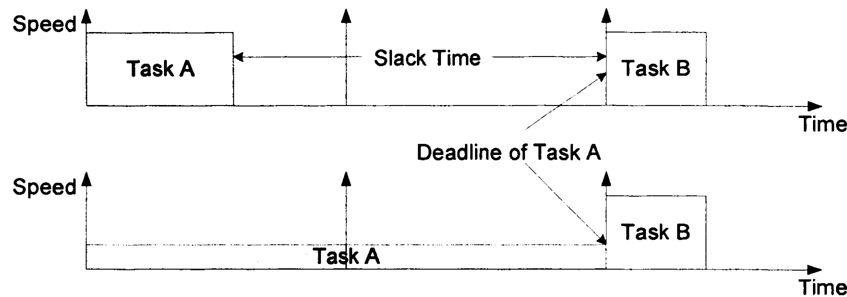

Figure 4.4: Example illustrating the possibility of task stretching

In theory, the available execution time for a task is simply the interval between its release time and the time of the release of the next task (e.g. see Figure 4.4). However, the overhead of the scheduling algorithm itself must also be considered in practical systems (Figure 4.5). The scheduling overhead which added the DVS processes is comparatively large when comparing to those of the conventional scheduler. For example, in the present study schedulers with a 1 ms "tick" interval were employed and – on the hardware platform used – the scheduler overhead exceeded 15% of the tick interval. This overhead consists of two key components: speed finding (including task scheduling) and frequency/voltage scaling processes.

The speed finding (also called voltage setting calculation (Andrei *et al.*, 2005; Cai *et al.*, 2006)) is a process which is used for calculating the appropriate CPU clock frequency to complete tasks within their timing constraints (i.e. deadlines). In general, speed finding is incorporated as part of a scheduler in online DVS schedulers while, in offline DVS schedulers, it can be done at compile time (Andrei *et al.*, 2005; Zhuo and Chakrabarti, 2005; Cai *et al.*, 2006). Whereas, the frequency/voltage scaling process (or frequency/voltage transition) is used for altering CPU clock frequency and also CPU core's supply voltage (according to the frequency/voltage pairs) before a task released. In practice, the time taken of the frequency/voltage scaling process mainly depends on hardware setup.

Typically, the timing behaviour of output voltage from a power supply depends on the values that use for the output and input capacitors, coils and feedback resistor network (Chew, 2002). For example, MAX1820, pulse-width-modulated DC-DC buck regulator, gives the output voltage settles in less than 30 μs for a full-scale change in voltage, 0.4V to 3.4V (Maxim, 2005) while the voltage settling time from other power supply can be longer, 100-200 μs (Dhar *et al.*, 2002). Generally, it is particularly interested in the rise time of output voltage when the required output voltage is increased according to run at higher speed. If the system runs at high speed with lower voltage than its voltage specified, it may halt or experience unusual behaviour. However, in this study, the overhead also consists of other processes, such as frequency scaling, controlling DAC to adjust output voltage of a DC-DC converter, that totally take about 160-190 μs. That make overhead of DVS scheduling significantly large when comparing to that of conventional scheduling.

To keep a number of voltage switches at minimal, the scheduling and DVS processes are run at the same speed of previous task. In Figure 4.5, After Task A finish, the operating frequency is still the same as previous but changed when releasing a new task, Task B.

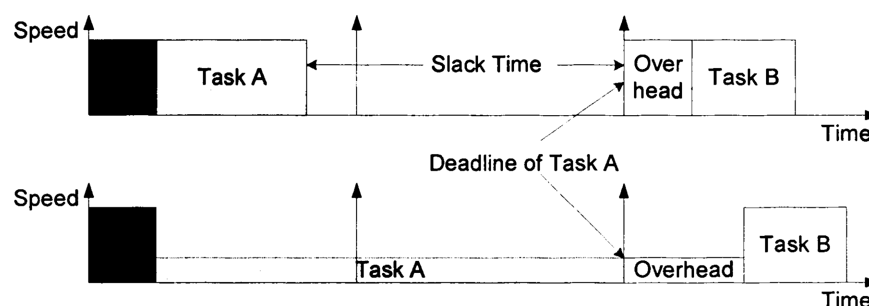

Figure 4.5: Illustrating a more realistic DVS implementation

4.4 DVS Algorithms

The overhead of the DVS scheduling system will be larger than that of conventional scheduling system. In this section, the purpose of study is to explore the impact of overhead when applying DVS and explore way to minimise this overhead.

There are many possible ways of implementing DVS and a number of possible designs were explored. Four of these designs are described in this section, and compared experimentally in Section 4.5.

The four designs were named as follows (for ease of reference):

- The Compute-Direct (CD) algorithm
- The Lookup Table (LT) algorithm
- The Circular Array (CA) algorithm, and,
- The Circular Skip (CS) algorithm

Each of these algorithms is described in turning this section.

4.4.1 The Compute-Direct (CD) algorithm

The first DVS algorithm considered in this study is the most straightforward, and will be referred to here as "Compute Direct" (CD).

The CD algorithm employed to calculate the DVS settings is run from the Dispatcher function (see (Pont, 2001)). At every clock tick, the characteristics of the task due for release (if any) are examined, to find the deadline. The CD algorithm will then aim to determine if the task can meet its deadline if executed at the lowest speed setting: if it can, the speed is set to this value and the task is released. If the deadline cannot be met at this speed, the next increment is tried. This process is repeated, as necessary, until the maximum speed setting is reached. Note that it is assumed that all tasks will meet their deadlines at the maximum speed setting.

Please note that the slack time is not constant in practical systems. It is dependent on the scheduling overhead which will vary to each speed setting. Thus, speed finding cannot be directly calculated.

A pseudo-code representation of this algorithm is shown in Listing 4.4

```
begin DISPATCH_TASKS:

    while Tick_Count > 0
       Number_of_task_run = 0 ;
       for loop all tasks in tasks array
           if -- Delay == 0 then
              Number_of_task_run ++ ;
           end if
           search Deadline;
       end for

       switch  (Number_of_task_run)
          case  0 :  CPU_Speed = MIN_SPEED ;
                     scale frequency and voltage ;
                     break
          case >0 :  CPU_Speed = MIN_SPEED ;
                     while !((Task_duration(CPU_Speed) < Deadline
                                         and CPU_Speed < MAX_SPEED)
                           CPU_Speed ++ ;
                     end while
                     break
       end switch

       for loop all tasks in tasks array
           if -- Delay == 0 then
              scale frequency and voltage ;
              release Task ;
              if it is a periodic task then
                 reload Delay = Period ;
              else
                 delete Task from array ;
              end if
            end if
          end for
          Tick_Count -- ;
    end while
    sleep ;

end DISPATCH_TASKS
```

Listing 4.4: Compute-Direct algorithm

4.4.2 The Lookup Table (LT) algorithm

The CD algorithm is conceptually simple but computationally expensive. To reduce the CPU load, a lookup table can be employed.

Using such a table, the task scheduling and frequency / voltage scaling execution time will be measured, calculated and stored. Note that the required calculations depend not only on the task to be executed, but also the context of this execution (for example, whether Task A is immediately followed by the execution of Task B).

To simplify the discussions here and later in the chapter, the task durations will be represented in "utilisation units". Each task slot is considered as 1U (Figure 4.6).

If a task employs the whole time slot at full speed (60 MHz in the system), it takes 100% of the available utilisation: such a task cannot run at lower speed because its utilisation will exceed 1U. Similarly, if a task takes 100% of the utilisation at 30 MHz, it has a load of 0.5U at 60 MHz.

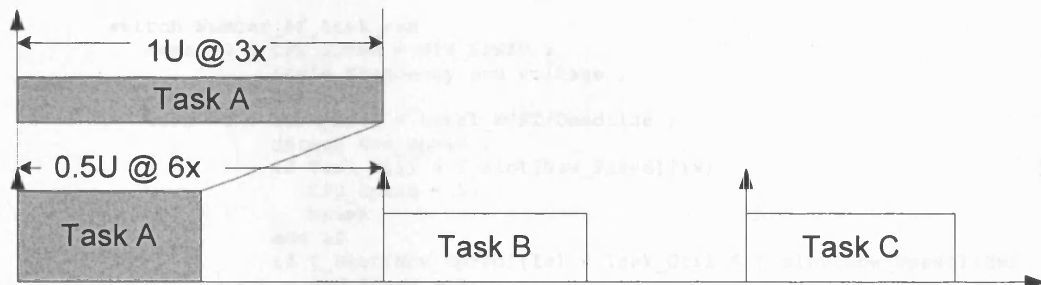

Figure 4.6: Considering intra-task utilisation

Figure 4.7 shows the maximum utilisation at various speeds, based on the maximum speed at 60 MHz. The available task slot at each speed can be found by deducting the scheduler overhead from these values. The available task slot values will then be stored in the lookup table. Listing 4.5 illustrates the LT algorithm.

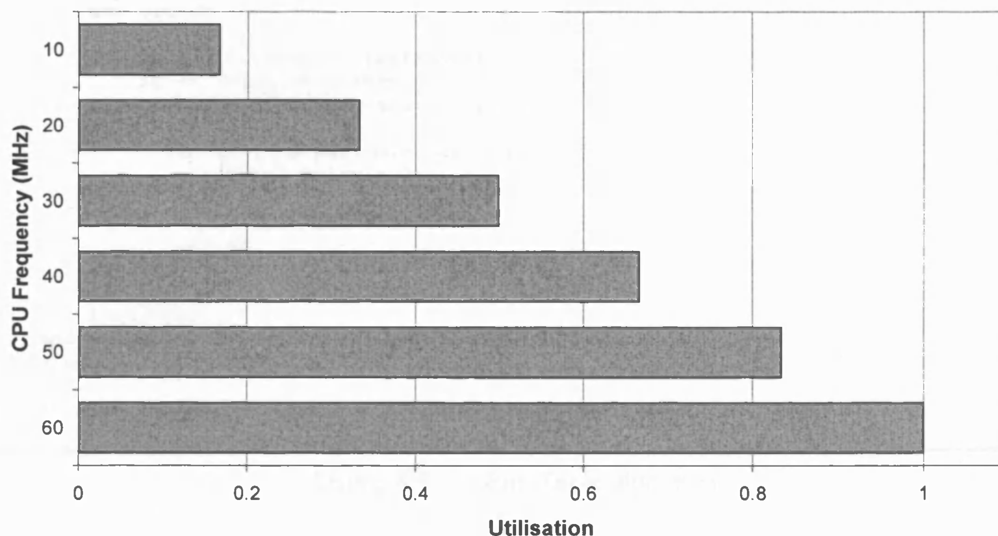

Figure 4.7: The boundary of utilisation based on 60 MHz

```
begin DISPATCH_TASKS:

     while Tick_Count > 0
        Number_of_task_run = 0 ;
        for loop all tasks in tasks array
            if -- Delay == 0 then
               Number_of_task_run ++ ;
            end if
            search Deadline;
         end for

         switch Number_of_task_run
            case  0 : CPU_Speed = MIN_SPEED ;
                      scale frequency and voltage ;
                      break
            case >0 : Task_Util = total_WCET/Deadline ;
                      detect Now_Speed ;
                      if Task_Util ≤ T_Slot[Now_Speed][1x]
                         CPU_Speed = 1x ;
                         break
                      end if
                      if T_Slot[Now_Speed][1x] < Task_Util ≤ T_Slot[Now_Speed][2x]
                         CPU_Speed = 2x ;
                         break
                      end if
                      if T_Slot[Now_Speed][2x] < Task_Util ≤ T_Slot[Now_Speed][3x]
                         CPU_Speed = 3x ;
                         break
                      end if
                      if T_Slot[Now_Speed][3x] < Task_Util ≤ T_Slot[Now_Speed][4x]
                         CPU_Speed = 4x ;
                         break
                      end if
                      if T_Slot[Now_Speed][4x] < Task_Util ≤ T_Slot[Now_Speed][5x]
                         CPU_Speed = 5x ;
                         break
                      end if
                      if Task_Util > T_Slot[Now_Speed][6x]
                         CPU_Speed = 6x ;
                         break
                      end if
                      break
         end switch

         for loop all tasks in tasks array
             if -- Delay == 0 then
                scale frequency and voltage ;
                release Task ;
                if it is a periodic task then
                   reload Delay = Period ;
                else
                   delete Task from array ;
                end if
             end if
         end for
         Tick_Count -- ;
     end while
     sleep ;

end DISPATCH_TASKS
```

Listing 4.5: Lookup Table algorithm

4.4.3 The Circular Array (CA) algorithm

Both the CD and LT algorithms execute every time a task is dispatched. In the Circular Array algorithm, the scheduling overhead is reduced by moving the core of the speed-finding procedure to a system initialisation function, which is run only once.

For example, suppose there are three periodic tasks, A(0,5), B(2, 10), C(5,15). Here the tasks are represented as *name(delay, period)*: task A, B and C start at 0, 2 and 10 ms (respectively) and repeat every 5, 10 and 15 ms (respectively). The cycle period is determined by finding the greatest common factor from the task periods. In the example, the cycle value is 30.

If these periodic tasks are assigned a running speed, they will be run at the same speed every cycle. Consequently, the speed of the task in each tick can be determined once and applied in every cycle. To implement this algorithm, a circular array, which has a size equal to the number of task slots in one cycle, can be used to store the required CPU speed. When the system is initialised, the speed- finding procedure will be run for a full cycle (without dispatching tasks) in order to calculate and store the required speed values. A scheme to synchronise the circular array pointer with task slot will then be run before the scheduler starts. The Circular Array algorithm is shown in Listing 4.6

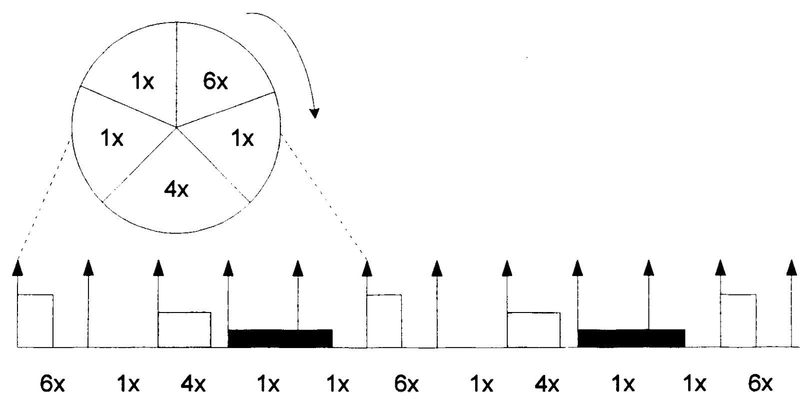

Figure 4.8: A schematic representation of the Circular Array algorithm

```
begin DISPATCH_TASKS:

    while Tick_Count > 0
        if Circular_Array_Pointer > MAJOR_CYCLE then
            reset Circular_Array_Pointer ;
            for loop all tasks in tasks array
                if -- Delay == 0 then
                    load Task_Speed from circular array ;
                    scale frequency and voltage ;
                    release Task ;
                    if  it is a periodic task then
                        reload Delay = Period ;
                    else
                        delete Task from array ;
                    end if
                end if
            end for
            Tick_Count -- ;
    end while
    sleep ;

end DISPATCH_TASKS
```

Listing 4.6: Circular Array algorithm

4.4.4 The Circular Skip (CS) algorithm

To minimise power consumption, it is possible to run all tasks at the minimum speed and never change the speed setting. This is rarely practical, but – with appropriate task scheduling – it can often reduce the number of speed changes required, thereby reducing the power consumption.

This idea is exploited in what it is called here the "Circular Skip" (CS) algorithm (see Listing 4.7). CS is applied after speeds have been calculated and stored in the circular array (using the CA algorithm). Using CS, it examines the array and look for instances where no speed changes are required between the execution of two tasks which are scheduled to run consecutively: when such a situation is found, the superfluous speed-change step is removed. This change itself reduces power consumption.

In addition, when the speed change is removed, this frees up CPU time and – as a consequence – it may be possible to run some tasks at a lower speed than was possible using the CA algorithm. In this way CS adds to the reduction in power consumption.

```
begin DISPATCH_TASKS:

    static Previous_Speed = 0 ;
    while Tick_Count > 0
        if Circular_Array_Pointer > MAJOR_CYCLE then
           reset Circular_Array_Pointer ;
           for loop all tasks in tasks array
                if -- Delay == 0 then
                    load Task_Speed from circular array ;
                    if Previous_Speed != Task_Speed then
                       scale frequency and voltage ;
                       save the Current_Speed to Previous_Speed ;
                    end if
                    release Task ;
                    if it is a periodic task then
                       reload Delay = Period ;
                    else
                       delete Task from array ;
                    end if
                end if
           end for
           Tick_Count -- ;
    end while
    sleep ;

end DISPATCH_TASKS
```

Listing 4.7: Circular Skip algorithm

4.5 Assessing and comparing the DVS algorithms

The experiments carried out to assess and compare the DVS algorithms described in Section 4.4 are described here.

4.5.1 Overhead analysis

Figure 4.9 illustrates the scheduling overhead for each of the DVS algorithms. To draw this figure, the voltage and frequency scaling time of all speed-setting and DVS algorithms were measured at 60MHz (based on 5 tasks execution) and then the scheduling overhead at all speeds was predicted. The resulting figures permit the calculation of the available execution time.

From Figure 4.9, the CD overhead is around 257.8 μs (162 μs for voltage/frequency scaling, and 97 μs for speed-finding and task scheduling) when run with the task set at 60 MHz. This figure increases when lower speeds are applied: it is 769.8 μs at 10 MHz. It is obvious that the range between minimum and maximum values is also high: this is due to the speed-finding calculation, which has a duration dependent on the number of calculations required. Compared to voltage/frequency scaling, the

times taken for this process at each speeds are not significantly different (approximately 160 to 190 μs).

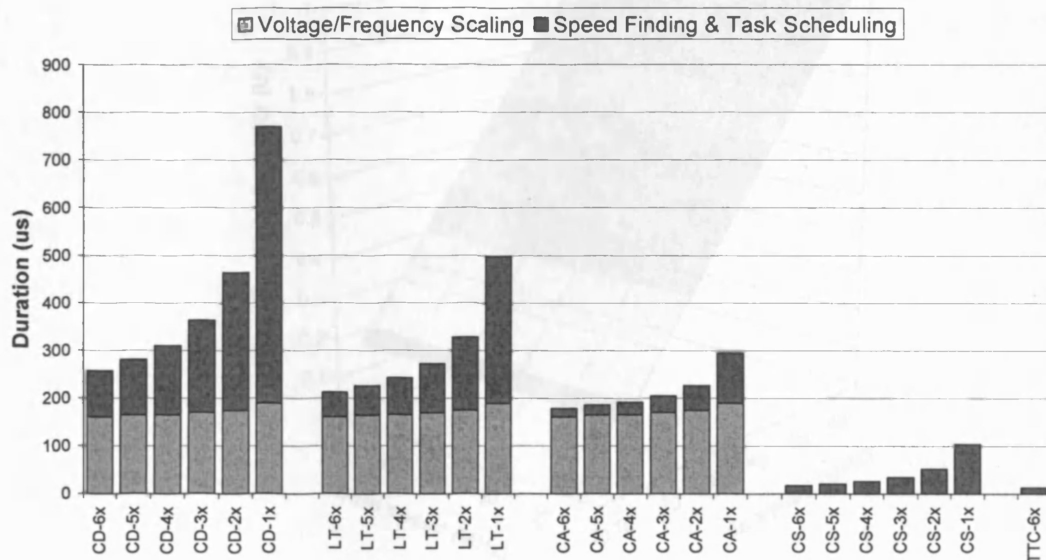

Figure 4.9: Scheduling overhead of DVS algorithms estimation

The available CPU time for tasks can be increased by reducing the time taken for speed-setting calculations, as the results from the LT and the CA algorithms show. The LT algorithm reduces calculation-time by using lookup table while the CA algorithm removes speed-finding part from runtime overhead. In case of the CA algorithm, it can reduce the overhead drastically. However, it is lack of capability to perform online speed calculations – all speeds must be pre-defined before the schedule starts. In the case of the CS algorithm, if voltage and frequency scaling can be avoided, the overheads are close to those of the original TTC scheduling algorithm.

From these scheduling overhead values, the available task utilisations of the various DVS algorithms can be calculated (Figure 4.11 - Figure 4.14)

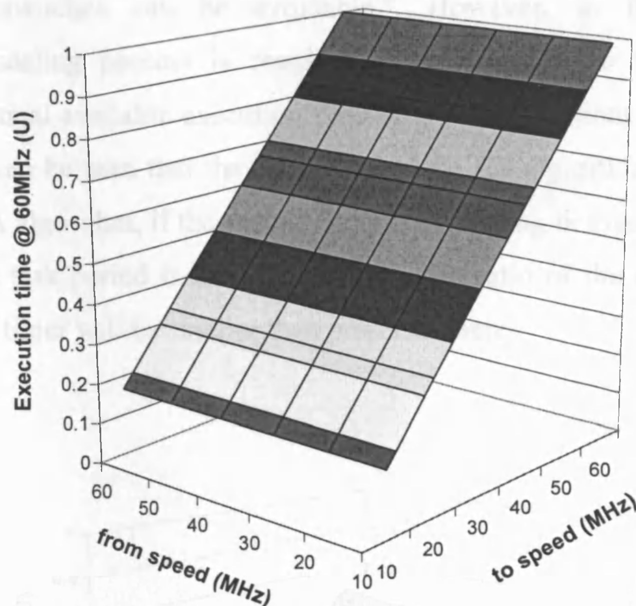

Figure 4.10: Maximum execution time of task considered at maximum frequency 60MHz

Figure 4.10 shows the ideal maximum utilisations of all frequencies running when they are considered comparatively at the maximum frequency 60 MHz. The maximum utilisation is equal 1.0 U when running at 60 MHz but it is about 0.16 U comparing with that of 60 MHz if the system runs at 10 MHz.

In, Figure 4.11 - Figure 4.15, the graphs are intended to show the practical utilisation of task which are available at each frequency switch, after deducting overheads. As in the experimental setup in Section 4.1.3, there are 36 cases of possible frequency switch. These graphs present the available execution times which are calculated under 1 ms task period scenario. For example, in Figure 4.11, the available utilisation of the CD algorithm is available for executing a task only about 0.12 U – 0.04 U when frequency switching from 60 - 10 MHz speed to 10MHz speed respectively. Available execution time will increase if tasks are executed at higher frequency.

The available execution times are improved by the LT algorithm (Figure 4.12) and more increased with the CA algorithm (Figure 4.13). In the CS algorithm, it reduces

unnecessary voltage/frequency scaling process for minimising overhead. Figure 4.14 shows that the available execution time of the CS algorithm will have if the voltage/frequency switches can be avoidable[10]. However, in this study, the voltage/frequency scaling process is required if the consecutive frequencies are different. The practical available execution time of the CS algorithm is illustrated as in Figure 4.15. It can be seen that the overheads of the CS algorithm are improved, compared to the CA algorithm, if the voltage/frequency scaling is avoided. Note that, in the case that the task period is longer than 1 ms, the ratio of the overhead to the available execution timer will be smaller than presented here.

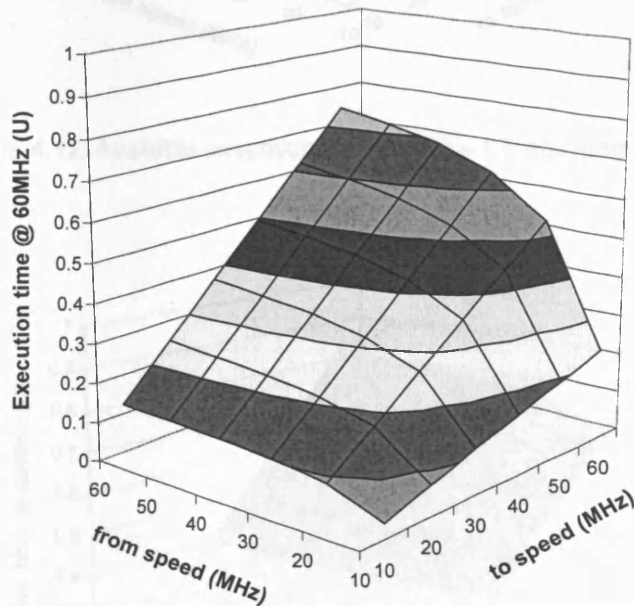

Figure 4.11: Available execution time of task – CD algorithm

[10] In practice, the overhead can be reduced by performing frequency scaling only if these running frequencies are in their voltage range. Generally, voltage scaling process mainly dominates the overhead when comparing to frequency scaling process.

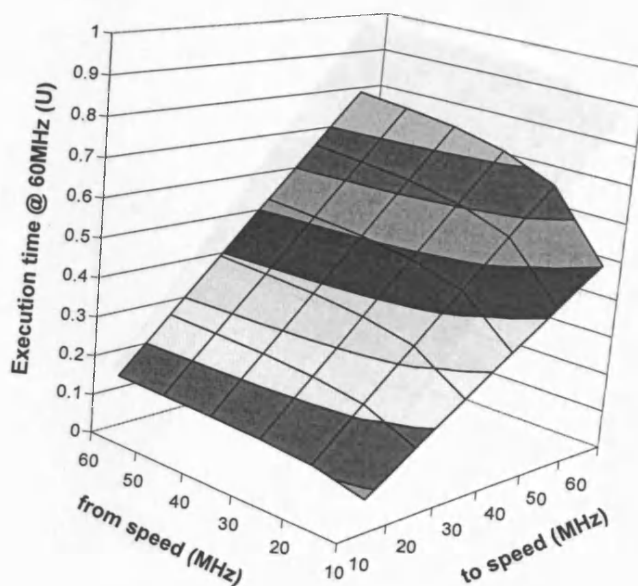

Figure 4.12: Available execution time of task – LT algorithm

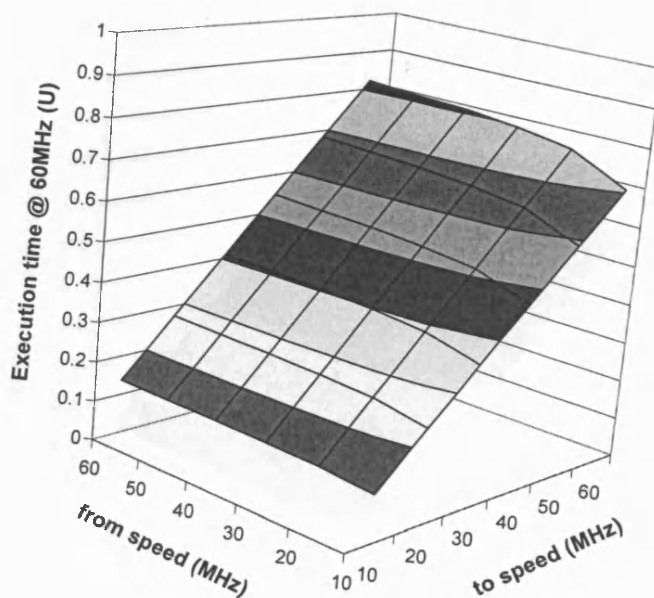

Figure 4.13: Available execution time of task – CA algorithm

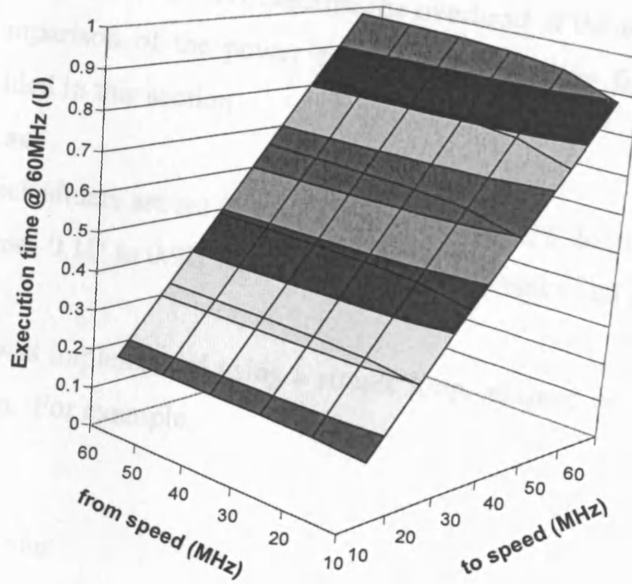

Figure 4.14: Available execution time of task – CS algorithm – excluded voltage/frequency scaling

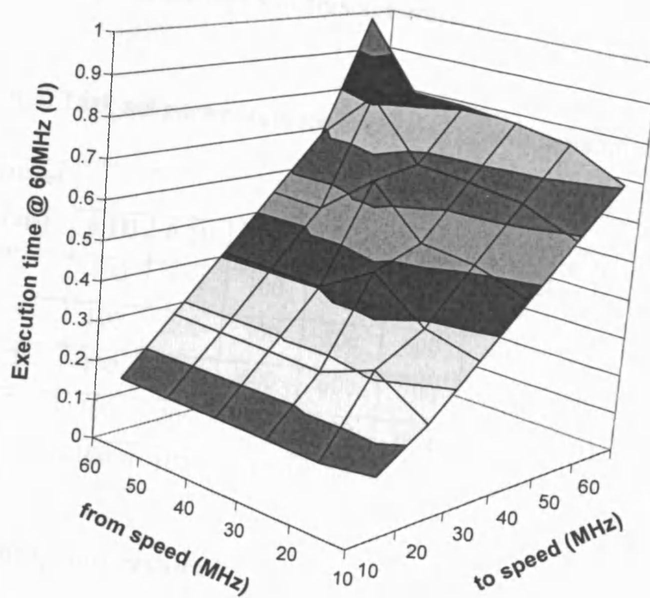

Figure 4.15: Available execution time of task – CS algorithm – practical

4.5.2 More general performance comparisons

The results presented in Section 4.5.1 describe the overhead of the various algorithms. A more direct comparison of the power consumption resulting from these various techniques is provided in this section.

4.5.2.1 The task set

For this study, all schedulers are set to have a 1 ms clock tick. 5 dummy tasks that had a total utilisation from 0.1U to 0.9U were then created. All tasks had a 5 ms period.

Each dummy task was implemented using a simple loop, adapted as required to give the required duration. For example:

```
Dummy Task:
        for loop
           {
           a = a + 1;
           }
```

Note that, in time-triggered co-operative scheduling systems, all task durations should be less than the scheduler tick interval. In this case, the duration of all tasks was less than 1 ms. The parameters of the tasks in this set are shown in Table 4.1.

Table 4.1: Task set parameters for assessing power consumption

Task	Period	Offset	Execution time (μs)								
	(ms)	(ms)	0.1U	0.2U	0.3U	0.4U	0.5U	0.6U	0.7U	0.8U	0.9U
A	5	0	100	100	100	200	300	500	700	800	900
B	5	1	100	200	300	500	600	600	700	800	900
C	5	2	100	400	600	600	700	700	700	800	900
D	5	3	100	200	300	500	600	600	700	800	900
E	5	4	100	100	200	200	300	600	700	800	900

4.5.2.2 Power consumption results

To perform this empirical comparison, task sets (with task durations from 0.1 U to 0.9 U) were set up and executed using the original TTC, CD, LT, CA and CS algorithms. The average power consumption of CPU core in each case was measured and is shown in Figure 4.16.

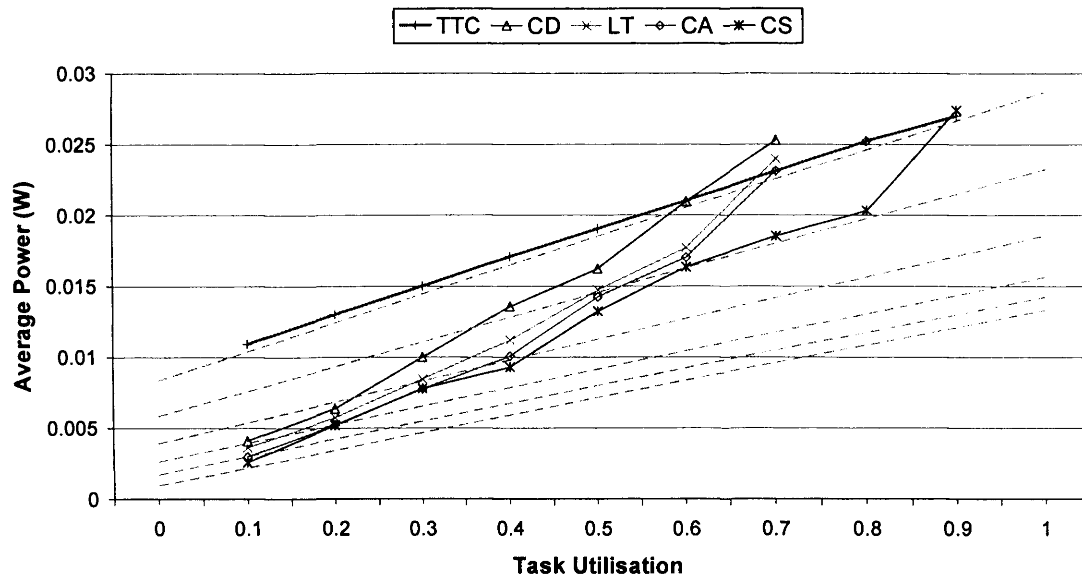

Figure 4.16: Power consumption of CPU core on different scheduling algorithms

Because it runs at a fixed 60 MHz frequency (when executing tasks) and in "idle" mode at other times, the TTC power consumption shown in this figure is almost linearly related to the workload. Overall, the power consumption of the TTC scheduling overhead is low: approximately 8.8 mW at 60 MHz.

At low levels of task utilisation all of the DVS algorithms succeed in reducing the system power consumption, when compared to the original TTC algorithm.

From the results, CD is the most power-hungry of the DVS algorithms considered here, while the LT, CA and CS are the second, the third and the fourth, respectively. At every workload, the CD consumes more power than the rest of variable-speed approaches. Its power is close to the TTC at 0.6 U and greater than TTC at 0.7 U.

CS is the most power-efficient of these DVS algorithms. It has the lowest power consumption at all levels of utilisation, except at 0.3 U. In this case, the CS slightly takes power more than the CA because the task set that was run had large gap between their durations (100, 300, 600, 300 and 100 μs). In this particular configuration, the CS could not optimise the speed. With this exception, CS saves power, until it reaches a utilisation level of 0.9 U: here, CS, which runs all tasks at full speed, consumes more power than the TTC (because of the algorithm overhead).

Note that straight-dot lines represent power consumption of loads at each operating frequency. The dummy loads were set up by running in the loop, without any scheduler, at different duration and then were measured. The results were averaged and plotted as in Figure 4.16.

4.6 The impact of DVS on system timing

In order to assess the impact of DVS on system's timing, a set of representative empirical studies was conducted. Among all those DVS algorithms, the algorithm selected for assessing in the studies was CS (Circular Skip) algorithm which was found (in the experiment presented here) to be most effective, in term of power-saving and to have the lowest overhead. From this point, the CS algorithm will be referred to as the "TTC-DVS" algorithm. The studies are described in this section.

4.6.1 General accessing the impact of DVS on jitter

To explore the impact of variable speed on tick and task, two tasks (Task A and Task B) were set up and run with TTC-DVS and a "standard" TTC scheduler (Pont, 2001). Task A was run every 2 ms while Task B was run at the same period but with 1 ms offset. In every case, the tick interval was set at 1 ms. To study the influence of the DVS, Task A and Task B were run with a random speed (10 to 60 MHz) while TTC run at a fixed speed of 60 MHz. The actual tick intervals were measured to assess tick jitter while the measuring interval between start times of Task A was for evaluating task jitter. Please note that the tick measurement in this study was probed at the beginning of scheduler ISR. The measurements were made using a National Instrumentation data acquisition card "NI PCI-6035E" (NI, 2000), used in conjunction with LabVIEW 6.1 (NI, 2001). The results are shown in Figure 4.17 - Figure 4.21.

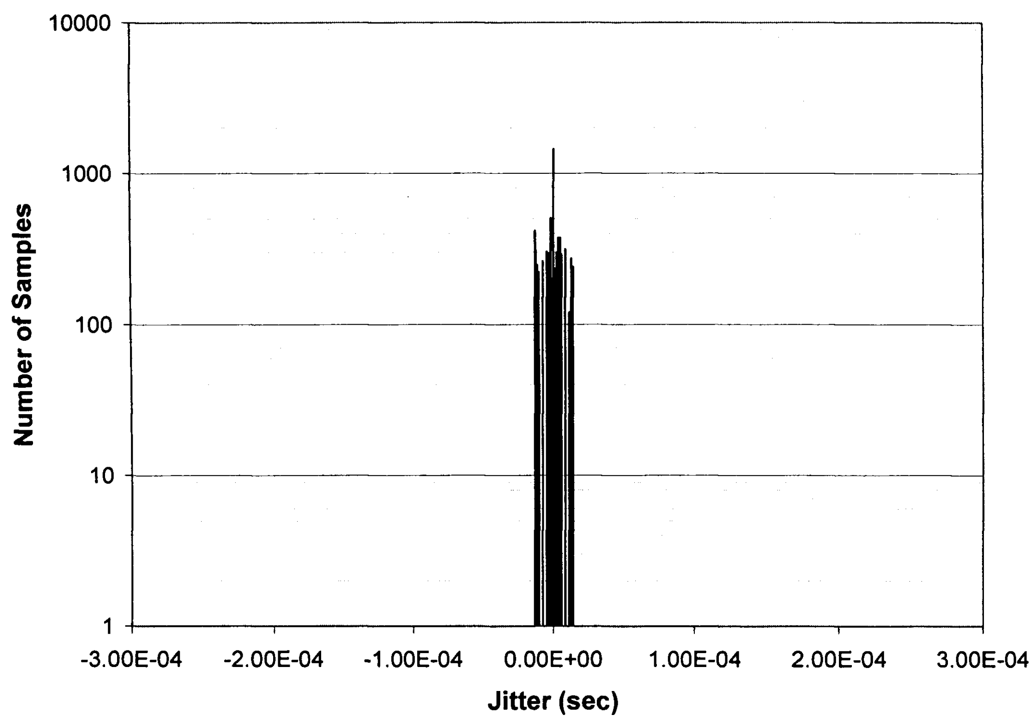

Figure 4.17: Histogram of tick jitter in TTC-DVS run with random frequency

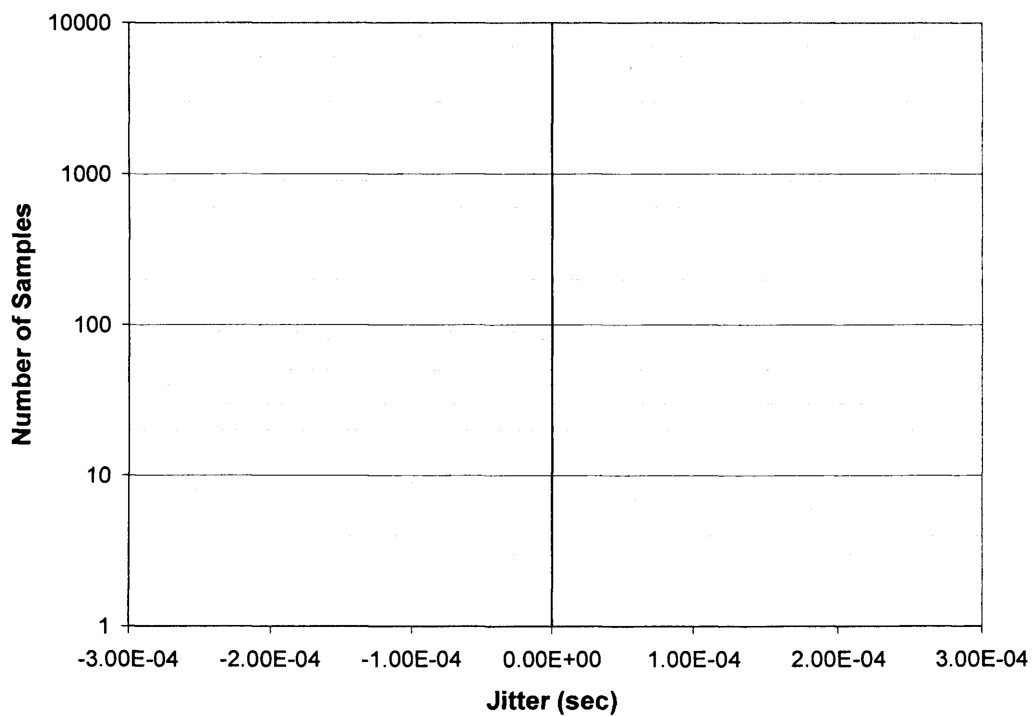

Figure 4.18: Histogram of tick jitter in TTC

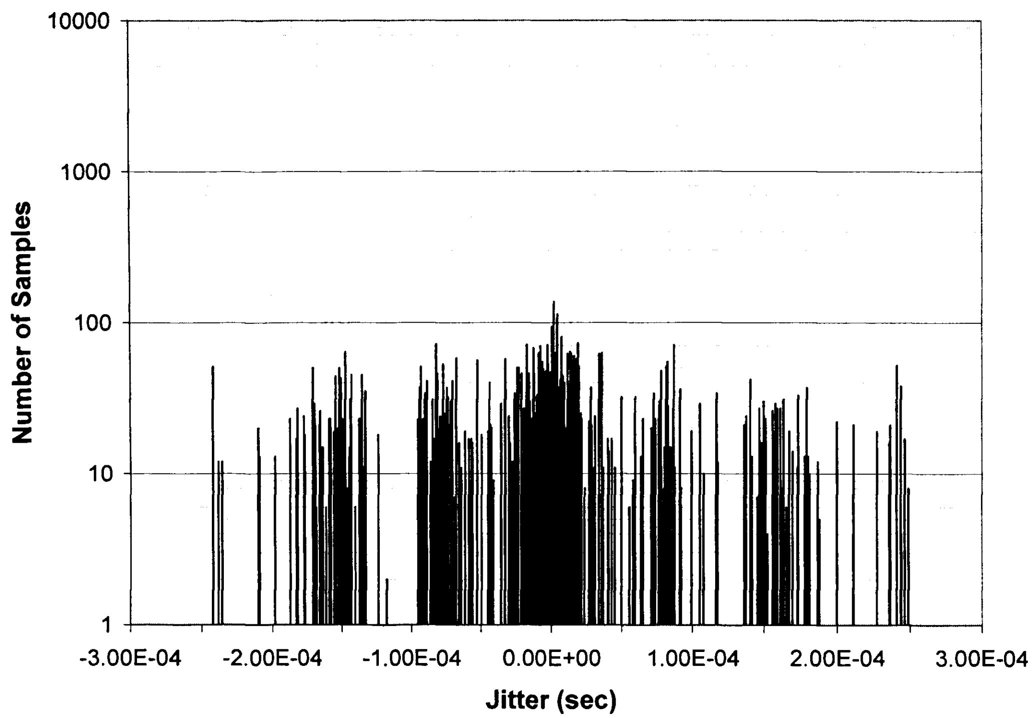

Figure 4.19: Histogram of task jitter in TTC-DVS run with random frequency

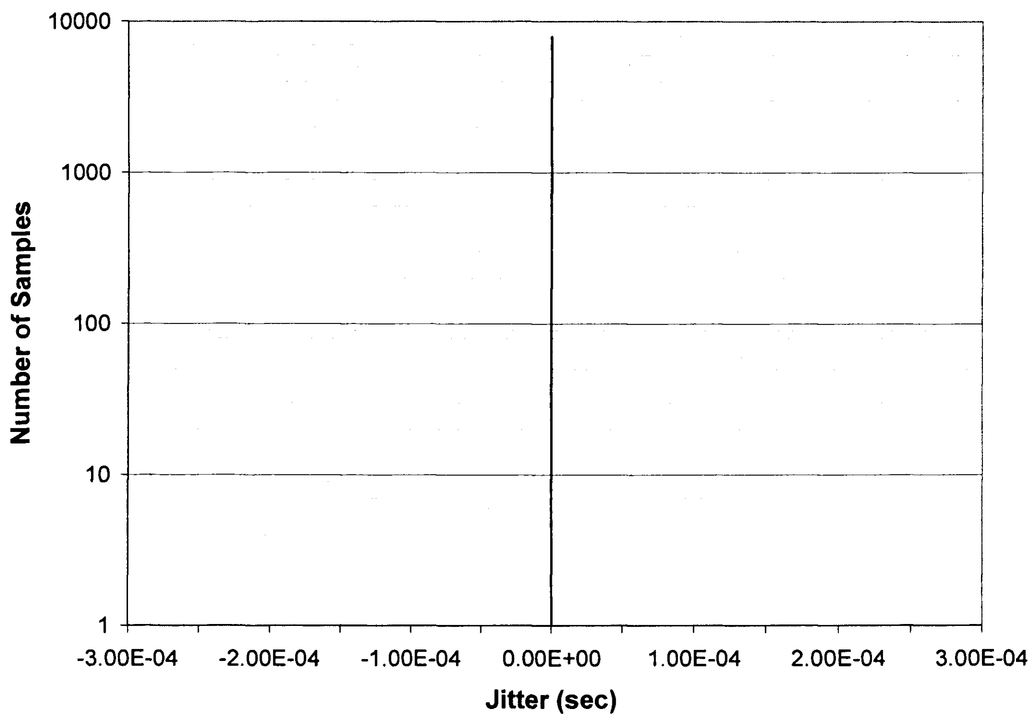

Figure 4.20: Histogram of task jitter in TTC

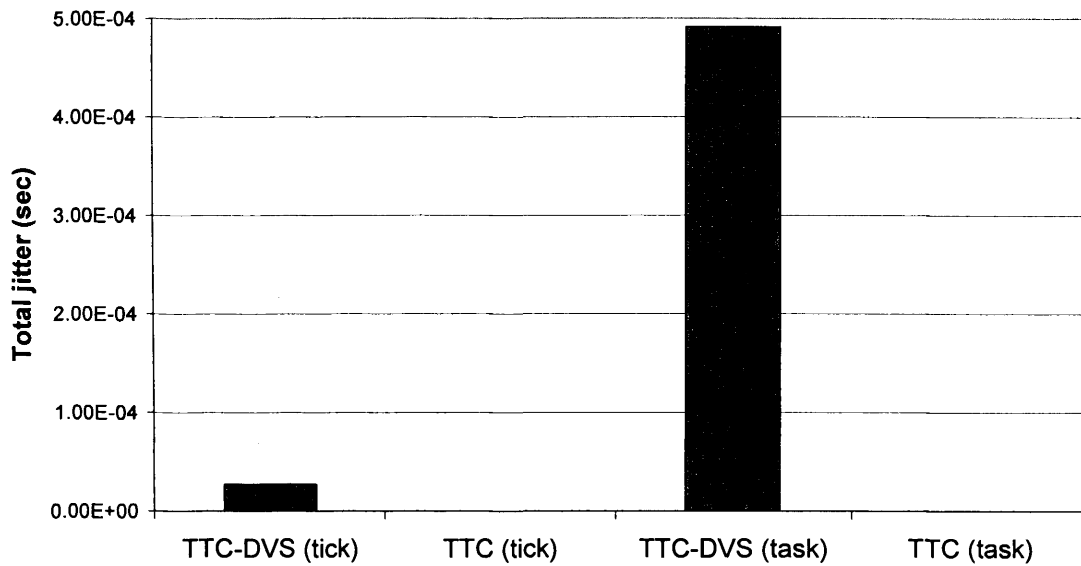

Figure 4.21: Total jitter of task in TTC-DVS and TTC systems

Table 4.2: Tick jitter and task jitter in TTC-DVS and TTC systems

	Tick jitter (μs)			Task jitter (μs)		
	Max	Min	Total	Max	Min	Total
TTC	0.00	0.00	0.00	0.10	0.00	0.10
TTC-DVS	13.90	-13.20	27.10	248.80	-242.40	491.20

Figure 4.17 and Figure 4.18 illustrate the occurrence of tick jitter taken from 10,000 consecutive samples from TTC-DVS and TTC algorithms respectively. No measurable tick jitter was obtained for the TTC algorithm. For the TTC-DVS algorithm, the measured jitter is illustrated in Figure 4.17. The detailed results from this comparison are provided in Table 4.2.

Figure 4.19 and Figure 4.20 illustrate histogram of task jitter taken from 10,000 consecutive samples from TTC-DVS and TTC algorithms. They show a significant impact of TTC-DVS on task jitter, more than 10% compared with its period. Figure 4.21 and Table 4.2 compare the impact of TTC-DVS on both tick and task with that of TTC.

The experimental results show that the TTC algorithm, which normally has low-jitter characteristic, suffers from significant increases in jitter when DVS is employed in

this naive manner. These findings are consistent with the jitter results reported by Mochocki *et al.* (2005) in which it was reported that all tasks scheduled by the RM algorithm suffered from jitter greater than 10%, even at a utilisation of 0.3 (see Section 2.4.1 for further details).

4.7 The knock-on impact of frequency scaling

Because the system operates with a variable clock frequency, jitter can occur in TTC-DVS systems. The resulting jitter can be divided into 3 categories:

- Tick jitter
- Sampling jitter
- Release jitter

Each of these categories is considered as the following.

4.7.1 Tick jitter, drift

In the TTC designs considered in this thesis, a clock tick is generated by a hardware timer that is linked to an interrupt service routine (Pont, 2001). This mechanism relies on the presence of a timer that runs constantly and accurately: in a DVS system, frequency switching is likely to disrupt such timing. For example, in many processors, it takes a variable amount of time for the phase-locked loop (PLL) to lock after the clock frequency is changed, and the timer operation is disrupted when the value of the prescaler register is adapted to match the new frequency (see Figure 4.22).

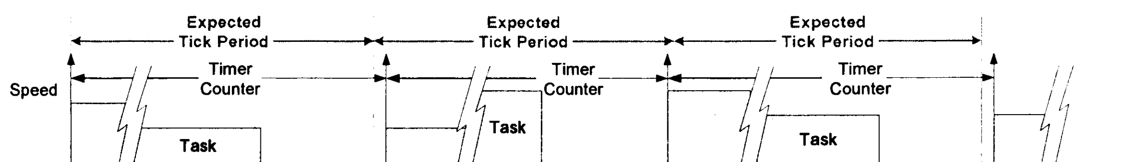

Figure 4.22: Tick drift in DVS systems. (Phatrapornnant and Pont 2006, IEEE Transactions on Computer, Vol.55(2), © 2006 IEEE)

4.7.2 Sampling jitter, Task duration variation

A direct consequence of DVS is that the task duration is increased if the processor speed is lowered and decreased if the processor speed is raised. This can have major side effects. For example, Figure 4.23 illustrates the impact of frequency changes on a system involving data sampling within a task.

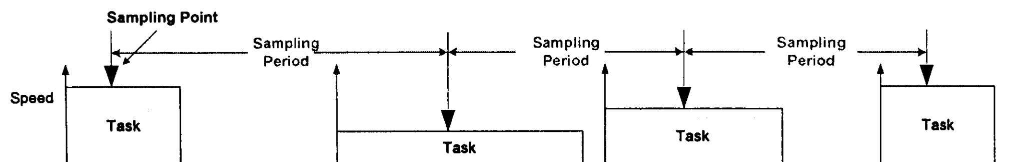

Figure 4.23: Sampling jitter caused by frequency scaling. (Phatrapornnant and Pont 2006, IEEE Transactions on Computer, Vol.55(2), © 2006 IEEE)

As previously noted, Cottet and David (Cottet and David, 1999) show that – during data acquisition tasks – jitter rates of 10% or more can introduce significant errors.

4.7.3 Release jitter, Scheduling overhead variation

The overhead of a conventional (that is, non-DVS) scheduler arises mainly from context switching. In systems employing DVS, it also needs to consider frequency / voltage scaling procedures and – possibly – speed-finding procedures. These procedures make the scheduling overhead of DVS systems comparatively large. Often of greater concern is the fact that these procedures may have a highly variable duration: for example, Figure 4.24 illustrates how a TTC-DVS system can suffer release jitter as a result of variations in the scheduler overhead.

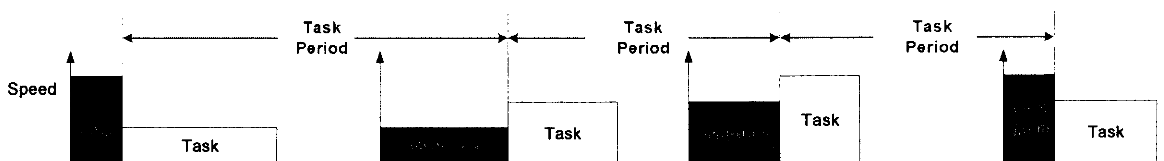

Figure 4.24: Release jitter caused by variation of scheduling overhead. (Phatrapornnant and Pont 2006, IEEE Transactions on Computer, Vol.55(2), © 2006 IEEE)

4.8 Conclusions

In this chapter, practical approaches to applying DVS in a TTC scheduler have been demonstrated. Both hardware and software have been set up to support dynamically varying CPU core's supply voltage. A number of DVS algorithms (CD, LT, CA and CS), which are applied in a TTC scheduling design have been explored. These DVS algorithms have been analysed and evaluated in terms of system overhead and power-saving performance comparing among of them and the original TTC design. From the experimental results, it has been found that the CS (referred as TTC-DVS) algorithm performs the best performance among those algorithms.

The impact of realistic DVS implementation, which involves the variation of DVS overhead and voltage/frequency scaling process, in TTC applications, has been illustrated. In particular, it has been demonstrated that use of DVS can cause a TTC scheduling system to suffer large jitter. Techniques for minimising jitter in DVS applications will be explored in Chapter 5.

Chapter 5

Design and evaluation of a reduced-jitter TTC/DVS scheduler

It was demonstrated in Chapter 4 that DVS can be very simply and effectively applied in embedded systems employing a time-triggered, co-operative (TTC) scheduler However, the results obtained showed that – while use of DVS reduced CPU energy consumption – this approach also increased levels of task jitter. The work described in this chapter considers ways of minimising the jitter levels in single-processor embedded systems employing a TTC scheduler and DVS.[11]

5.1 Minimising jitter caused by DVS

DVS has been used effectively for reducing dynamic power consumption by lowering CPU frequency and voltage. As noted previously, these frequency changes can cause the systems employing DVS suffer to jitter. As described in Section 4.7, jitter which may occur in TTC applications can be categorised as: (i) tick jitter, (ii) sampling jitter, and (iii) release jitter.

In order to develop TTC/DVS scheduler for applications which require low levels of task jitter, it is assumed that it may not be necessary to run all tasks with low jitter,

---

11 Part of this chapter has previously published in Phatrapornnant, T. and Pont, M.J. (2004a) and Phatrapornnant, T. and Pont, M.J. (2006).

depending on the application. Tasks that are required to run "jitter free" are defined as "reduced-jitter tasks" (RJTs).

The jitter reduction approaches applied to RJTs are as follows.

First, the jitter arising from variations in the scheduler overheads can be alleviated by inserting a delay (called a "jitter guardian") before the RJTs are activated. This results that the release jitters are minimised.

Second, in realistic implementations, the scheduling "tick" can suffer jitter when a timer is re-programmed to match the new frequency. By re-programming the timer, the tick jitter can be reduced.

Third, setting the operating frequency of the RJTs to be constant can avoid jitter caused by task execution-time variation.

The studies in this chapter illustrate ways to implement the jitter reduction techniques in TTC/DVS scheduler – particularly concerned with the impact of DVS on tick jitter, and with the knock-on effects of DVS-induced variations in scheduler overhead and task duration.

5.2 The TTC-jDVS algorithm

The development of the algorithm in this chapter is aimed at limited-resource embedded applications which require low-jitter and low-energy, e.g. signal sampling applications.

Various techniques for implementing DVS in TTC systems were proposed and explored in Chapter 4. Of these algorithms, a version called "Circular Skip" – which will be referred to here as "TTC-DVS" – was found to be the most effective and lowest overhead. TTC-DVS will form the basis of a low-jitter TTC-DVS algorithm described in this chapter.

The new algorithm – referred to here as TTC-jDVS – employs a three-step technique for jitter reduction. The three steps are described in this section.

5.2.1 Adding tick compensation

As noted in Section 4.7.1, the hardware timer in typical processors can give rise to tick drift when DVS is employed. A compensation process is needed to minimise the tick jitter which results from this.

As part of the TTC-jDVS algorithm, a tick-compensation process illustrated schematically in Figure 5.1 is proposed.

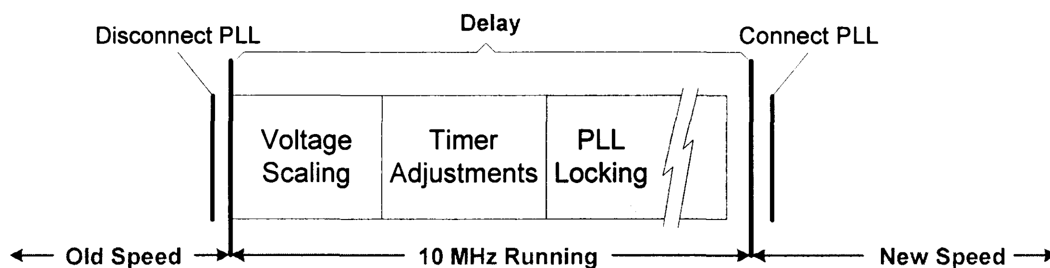

Figure 5.1: Voltage and frequency scaling steps. (Phatrapornnant and Pont 2006, IEEE Transactions on Computer, Vol.55(2), © 2006 IEEE)

The compensation is carried out within a "sandwich delay", which has a duration set to match the (combined) worst-case execution time of the three stages (voltage scaling; timer adjustments; PLL locking). Note that the PLL is locked after the voltage scaling and timer adjustments are carried out: these calculations are therefore carried out at a known (base) frequency.

The timer-adjustment process loads new timer values whenever the frequency is changed. Each frequency-switch pair requires different compensation values: in the study described in this thesis, these values were obtained (largely by trial and error) and stored in a lookup table[12].

---

[12] For reference: In the empirical study described later in this study, the timer (TMR0) reload value was 10,000. In this case, the range of compensation values required was from −112 to +186 (that is, the reload values varied from 9888 to 10112).

To obtain the compensation values, CPU clock frequency was forced to switch between two frequencies at every tick. The tick was probed (using a data-acquisition card "NI PCI-6035E" (NI, 2000) in conjunction with LabVIEW 6.1 (NI, 2001)) to measure jitter. To compensate for the time disruption caused by the frequency changes, the timer / counter was adjusted by adding or subtracting a small number of units and the tick interval was then measured again. This "trial and error" process continued until the jitter was reduced below the required level. The compensation values of this frequency-switch pair were then noted. To obtain the compensation values of other frequency-switching pairs, the procedures described above was repeated. Note that these compensation values were specifically dependent on program code for the tick-compensation procedure.

5.2.2 Adding a jitter guardian

It is assumed that it may not be necessary to run all tasks with low jitter. "Reduced-jitter tasks" (RJTs) is therefore defined as those that must be scheduled to run "jitter free". In this study, an RJT flag was added in the scheduling algorithm to identify such tasks.

Having identified RJTs, the next step is to reduce the variation in scheduler overhead prior to the release of such tasks. To do this a "jitter guardian" is used, again based on a sandwich delay. This is inserted before the RJT release (see Figure 5.2). The duration of the required delay is based on the maximum scheduler overhead (including frequency and voltage scaling) for the particular set of tasks in the circular array (this is discussed further in Section 4.4). Note that – inevitably – the jitter guardian will reduce the available execution time for tasks, and will therefore reduce the power-saving performance of the DVS system.

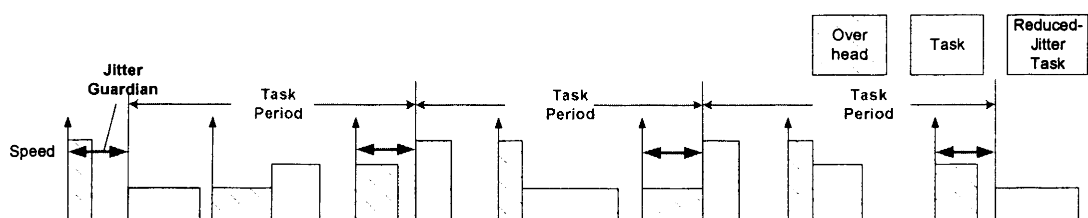

Figure 5.2: Minimising release jitter through use of a "jitter guardian". (Phatrapornnant and Pont 2006, IEEE Transactions on Computer, Vol.55(2), © 2006 IEEE)

5.2.3 Fixing the running speed of RJTs

The final stage of the TTC-jDVS algorithm involves determining the required execution speed of the RJTs. To deal with the problems caused by variations in the task duration (see Section 4.7.2), each RJT is run at the same speed every time it is released (see Listing 5.1). To determine the required running speed, the circular array is first examined to find the maximum speed at which each RJT executes: this speed is then applied every time this RJT is released. The array is then examined again, to find the lowest speed at which a task in the slot before the RJT is executed. Using the information about the speed change before the RJT is executed allows the required length of the jitter guardian (delay) to be calculated.

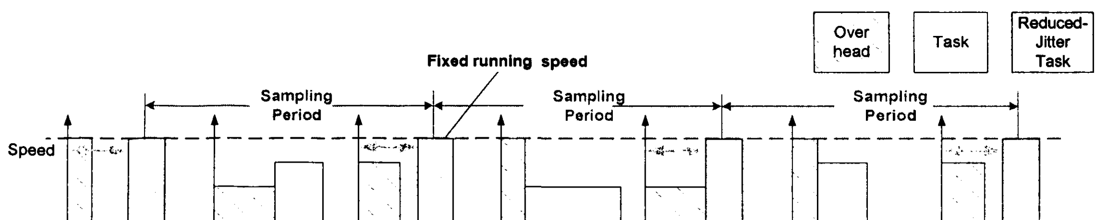

Figure 5.3: Minimising sampling jitter by fixed running speed. (Phatrapornnant and Pont 2006, IEEE Transactions on Computer, Vol.55(2), © 2006 IEEE)

```
begin SET_RJT_SPEED:

  for all slots in Circular array
      for all tasks in tasks array
          if RJT included in tasks due to run then
             find the Lowest_Speed of slot before RJT slot ;
             find the Fastest_Speed of RJT slots;
          end if
      end for
  end for

  for all slots in Circular array
      for all tasks in tasks array
          if RJT included in tasks due to run then
             assign the Fastest_Speed to current slot ;
          end if
      end for
  end for
  calculate Jitter_Guardian_Delay(Lowest_Speed) ;

end SET_RJT_SPEED
```

Listing 5.1: Setting the execution speed of "reduced-jitter" tasks (RJTs)

5.3 Implementing the TTC-jDVS scheduler

This section demonstrates the way to implement the jitter reduction algorithm in the TTC scheduler.

```
begin DISPATCH_TASKS:

  if Circular_Array_Pointer > MAJOR_CYCLE then
     reset Circular_Array_Pointer;
  end if

  while Tick_Count > 0
        for all tasks in tasks array
            if -- Delay == 0 then
               load Task_Speed from circular array ;
               if Previous_Speed != Task_Speed then
                  scale frequency and voltage (and perform tick compensation) ;
               end if
               if task is an RJT then
                  insert Jitter Guardian ;
               end if
               release Task;
               if task is periodic then
                  reload Delay = Period ;
               else
                  delete Task from array ;
               end if
            end if
        end for
        Tick_Count -- ;
  end while
  sleep ;

end DISPATCH_TASKS
```

Listing 5.2: Dispatching tasks in TTC-jDVS

Listing 5.2 shows a pseudo-code representation of the dispatcher function with jitter-compensation incorporated. In the code, the jitter guardian is inserted before the release of RJTs.

5.4 Evaluating the TTC-jDVS algorithm

To evaluate the TTC-jDVS algorithm, a set of representative empirical studies were conducted. The studies are described in this section.

5.4.1 Assessing the impact of TTC-jDVS on jitter

To explore the impact of the jitter compensation algorithm, a series of tests using the platform described in the previous section were conducted. The empirical studies reported in this chapter were conducted using the hardware platform described in Section 4.1.

5.4.1.1 Tick jitter

In order to measure tick jitter, a task using a random clock speed (from 10 to 60 MHz) using TTC-DVS and TTC-jDVS was run. This task was also run with a "standard" TTC scheduler (Pont, 2001) at a fixed speed of 60 MHz. In each case the required tick interval was set at 1 ms, and the actual tick intervals were measured. Please note that the tick measurement in this study was probed at the beginning of scheduler ISR. The measurements were made using a National Instrumentation data acquisition card "NI PCI-6035E" (NI, 2000), used in conjunction with LabVIEW 6.1 (NI, 2001).

No measurable jitter was obtained for the TTC algorithm. For the TTC-DVS algorithm, the measured jitter is illustrated in Figure 5.4: the corresponding test run using the TTC-jDVS algorithm (Figure 5.5) shows a significant reduction. Figure 5.6 and Figure 5.7 illustrate the occurrence of tick jitter taken from 10,000 consecutive samples. Table 5.1 provides detailed results from this comparison.

Table 5.1: Comparing tick jitter run by TTC-DVS, TTC-jDVS, and TTC algorithms

	Jitter (μs)		
	Max	Min	Total
TTC	0.0	0.0	0.0
TTC-DVS	13.9	-13.2	27.1
TTC-jDVS	0.2	-0.6	0.8

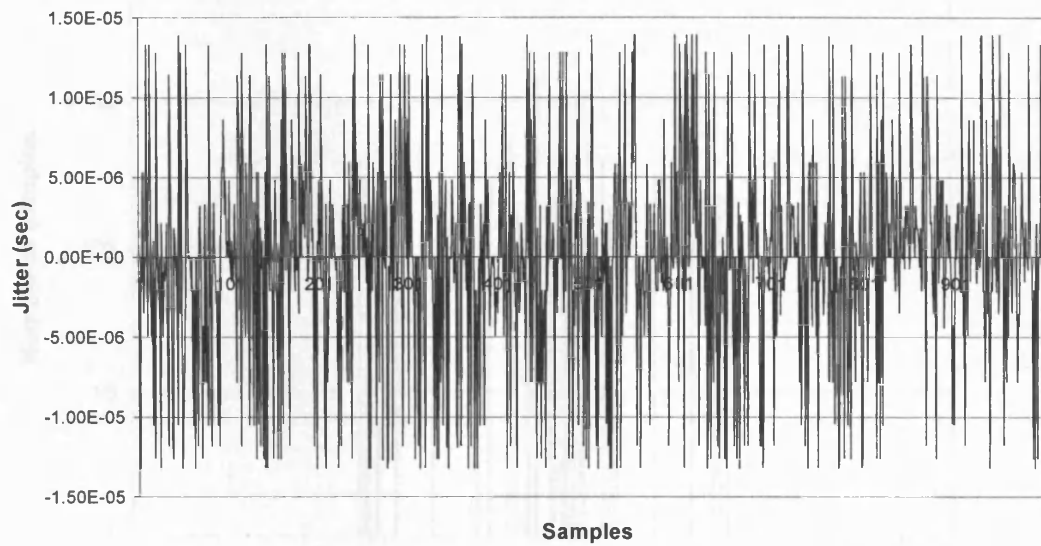

Figure 5.4: Tick jitter in TTC-DVS. (Phatrapornnant and Pont 2006, IEEE Transactions on Computer, Vol.55(2), © 2006 IEEE)

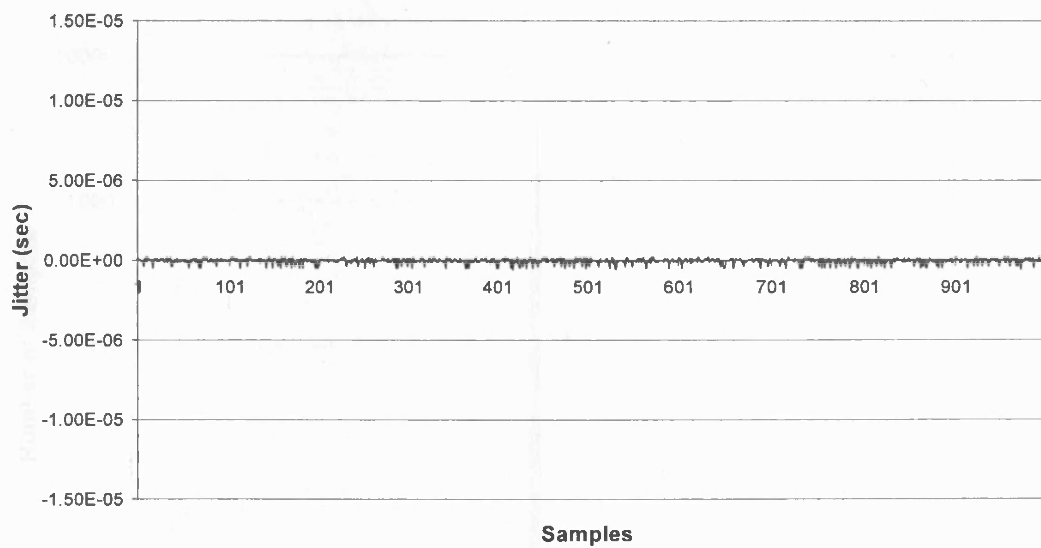

Figure 5.5: Tick jitter in TTC-jDVS. (Phatrapornnant and Pont 2006, IEEE Transactions on Computer, Vol.55(2), © 2006 IEEE)

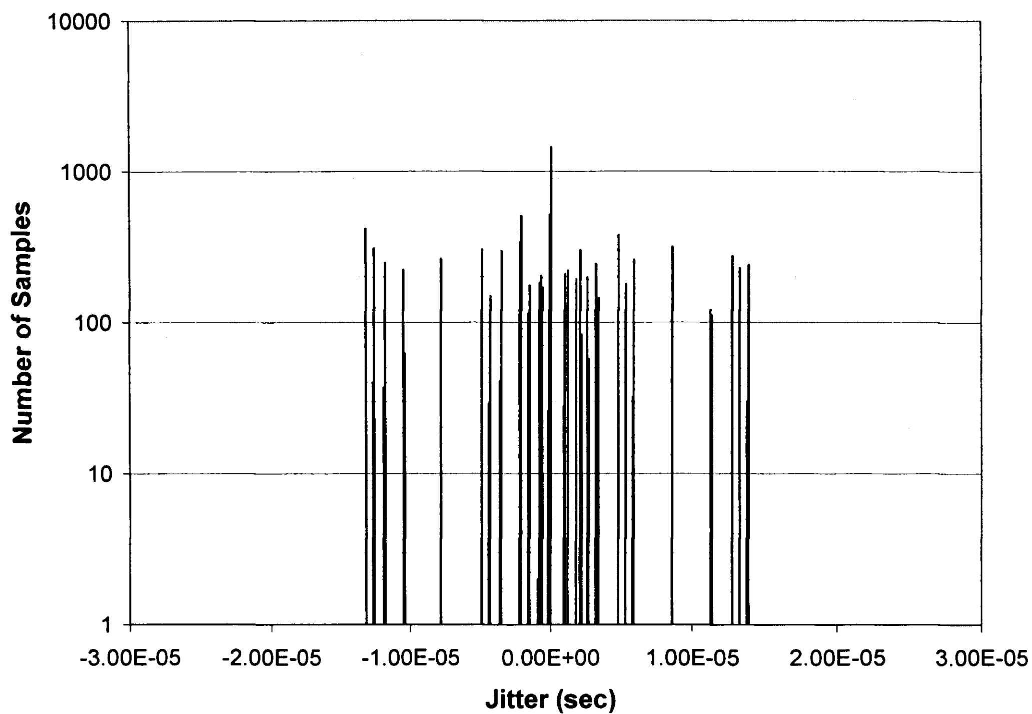

Figure 5.6: Histogram of tick jitter in TTC-DVS. (Phatrapornnant and Pont 2006, IEEE Transactions on Computer, Vol.55(2), © 2006 IEEE)

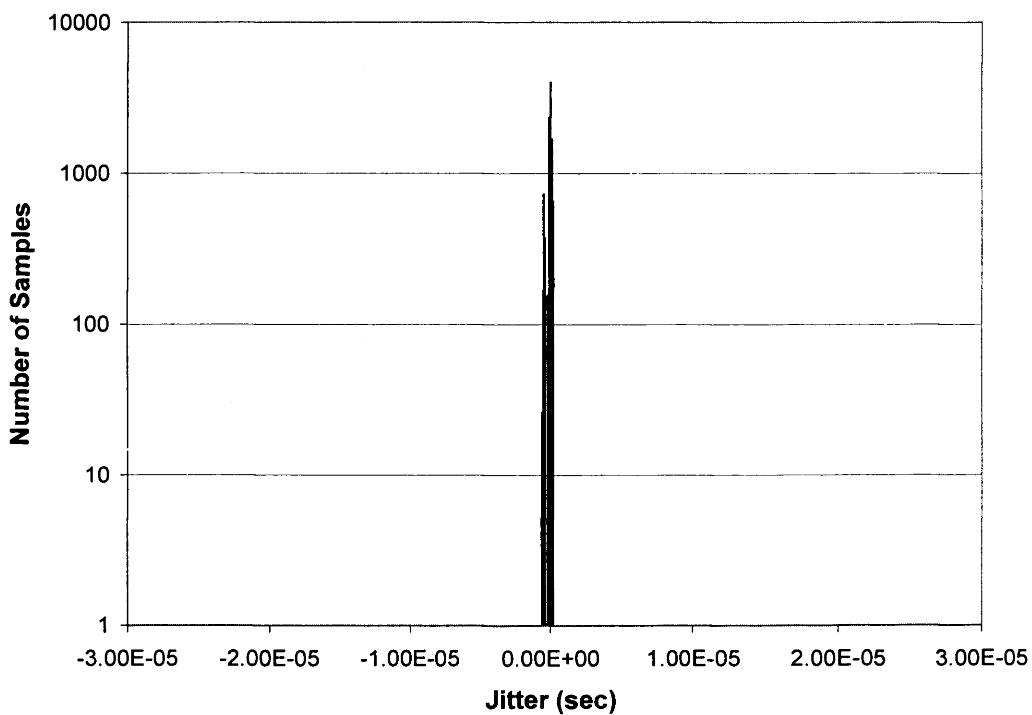

Figure 5.7: Histogram of tick jitter in TTC-jDVS. (Phatrapornnant and Pont 2006, IEEE Transactions on Computer, Vol.55(2), © 2006 IEEE)

5.4.1.2 Task jitter

To explore the impact of variable speed on task release jitter, two tasks were set up (see Table 5.2). Task A was an RJT and Task B was a normal task. Task A was run every 2 ms while Task B was run at the same period but with 1 ms offset. To study the influence of the speed of the preceding task on the RJT, Task B was run with a random speed (10 to 60 MHz) while the running speed of Task A was fixed for each experiment (again in the range 10 to 60 MHz).

The interval between start times of the RJT were measured: the results are shown in Figure 5.8.

Table 5.2: Task set parameters

Task	Period (ms)	Offset (ms)	RJT
A	2	0	Y
B	2	1	N

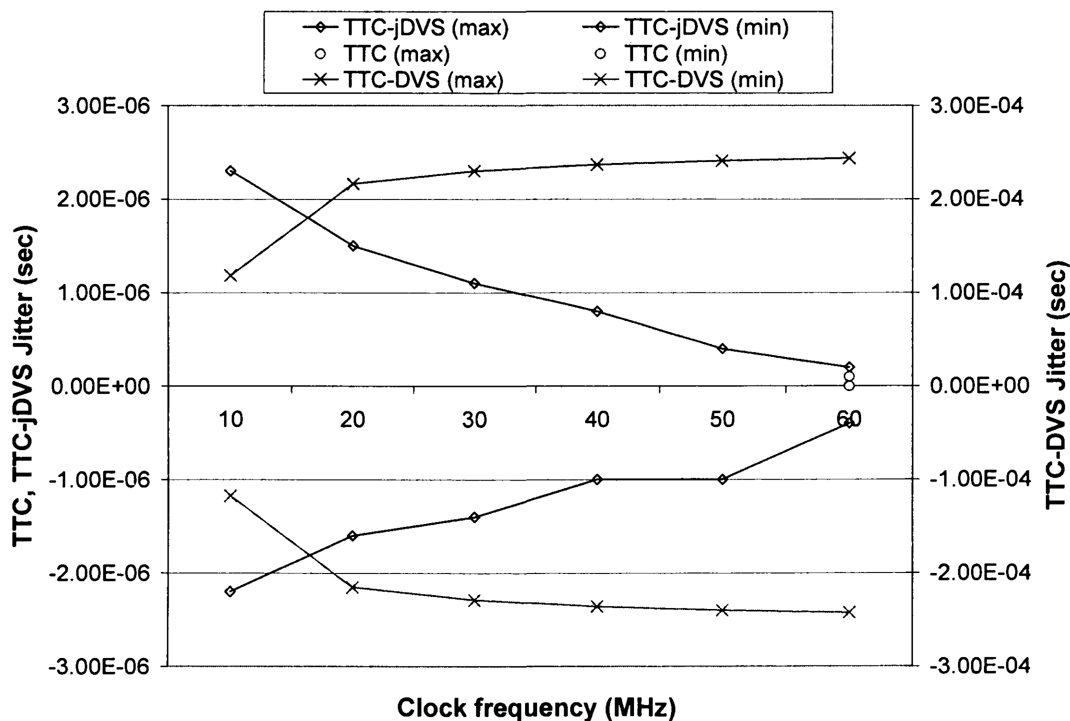

Figure 5.8: Minimum and maximum jitter level of RJT at speed 10-60 MHz run by TTC-jDVS, TTC-DVS and TTC. (Phatrapornnant and Pont 2006, IEEE Transactions on Computer, Vol.55(2), © 2006 IEEE)

Figure 5.8 shows that, in this study, average release jitters from TTC-DVS are in the region of +/-230 μs, while those of TTC-jDVS are in the region of +/-1 μs. Overall, TTC-jDVS reduced the level of release jitter by a factor of approximately 200 when compared with TTC-DVS.

5.4.2 Assessing the impact of TTC-jDVS on CPU power consumption

In order to begin to assess the power-saving ability of the TTC-jDVS algorithm, three schedulers with 1 ms tick intervals (TTC, TTC-DVS, TTC-jDVS) were again used, as in the study described in Section 5.4.1.

To compare the power consumption, 5 dummy tasks (Task A to Task E) which utilised between 10% and 90% of the available CPU activity (when run at the highest speed) in the tick interval (represented as 0.1 U to 0.9 U in Table 5.3) were created. Each dummy task was implemented using a simple loop, adapted to give the required duration.

Note that, in a TTC design, a designer will usually wish to ensure that the WCET of each task is less than the scheduler tick interval: in this case, the duration of all tasks was less than 1 ms. Note also that Task A was (when scheduled using TTC-jDVS) viewed as an RJT. This task was run every 2 ms in (all systems). The remaining tasks (not RJTs) were run every 8 ms with different offsets.

The execution times and other parameters of this task set are shown in Table 5.3.

Table 5.3: Task set parameters for assessing power consumption

Task	Period (ms)	Offset (ms)	RJT	Execution time (μs)								
				0.1U	0.2U	0.3U	0.4U	0.5U	0.6U	0.7U	0.8U	0.9U
A	2	0	1	100	200	300	400	500	600	700	800	900
B	8	1	0	100	100	200	200	400	300	700	800	900
C	8	3	0	100	200	300	600	600	600	600	900	900
D	8	5	0	100	400	400	200	700	800	700	900	900
E	8	7	0	100	100	300	600	300	700	800	800	900

To perform this empirical comparison, the task sets in Table 5.3 were executed using TTC, TTC-DVS and TTC-jDVS algorithms. The average power consumption of the CPU core was measured in each case and is shown in Figure 5.9.

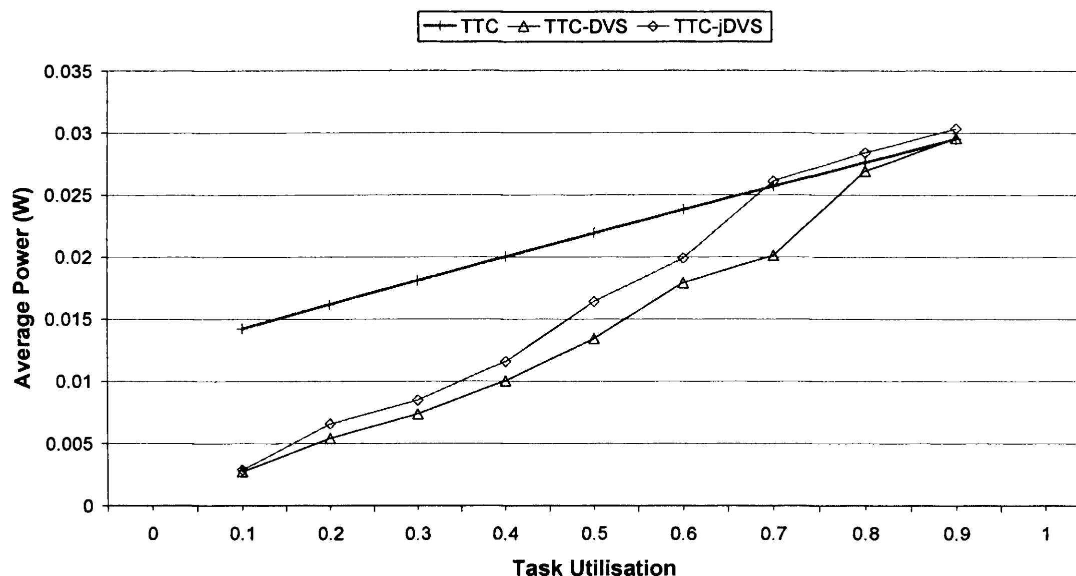

Figure 5.9 : CPU power consumption comparison of scheduling algorithms at different load. (Phatrapornnant and Pont 2006, IEEE Transactions on Computer, Vol.55(2), © 2006 IEEE)

In Figure 5.9, it is clear that the TTC power consumption is almost linearly related to the workload. Also from this figure it can be seen that TTC-DVS is the most power-efficient of the algorithms in this study, followed by TTC-jDVS. At less than 0.6 U, all DVS algorithms consume less power than the TTC algorithm. The TTC-DVS algorithm consumes less power than TTC up to 0.8 U, while TTC-jDVS consumes less power than TTC up to 0.7 U.

The TTC-jDVS algorithm is more complicated than the TTC-DVS algorithm. The TTC-jDVS algorithm includes a tick-compensation process which is performed whenever the frequency changes. Furthermore, the TTC-jDVS algorithm has to expend some slack time in order to offset the activation of the RJTs and run the RJTs with fixed frequency. These jitter-reduction mechanisms introduced in the TTC-jDVS algorithm inevitably degrade the power-saving performance of the original DVS algorithm. In practice, as can be seen from the power consumption results, the TTC-jDVS algorithm consumes more power than the TTC-DVS algorithm over most of the workload range (up to a utilisation of 0.7). If the application has high workload (i.e. utilisation greater than 0.7) the TTC algorithm should be considered (in place of TTC-jDVS) because of its simplicity and low-jitter characteristics.

5.5 Impact of jitter on sampled data systems

As discussed in Section 3.2, jitter has direct impact in data sampling applications by degrading an accuracy of the use of ADCs. The sampling jitter (caused by clock jitter, delay in task scheduling and so on) can impose the error on the sampled data. The error of signal in ADC can be determined by:

$$V_{error} = \text{slew rate} \cdot t_{jitter} \tag{5.1}$$

Assume that a sine wave signal to be sampled is:

$$v(t) = A\sin(2\pi ft) \tag{5.2}$$

The maximum slew rate of a sine wave is at the zero crossing, $t = 0$, and the slew rate is defined by the first derivative of the signal function (Brannon and Barlow, 2006):

$$\frac{d}{dt}v(t) = A2\pi f \cos(2\pi ft) \tag{5.3}$$

The maximum amplitude (A_{FS}) of the sampled signal using an N-bit ADC can be represented by:

$$A_{FS} = 2^N Q \tag{5.4}$$

Where Q is quantum or the minimum voltage step (LSB, least significant bit) (ATMEL, 2003). To ensure conversion accuracy, the V_{error} should be less than $Q/2$. From Equation (5.1), it can infer that the sampling jitter t_{jitter}, determined at time $t = 0$, must satisfy this condition:

$$t_{jitter} < \frac{1}{2^N} \cdot \frac{1}{2\pi f} \tag{5.5}$$

For example, suppose there is a 500 Hz tone to be sampled using an 8-bit ADC with 1kHz sample rate. Maximum allowable jitter will be:

$$\frac{1}{2^8} \cdot \frac{1}{2 \times 500\pi} = 1.24\ \mu s$$

If the system cannot meet this jitter requirement, the data obtained will be degraded: in other words, it is not worth using an ADC with this resolution.

5.6 Wireless ECG: A case study

The studies described in earlier parts of this chapter are based on artificial task sets. To explore the potential application of the TTC-jDVS algorithm, a more realistic case study was carried out: this required both accurate data sampling (that is, low jitter) and energy efficiency. The case study was based on a wireless system for monitoring electrocardiograms (ECGs).

Briefly, an ECG is an electrical recording that is used for investigating heart disease (Hampton, 1998). In a hospital environment, ECGs normally have 12 leads (standard leads, augmented limb leads and precordial leads) and are sampled at 250 Hz (minimum requirement). In this study, the wireless ECG design was intended to allow the recording of the three standard leads (Lead I, Lead II, and Lead III) at 500 Hz: this type of recording is often considered to be sufficient for an initial diagnosis. In this study, the electrical signal from the heart was quantised by a 12-bit ADC and all 3 channels of data were passed to a "HandyCore" (HandyWave, 2004) Bluetooth module for transmission ("live") to a PC's serial port. The data were then plotted on the PC screen.

The interface to the Bluetooth module employs an "RS-232" protocol, which can support baud rates from 2,400 – 115,200 baud. On the PC, LabView 6.1 (NI, 2001) was used to create a program for displaying the ECG traces. The program again received ECG data via a PC's serial port at 115,200 baud.

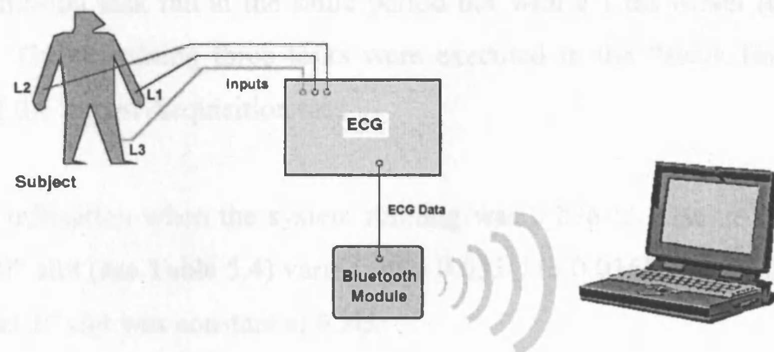

Figure 5.10: A schematic representation of a system for ECG monitoring.

Note that no ECG recordings were made in the study described here. Instead, to allow an accurate measurement of jitter, a National Instruments "NI PCI-6035E" (NI, 2000) and LabVIEW 6.1 (NI, 2001) were used to generate 100 Hz sinusoidal signals which were subsequently processed using a passive low-pass filter (500 Hz cut-off). The signals were then sampled by the ECG unit, and passed (via the Bluetooth link) to a PC where they were subsequently analysed.

From the requirements of the design in the case study, the maximum level of jitter which will not degrade any accuracy of ADC is 0.15 μs (considering at 250 Hz input frequency: see Equation 5.5).

5.6.1 The task set

All the ECG tasks were periodic, and had the characteristics shown in Table 5.4. The worst-case execution time was measured using an oscilloscope when the system ran at full speed (60 MHz).

Table 5.4: ECG task set parameters

Task	Period (ms)	Offset (ms)	WCET(μs)	RJT	Utilisation
Signal Acquisition	2	0	70	Y	0.035
Transmission	2	1	400	N	0.2
Switch Read	100	0	20	N	0.0002
Link Check	200	0	20	N	0.0001
Status Display	400	0	30	N	0.000075

The system's clock tick interval was 1 ms. The Signal Acquisition task was defined as an RJT and ran every 2 ms to meet the requirement of a 500 Hz sampling rate. The

Data Transmission task ran at the same period but with a 1 ms offset (to avoid task collisions). The remaining three tasks were executed in the "slack time" after the execution of the Signal Acquisition task.

The overall utilisation when the system running was 0.236 U. The utilisation during the "Offset 0" slot (see Table 5.4) varied from 0.035U to 0.036U, while the utilisation in the "Offset 1" slot was constant at 0.2U.

5.6.2 Comparing jitter levels

For assessing jitter in this case study, the ECG application was run with the three scheduling algorithms measured the intervals between the start times of the signal-acquisition task, again using the PCI-6035E data acquisition card. Measurements were taken from 10,000 consecutive samples: the results are shown in the form of histograms in Figure 5.11 to Figure 5.13.

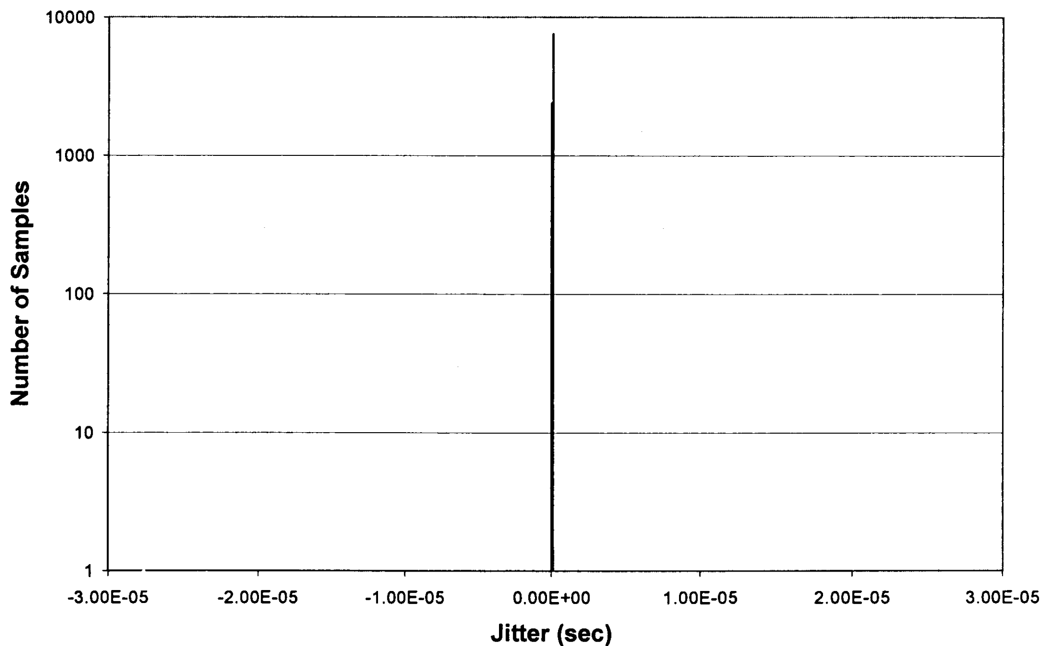

Figure 5.11: Histogram of period jitter for ECG study (TTC). (Phatrapornnant and Pont 2006, IEEE Transactions on Computer, Vol.55(2), © 2006 IEEE)

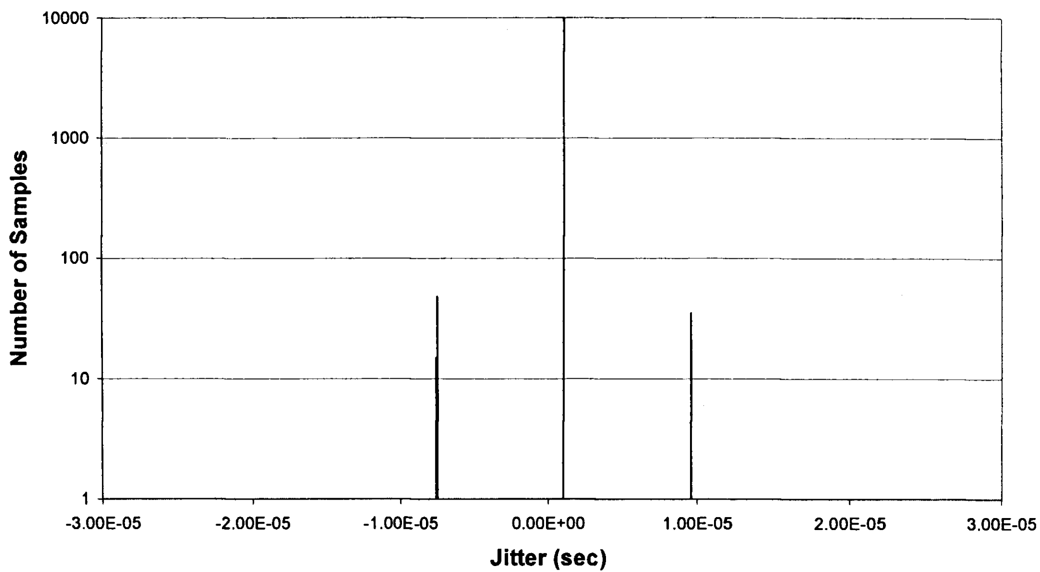

Figure 5.12: Histogram of period jitter for ECG study (TTC-DVS). (Phatrapornnant and Pont 2006, IEEE Transactions on Computer, Vol.55(2), © 2006 IEEE)

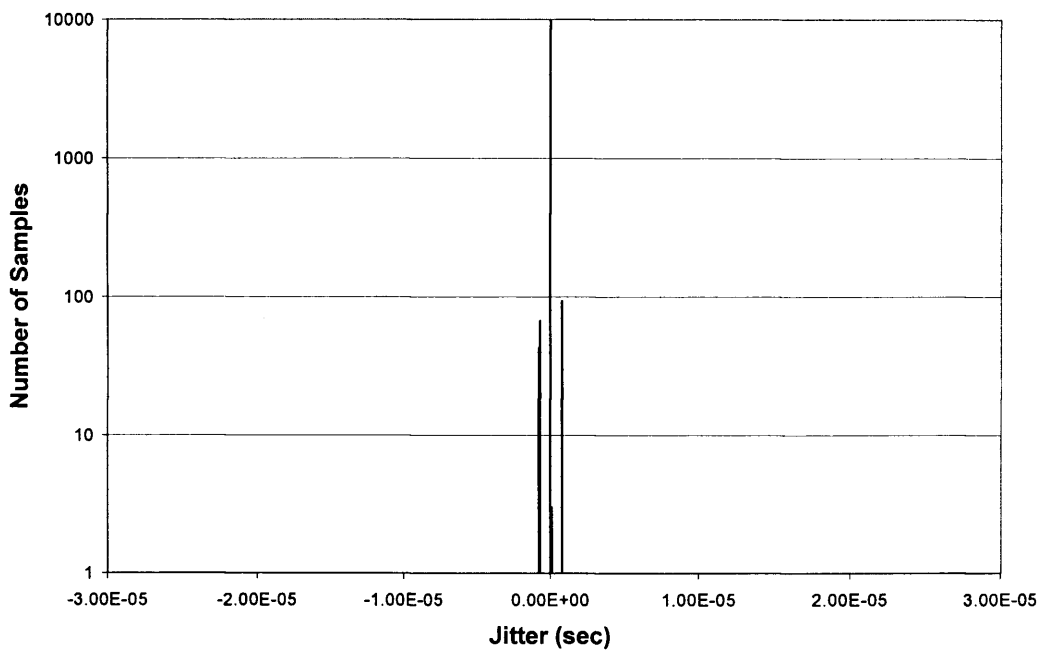

Figure 5.13: Histogram of period jitter for ECG study (TTC-jDVS). (Phatrapornnant and Pont 2006, IEEE Transactions on Computer, Vol.55(2), © 2006 IEEE)

From Figure 5.11 it is clear that jitter measured from the TTC algorithm is approximately +/-0.1 μs. The jitter level in the system using TTC-DVS (Figure 5.12) is approximately -7.6 μs to +9.6 μs, while the level for the system using TTC-jDVS

(Figure 5.13) is from approximately -0.7 μs to +0.8 μs. Overall, these results are consistent with the findings from the artificial task set (in Section 5.4).

The jitter level that can have an impact on data accuracy in the ECG system can be calculated using Equation 5.5 and information about the ECG setup given in Section 5.6. With a 250 Hz sampling rate and a 12-bit ADC, the maximum allowable jitter will be 0.155 μs. In other words, if jitter levels of the sampling task are less than 0.155 μs, they will not have an impact on data accuracy. From the results, the TTC algorithm had a total jitter of approximately 0.2 μs: this means that there may be an impact on the least significant bit (LSB) in the sampled data. When employing the TTC-DVS algorithm, the data loss reaches a level of 7 LSBs, while with the TTC-jDVS algorithm can recover 3 bits (up to 4 LSBs are lost).

Note that modified algorithms for TTC-jDVS (TTC-jDVS2 and TTC-jtDVS2) which can improve levels of jitter are presented in Appendix C and Chapter 7.

5.6.3 Comparing CPU power consumption

To compare the power consumption of the CPU core in this study, a National Instruments PCI-6035E data acquisition card was used to measure the current and voltage characteristics of the core.

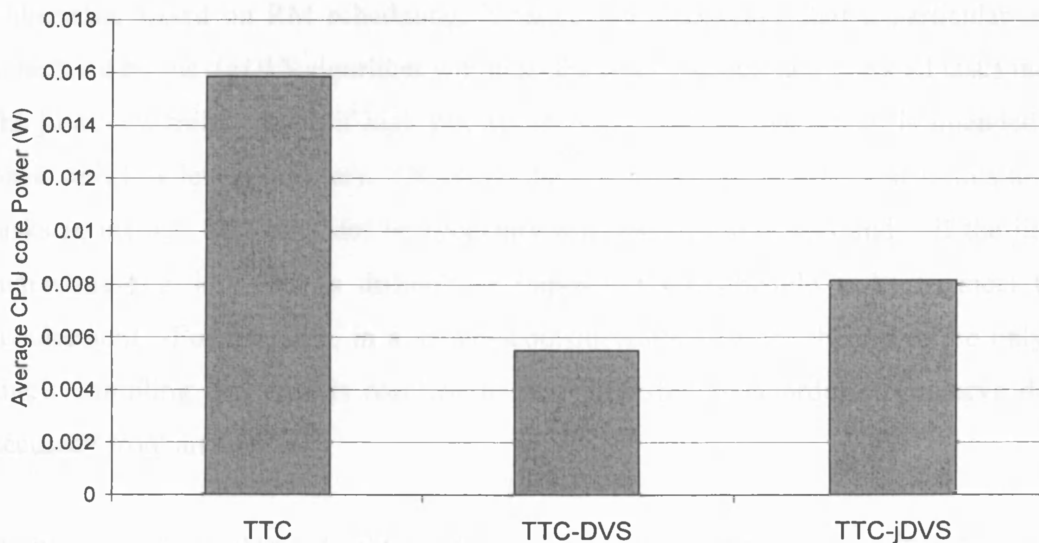

Figure 5.14: The comparison of CPU core power consumption of TTC, TTC-DVS and TTC-jDVS. (Phatrapornnant and Pont 2006, IEEE Transactions on Computer, Vol.55(2), © 2006 IEEE)

The average CPU power consumption was found to be 5.498 mW, 8.165 mW and 15.861 mW for the TTC-DVS, TTC-jDVS and TTC algorithms respectively (see Figure 5.14).

5.7 Discussion

In this section, the results presented in this chapter are compared with those obtained by other researchers. Ways in which TTC-jDVS might be applied on a wider range of hardware platforms is also discussed.

5.7.1 Other work on jitter and DVS

There have been only a small number of studies which have considered jitter and DVS. As discussed in Section 2.4.1, Lee *et al.* (2005), Son *et al.* (2001), Pouwelse *et al.* (2001c) *and* Nurvitadhi *et al.* (2003) have studied the impact of DVS on a video decoding application, focusing on decoding-time prediction. The only previous study to have considered the impact of DVS on real-time scheduling is by Mochocki *et al.* (2005).

Mochocki *et al.* (2005) introduced *JADVS* (RM-based) algorithm to schedule tasks in order to meet jitter constraints. Unlike the studies presented in this thesis, the main aim of the Mochocki algorithm was to minimise jitter in the finishing times of tasks. Otherwise, based on RM scheduling, it cannot be guaranteed that a particular task scheduled by the *JADVS* algorithm will meet the jitter constraints unless all tasks meet the jitter constraints (even if high priority is assigned to a task which is intended to have with low levels of jitter). This is because *JADVS* applies jitter constraints to all tasks in the set. This may not be necessary for many applications and – if the jitter requirement is low – it is difficult or impossible to schedule tasks to meet the requirement. For example, in a signal-acquisition application, there may be only a single sampling task that is required to run "jitter-free", in order to preserve data accuracy from an ADC.

Furthermore, in *JADVS* scheduling, the execution jitter (and also sampling jitter) can be large because: (i) tasks are allowed to run with various frequencies and (ii) task pre-emption can occur. In addition, Mochocki *et al.* (2005) did not study the impact

of DVS on system overheads (task scheduling, frequency/voltage calculation and transition times): as has been demonstrated in this thesis, these factors can have a significant impact on both task timing and energy-savings.

5.7.2 Power consumption results

Although TTC-jDVS has demonstrated good levels of jitter reduction (when with TTC-DVS) such reductions are of little value if the savings in CPU power consumption obtained in TTC-DVS are lost when the jitter-reduction algorithm is incorporated.

Looking at the power consumption results presented in this chapter (Section 5.6.3), it can be expected that applications scheduled using TTC-jDVS will consume more power than those scheduled using TTC-DVS, because of the impact of the jitter guardian and the fact that the RJTs run at a higher speed. However, when compared with the TTC algorithm, TTC-jDVS still saves power at both low and medium workloads. For high workloads, where there is no available slack time, TTC-jDVS runs at the highest speed (like TTC) but consumes more power, due to the more complex scheduling algorithm. Overall – in the case study – when compared with the TTC power consumption, TTC-jDVS showed a reduction of approximately 48%.

It is difficult to make a detailed comparison between these results and those from other DVS studies (because the algorithms and application areas vary considerably). However, the available results do suggest that – even with the additional scheduler load required to reduce jitter – the results obtained here are in line with those from other DVS studies. For example, in (Lorch and Smith, 2001), using simulations based on real workloads, basic DVS schemes have been shown to reduce the CPU energy consumption of by – on average – around 54%, with a more advanced algorithm (PACE) further reducing CPU energy consumption by an additional 20%. Other researchers have demonstrated enrgy savings – using advanced DVS schemes (such as feedback-based EDF scheduling) – of up to 64% when compared with simple DVS schemes (Zhang and Chanson, 2003). Similarly, Pering showed his algorithms reduce system energy by about 46% while maintaining the peak performance demanded by general purpose systems (Pering *et al.*, 1998a).

Please note that none of the alternative algorithms mentioned has been designed to reduce jitter levels. Please also note that Mochocki *et al.* (2005) do not report on power consumption in their study.

5.7.3 Working with other hardware platforms

The empirical results presented in this chapter have been demonstrated only on a single platform. However, this is based on a popular (ARM) core, with the consequence that TTC-jDVS can be directly applied in a wide range of existing microcontrollers and microprocessors (e.g. Intel StrongARM SA11x0, STMicroelectronics STR71x, Philips LPC 2000 series) without difficulty. On other platforms with support for frequency scaling these techniques can be readily adapted: such platforms include x86-architecture processors (e.g. AMD : K6-2, K6-3, Duron, Athlon, Intel: Pentium III, Pentium 4, Pentium M), Transmeta Crusoe, Intel XScale, UltraSPARC-III.

More generally, an external frequency scaling device (such as that described in (Lattice, 2005)) allow the techniques described in this study to be applied with an even wider range of "off the shelf" processors.

To overcome the limitation of software, it is possible to improve performance of the TTC-jDVS algorithm by implementing hardware. For example, conventional microcontroller designs tend to have the inputs to the timers and the CPU clock linked together, meaning that the timer reload values (for the scheduler ISR) must be adjusted when the clock frequency is changed: as it was discussed in Section 4.7.1 this is the main cause of tick jitter problems. The details of hardware development to improve the algorithm performance are illustrated in Chapter 7.

For more advanced improvement, it can have the option of moving the TTC-jDVS algorithm into SoC designs. For example, it can have dedicated timer units to facilitate jitter sensitive tasks or any dedicated hardware to reduce complexity of hardware. As recent studies have shown, DVS algorithms may be readily applied in such designs (Burd *et al.*, 2000; Schmitz and Al-Hashimi, 2000; Lee *et al.*, 2002; Flautner *et al.*, 2003). Direct application of the TTC-jDVS algorithm in SoC designs is therefore a straightforward proposition.

5.8 Conclusion

In this chapter, the impact that the use of DVS has on the levels of both clock and task jitter in TTC applications has been considered. A modified DVS algorithm (TTC-jDVS) which can be used where low jitter is an important design consideration has been described. The effectiveness of the modified algorithm has been demonstrated on both a data set made up of artificial tasks, and in a more realistic case study. The results obtained here have been compared with those from other studies and ways in which TTC-jDVS can be applied using a range of SoC and "off the shelf" embedded platforms have been considered.

Overall, it can be concluded that use of DVS is a practical way of reducing CPU energy consumption even in embedded systems with severe resource constraints and for which low levels of jitter are an important design consideration.

Chapter 6

Working with a hybrid scheduler

In Chapter 5, the TTC-jDVS, algorithm for reducing jitter in TTC systems employing DVS, was demonstrated. In this chapter, alternative solutions for situations in which a TTC-jDVS design is not appropriate are presented. Specifically, the studies in this chapter explore ways to implement DVS in a TT "hybrid" design[13].

6.1 From TTC to TTH

Chapter 5 explored ways in which DVS could be incorporated in a very simple "time triggered co-operative" (TTC) scheduling algorithm in order to reduce power consumption and also maintain the low-jitter characteristics. The resulting TTC-jDVS algorithm was shown to achieve a 48% power reduction (compared to an equivalent system based on TTC) in a representative case study.

Of course, a TTC solution is not always appropriate. As Allworth has 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." (Allworth, 1981). It can be expressed this concern 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

---

[13] Part of this chapter also in Phatrapornnant, T. and Pont, M.J. (submitted a).

events then, in situations where $t < e$, a pure co-operative scheduler will not generally be suitable.

For example, for signal-processing applications, the execution of processing tasks (e.g. Fast Fourier Transform, data encryption) normally takes longer than the sampling interval of the signal acquisition task. A TTC design cannot match this combination of sampling and processing requirements, unless developers manually partition the signal processing task to match the sample rate for the signal acquisition task. Such a partitioning process may be difficult to produce and maintain.

In such cases, the TTH design (introduced in Section 3.4.4) provides an alternative. In the TTH design, limited pre-emption is allowed and it is possible to have different operating frequencies for the pre-empting and co-operative tasks.

From the point of view of a DVS algorithm, it is clear that slack time (alone) may not be the best source of information when determining appropriate voltage / frequency pairs for use with a TTH scheduler. As an alternative, a "power model" is proposed in this chapter for jointly calculating the appropriate frequency/voltage pairs for running tasks in the TTH scheduler.

This chapter explores and evaluates an alternative low-energy, low-jitter scheduler for embedded systems where the TTC design is not appropriate.

6.2 A TTH Scheduler

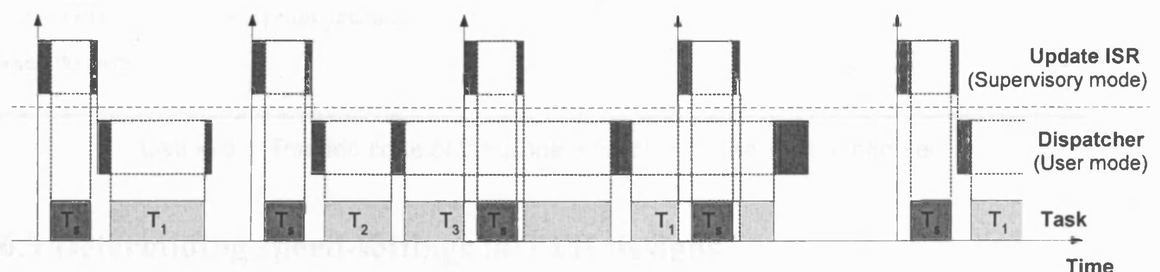

Figure 6.1: TTH scheduling diagram

As discussed in Section 3.4.4, the TTH scheduler is a modification of the TTC scheduler. In the TTC scheduler, the *Update* is used to update a flag only but, in the TTH design, it allows to execute a single short task to pre-empt another co-operative tasks, (see illustrated in Figure 6.1). Thus, the TTH scheduler supports any number of co-operatively scheduled tasks as usual, and also supports a single pre-emptive task – it breaks the nature of purely co-operative scheduler. For the important rule of the TTH design, the task dispatched by the *Update* must be the highest frequency and very short (e.g. less than 50 percents of tick interval – preferably much less), otherwise overall system performance will be impaired (Pont, 2001). Analogous to the rate monotonic algorithm, the pre-emptive task is viewed as the highest priority task. The advantage of the TTH scheduler is that it greatly simplifies the system architecture compared to a fully pre-emptive solution. In particular, it does not need to implement a context switch mechanism.

The *Update* function of the TTH scheduler is shown in Listing 6.1. The part of the pre-emptive task schedule is added into the TTC's *Update.* After the tick interrupt taking place, the Update will check the due to run of the pre-emptive task. If it is due, the pre-emptive task is executed within the ISR while the mechanism to execute co-operative tasks is the same as running in the TTC scheduler.

```
begin UPDATE ISR:

     Tick_Count ++ ;
     if -- Delay.preemp_Task == 0 then
        release preemp_Task ;
        if it is periodic task then
           reload Delay.preemp_Task = Period.preemp_Task ;
        end if
     end if
     reset timer_interrupt_FLAG ;

end  UPDATE
```

Listing 6.1: Pseudo code of the Update function in the TTH scheduler

6.3 Determining speed-settings in TTH designs

In a TTH schedule, it is more difficult to determine the available slack time because task pre-emption can take place and it must be taken into account the number of pre-

emptive task calls that occur during the execution of a given "co-operative" task (Figure 6.2).

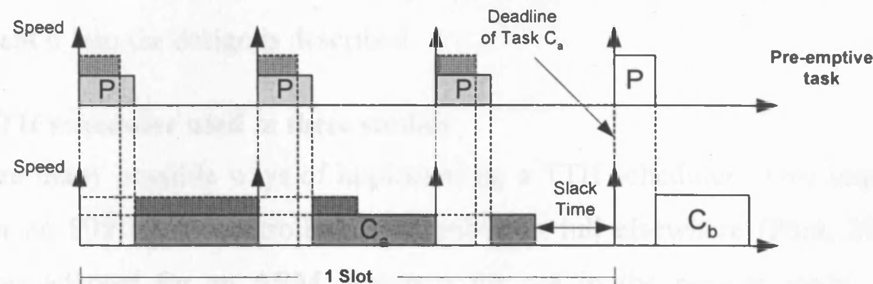

Figure 6.2: Example illustrating the use task stretching in a TTH scheduler

Having determined the available slack time, it is then needed to determine the clock frequency for both co-operative and pre-emptive tasks. Because the clock speed of the pre-emptive task will have an impact on the co-operative task, and vice versa, it must be determined the two speeds simultaneously. This is challenging because it is possible to have two or more combinations of speed settings which will allow the tasks to complete within the available slack time and these combinations may not consume the same amount of energy.

For example, consider two possible design options for a hypothetical system. In the first, the pre-emptive task is run at the lowest frequency (to save power) with the result that the co-operative task must run at full clock speed in order that it meets its deadline. In the second design option, both tasks are run at a medium speed (and again meet the deadlines). Even in such a general example, it is difficult to say which of these options will result in lower energy consumption. In addition – in a real system – there will be further constraints: for example, there will be a limited number of possible frequency settings.

To deal with these issues, a solution based on what it is referred to here as a "power model" of the system is proposed. The details of the power model are presented in the following section.

6.4 Implementing DVS in TTH systems

In this section, it considers the challenges involved in incorporating DVS in systems with a TTH software architecture. A power model is introduced and the way to implement it into the design is described.

6.4.1 TTH scheduler used in these studies

There are many possible ways of implementing a TTH scheduler. One implementation (for an 8051 microcontroller) is described in full elsewhere (Pont, 2001): this code was adapted for an ARM processor for use in the present study. A brief overview of the ARM implementation will be provided in this section.[14]

During normal operation of systems using this TTH scheduler implementation, the first function to be run (after the startup code) is `main()`. Function `main()` then calls `Dispatch()` which in turn launches the co-operative task(s) currently scheduled to execute: it will be assumed in this discussion that a task `C_Task()` may be called. Once any co-operative tasks have completed their execution, `Dispatch()` calls `Sleep()`, placing the processor into a suitable "idle" mode. A timer-based interrupt occurs every millisecond (in typical implementations) which either wakes the processor up from the idle state or pre-empts a long co-operative task. In either case, the ISR `Update()` is invoked, by means of an "IRQ wrapper" function. `Update()` then directly calls the pre-emptive task (here it will be assumed this is `P_Task()`). Once the pre-emptive task is complete, `Update()` increments a tick counter. The function calls then "unwind" back to `main()`, and `Dispatch()` is called again. The cycle thereby continues. This process is summarised in Figure 6.3.

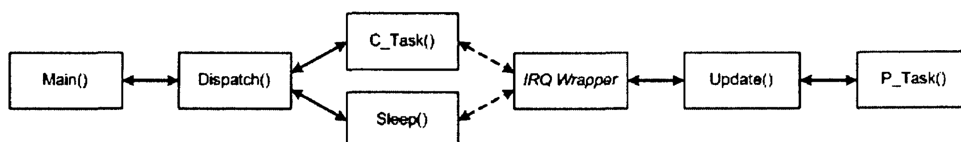

Figure 6.3: Function call tree for the TTH scheduler (normal operation)

[14] The hardware testbed employed in these studies is described in Section 4.1.

Please note that, in most designs, it would be aimed – under normal conditions – to have the pre-emptive task occupy no more than approximately 10% of the tick interval.

6.4.2 Applying DVS in TTH designs

If it is assumed that Figure 6.2 provides a model of the system operation, then the schedulability of the task set can be determined by means of Equation 6.1. In this case, if the total duration of tasks is less than T_{slot}, the task set can be scheduled.

$$T_{slot} > \frac{WCET_C . f_{\max}}{f_C} + \frac{(WCET_P . n_{\text{int}}) . f_{\max}}{f_P} \qquad (6.1)$$

$$n_{int} = \frac{T_{slot}}{R_{\text{int}}} \qquad (6.2)$$

where

T_{slot} is duration of a given task slot in the schedule[15].

$WCET_C$ are worst case execution time of co-operative task(s).

$WCET_P$ are worst case execution time of pre-emptive task(s).

n_{int} is the number of interruption in the task slot.

R_{int} is an interruption rate.

f_C is the clock frequency for running the co-operative task, $f_{min} \leq f_C \leq f_{max}$.

f_P is the clock frequency for running pre-emptive task, $f_{min} \leq f_P \leq f_{max}$.

f_{max} is the maximum clock frequency of the system.

f_{min} is the minimum clock frequency of the system.

However, in a more realistic DVS implementation (see Figure 6.4), it is necessary to take into account the system overheads, such as the voltage/frequency switching time and task scheduling overhead.

---

[15] In this context a "slot" begins with the tick that releases a given co-operative task and ends with the tick that releases the next co-operative task. For example, in Figure 6.4 the slot is of length 3 tick intervals.

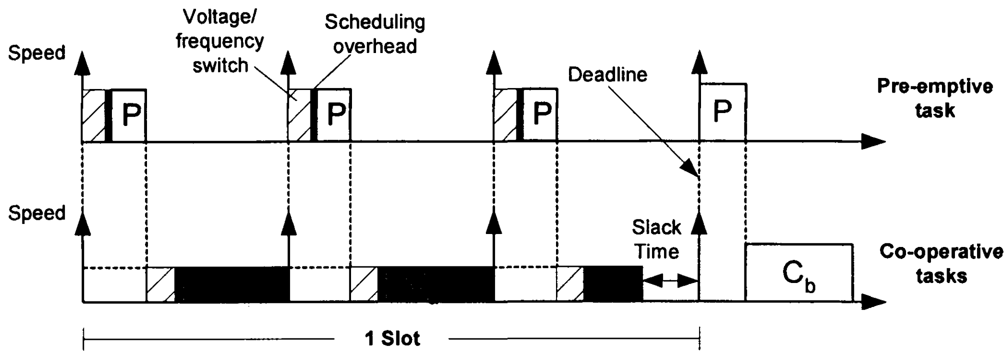

Figure 6.4: Realistic DVS implementation

Based on this more realistic model of the system operation, Equation 6.3 can be derived.

$$T_{slot} > total_t_{sw} + \frac{(WCET_C + (sch_C.n_{int})).f_{max}}{f_C} + \frac{(WCET_P + sch_P).n_{int}.f_{max}}{f_P} \quad (6.3)$$

where:

$total_t_{sw}$ is total voltage/frequency switching time in the slot that has value between 0 and $2\,t_{sw}.n_{int}$.

t_{sw} is voltage/frequency switching time.

sch_C is scheduling overhead to schedule co-operative tasks.

sch_P is scheduling overhead to schedule pre-emptive tasks.

The switching time $total_t_{sw}$ can be zero when all tasks are run at the same frequency or can be t_{sw} when the frequencies in a slot are the same but those between slots are different. The maximum value of $total_t_{sw}$ is $2\,t_{sw}.n_{int}$.

It is important that it is taken into account the practical realities of the DVS process. On the type of small processor with which it is concerned in this thesis (typically ARM7TDMI architecture), voltage/frequency switching, t_{sw}, will typically take around 100-200 μs. This amounts to around 20% of a typical (1 ms) tick interval. Clearly, for reasons of system performance (as well as jitter reduction) it is needed to keep the number of voltage / frequency switches to a minimum.

6.4.3 Creating and assessing a power model

In order to identify the optimal running speeds for both the pre-emptive and co-operative tasks, a "power model" was created, based on data from an empirical study. To do this, a target CPU was instrumented. Tasks were executed that required from 0% to (almost) 100% of the tick intervals, using the range of possible operating frequencies that the target CPU allowed. The power consumption was measured for the different options and then plotted: note that, as it would be expected, the results are consistent with Equation 2.3. In this study, the voltage levels of each speed from 10 to 60 Mhz are 1.023, 1.034, 1.067, 1.144, 1.254 and 1.375 volts respectively, while the effective load capacitance was found to be approximately 0.18 nF.

Please note that such a model assumes that the characteristics of different tasks in the system will always be the same. This is a simplification, since it is known from previous studies that each processor instruction consumes different power: for example Load and Store instructions are more power intensive than other instructions (Sinha and Chandrakasan, 2001). However, most realistic tasks consist of various combinations of instructions and it is assumed here that – over the execution period of a given task – the average power consumption will be similar.

Given this assumption, Figure 6.5 shows the load characteristics from one CPU core (LPC2106) which has six speed steps (from 10 MHz to 60 MHz). The load lines are represented in term of utilisation (% of tick interval, 1 ms in this case) when run at maximum speed (60 MHz in this example). Therefore, if a task is run at 10 MHz with full load, it takes around 166 μs at 60 MHz. The results in Figure 6.5 illustrate that significant average power (or energy) can be saved when running at lower speed (and also lower voltage). For example, running with full load at 30 MHz consumes the same average power as running with no load at 60 MHz.

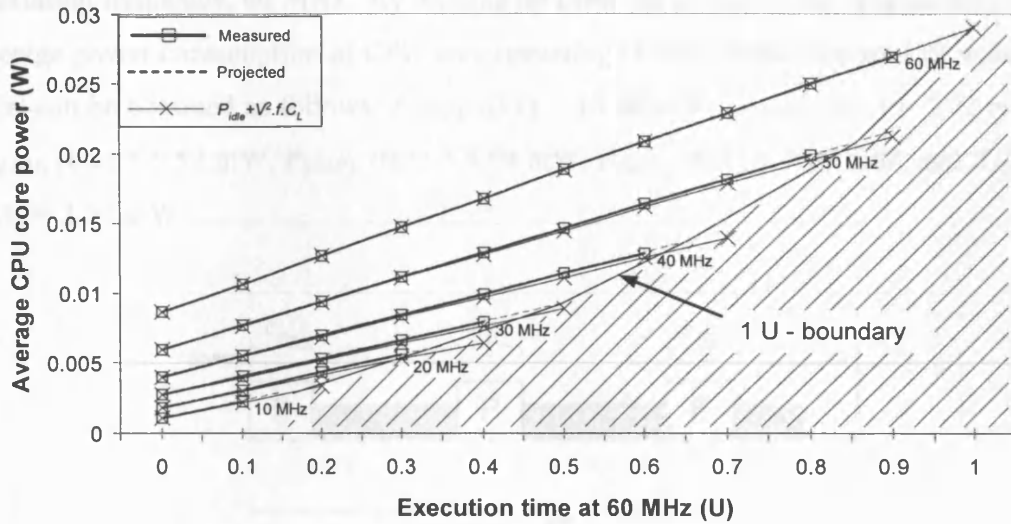

Figure 6.5: Load characteristics of a Philips LPC2106 processor (ARM7TDMI core) with 6 speed steps

Given such a model, the power consumption of tasks in one task slot can be predicted by means of Equation 6.4.

$$P_{total} = P_{freq}(u_P) + P_{freq}(u_C) \tag{6.4}$$

where:

u_P is utilisation of pre-emptive tasks in the given task slot.

u_C is utilisation of co-operative tasks in the given task slot.

P_{freq} is power function of CPU which running at clock frequency *freq*.

$$u_P = \frac{\sum_{i=1}^{n} e_{Pi}}{T_{slot}} \tag{6.5}$$

$$u_C = \frac{\sum_{i=1}^{n} e_{Ci}}{T_{slot}} \tag{6.6}$$

The utilisation of pre-emptive and co-operative tasks is defined in Equation 6.5 and Equation 6.6, where e_P and e_C are duration of pre-emptive and co-operative tasks respectively and T_{slot} is the length of the corresponding task slot (see Figure 6.6). For example, if there is a (pre-emptive or co-operative) task which runs periodically every 10 ms and its execution time is 1 ms, its utilisation equals 0.1 when considered at

maximum frequency, 60 MHz. By looking up from the power model (Figure 6.5), the average power consumption of CPU core operating at each frequency (and its voltage pair) can be obtained as follows: P_{60MHz} (0.1) = 10.68 mW, P_{50MHz} (0.1) = 7.72 mW, P_{40MHz} (0.1) = 5.52 mW, P_{30MHz} (0.1) = 4.04 mW, P_{20MHz} (0.1) = 3.09 mW, and P_{10MHz} (0.1) = 2.30 mW.

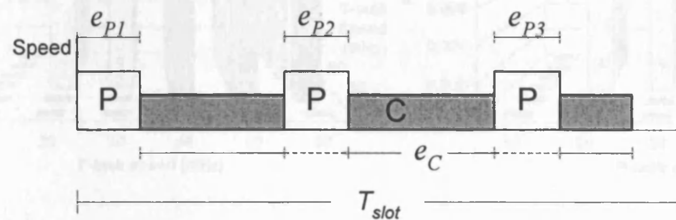

Figure 6.6: Utilisation of tasks in one task slot

In order to identify the appropriate execution speeds, it is necessary to know the (worst case) execution times of all tasks, when run at full speed. The power model can then be used to explore different speed options and determine (if the tasks can be scheduled) what the resulting power consumption will be. Please note that the search process is simplified by assuming that (at most) two speeds will be used in each task slot, one for the pre-emptive task and one for any / all co-operative tasks. Please also note that the scheduler overhead is not taken into account (at this stage).

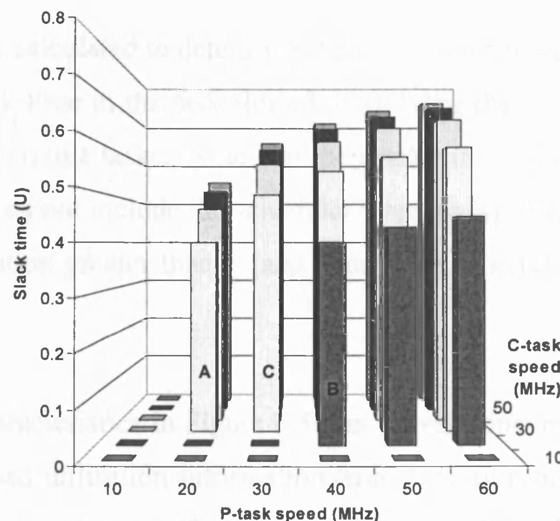

Figure 6.7: Slack time presented after applying running speeds of an example: e_P = 0.1ms, e_C = 0.3ms, T_{slot} = 2ms, and R_{int} = 1ms

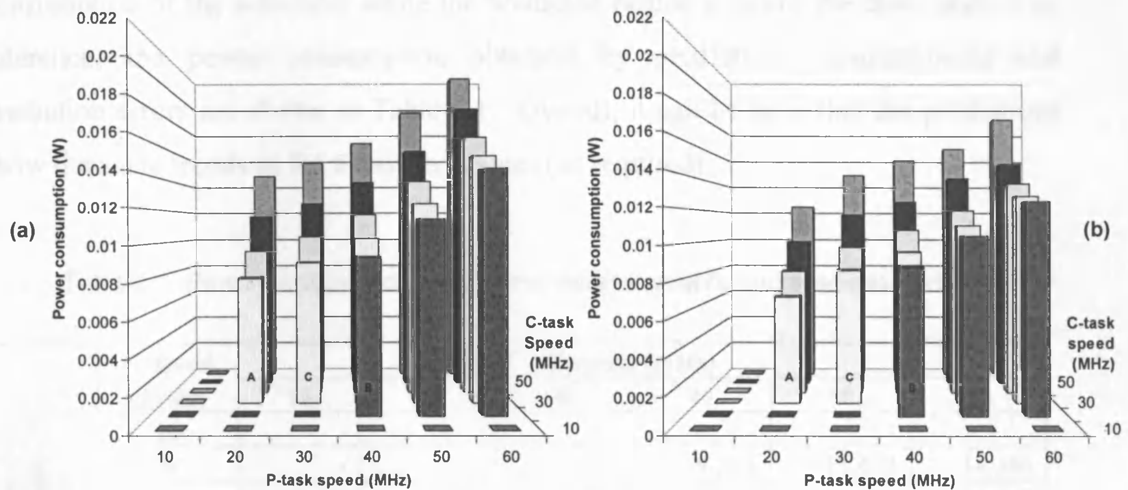

Figure 6.8: CPU power consumption: (a) predicted by using power model, (b) measured of an example: e_P = 0.1ms, e_C = 0.3ms, T_{slot} = 2ms, and R_{int} = 1ms

6.4.4 Using the power model: example

The use of the power model is demonstrated in Figure 6.7 and Figure 6.8. For this example, loads consist of pre-emptive, e_P, and co-operative, e_C, tasks which have execution times of 0.1 ms and 0.3 ms respectively (when run at the maximum speed of 60 MHz) while the task slot duration, T_{slot}, is 2 ms. The interruption rate (R_{int}) is 1 ms.

First, the loads are calculated to determine the execution time at all speeds. Figure 6.7 illustrates the slack time in the task slot after applying the running speeds to the pre-emptive and co-operative tasks. Note that the slack time presented takes into account the task only (it does not include the scheduler overheads). Please also note that cases which have utilisation greater than 1 (and cannot be scheduled) are not presented in the graphs.

Using the load characteristics in Figure 6.5, the power consumption can be calculated by applying the load utilisation information from Equation 6.4. Figure 6.8(a) shows power-consumption predictions (for the tasks only) for all load combinations in which pre-emptive and co-operative tasks are run from 10 MHz to 60 MHz. Figure 6.8(b) shows the corresponding power–consumption measurements. Please note that a perfect match cannot be expected since the measured results include the power

consumption of the scheduler while the predicted results concern the tasks only. For reference, the power consumption obtained by prediction, measurements and prediction errors are shown in Table 6.1. Overall, it can be seen that the predictions show the same trends as the measured values (as required).

Table 6.1: Power consumption predictions, measurements and prediction errors

	C-speed (MHz)	P-speed (MHz) 10	20	30	40	50	60
Power Predicted (mW)	10	-	-	-	-	-	-
	20	-	-	-	9.228	11.432	14.396
	30	-	7.782	8.741	10.212	12.416	15.380
	40	-	9.336	10.295	11.766	13.970	16.934
	50	-	11.676	12.635	14.106	16.310	19.274
	60	-	14.787	15.746	17.217	19.422	22.386
Power Measured (mW)	10	-	-	-	-	-	-
	20	-	-	-	8.658	10.416	12.461
	30	-	6.623	8.272	9.428	11.051	12.914
	40	-	8.034	9.596	10.774	12..043	13.880
	50	-	9.904	11.803	12.759	14.410	15.396
	60	-	12.446	14.829	15.969	16.822	18.962
Prediction Errors (%)	10	-	-	-	-	-	-
	20	-	-	-	6.57	9.75	15.52
	30	-	17.50	5.67	8.32	12.35	19.10
	40	-	16.21	7.28	9.21	16.00	22.00
	50	-	17.89	7.05	10.56	13.19	25.19
	60	-	18.81	6.18	7.82	15.45	18.06

It should be emphasised that stretching tasks in order to employ as much slack time as possible does not always result in minimal power consumption. This is a consequence of the more complex nature of the algorithm that results from the possibility of pre-emption (and is in contrast to the results obtained with the simpler TTC-jDVS algorithms and most other studies in this area (Ishihara and Yasuura, 1998a; Pering and Brodersen, 1998; Shin and Choi, 1999; Lee and Sakurai, 2000; Kawaguchi *et al.*, 2001; Pillai and Shin, 2001; Swaminathan and Chakrabarty, 2001; Zhang and Chanson, 2003)).

As an example of this phenomenon, note that, in Figure 6.7, Sample A (in which P-task run at 20 MHz and C-task run at 30 MHz), and Sample B (in which P-task run at 40 MHz and C-task run at 20 MHz), have the same utilisation, but Sample A

consumes less power than Sample B. The tasks in Sample C (which are "stretched" to a lesser extent than those in Sample B) consume less power than those in Sample B.

6.5 The TTH-jDVS algorithm

In this section, the power model (described in Section 6.4.3) is used to reduce power consumption in systems scheduled using a TTH algorithm. The section begins by outlining a simple TTH-DVS algorithm. This algorithm is then refined in order to reduce the levels of task jitter: the resulting algorithm is referred to here as "TTH-jDVS".

6.5.1 TTH-DVS scheduling

In TTH scheduler implementation considered in this chapter, the pre-emptive task is called from (or implemented by means of) an interrupt service routine (ISR) while co-operative tasks are called from a "Dispatcher" routine which runs in an endless `while` loop.

In the ISR, the CPU speed is changed before the pre-emptive task is released. Before returning from the ISR, the speed is set to that of the corresponding co-operative task.

To determine the speed of these tasks, the power model is used to find the running speed of pre-emptive and co-operative tasks that gives the minimum power consumption. The speed-finding process explores all combinations of speed setting (on a task-slot by task-slot basis), until it completes the whole cycle. This process is executed (without dispatching tasks) when the system is initialised. The required speeds settings are stored in a "circular array" (see Section 4.4.3), which has a size equal to the number of tick intervals in one complete periodic task cycle. In this case, the speeds of pre-emptive and co-operative tasks are set individually. A scheme to synchronise the circular array pointer with the task slot will then be run before the scheduler starts. It is emphasised again that each task slot may be longer than the tick interval.

6.5.2 Reducing jitter levels in the TTH-DVS algorithm

In the TTH-jDVS algorithm, a three-step technique is incorporated in TTH-DVS in order to minimise the jitter caused by DVS. The three steps are described in this section.

6.5.2.1 Assumption

It is assumed that it may not be necessary to run all tasks with low jitter. Specifically, in the TTH-jDVS algorithm, the intention is to minimise the jitter associated only with the pre-emptive task: co-operative tasks are run as normal. This is consistent with the fact that, in a TTH design, the pre-emptive task will generally be used for time-sensitive operations (notably data sampling).

6.5.2.2 Tick compensation

Tick compensation is a process which is used to minimise the (tick) jitter which can result from alterations to the timer registers in a scheduled system when DVS is employed. The tick drift is minimised by re-adjusting the timer registers when the CPU operating frequency is changed. The process is described in detail in Section 5.2.1.

Please note that in TTH designs, it is possible to have more than one frequency switch in each tick interval, if the speeds of pre-emptive and co-operative tasks are different.

6.5.2.3 Jitter guardian

A "jitter guardian" is a mechanism used to reduce release jitter which can result in scheduled systems when DVS is employed.

In TTH-based schedulers, the jitter guardian is inserted in order to reduce the variation in the scheduler overhead prior to the release of the (jitter-sensitive) pre-emptive task. Normal (co-operative) tasks are dispatched without any delay. Figure 6.9 illustrates how the jitter guardian protects the DVS system suffered from release jitter as a result of variations in the scheduler overhead.

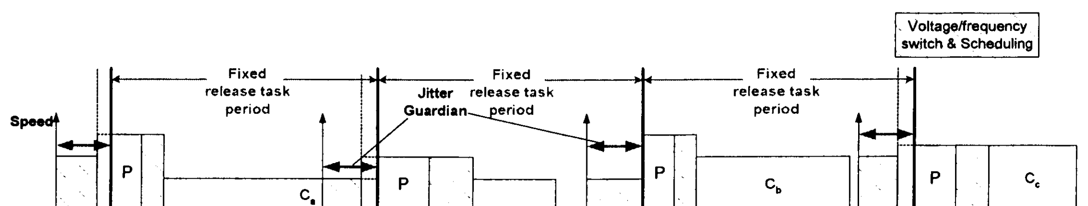

Figure 6.9: Reducing release task jitter by jitter guardian

6.5.2.4 Determining the running speed of the pre-emptive task

Suppose that it has a (pre-emptive) task which has a release time that is completely free of jitter. Further suppose that, in the middle of this task, it performs a data-sampling action. Unless the task runs at the same speed every time it is executed, there will be jitter in the sampling action. To avoid such problems, the task must always be run at the same speed.

In the TTH-jDVS algorithm (see Figure 6.10) the pre-emptive task's speed is set first. Possible speeds for the co-operative tasks are then explored and the power consumption is calculated throughout the whole cycle. This process is repeated, as necessary, starting from different pre-emptive speeds. The combination of speeds which consumes the least power will be selected. This speed-finding process is carried out when the system is initialised and the values are stored in a circular array.

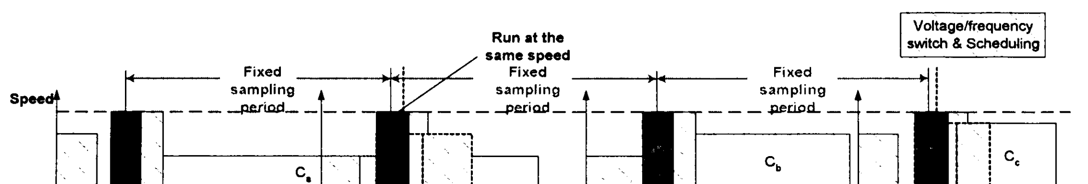

Figure 6.10: Reducing sampling period jitter by fixed running speed

6.5.3 Pulling it all together: TTH-jDVS scheduling

The TTH-jDVS algorithm incorporates all of the features discussed in this section. In addition, in the TTH-jDVS ISR, the jitter guardian is inserted before the pre-emptive task is released. In this chapter, the jitter guardian delay, which is pre-calculated in the initialisation phase, is set by a timer and the system then enters "idle" mode. When the timer matches the delay value, the system is awakened up by the interrupt and runs the pre-emptive task. This process helps to minimise both power consumption and jitter.

6.6 Evaluating the TTH-jDVS algorithm

To evaluate the TTH-jDVS algorithm, a series of representative empirical studies was conducted. The studies are described in this section. Please note that the hardware platform employed in these studies is detailed in Section 4.1.

6.6.1 Assessing the impact of TTH-jDVS on jitter

To explore the impact of the jitter compensation algorithm, a series of tests using an implementation of TTH-jDVS adapted from the TTH scheduler described in the previous section was conducted.

6.6.1.1 Tick jitter

In order to measure tick jitter, two tasks (one co-operative, one pre-emptive) were run at various random speeds (10 to 60 MHz). The tick interval was set to 1 ms. There were two frequency switches within each tick interval. The tasks were scheduled using TTH-DVS and TTH-jDVS. For comparison, the tasks were also scheduled using a TTH scheduler (Pont, 2001), in this case at a fixed speed of 60 MHz. The jitter measurements were based, in each case, on 10,000 consecutive samples. Please note that the tick measurement in this study was probed at the interrupt signal generated by a timer.

For the TTH-DVS algorithm, as illustrated in Figure 6.11, the jitter the range was +/- 3 µs (see Table 6.2). The corresponding test run using the TTH-jDVS algorithm (Figure 6.12) gave a jitter range of +/- 0.3 µs. No measurable jitter was found for the TTH algorithm in these tests.

Table 6.2: Comparing tick jitter run by TTH-DVS, TTH-jDVS, and TTH algorithms

	Jitter (µs)		
	Max	Min	Total
TTH	0.0	0.0	0.0
TTH-DVS	2.9	-3.0	5.9
TTH-jDVS	0.3	-0.3	0.6

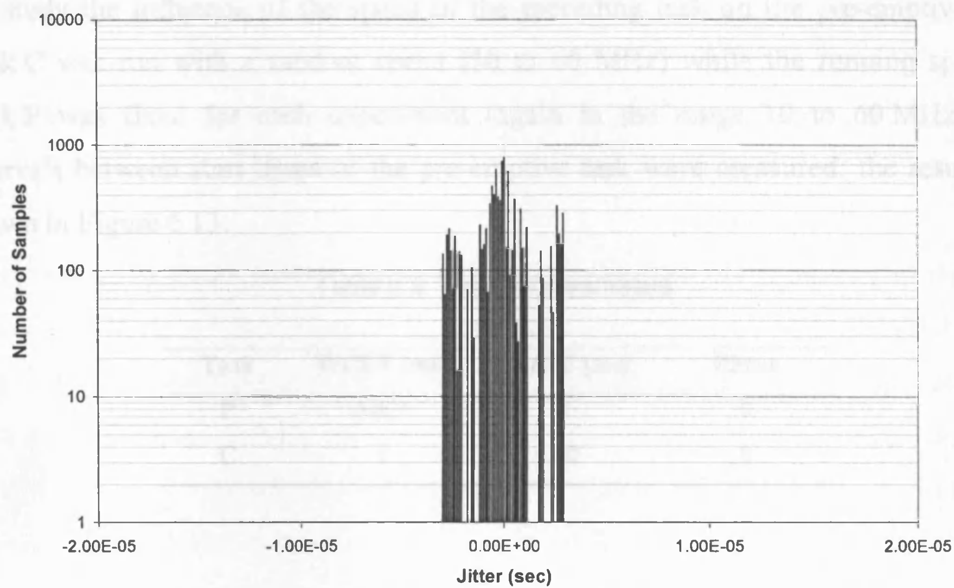

Figure 6.11: Histogram of tick jitter in TTH-DVS

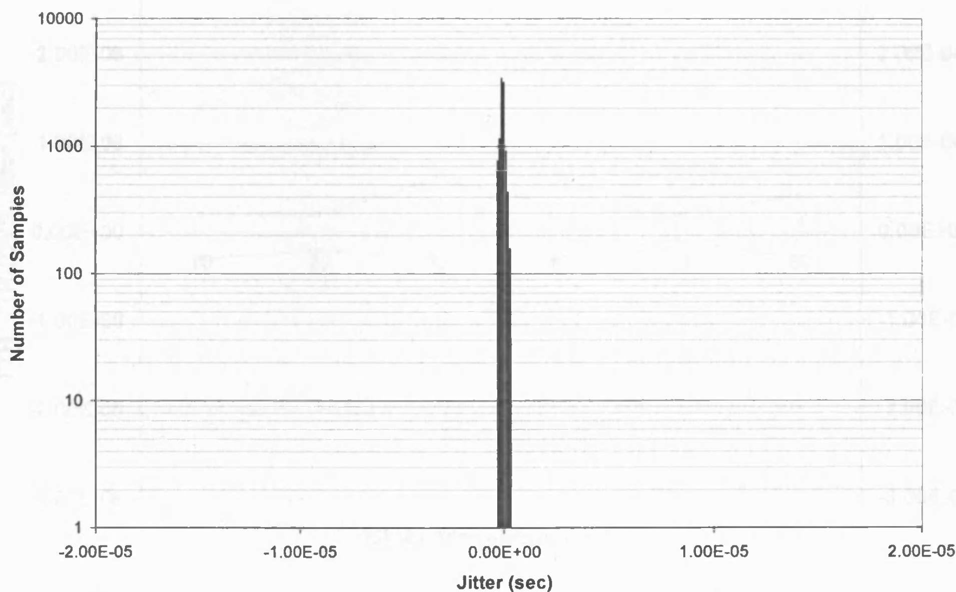

Figure 6.12: Histogram of tick jitter in TTH-jDVS

6.6.1.2 Release jitter

To explore the impact of variable speed on task release jitter, two tasks were again set up (see Table 6.3). In this case Task P was a pre-emptive task and Task C was a co-operative task. Task P was run every 1 ms while Task C was run every 10 ms.

To study the influence of the speed of the preceding task on the pre-emptive task, Task C was run with a random speed (10 to 60 MHz) while the running speed of Task P was fixed for each experiment (again in the range 10 to 60 MHz). The intervals between start times of the pre-emptive task were measured: the results are shown in Figure 6.13.

Table 6.3: Task set parameters

Task	WCET (ms)	Period (ms)	Phase
P	0.025	1	0
C	1	10	0

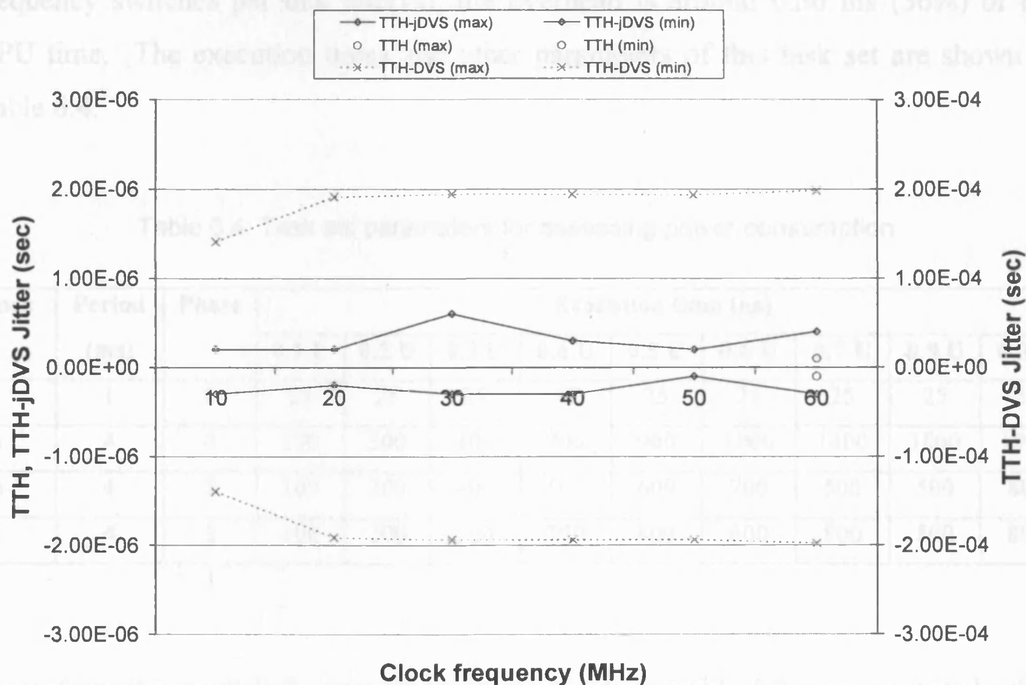

Figure 6.13: Minimum and maximum release jitter level of pre-emptive task at speed 10-60 MHz run by TTH-jDVS, TTH-DVS and TTH

Figure 6.13 shows that, in this study, average release jitter from TTH-DVS was in the region of +/- 200 µs, while that of TTH-jDVS was in the region of +/-0.6 µs and that of TTH was in the region of +/-0.1 µs. Overall, the use of TTH-jDVS reduced the level of release jitter by a factor of approximately 500 when compared with TTH-DVS.

6.6.2 Assessing the impact of TTH-jDVS on CPU power consumption

In order to begin to assess the power-saving ability of the TTH-jDVS algorithm, three schedulers with 1 ms tick intervals (TTH, TTH-DVS, TTH-jDVS) were again used, as in the study described in Section 6.6.1.

To compare the power consumption, 4 dummy tasks (one pre-emptive and three co-operative) which utilised between 10% and 90% of the available CPU time (when run at the highest speed) in the tick interval (represented as 0.1 U to 0.9 U in Table 6.4) were created. Each dummy task was implemented using a simple loop, adapted to give the required duration. The 1 ms period of the pre-emptive task was set to test the scenario in which the scheduling overhead is high. In this case, with two voltage/ frequency switches per tick interval, the overhead is around 0.36 ms (36%) of the CPU time. The execution times and other parameters of this task set are shown in Table 6.4.

Table 6.4: Task set parameters for assessing power consumption

Task	Period (ms)	Phase	Execution time (μs)								
			0.1 U	0.2 U	0.3 U	0.4 U	0.5 U	0.6 U	0.7 U	0.8 U	0.9 U
P	1	0	25	25	25	25	25	25	25	25	25
Ca	4	0	100	300	400	700	900	1000	1400	1800	1900
Cb	4	2	100	200	400	500	600	700	500	500	800
Cc	4	3	100	200	300	300	400	600	800	800	800

To perform this empirical comparison, the task sets in Table 5.3 were executed using TTH, TTH-DVS and TTH-jDVS algorithms. The average power consumption of the CPU core was measured in each case and is shown in Figure 6.14.

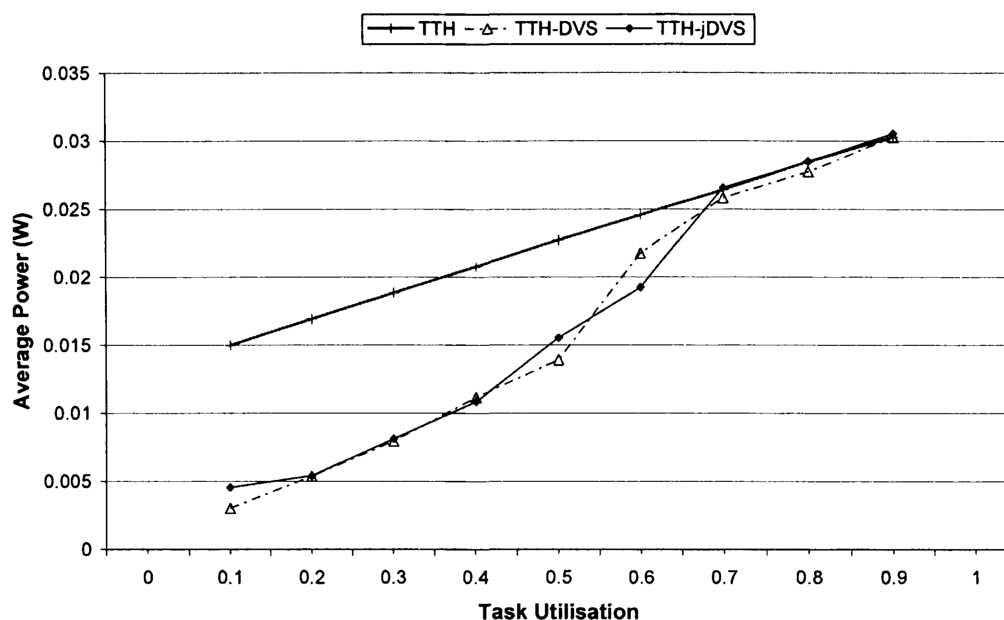

Figure 6.14: Power consumption comparison of scheduling algorithms at different load

From Figure 6.14, it is clear that the TTH power consumption is almost linearly related to the workload. It can also be seen that TTH-DVS is the most power-efficient of the algorithms in this study, followed (closely) by TTH-jDVS. The TTH-DVS algorithm consumes less power than TTH up to 0.8 U because, at 0.7 U and 0.8 U, Task Cb, which has 500 μs duration, could be run at lower speed while the rest were run at the maximum speed. TTH-jDVS consumes less power than TTH up to 0.6 U while, at 0.7 U – 0.9 U, consumes slightly more power than TTH. At 0.6 U, TTH-jDVS can consume less power than TTH-DVS because the former algorithm calculates the power consumption over the whole task cycle.

6.7 Case study: Encrypted wireless ECG monitoring

To illustrate the discussions in this chapter, the TTH-jDVS algorithm is applied in a real-world case study. The problem is discussed in Section 6.7.1, and the task set used is described in Section 6.7.2. The results are compared and discussed in Section 6.7.3.

6.7.1 Problem description

The case study involves a wireless electrocardiogram (ECG) system and is based on the design described in Section 5.6.

In the system considered here, three standard leads (Lead I, Lead II, and Lead III) were recorded at 1 kHz. The 3-channel electrical signals were sampled using a (12-bit) ADC. After sampling, the samples were encrypted (16 samples at a time) using a "blowfish" algorithm (see (Schneier, 1993; Kocher, 1997)). A synchronising byte was then added and the data were passed to a "Bluetooth" module for transmission to a notebook PC, for analysis by a clinician.

In the version of this system discussed here, the following tasks were employed:

- Sample the data continuously at a rate of 1 kHz. Sampling takes less than 0.1 ms.
- When it has 16 samples (that is, every 16 ms), encrypt the data, a process which takes a total of 1.5 ms while the data transmission process which sends 97 bytes of encrypted data with a synchronising byte takes 8.5 ms.

The execution time of both data-processing tasks is greater than the sampling period. It is assumed that the tasks cannot be divided. Thus, a TTC design is not appropriate.

6.7.2 The task set

The wireless ECG unit developed for this study executed 6 tasks. All the ECG tasks were periodic, and had the characteristics shown in Table 6.5. The worst-case execution time was measured using an oscilloscope when the system ran at full speed (60 MHz).

Table 6.5: ECG task set parameters

Task	Period (ms)	Phase	WCET(μs)	Pre-emptive
Signal Acquisition	1	0	30	Y
Encryption	16	0	1200	N
Transmission	16	3	8500	N
Switch Read	80	0	20	N
Link Check	160	0	20	N
Status Display	400	0	30	N

6.7.3 Results

In this case study, the ECG application was run in order to assess the tick and task jitter. The runs involved one of three scheduling algorithms (TTH, TTH-DVS or TTH-jDVS). In each case, the intervals between the start times of the tick (interrupt)

and start times of the signal-acquisition task were measured. The results are shown in Table 6.6.

The TTH results show that there is no measurable tick jitter in this system. However, there is task jitter (in the range of +/- 0.4 μs). This task jitter occurs because the time taken to respond to an interrupt depends which instruction is currently being executed[16]. With the TTH-DVS algorithm, the voltage/frequency switch causes tick jitter of around +/- 2.8 μs, while the task jitter is in the range +/- 188 μs. With the TTH-jDVS algorithm, the tick jitter was in the range of +/- 0.1 μs while task jitter was in the region of - 0.1 μs. Overall, the results from the TTH-jDVS algorithm were significantly better than those obtained for the TTH-DVS algorithm in this study.

From the jitter levels shown in Table 6.6, the 12-bit ADC used with the TTH algorithm will function as a 9-bit ADC (because the three least significant bits will be lost). The TTH-DVS algorithm will suffer more: here, with jitter levels of 318.2 μs, up to 11 bits (out of 12) will be lost. By contrast, with the TTH-jDVS algorithm, a full 12 bits of data resolution will be obtained.

Table 6.6: Tick and release jitter measured

	Tick-jitter (sec)		**Release-jitter (sec)**		**Power consumption (watt)**
	max	**min**	**max**	**min**	
TTH	0.00e-06	0.00e-06	0.40e-06	-0.40e-06	0.0242
TTH-DVS	2.40e-06	-2.80e-06	188.60e-06	-129.60e-06	0.0205
TTH-jDVS	0.10e-06	-0.10e-06	0.00e-06	-0.10e-06	0.0167

The power-consumption figures from this study are also shown in Table 6.6. It is clear that the TTH algorithm results in the highest average (CPU) power requirements (around 24 mW). The TTH-DVS algorithm results, in this case, a power consumption of around 20 mW while the TTH-jDVS algorithm results in a power consumption of around 17 mW. Overall, in this study, use of TTH-jDVS reduces the power consumption by around 30% when compared with the use of the original TTH algorithm.

[16] This is behaviour that it would be expected to see in a simple TTH implementation. Techniques for reducing such jitter are discussed elsewhere (Maaita and Pont, 2005).

6.8 Discussion

The use of power model in this study is limited for the specific processor model "Philips LPC2106". To work with another processor model, it is required to reconstruct it own power model. The power model can be obtained by measuring power consumption from the real system. Firstly, the idle power values of CPU core operating at each frequency/voltage pair are needed to know – by measuring when CPU is idle. Also, the dynamic power can be obtained by measuring the CPU core power consumption at full load of each frequency/voltage pair. Thus, the load lines (from 0 to 1 utilisation) of each frequency/voltage pair can then be drawn. Alternatively, the load line can be generated if the effective capacitance is known – by measuring only one dynamic power value (at full load) and then using dynamic power equation (Equation 2.3) to find the effective capacitance of processor, C_{eff}. With the idle power of each frequency/voltage pair and the effective capacitance, the load lines can be drawn by using the equation.

In the case that the power model cannot be generated (or creation of such a model proves to be too difficult), lowering the CPU frequency/voltage to exploit slack time as much as possible is still an effective way for saving energy (even though it may not be the optimal approach).

Moreover, using the power model for determining the running frequency/voltage pairs is costly (in terms of time). It may be possible to adapt this technique for online DVS scheduling but the time overhead is an issue that would need to be considered.

6.9 Conclusions

As discussed in the beginning of this chapter, a TTC scheduler is not always suitable and – in a range of designs (including a wide range of control and condition-monitoring applications) a TTH scheduler is more appropriate. The 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-emptive task. In programming terms, the difference between a TTC and TTH design is very small: however, the possibility of pre-emption leads to substantial changes in system

behaviour, and makes it significantly more difficult to incorporate a DVS algorithm in low-jitter systems.

The solution – the TTH-jDVS algorithm, presented here – is a novel voltage scheduling based on a "power model", constructed in advance (from empirical measurements) and giving data which is used, before the system begins executing, to determine an appropriate task schedule. The effectiveness of this algorithm using both an artificial task set and in a more realistic case study has been demonstrated. Overall, in the case study, the TTH-jDVS algorithm was found to reduce power consumption to a level of about 70% of that seen for the "raw" TTH algorithm, while keep levels of task jitter at minimal levels.

Chapter 7

Further reductions of jitter in TTC/DVS scheduler

In Chapter 5 and Chapter 6, the DVS algorithms which minimised both energy consumption and jitter level were presented. However, using these algorithms, jitter could not be completely eliminated. In this chapter, techniques intended to further reduce jitter are explored: these techniques are based on the use of an independent (hardware) timer unit.[17].

7.1 The need for additional hardware

As has been seen in previous chapters, the designer faces some significant challenges, if precise control of system timing is an important consideration. This is because – in modern, general-purpose processors – the CPU core and "peripherals" (such as a timer, UART, analog-to-digital converter, CAN module, etc) are tightly integrated onto a single chip, in order to maximize performance and minimize cost[18]. In almost all cases, the CPU core and peripherals share a common clock source which is expected to remain largely fixed as the device operates. In the event of high-

---

17 Part of this chapter also in Phatrapornnant, T. and Pont, M.J. (submitted b).

18 This problem is not restricted to a small number of obscure processors. Advanced modern processors/microcontrollers (such as those in the STMicroelectronics STR71x and Philips LPC 2xxx families) provide an on-chip timer linked to the processor clock.

frequency changes to this clock source (as occurs when DVS[19]-based techniques are employed) it becomes very difficult to maintain fixed timing in peripheral components (such as timers), with the consequence that some level of jitter in task timing is unavoidable. In previous chapters, they demonstrate that, while careful software design can greatly reduce the impact of DVS on system task timing, it cannot eliminate such effects completely.

This chapter describes how use of an independent timer unit can eliminate jitter in TTC-DVS designs.

7.2 Previous work on hardware support for DVS

In support of DVS-based techniques, various "hardware" solutions have previously been developed. These include special power supplies and clock generators (see, for example, ispPAC (Lattice, 2003) and ispClock (Lattice, 2004)). Similarly, Adaptive Voltage Scaling (AVS) approaches use a 2-wire communication interface between the ARM processor and the energy management unit (the PowerWise Interface (Hartman and Dhar, 2004)), to adjust the power supply voltage according the load requirements. Such approaches can provide effective support for DVS and related techniques: however – unlike the simple solution proposed here – they do not tackle the underlying timing problems.

7.3 Including an independent timer in a TTC-jDVS2 design

Although the TTC-jDVS and TTC-jDVS2 algorithms (see Appendix C) have good levels of performance, it is clear that software alone cannot completely eliminate jitter in an embedded system in which the timer clock frequency is varied.

One solution is to employ an independent timer (with its own fixed-frequency clock source) to drive the system scheduler. Such a solution can be easily implemented using standard microprocessors or microcontrollers, or in system-on-chip designs.

---

[19] Dynamic Frequency Scaling involves changing the operating frequency of a processor only (supply voltage of CPU core is fixed).

For example, consider a standard "8254", 16-bit programmable interval timer / counter (Intersil, 2005). An 8254 contains three fully-independent counters which have six operation modes. For use with in conjunction with what it will be called the "TTC-jtDVS2" algorithm, the 8254 is used to generate clock tick (scheduling tick) and jitter guardian signals ("Clock_Tick" and "JGuard" respectively). The process is illustrated in Figure 7.1.

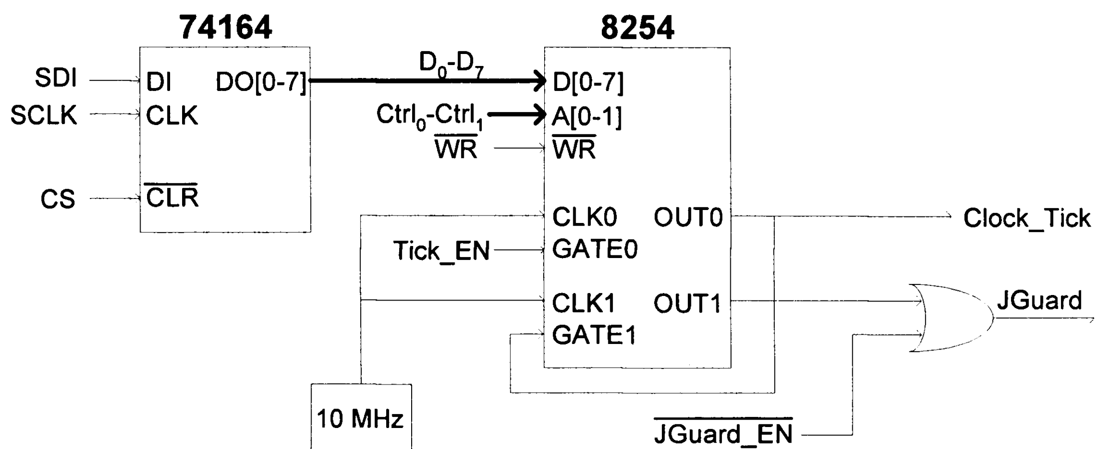

Figure 7.1: Using a COTS timer

To generate the clock tick, Counter 0 of the 8254 is programmed in Mode 2 (Rate Generator) which is typically used to generate real-time clock interrupts. An output will initially be high. When the initial count has decremented to 1, the output goes low (for one clock pulse) and then goes high again. The initial count is then reloaded, and (in Mode 2), this process is repeated indefinitely. The Counter is enabled if GATE is high and disabled if it is low.

To generate the jitter guardian signal, Counter 1 of the 8254 is programmed to run in Mode 5 (Hardware Triggered Strobe). An output signal is initially high. Counting is triggered by a rising of GATE1 which is connected to "Clock_Tick" in order to create a delay related to the clock tick (see Figure 7.1). When the delay count has expired, the output of the Counter 1 goes low for one clock pulse and then goes high again. The output is OR-ed with "JGuard_EN" which is used to control the "JGuard" (see Figure 7.1). "Clock_Tick" and "JGuard" are connected to external interrupts of the system processor.

In the present study, the data were sent serially from the system processor via an SPI bus and then converted by means of an 8-bit serial-in, parallel-out shift register (74164). Other controlling signals were generated using GPIO pins on the system processor. The frequency of the oscillator that feeds to the timer / counter was 10 MHz.

Please note that the behaviour of the 8254 can also be reproduced very easily in an SoC design (see Appendix D).

7.4 Assessing the modified algorithms

In this section, it presents the results obtained from an assessment of the TTC-jDVS2 and TTC-jtDVS2 algorithms in this section. The hardware platform used in this section is identical to that described in Section 4.1.

7.4.1 Impact on jitter

The impact of the modified algorithms on jitter behaviour was first explored.

7.4.1.1 Tick jitter

In order to measure tick jitter, a task was run using a random clock speed (from 10 to 60 MHz) using TTC-jDVS, TTC-jDVS2 and TTC-jtDVS2. This task was also run with a TTC at a fixed speed of 60 MHz. In each case the required tick interval was set at 1 ms, and the actual tick intervals were measured. Please note that the tick measurement in this study was probed at the interrupt signal generated by a timer. The measurements were made using identical equipments that described in Section 5.4.1.1.

Table 7.1 provides detailed results from this comparison, based on an analysis of 10,000 samples in each case. Please note that the tick compensation schemes of TTC-jDVS and TTC-jDVS2 are identical, and that the tick-compensation values used in this study were recalculated for use with the modified algorithms. Please also note that no measurable jitter was obtained for TTC-jtDVS2 and TTC.

Table 7.1: Comparing tick jitter from the TTC, TTC-jDVS, TTC-jDVS2 and TTC-jtDVS2 algorithms

	Jitter (μs)		
	Max	**Min**	**Total**
TTC	0.0	0.0	0.0
TTC-jDVS	0.2	-0.2	0.4
TTC-jDVS2	0.2	-0.2	0.4
TTC-jtDVS2	0.0	0.0	0.0

7.4.1.2 Task jitter

To explore the impact of variable speed on task release jitter, the experimental setup carried out in this section is identical to that described in Section 5.4.1.2

The interval between start times of the RJT were measured: the results in Figure 7.2 show that TTC-jtDVS2 (which uses the independent timer) has zero release jitter, except when running at 10 MHz, at which point the jitter was in region of 0.1 μs. By contrast, the TTC-jDVS2 algorithm has jitter in the range of +/- 0.4 μs over the speed range.

Both TTC-jtDVS2 and TTC-jDVS2 have lower release jitter than TTC-jDVS.

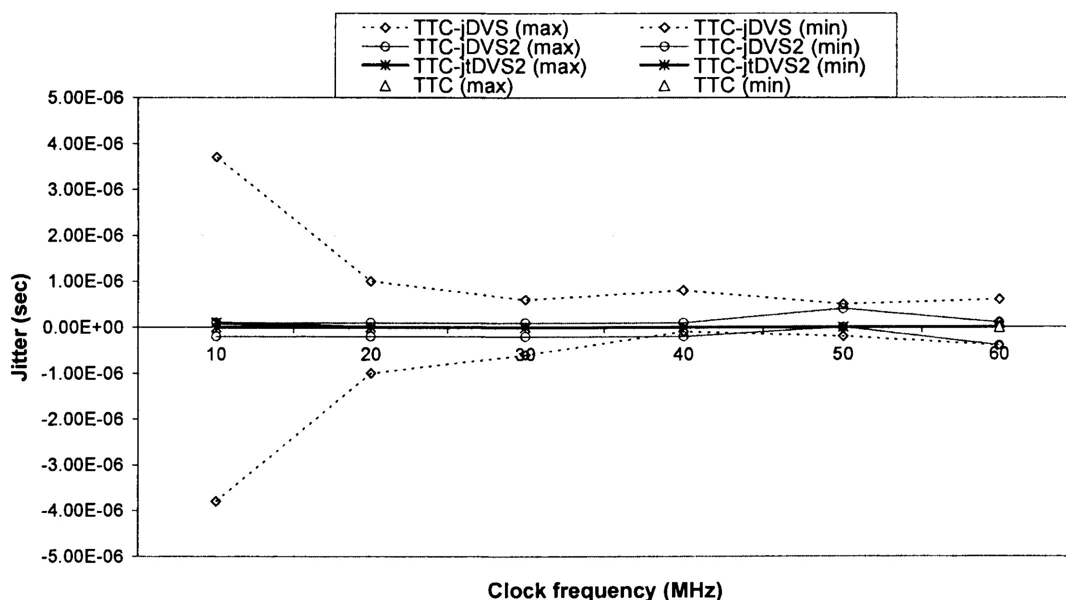

Figure 7.2: Minimum and maximum jitter level (from RJTs) at speeds 10-60 MHz, from TTC-jDVS, TTC-jDVS2, TTC-jtDVS2 and TTC (at 60MHz) algorithms

7.4.2 Assessing the impact of the independent timer on CPU power consumption

To compare the power consumption, 5 dummy tasks (Task A to Task E) were created which utilised between 10% and 90% of the available CPU activity (when run at the highest speed) in the tick interval. Each dummy task was implemented using a simple loop, adapted to give the required duration. In order to begin to assess the power-saving ability of these algorithms, schedulers with 1 ms tick intervals were used again (TTC, TTC-jDVS, TTC-jDVS2 and TTC-jtDVS2).

Note that, in any TTC design, a designer will usually wish to ensure that the WCET of each task is less than the scheduler tick interval: in this case, the duration of all tasks was less than 1 ms. Note also that Task A was viewed as an RJT (where appropriate). This task was run every 2 ms in (all systems). The remaining tasks (not RJTs) were run every 8 ms with different offsets.

To perform this empirical comparison, the task sets were executed using TTC, TTC-jDVS, TTC-jDVS2 and TTC-jtDVS2 algorithms. The average power consumption of the CPU core was measured in each case and is shown in Figure 7.3.

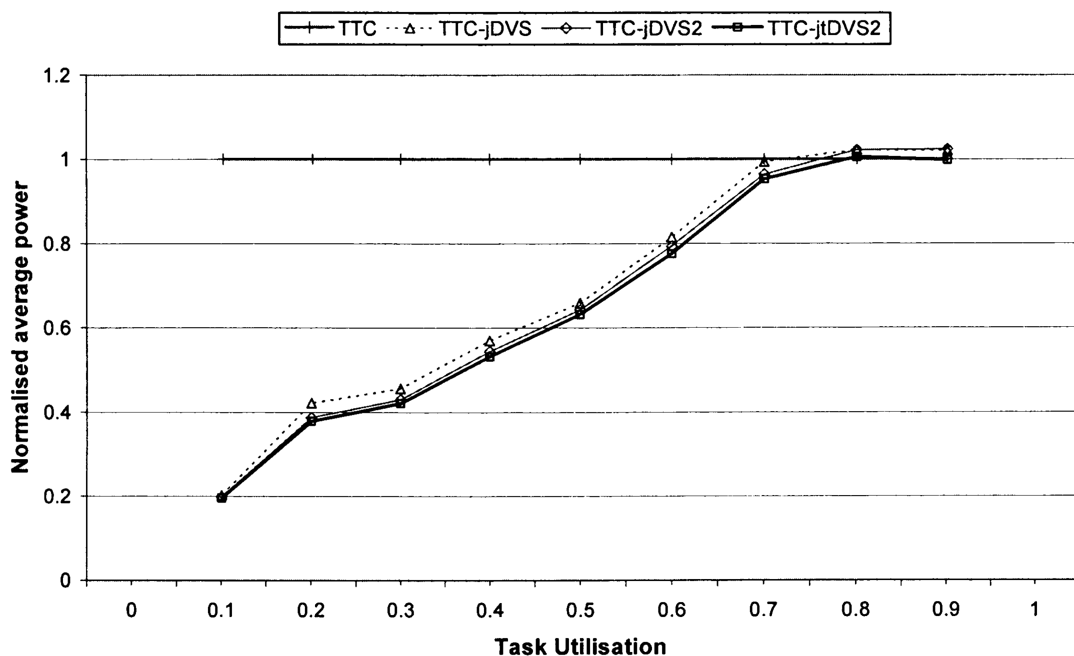

Figure 7.3: Power consumption comparison of scheduling algorithms at different load

Using the modified jitter guardian in TTC-jDVS2 is seen to result in a slight reduction in power consumption when compared with TTC-jDVS. A further power reduction is achieved using TTC-jtDVS2 (where the tick-compensation process is not required).

Note that, at high loads, TTC-jDVS and TTC-jDVS2 can exceed the power consumption of the TTC algorithm. In these circumstances, the power consumption of TTC-jtDVS2 is very close to that of the original TTC algorithm.

7.4.3 Impact on code complexity

The use of independent timer has an impact on the code size, especially in the frequency-scaling procedure (which is run whenever the processor's speed is changed). In this case, when using TTC-jDVS2, the code size is 173 lines (excluding comments). When employing TTC-jtDVS2, the code size is 27 lines (again excluding comment lines). Please see Listing 7.1 and Listing 7.2 for details.

```
begin FREQ_SCALE (New_freq_MULTIPLIER):

      set PLLCFG_Register = New_freq_MULTIPLIER ;

      disconnect PLL to run with raw frequency ;
      update PLL_registers with feed sequence ;

      reset Timer0_Prescaler = 0 ;
      reset Timer0_Prescaler_Counter = 0 ;

      start Timer1_Counter ;

      call function VOLTAGE_SCALE (New_freq_MULTIPLIER) ;

      while !(PLL_locked)
            update PLL_registers with feed sequence ;
      end while

      while !(Timer1_IntFlag_of_VoltageDelay)
            do nothing ;
      end while
      reset Timer1_IntFlag_of_VoltageDelay = 0 ;

      connect PLL ;
      update PLL_registers with feed sequence ;
      set Timer0_Prescaler = New_freq_MULTIPLIER ;

end FREQ_SCALE
```

Listing 7.1: Pseudo code of Frequency scaling function of TTC-jtDVS2

```
begin FREQ_SCALE_w_compensation (New_freq_MULTIPLIER):

      Set Current_freq_MULTIPLIER = PLLCFG_register ;
      set PLLCFG_Register = New_freq_MULTIPLIER ;

      disconnect PLL to run with raw frequency ;
      update PLL_registers with feed sequence ;

      reset Timer0_Prescaler = 0 ;
      reset Timer0_Prescaler_Counter = 0 ;

      start Timer1_Counter ;

      call function VOLTAGE_SCALE(New_freq_MULTIPLIER) ;

      switch (Current_freq_MULTIPLIER)
         case 6x:  if  New_freq_MULTIPLIER == 1
                       Timer0_Counter = Timer0_Counter + 6x_to_1x_compensation ;
                       break
                   end if
                   if  New_freq_MULTIPLIER == 2
                       Timer0_Counter = Timer0_Counter + 6x_to_2x_compensation ;
                       break
                   end if
                   if  New_freq_MULTIPLIER == 3
                       Timer0_Counter = Timer0_Counter + 6x_to_3x_compensation ;
                       break
                   end if
                   if  New_freq_MULTIPLIER == 4
                       Timer0_Counter = Timer0_Counter + 6x_to_4x_compensation ;
                       break
                   end if
                   if  New_freq_MULTIPLIER == 5
                       Timer0_Counter = Timer0_Counter + 6x_to_5x_compensation ;
                       break
                   end if
                   break
         case 5x:     .
                      .
                      .
           .       break
           .
           .
         case 1x:        .
                         .
                         .
                   break
      end switch

      while !(PLL_locked)
            update PLL_registers with feed sequence ;
      end while

      while !(Timer1_IntFlag_of_VoltageDelay)
            do nothing ;
      end while
      reset Timer1_IntFlag_of_VoltageDelay = 0 ;

      connect PLL ;
      update PLL_registers with feed sequence ;
      set Timer0_Prescaler = New_freq_MULTIPLIER ;

end FREQ_SCALE_w_compensation
```

Listing 7.2: Pseudo code of Frequency scaling function of TTC-jDVS2 with tick compensation

Overall, the total (executable) code sizes of TTC-jDVS2 and TTC-jtDVS2 are about 40,868 bytes and 39,923 bytes, respectively. That is, the use of an independent timer reduces the code size by approximately 1Kbyte.

7.5 Conclusions

This chapter has described the "TTC-jtDVS2" algorithm, which employs an independent timer unit to reduce the jitter below measurable levels in most cases. This hardware enhancement was also seen to further reduce both power consumption and software complexity when compared with the algorithms presented in previous chapters of this thesis.

Chapter 8

Discussion and Conclusions

In this concluding chapter, the work presented in this thesis is summarised and evaluated. Suggestions for future work are also made.

8.1 Introduction

The work described in this thesis is concerned with the development of energy reduction techniques for use in low-cost embedded systems. More specifically, it is focused on ways to implement DVS techniques in order to minimise CPU energy consumption in embedded systems which employ a time-triggered software architecture.

As has been noted throughout this document, DVS reduces CPU energy consumption in a process which involves variations in the CPU clock frequency: these frequency changes can lead to jitter. The work described in this thesis has developed and assessed various techniques (involving software and hardware) for reducing the impact of DVS on system jitter.

This chapter reviews some of the key contributions from the studies presented in this documented and discusses the extent to which the initial aims of the thesis were achieved. In addition, a few suggestions for future work in this area are made.

8.2 Knock-on impact of DVS implementation

The fundamental aim of the work described in this document was to minimise jitter in low-cost embedded systems which employed DVS. In order to study the impact of DVS implementation, the work began by exploring ways in which DVS could be applied in a TTC scheduler (which has a small overhead and low-jitter characteristics).

In practice, DVS processes require a comparatively large amount of time for supply-voltage switching and speed-finding calculations. These processes become part of the scheduler overhead and are one of the causes of task jitter.

During the research, attempts were made to minimise this scheduler overhead. To this end, a number of DVS algorithms (CD, LT, CA and CS) were developed and evaluated. The results obtained suggest that the algorithm which has small overhead can assist in power reduction: this is not only because a smaller algorithm itself consumes less power as it is executed, it is also because there is more time left to run the system tasks at lower speed settings. In the study, it was found that the CS algorithm (subsequently referred to as "TTC-DVS"), algorithm was the most effective at reducing power consumption.

8.3 Minimising jitter in DVS systems

The focus of the work then shifted to jitter reduction. The work in this area is summarised in this section.

8.3.1 Causes of jitter

Three causes of jitter in TTC / DVS designs were identified.

First, it is found that a scheduling tick generated by a hardware timer which is connected together with CPU clock can be disrupted when it is adjusted to match with clock frequency scaled. The results show that the scheduling tick in the DVS system significantly suffers jitter.

Second, the overheads from DVS processes are varied when frequency and voltage scaling is performed and this then causes release jitter. Even though the empirical studies employed a low-jitter scheduler, the results show that the jitter levels are large because the overheads are large.

Third, it is also found that the variation of operating clock frequencies changes execution time of tasks and this then causes the operating point inside a task (e.g. sampling points) to suffer jitter.

8.3.2 Software algorithms

In Chapter 4 and Chapter 6, the jitter impact of DVS implementation in TTC and TTH schedulers (respectively) was addressed using software algorithms.

In Chapter 5, a three-step jitter reduction scheme was developed to minimise jitter levels in TTC applications employing DVS: this algorithm was called TTC-jDVS. This involved: (i) tick compensation, (ii) jitter guardian insertion, and (iii) fixed speed of jitter-sensitive task. The effectiveness of the TTC-jDVS was demonstrated both on a data set made up of artificial tasks, and in a more realistic case study. The results show that the TTC-jDVS can significantly reduce jitter levels close to the TTC algorithm and also save CPU power when it is compared to the original TTC algorithm. Note that a slightly improved version of the TTC-jDVS algorithm ("TTC-jDVS2") is also presented in this document (in Appendix C).

In Chapter 6, a TTH-jDVS algorithm was developed, for use with "hybrid" schedulers. This algorithm was based on a power model. The TTH-jDVS algorithm has also been demonstrated on both an artificial task set and in a more realistic case study. The results has been proved that it can reduce power consumption to a level of that seen for the original TTH algorithm and also keep levels of task jitter at minimal levels.

8.3.3 Hardware support

The studies in Chapter 5 and Chapter 6 demonstrated that – while task jitter arising from DVS can be significantly reduced by using software algorithm, it could not be completely eliminated.

In Chapter 7, a hardware solution was introduced. This solution employed an independent timer to drive the system scheduler: this avoided the need to re-program the timer whenever the clock frequency was scaled. The resulting algorithm was shown to further improve both jitter performance and power consumption.

Overall, the results suggest that using hardware support can improve jitter performance of the scheduling algorithms. The use of hardware also reduced software complexity which resulted in a further power reduction.

8.4 Limitations of the work

Although it has been demonstrated that the TTC-jDVS / TTC-jDVS2 and TTH-jDVS algorithms were effective in reducing both jitter and power consumption, the work has a few limitations.

First, because they are based on a time-triggered software architecture, the TTC-jDVSx and TTH-jDVS algorithms support only periodic task basis. CPU speeds stored in the Circular array, which aimed to reduce overhead of DVS processes, were pre-calculated for periodic tasks only, and the speed settings were fixed throughout the system operation.

Another limitation of the TTC-jDVSx and TTH-jDVS algorithms is that they have been designed to support only one reduced-jitter task. In case of the TTH-jDVS algorithm, this limit is fundamental, and a consuequence of the scheduler architecture. However, with the TTC-jDVSx algorithm, it would be possible to support multiple reduced-jitter tasks by inserting another jitter guardian and fixing the running speed of the additional reduced-jitter task.

It should also be noted that the TTC-jDVSx and TTH-jDVS are all based on a priori knowledge: in particular, knowledge of WCET is required in order to determine appropriate CPU speeds. A consequence is that the algorithms cannot adapt to situations in which tasks execute for times less than the WCET. One of some previous works on run-time voltage hopping (Lee and Sakurai, 2000), describes ways in which energy can be saved when the task's execution time is less than the predicted WCET.

Note, however, that there is a risk that incorporating such "enhancements" to the present algorithms would be counter productive because of the increase system overhead which would result from having additional voltage-switching operations.

Finally, it should be noted that the use of DVS in TTC designs may have an impact on failure-detection mechanisms. In TTC designs, task overruns (or processor overloads) can be detected by observing that the system is executing a task – that is, it is not idle – when the timer interrupt occurs (Bate, 1998; Hughes and Pont, 2004). After applying DVS, it is possible that tasks that are executed with a lower speed can be stretched out over their minor cycle. Thus, mechanisms for detecting task overruns in TTC designs may need to be modified.

8.5 Potential applications of the work

As demonstrated and discussed previously, the impact of DVS can greatly degrade the accuracy of sampled data. The techniques proposed in this thesis have potential applications in signal-sampling and signal-processing systems which require both accuracy of data and also require long battery life. One of the most promising examples is medical monitoring applications, including wearable computing. In these applications, sampling rates of signal acquisition are relatively low (from a few Hertz to a few hundreds of Hertz), but reliability must be high (Raskovic *et al.*, 2004). Furthermore, such applications require high accuracy of data in order to support subsequent processing and data analysis, e.g. detecting asphyxia by using heart rate and heart rate variability (HRV) processed from ECG data (Boardman *et al.*, 2002).

8.6 Future work

In penultimate section of this thesis, some suggestions for the further studies are discussed.

8.6.1 Improvements to the TTC-jDVSx and TTH-jDVS algorithms

Although the TTC-jDVSx and TTH-jDVS algorithms described in this thesis have been shown to be effective in minimizing jitter and power consumption in low-cost embedded systems, there is still room for further improvements in both areas.

The current version of the TTC-jDVSx and TTH-jDVS algorithms are based on the assumption that all tasks consume the same amount of energy. In fact, each task consists of various instructions which relate to the system requirements. Previous studies have shown that different instructions consume different power: e.g. in the Intel StrongARM SA1100, *Store* typically consumes more power than *Branch* (Sinha and Chandrakasan, 2001). Please note that power consumption of chips also depends on the process technology (Jan *et al.*, 2005). As a consequence, it may be possible to improve the performance of the algorithms by weighting the tasks according to their relative power-consumption values.

Measurements of power consumption for each task could be done at the compilation stage and then used as a pre-defined parameter. However, this method may not obtain the actual power consumption (for example, in situations where tasks have different inputs). A more advanced algorithm could measure the power consumption of each task online and feed the measured power back to the scheduler, in order to give precise information for making decisions about speed settings.

In order to further improve jitter performance, hardware will be required. With the simplicity of the TTC scheduler, a hardware translation can be created (e.g. see (Hughes and Pont, 2005)). Using parallel processing in hardware can improve performance. For example, in DVS-based systems, scheduling overhead arising from software complexity has a significant impact on release jitter: however, an SoC design with dedicated scheduler hardware – and with support for multiple jitter guardians – should be able to release tasks at precise times, without interference from voltage scaling. Furthermore, it would be possible to design the jitter-sensitive part to run independently, in order to further reduce the complexity of the software algorithm.

8.6.2 Maximising the use of DC-DC converter

The development of advanced power-reduction techniques might be pointless if the design of the power supply itself is not considered.

Generally, systems employing DVS techniques require DC-DC converters to vary supply the voltage on the fly. In practice, the use of a DC-DC converter must be

considered carefully, switching losses[20] in such converters may not be significant at full load but, with light loads and under standby conditions, the switching losses can be a serious source of power loss (Arbetter *et al.*, 1995; Erickson and Maksimovic, 1995). Systems employing DVS impose a wide range of loads and – unless care is taken – power saved by DVS might be wasted on the DC-DC converter. It would therefore be interesting to explore a DVS algorithm that took into account the DC-DC converter characteristics when determining the CPU speed.

Another (potentially very simple) improvement could be lowered supply voltage when the system is idle. Algorithms based on TTC and TTH schedulers will be set to idle mode after all scheduled tasks have been executed. However, during the idle mode, the voltage level will not generally be altered. It would be straightforward to ensure that the supply voltage was set to the lowest level after entering the idle mode, in order to reduce the leakage power (see Equation 2.8). Such techniques could be compared with power gating (Abdollahi and Pedram, 2006), as discussed in Section 2.3.4.

8.7 Conclusions

The project described in this thesis has made three major contributions to this field. First, it has developed and assessed novel DVS scheduling algorithms with low levels of task jitter (TTC-jDVS, TTC-jDVS2). Second, it has developed and assessed a novel algorithm (TTH-jDVS) as an alternative solution for use in situations where a TTC scheduler is not appropriate. Third, it has developed and assessed hardware-based techniques in order to further reduce jitter levels and energy consumption. This thesis has also discussed in many more interesting issues and also provided the necessary inspiration for further research in this important area.

[20] The total switching power loss is equal to total energy lost during the turn-on and turn-off of switching transistors, multiplied by the switching frequency (Erickson and Maksimovic, 1995).

APPENDICES

Appendix A

Jitter model

This appendix gives a fundamental background of jitter which can be categorised in various types.

A.1 The jitter model

Typically, the total jitter can be divided into two categorises which are random jitter and deterministic jitter. The deterministic jitter is further divided into three categories: periodic jitter, data-dependent jitter, and bounded uncorrelated jitter. The characteristics for these jitter types are described as the following (Wavecrest, 2001; Ou *et al.*, 2004).

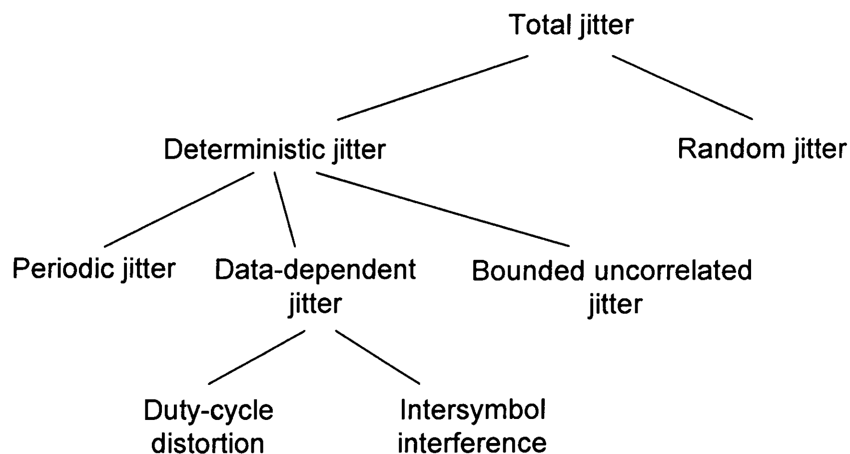

Figure A.1: Hierarchy of jitter

A.1.1 Random jitter

Random jitter arises from device noise source which has no discernable pattern. Thus, it cannot be predicted. Generally, device noise comes from short noise, related to a transistor's fluctuation in the current flow, thermal noise, caused by higher temperature resulting in greater atom vibration, collision and then electron scattering, and flicker noise, resulting from the random capture and emission of carriers from oxide interface traps that affects carrier density in a transistor (Hajimiri and Lee, 1998; Ou *et al.*, 2004). Random jitter is commonly modelled by the Gaussian distribution (bell curve) according to the distribution of thermal noise (Tektronix, 2002).

A.1.2 Deterministic jitter (Overview)

On the contrary to random jitter, deterministic jitter is repeatable and predictable because the peak-to-peak value of this jitter is bounded and the bounds can usually be observed or predicted based on a reasonably low number of observations (Tektronix, 2002). It primary causes arise from the interference of other systems, such as electro-magnetic interference, crosstalk (Bogatin and Garat, 2004). On an oscilloscope, random jitter gives a clean picture of a repetitive time-domain waveform (even slightly distorted) but deterministic jitter is the difference in time between the actual time and ideal time for each transition in the repetitive sequence (Johnson, 2002).

Depending on the noise sources, deterministic jitter is subcategorized as periodic jitter, data-dependent jitter and bounded uncorrelated jitter. An overview of each of these forms of jitter is presented in the sections which follow.

A.1.3 Deterministic jitter - Periodic jitter

Periodic jitter (or sinusoidal jitter) refers to variations of signal edge position over time. In general, it is caused by external noise sources (e.g. power supplies coupling, a strong local RF carrier) into a system (Tektronix, 2002).

A.1.4 Deterministic jitter - Data-dependent jitter

Data-dependent jitter occurs by the previous transmitted data symbols. It depends on the bit pattern transmitted on the link and does not describe the jitter induced by crosstalk resulting from adjacent signal paths. The major causes of problem come

from other devices and systems (Wavecrest, 2001; Buckwalter *et al.*, 2004; Ou *et al.*, 2004). Data-dependent jitter is divided into two subcategories as the follows:

Duty-cycle distortion: described a jitter amounting to a signal having asymmetrical pulse widths of rising and falling edges. The causes of the jitter come from the slew rates for the rising edges and falling edges which are different, and the decision threshold for a waveform which is inappropriate (Tektronix, 2002).

Intersymbol interference: depends on transmitted bit patterns (referred to the timing of the edge of transmitted signal). It causes mainly depend on bandwidth limitation of the transmission medium, the nonlinear phase response of transmission media, and reflection from imperfect transmission line terminations (Ou *et al.*, 2004).

A.1.5 Deterministic jitter - Bounded uncorrelated jitter

Bounded uncorrelated jitter is typically due to coupling from adjacent data-carrying links or on-chip random logic switching. It is bounded because of the finite coupling strength (Ou *et al.*, 2004).

Appendix B

Real-time scheduling architectures

This appendix gives an overview of real-time scheduling architectures which are relevant to the work in this thesis .

B.1 Introduction

Real-time systems are computer systems in which the correctness of the system behaviour depends not only on the correct results of the computations, but also on the time at which these results are produced (Kopetz, 1997). A reaction that occurs too late could be useless or even hazardous. A good example is the airbag deployment system in automobiles which late deployment defeats the entire purpose of airbag protection (Cooling, 2003). Furthermore, predictability is one of the important objective of a real time system (Stankovic and Ramamritham, 1990), i.e. the ability to determine whether the system is capable of meeting all its timing requirements, such as, message latency, protocol processing delays, and access to shared resources involved in communication. If the critical time constraints that are designed cannot be met when system is running, the consequences of a failure can sometimes be catastrophic (Pont, 2001). For example, a prototype of a fly-by-wire fighter plane responded to the pilot's commands too slow (missed timing requirements), so it then caused the plane crashed (Neumann, 1995)

Today, real-time systems involves in various applications, such as chemical and nuclear plant control, railway switching systems, automotive applications, flight control systems, telecommunication systems, industrial automation, robotics, environmental acquisition and monitoring systems, multimedia systems, and so on (Buttazzo, 2004).

B.2 A basic concept of real-time system

Generally, embedded systems are widely associated with real-time applications which are required responsiveness and predictable behaviour. The Oxford dictionary of computing gives the definition of real-time systems as: "Any system in which the time at which the output is produced is significant. This is usually because the input corresponds to some movement in the physical world, and the output has to relate to that same movement. The lag from input time to output time must be sufficiently small for acceptable timeliness" (Daintith, 2004).

Real-time systems consist of tasks which are executed by a CPU in a sequential manner and a task is a computation that is constructed from program codes (Marti, 2002). In fact, the application may have two or more tasks that are requested to execute at the same time. The CPU time has to be shared the processing time for executing the concurrent tasks in order to meet individual task's timing constraints. Thus, to execute such tasks according to predefined timing criterion, real-time system requires a scheduling policy and scheduler to manage task execution orderly. The detail of scheduling will be described later in Section B.3.

B.2.1 Timing constraints

In real-time systems, computational activities must be completed within stringent timing constraints in order to achieve the desired behaviour. An important timing constraint on a task is the deadline, which is the time that a task should complete its execution without causing any damage to the system (Liu, 2000; Buttazzo, 2004). Usually, tasks can be characterised according to their criticality, depending on the consequence of missed deadline, as hard real-time tasks and soft real-time tasks (Bennett, 1994; Kopetz, 1997; Marti, 2002; Buttazzo, 2004). If a task misses a given deadline that is critical for the system operation and may cause catastrophic

consequences, it is considered to be hard. If a task misses the time constraints that are required but it does not cause any serious damage, it is then considered to be soft.

B.2.2 Type of task

A task is a sequential program (e.g. in a control system, a task may consist of single or multiple processes: signal acquisition at the beginning, position calculation in the middle, and motor controlling at the end) that is activated for execution by the occurrence of a particular event (Heath, 1997; Kopetz, 1997). Typically, a task is commonly referred to a process or one part of a process which is a more complex computational activity (Buttazzo, 2004). Real-time tasks are also categorised by its regularity of activation as the following:

- **Periodic task**

Periodic tasks are executing periodically with a specific time and all future release times of a periodic task are known a prior. They are commonly found in activities such as sensory acquisition or control loops, which require accurate rates to insure system stability. Thus, periodic tasks often associate with hard deadlines that must be met under all anticipated circumstances (Spuri *et al.*, 1995; Pont, 2001).

- **Aperiodic task**

Aperiodic is call for tasks which have irregular activations or arrival times. Aperiodic tasks are typically employed for random processing requirements, such as operator requests or displaying activities. They are usually implemented for the less critical activities that have soft deadlines or no deadline at all (Spuri and Buttazzo, 1996; Nolte, 2003).

- **Sporadic task**

The activation times of sporadic tasks are not known a priori but a minimum interval time between any two activation times of sporadic tasks are known. Typically, sporadic tasks are associated with event-driven processing, such as a response to use inputs or non-periodic device interrupts. Their activations occur repeatedly but the time interval between consecutive occurrences is not constant as periodic tasks and can vary arbitrarily large (Jeffay *et al.*, 1991; Nolte, 2003).

B.3 A real-time scheduler

In embedded systems, real-time schedulers are commonly employed to manage such applications (Cooling, 2003). A scheduler is comparable to an operating system (OS) in a desktop PC but it is a very simple operating environment for embedded applications. Generally, the desktop PC does not require an OS but it is for flexibility to run many thousands of different applications. The desktop OSs provide the common code (for printing, file storage, display, sound, and so on) that is required by application programs (Pont, 2001). Specifically, OS support has a major impact on software dependability, productivity and maintainability. Whereas embedded systems are designed for specific applications, run on small and dedicated hardware and often require real-time properties (e.g. responsiveness, reliability), real-time schedulers are then the reasonable way (Cooling, 2003).

To execute tasks in order to meet timing constraints, a scheduling algorithm (a set of rules) is used to determine a queue or schedule in which tasks will be executed (Buttazzo, 2004). For example, if there is a high-priority task is requested to execute, the scheduling algorithm will make decision, based on the policy that is prior set, to interrupt the running task for executing the high-priority task first, or execute the high-priority task after the running task completed. Otherwise, the scheduling algorithm may make a decision by executing the task that will meet its deadline first for guaranteeing that no task misses its deadline. Each scheduling algorithm has advantages and disadvantages by itself. Designers have to select the appropriate scheduling algorithm for their applications.

Among the grate variety of scheduling algorithms, they can be classified into these main types which are described as the following.

B.3.1 Event-triggered versus Time-triggered

A trigger is a temporal control signal or an event to activate some tasks in a node (e.g. data transmission or analogue-to-digital signal conversion) (Kopetz, 1997). In real-time scheduling systems, the triggering mechanisms can be divided into two different approaches: event-triggered and time-triggered (Kopetz, 1991; Bennett, 1994; Pont, 2001).

In the event-triggered system, all tasks are activated in response to significant external incidents (e.g. the depressing of a push button by a user, the activation of a limit switch, or the arrival of a new message at a node) which can take place at any time (Kopetz, 1997). In practice, an event-trigger signal can be obtain from the sensor for detecting environmental activities and presents to the system in form of interrupt signal. To deal with unpredictable input of dynamic environment, the event-triggered systems require a dynamic scheduling strategy to activate the task in order to response to the event. The main advantages of event-triggered systems are an immediate response, flexibility, and suitability for sporadic data (Kopetz, 1997). However, the event-triggered systems have more overhead which can have an opportunity to fail under heavy load conditions (Marti, 2002).

In a time-triggered system, all tasks are activated at specific time instants which are know a priori before the system starts execution. Generally, a hardware timer can be used to set to expire periodically as a periodic clock interrupt (Bennett, 1994)while, in a distributed real-time system, all time-triggered nodes may use clocks which are synchronised to form a global notion of time (Kopetz, 1997; Pont, 2001). According to schedule tasks at each time instant, scheduling overhead of time-triggered systems during run-time can be minimised. Furthermore, an advantage of the time-triggered systems is easy to validate, test, and certify because the times related to the tasks are deterministic (Liu, 2000). However, the time-triggered systems are proper to implement in static environments which all system activities must be known in the design phase and require careful planning during the design phase (Kopetz, 1991).

B.3.2 Co-operative versus Pre-emptive scheduling

In real-time scheduling systems, there are two different scheduling strategies: co-operative scheduling and pre-emptive scheduling (Liu, 2000; Pont, 2001). They are described as the following.

In co-operative (also known as non-preemptive) scheduling, tasks cooperate with each other to relinquish control of the CPU, in other words, a task which is currently executing will not be interrupted until its process is completed (Pont, 2001). It is simple and straightforward to implement because each task can be executed to completion without interruption from another task. Thus, a co-operative scheduler

requires little necessity to protect shared data and concern of corruption by another task. Moreover, another its advantage is that interrupt latency is typically low, as no unexpected context switches, this, therefore, makes the scheduling overhead low (Locke, 1992; Labrosse, 1998). However, it is recommended that co-operative scheduling is suitable in a system where many short tasks are executed because a co-operative scheduler cannot switch to any task while the long task is being executed and this then result in poor responsiveness. The shortest guaranteed responsiveness of co-operative scheduling in single processor systems is the sum of the longest and the shortest task execution time (Kopetz, 1997).

In pre-emptive scheduling, the highest priority task ready to run is always given control of CPU time. When a higher priority is ready to run, the currently running task is suspended (pre-empted) and the system must save its state (e.g. program counter and register contents) for resuming the pre-empted task from that state. Then, higher priority task is executed until its completion and the pre-empted task is resumed (Labrosse, 1998; Liu, 2000). Accordingly, pre-emptive scheduler always requires high memory allocation for temporarily saving a context switch (the content of the CPU status register and program counter) every time of pre-emption. The system overhead (context-switch time), including memory usage, will increase if there are many levels of preemption taking place. Moreover, the pre-emptive scheduling require complicated algorithm to avoid resource conflicts and blocking (Kopetz, 1997). However, in term of responsiveness, a significant event can immediately react by interrupt the running task to invoke the scheduler to make a new scheduling decision.

B.3.3 Static versus Dynamic scheduling

Among various types of real-time scheduling architecture, scheduling algorithms can be categorized by determining when the scheduling decisions are taken. They are divided into static and dynamic scheduling (Liu, 2000).

In static scheduling, all the task set parameters, such as execution times, deadlines, periods and so on, have been known at the design stage. A task scheduling decisions are computed at pre-runtime and the entire schedule is stored in the table for use at runtime (Stankovic *et al.*, 1995; Kopetz, 1997; Nolte, 2003). If the operation mode of the system is changed, the new schedule is needed to re-compute. The advantage of

static scheduling is predictable because all activities in the system are deterministic. Moreover, the runtime scheduling overhead is simple and low that is suitable for limited resource systems. However, the static scheduling has drawback that all activities have to be planned before startup. It then lacks of flexibility to adapt itself to new environments (Liu, 2000). Off-line scheduling is often equated static scheduling by misunderstanding. For analysis, off-line scheduling should always be done regardless of whether the final runtime algorithm is static or dynamic in building any real-time systems (Stankovic *et al.*, 1995).

In dynamic scheduling (or online scheduling), the scheduling decisions are made at runtime. A dynamic scheduler computes each scheduling without knowledge of new arrival tasks that may occur in the future – the parameters of each task become known to the dynamic scheduler only when a task arrives at runtime (Stankovic *et al.*, 1995; Kopetz, 1997). These schedulers are flexible and suitable for the systems which need to interact with evolving environment or the future load is unpredictable (Liu, 2000). Even dynamic schedulers are flexible and adaptive but they prone to have high scheduling overhead because the complicated scheduling algorithm must be processed at runtime. In addition, it is also difficult to predict system's behaviors of dynamic scheduling under diverse conditions of external environments.

B.3.4 Fixed-priority versus Dynamic-priority

An online scheduler is a priority-driven scheduler which schedules tasks according to some priority-driven algorithm. It assigns priorities to tasks at each scheduling decision time and places the tasks in a ready task queue in priority order (Liu, 2000). Both fixed and dynamic priority algorithms are different from each other in how priorities are assigned to tasks.

A fixed-priority algorithm assigns the same priority to all instances in each task. In other words, the priority of each task is fixed relative to other tasks during the application's execution (Labrosse, 1998; Liu, 2000). A well-known fixed priority algorithm is *rate-monotonic* (RM) algorithm(Liu and Layland, 1973; Bate, 1998). In order to define task priority, RM algorithm assigns priorities to tasks based on their periods. Then, the task which have shortest period will be assigned highest priority (Buttazzo, 2005). For example, task A which run at higher rate than task B has higher

priority than task B. Task A is always scheduled and executed immediately when it is released and the priorities of tasks are not changed all time of their execution.

In contrast, a dynamic-priority algorithm assigns different priorities to the individual task instances in each task. For this reason, the priority of the task with respect to that of the other tasks can change when the task instances are released (Liu, 2000). The *earliest-deadline-first* (EDF) is a well-known dynamic algorithm. The EDF algorithm dynamically assigns priorities to individual task instances according to their absolute deadlines (Pedreiras and Almeida, 2002; Buttazzo, 2005). The highest priority will be assigned to the task which will reach its deadline first. At each scheduling decision, the scheduler has to calculate the time to the deadline of each task and reorder the priorities of tasks in a queue. Hence, in EDF, the priorities of tasks are kept changing all the time.

Appendix C

The TTC-jDVS2 algorithm

This appendix describes a simple modification to the original TTC-jDVS algorithm which results in an improvement in jitter behaviour and a reduction in energy consumption: the revised algorithm will be referred here as TTC-jDVS2.

C.1 The TTC-jDVS2 scheduling algorithm

In this study, the aim was to improve the performance of the TTC-jDVS algorithm (described in Chapter 5) by modifying the "jitter guardian" function. In the TTC-jDVS algorithm, the jitter guardian is a sandwich delay which is executed before the reduced-jitter (RJ) tasks are released: this delay is generated by setting a hardware timer to expire with a constant delay time. In the TTC-jDVS algorithm, it keeps polling the timer flag during the delay period: when the flag is set, the RJ tasks will be then released.

In the TTC-jDVS2 algorithm (see Listing C.1), the performance is slightly improved by placing the CPU in idle mode during the jitter-guardian delay period. The CPU will then be wakened when the timer expires, and the RJ task will be released. By incorporating the idle-time mode change, it both saves power and reduce the level of release jitter (see (Maaita and Pont, 2005) for further details of the underlying mechanism).

```
begin DISPATCH_TASKS:

    if Circular_Array_Pointer > MAJOR_CYCLE then
       reset Circular_Array_Pointer ;
    end if

    while Tick_Count > 0
          for all tasks in tasks array
              if -- Delay == 0 then
                 load Task_Speed from circular array ;
                 if Previous_Speed != Task_Speed then
                    scale frequency and voltage (and perform tick compensation) ;
                 end if
                 if task is an RJT then
                    enable Timer ;
                    sleep ;
                    disable Timer ;
                 end if
                 release Task;
                 if task is periodic then
                    reload Delay = Period ;
                 else
                    delete Task from array ;
                 end if
              end if
          end for
          Tick_Count -- ;

    end while
    sleep ;

end DISPATCH_TASKS
```

Listing C.1: Dispatcher function in TTC-jDVS2

Compared to the original Dispatcher function in the TTC-jDVS algorithm (see Listing 5.2), TTC-jDVS2 modifies the section that inserts the jitter guardian. The improvement to TTC-jDVS2 is obtained by employing idle mode instead of a delay (see Table C.1). With this method, the timer interrupt will wake the processor up when the timer counter expires. The task is then executed.

Table C.1: The code of jitter guardian insertion

TTC-jDVS	TTC-jDVS2

`While (TIMER <= JGuard);`	`enable TIMER;`
	`SCH_Go_To_Sleep();`
	`disable TIMER;`

C.2 Assessing the modified algorithms

In this section, it presents the results obtained from an assessment of the TTC-jDVS2 and original TTC-jDVS algorithms on both jitter and power consumption. The hardware platform used in this section is identical to that described in Section 4.1.

C.2.1 Impact on jitter

The impact of the modified algorithms on jitter behaviour was first explored.

C.2.1.1 Tick jitter

In order to measure tick jitter, a task was run using a random clock speed (from 10 to 60 MHz) using TTC-jDVS and TTC-jDVS2. In each case the required tick interval was set at 1 ms, and the actual tick intervals were measured. Please note that the tick measurement in this study was probed at the interrupt signal generated by a timer. The measurement setup used in this section is identical to that described in Section 5.4.1.1.

Table C.2 provides detailed results from this comparison, based on an analysis of 10,000 samples in each case. The tick compensation schemes of TTC-jDVS and TTC-jDVS2 are identical. Therefore, they both gave the same levels of tick jitter. Please note that the tick-compensation values used in this study were recalculated for use with the modified algorithms.

Table C.2: Comparing tick jitter from the TTC-jDVS and TTC-jDVS2 algorithms

	Jitter (μs)		
	Max	**Min**	**Total**
TTC-jDVS	0.2	-0.2	0.4
TTC-jDVS2	0.2	-0.2	0.4

C.2.1.2 Task jitter

To explore the impact of variable speed on task release jitter, the experimental setup carried out in this section is identical to that described in Section 5.4.1.2

The interval between start times of the RJT were measured: the results in Figure C.1 show that TTC-jDVS2 presents lower release jitter than TTC-jDVS. By contrast, the

TTC-jDVS2 algorithm has jitter in the range of +/- 0.4 μs over the speed range while the original TTC-jDVS typically has greater levels of jitter – especially at 10 MHz, it is close the range of +/- 4 μs.

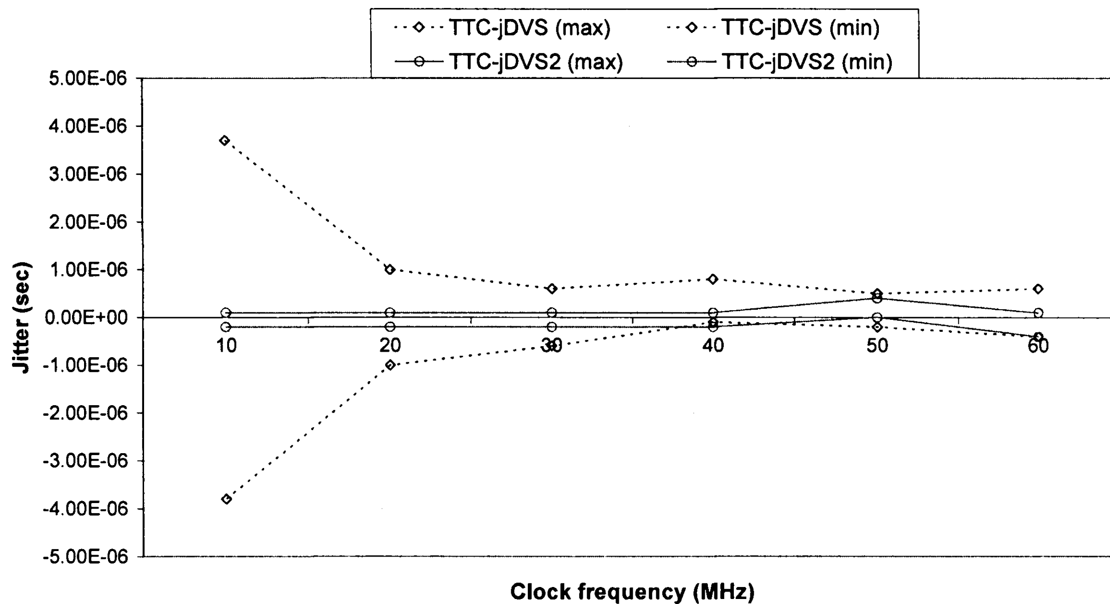

Figure C.1: Minimum and maximum jitter level (from RJTs) at speeds 10-60 MHz, from TTC-jDVS and TTC-jDVS2algorithms

C.2.2 Assessing the impact of the independent timer on CPU power consumption

To compare the power consumption, it is again used the experimental setup identical to that described in Section 5.4.2 and rerun.

To perform this empirical comparison, the task sets were executed using TTC, TTC-jDVS and TTC-jDVS2 algorithms. The average power consumption of the CPU core was measured in each case and is shown in Figure C.2.

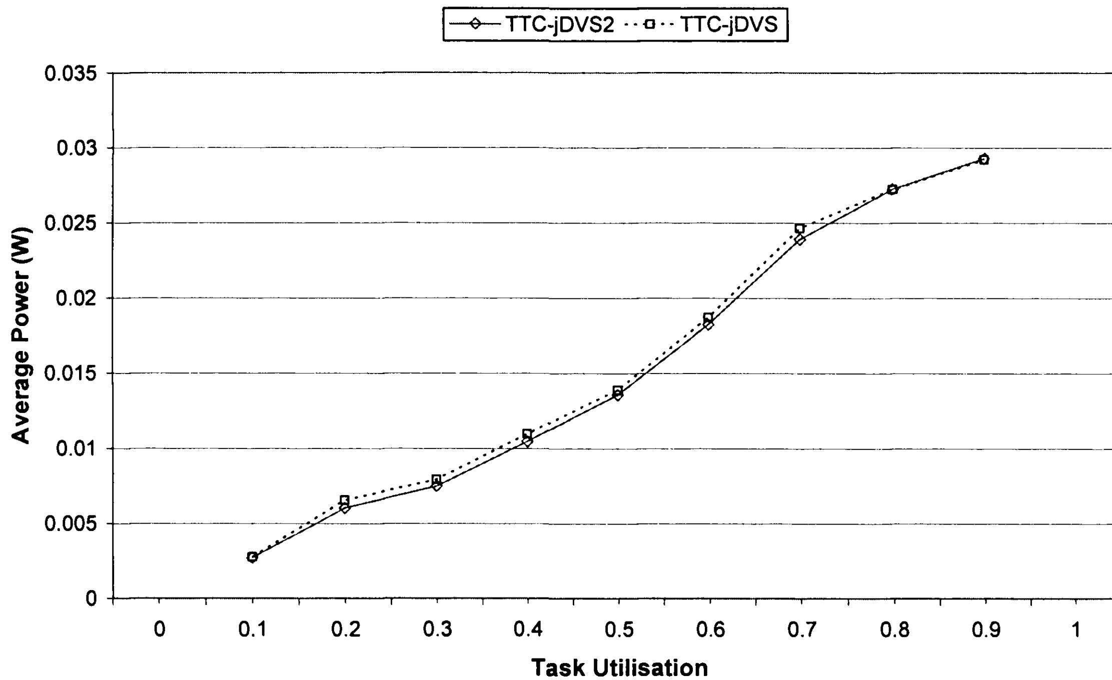

Figure C.2: Power consumption comparison of TTC-jDVS and modified TTC-jDVS algorithms at different load

Using the modified jitter guardian in TTC-jDVS2 is seen to result in a slight reduction in power consumption when compared with TTC-jDVS.

C.3 Conclusions

This appendix has described modifications to the "TTC-jDVS" algorithm. The modified "TTC-jDVS2" algorithm described here was seen to demonstrate small improvements in both jitter performance and power consumption when compared to TTC-jDVS.

Appendix D

Incorporating an independent timer in an FPGA-based SoC design

This appendix describes how an independent timer can be incorporated in an SoC design.

D.1 The hardware platform

The test platform employed in this study was based on a PH Processor (Hughes *et al.*, 2005). Briefly, this processor is a MIPS-based, 32-bit design with 32 registers and a 5-stage pipeline which is based on an outline provided by Patterson and Hennessy (Patterson and Hennessy, 2004). The processor also includes the system coprocessor CP0, to support precise exceptions.

The implementation of the PH processor used in these studies was created using VHDL with Xilinx ISE tools. The target was a Xilinx 200K gate (Spartan 3) FPGA chip on a Digilent Spartan 3 development board (Digilent, 2004). The version of the processor used in the present study had 216 KBits of block RAM on the chip and 1 MB of SRAM on-board. The board also contained a serial port, LEDs, seven-segment display, buttons and switches.

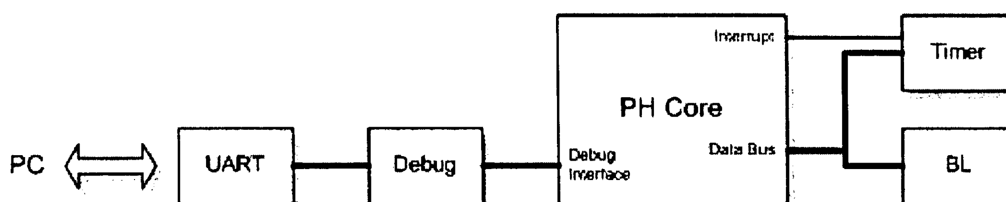

Figure D.1: Layout of the PH core and supporting systems

The design outline of the processor was set up on an FPGA as shown in Figure D.1. Implemented in this way, the PH core contains its own instruction and data memories and the device can be programmed and debugged through the serial UART, which connects to a PC-based debugger. The BL (Buttons and Lights) block uses memory mapped I/O to interface to the onboard LEDs and I/O pins. In the original PH Processor, the timer is attached on the data bus where the necessary registers are easily addressed through normal memory load and store instructions. The Core plus the additional Timer, UART, Debugger Control Unit and BL controller utilizes about 72% of 200K Spartan chip. This includes the large distributed RAM register file implementation, 2KB of instruction RAM and 8KB of data RAM, and has only tested running at a safe 25 MHz.

D.2 The jDFS Timer

The jDFS Timer separates the clock input which drives the timer unit (which it is assumed that will be used to drive the scheduler) from the input to the main CPU clock. The timer clock is (it is further assumed) driven by a fixed-frequency source. The structure of the jDFS Timer block diagram is shown in Figure D.2.

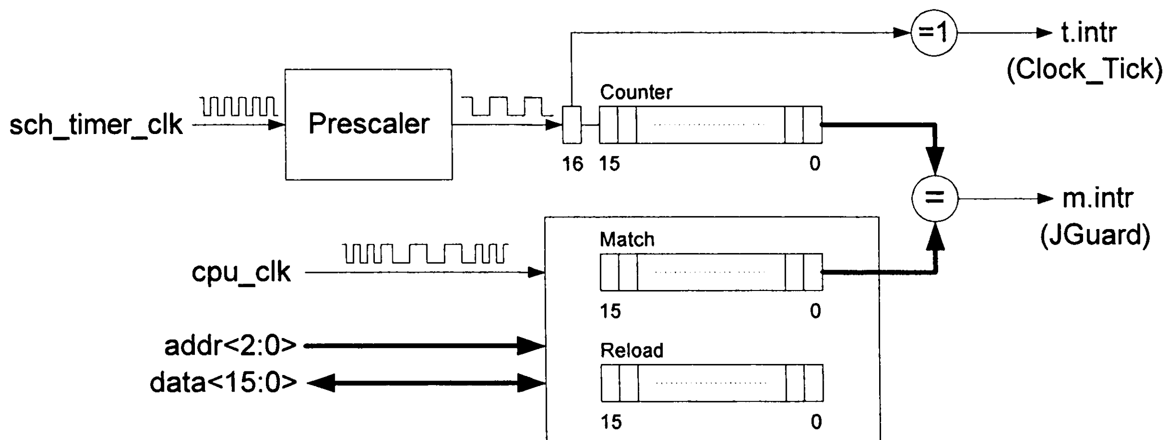

Figure D.2: The jDFS Timer block diagram

As an example, the jDFS Timer unit used in this study is a based on a 16-bit timer which operates in count-down mode. The input clock for the jDFS Timer is derived from the peripheral clock "sch_timer_clk", divided by a (programmable) prescaler. To initialize the jDFS Timer, it can be configured by input a count value into a Reload register and this value will then be loaded into the Counter automatically. If the timer is enabled, the Counter's value is decreased (in response to the sch_timer_clk clock signal) until the Counter underflows. Timer interrupt "t.intr" signal is then generated and the value in Reload register will be automatically reloaded into the Counter and 16th bit of Counter is cleared. This process will repeat until the timer is disabled (or power is removed).

Optionally, the jDFS Timer may be used to perform other actions at specified timer values, based on the use of a Match register. The specified time, which can provide for precise event control, can be configured by loading the time value into the Match register. When the Match register equals the timer's Counter, the match output interrupt "m.intr" is generated. Use of this facility will be demonstrated in later examples.

D.3 The TTC-jDFS scheduling algorithm

The aim of this study is to explore the implementation of the TTC-jDVS algorithm with independent timer in an FPGA-based SoC design. Generally, applying DVS on a commercial-off-the-shelf FPGA is not straight forward. As mention in the first implementation of DVS for COTS FPGAs (Chow *et al.*, 2005), the voltage controller is responsible for ensuring the voltage supply to the FPGA is not lowered so much that the FPGA ceases to operate properly, or it dose not meet the frequency requirements of the application. A Logic Delay Measurement Circuit (LDMC) has then proposed by (Chow *et al.*, 2005) to determine the critical delay time of the circuit while lowering supply voltage. However, The FPGA (Xilinx, 2005) does not support power supply partitioning between PH Core and peripheral devices in this implementation. If the DVS is applied, scaled voltage will affect the whole design, including the part intended to run with fixed voltage. Thus, in this study, it is mainly focused on jitter impact on the use of an independent timer with the algorithm without varying supply voltage.

Based on the TTC-jDVS algorithm (Phatrapornnant and Pont, 2006), the TTC-jDFS algorithm is a modified version which removes the voltage scaling process. TTC-jDFS still contains a timer-adjustment process to reduce tick jitter, jitter guardian insertion for minimizing release jitter, and a process for assigning speed of "reduced-jitter" tasks to reduce task duration variation. Without voltage scaling process, the overhead of TTC-jDFS is significantly small when comparing to TTC-jDVS.

D.4 Supporting frequency scaling on the PH processor

To explore the proposed technique, the additional hardware was added on PH processor to support Dynamic Frequency Scaling (DFS).

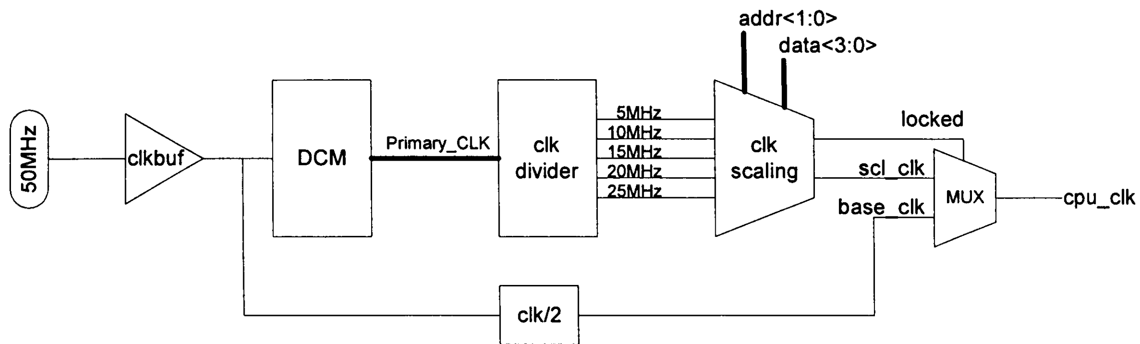

Figure D.3: Frequency scaling added on PH processor

The 50 MHz clock signal generated by an on-board oscillator is passed to a clock buffer to eliminate clock skew before it is supplied to a Digital Clock Manager (DCM) (Xilinx, 2005). The DCM which contains a digital frequency synthesizer component generates primary clock signal frequencies and then passes to a clock divider to generate the required clock signal frequencies, for example 5, 10, 15, 20, and 25 MHz in this study. The required clock outputs are multiplexed by a clk_scaling component to support variable frequency clock. The output signal "cpu_clk" is switched to base_clk, raw-based frequency clock, while a scaling clock "scl_clk" is changing frequency. When the frequency of scaling clock has changed, the locked signal will trigger the MUX to connect the scaling clock to the output. Please note: there is no PLL multiplier using for generated clock frequency required. Frequencies switch using multiplexer performs faster than using PLL multiplier in conventional processors.

D.5 Incorporating the jDFS Timer

The jDFS Timer described in Section D.2 is implemented together with the PH-processor core in FPGA development board (Digilent, 2004). Data and address signals of the jDFS Timer were directly connected to the data and address bus of the system. The sch_timer_clk of the jDFS Timer was connected to the peripheral clock bus which provided fixed frequency while cpu_clk of jDFS Timer was connected to scaling clock bus. Please note that a series of tests of TTC-jtDVS2 were run with the jDFS Timer described above while another algorithm used the original design timer which drive with scaling CPU clock only.

D.6 Assessing the impact of the jDFS Timer on jitter

To explore the impact of the jitter compensation algorithm comparing between a conventional and jDFS Timer, a series of tests using the platform described in this section above were conducted.

In these studies, system clocks of 5, 10, 15, 20 and 25 MHz were generated by DCM and used multiplexer to select the required clock frequency. However, to conduct the experiments in this section, the dynamic frequency scaling (DFS) was only performed, all supply voltages were fixed.

D.6.1 Tick jitter

In order to measure tick jitter, a task was run using a random clock speed (from 5 to 25 MHz) using TTC-DFS, TTC-jDFS with the conventional timer and TTC-jDFS with a jDFS Timer (TTC-jtDFS). This task was also run with a "standard" TTC architecture (Pont, 2001) at a fixed speed of 25 MHz. The measurements in this section are identical to those in Section 7.4.1.1.

The tick jitter measurement results in this section are similar to the results in Section 7.4.1.1. No measurable jitter was obtained for the TTC and TTC-jtDFS. The corresponding test run using the TTC-jtDFS shows a significant improvement comparing with TTC-DFS and TTC-jDFS using the conventional timer. However, the total tick drift range of TTC-DFS is 1 μs which is smaller than that of TTC-DVS, 4.9 μs. Figure D.5 and Figure D.6 illustrate the occurrence of tick jitter taken from 10,000 consecutive samples and Table D.1 provides detailed results from this comparison.

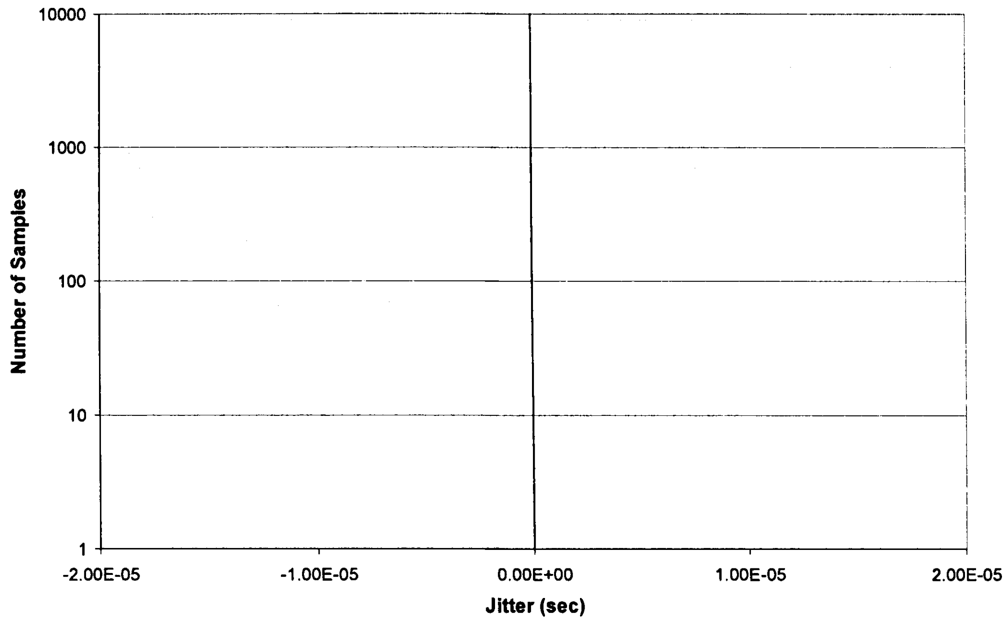

Figure D.4: Tick jitter in TTC (PH-processor)

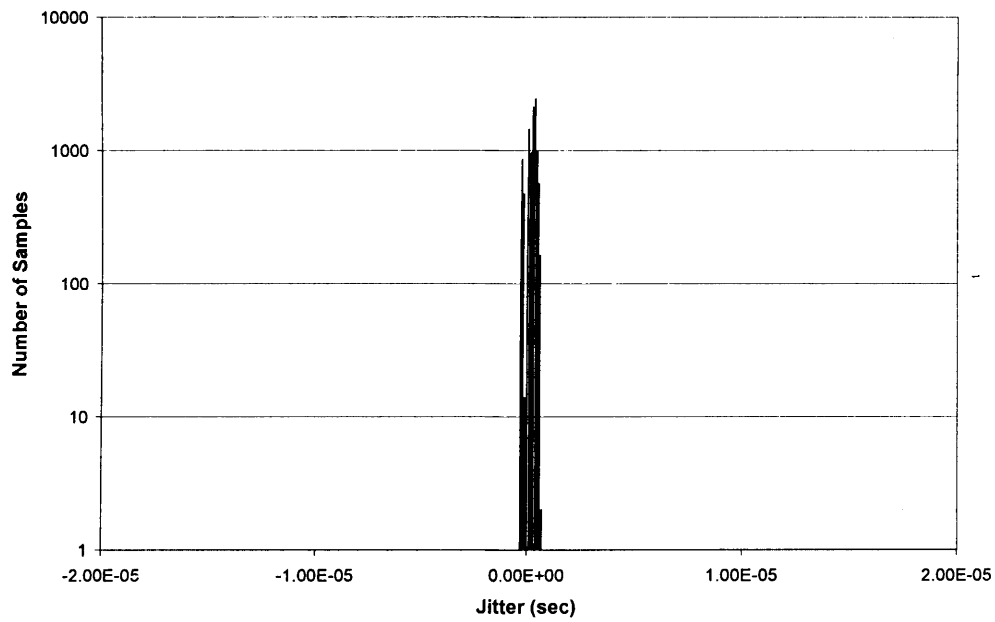

Figure D.5: Tick jitter in TTC-DFS (PH-processor)

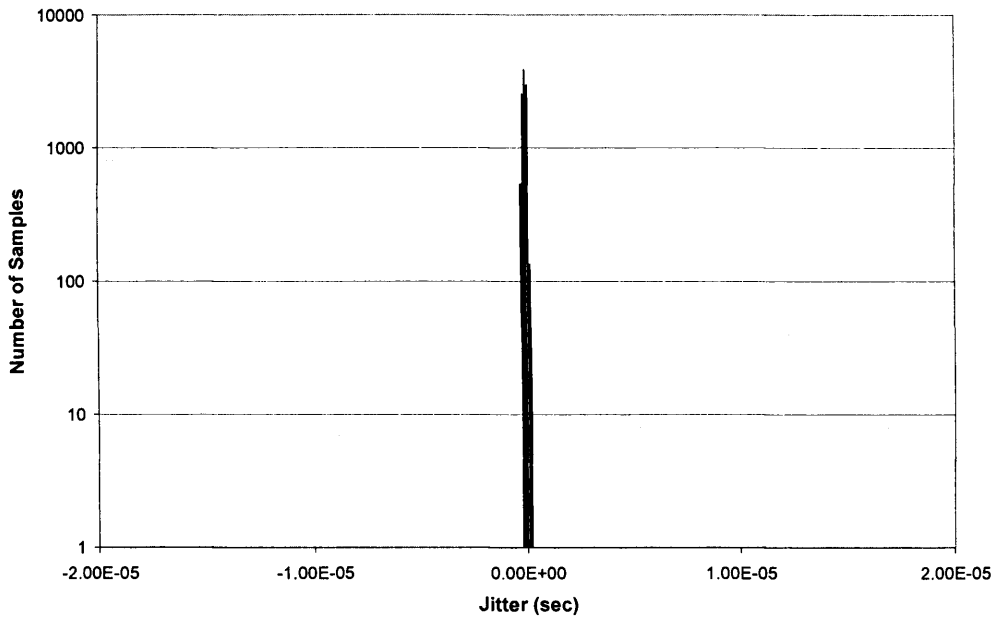

Figure D.6: Tick jitter in TTC-jDFS (PH-processor)

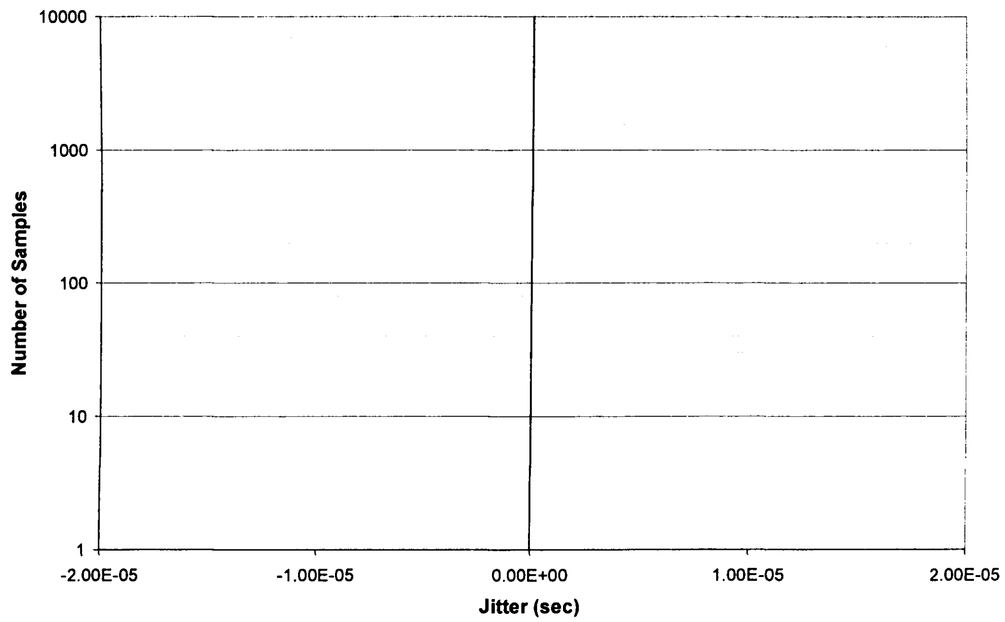

Figure D.7: Tick jitter in TTC-jtDFS (PH-processor)

Table D.1: Comparing tick jitter run by TTC-DFS, TTC-jDFS, TTC-jtDFS and TTC algorithms

	Jitter (μs)		
	Max	Min	Total
TTC	0.0	0.0	0.0
TTC-DFS	0.7	-0.3	1.0
TTC-jDFS	0.2	-0.2	0.4
TTC-jtDFS	0.0	0.0	0.0

D.6.2 Task jitter

To explore the impact of variable speed on task release jitter, the set up is identical to 7.3.1.2 but a random speed was changed to 5 - 25 MHz corresponding to the hardware setup in this section.

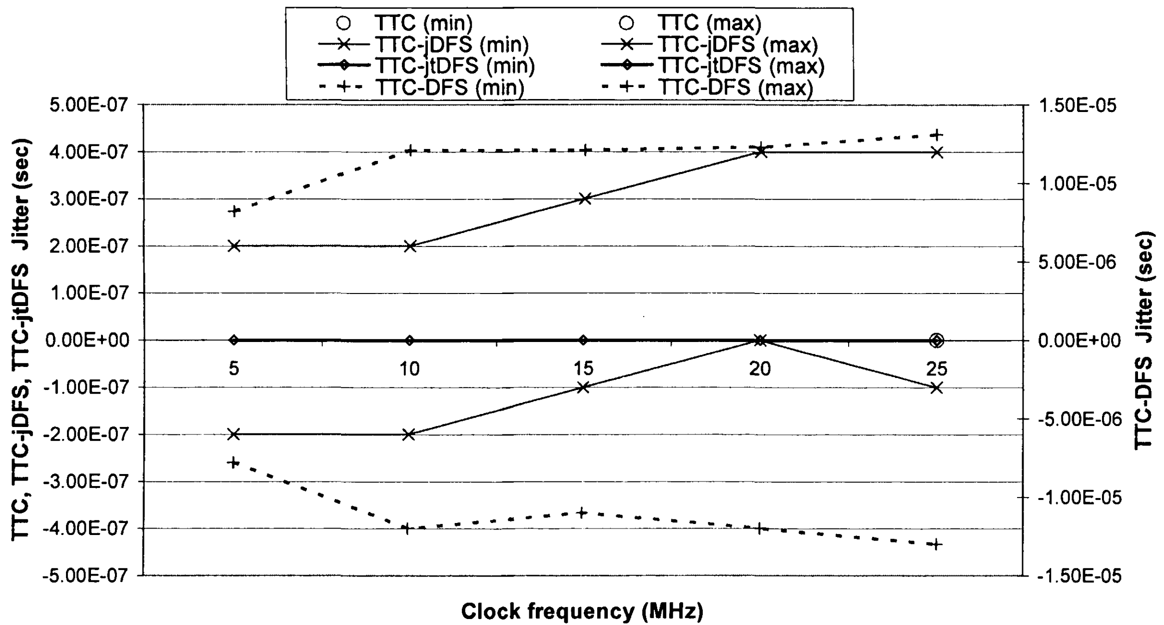

Figure D.8: Minimum and maximum jitter level of RJT at speed 5-25 MHz run by TTC, TTC-DFS, TTC-jDFS and TTC-jtDFS

The results measured the interval between start times of the RJT are shown in Figure D.8. They present the same trend as those shown in Section 7.4.1.2. There is no jitter in both TTC-jtDFS and TTC while TTC-jDFS and TTC-DFS obtain jitter in range of +/-0.4 μs and +/-14 μs respectively. The variation of release jitters in TTC-DFS is considerably less than those in TTC-DVS because no voltage scaling process is included in the TTC-DFS scheduling overhead.

D.7 Assessing the impact of the jDFS Timer on CPU power consumption

To assess the power-saving ability of the algorithms, 4 schedulers with 1 ms tick intervals (TTC, TTC-DFS, TTC-jDFS, TTC-jtDFS) were again used. The experimental setup in this section is identical to that in Section 7.4.2.

To measure power consumption, current and voltage supplied to the FPGA's V_{CCINT} voltage input (1.2 volts typical) were measured, which power the FPGA's core logic (Digilent, 2004). Figure D.9 presents power consumption comparison of the algorithms, TTC, TTC-DFS, TTC-jDFS and TTC-jtDFS.

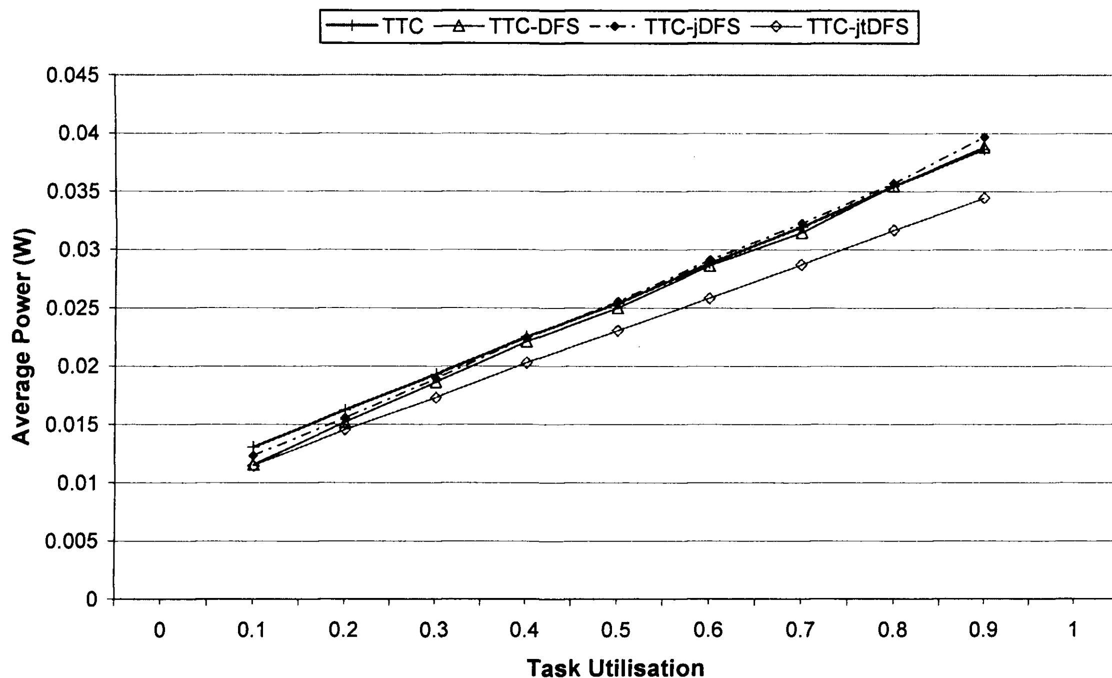

Figure D.9: Power consumption comparison of scheduling algorithms at different load

From the results, TTC, TTC-DFS and TTC-jDFS closely consume power. The TTC-DFS consume less power than TTC at all load while TTC-jDFS still does at load range up to 0.5U. TTC-jtDFS consumes the least power among those algorithms. However, the design of the jDFS Timer using in TTC-jtDFS differs from the typical design using in those algorithms whereas other parts of the design are identical.

D.8 Conclusions

This appendix has described a way to incorporate an independent timer in an SOC design, in order to reduce jitter in scheduler designs which incorporate dynamic frequency scaling. The results have shown that the jDFS Timer can reduce both tick and task jitter to be zero. It is believed that the use of independent timer with dynamic frequency scaling can also be incorporated into any SOC design.

References

Abdollahi, A. and Pedram, M.(2006), *"Power minimisation techniques at the RT-level and below".* In: Al-Hashimi, B. M. (Ed.) System-on-Chip: Next Generation Electronics, Vol. 18, The IEE, London, pp. 387-414.

Agilent (2000), *"Agilent 4352S VCO/PLL Signal Test System: Optimizing VCO/PLL Evaluations and PLL Synthesizer Designs, Application Note",* Agilent Technologies.

Agilent (2002), *"Understanding Jitter and Wander Measurements and Standards",* Agilent Technologies.

Agrawal, S. and Bhatt, P. (2001), *"Real-time embedded software systems: An introduction"*, Technology Review #2001-04, Tata Consultancy Services, August 2001.

Allworth, S. T. (1981), *"An Introduction to Real-Time Software Design",* Macmillan, London.

Andrei, A., Schmitz, M. T., Eles, P., Peng, Z. and Al-Hashimi, B. M. (2005), *"Quasi-static voltage scaling for energy minimization with time constraints",* Proceedings of the conference on Design, Automation and Test in Europe, Vol. 1, pp. 514-519.

Andrei, A., Eles, P., Peng, Z., Schmitz, M. T. and Al-Hashimi, B. M. (2007), *"Energy Optimization of Multiprocessor Systems on Chip By Voltage Selection",* IEEE Transactions on VLSI Systems.

Arbetter, B., Erickson, R. and Maksimovic, D. (1995), *"DC-DC Converter Design for Battery-Operated Systems",* IEEE Power Electronics Specialists Conference, pp. 103-109.

ARM (2003), *"Intelligent Energy Manager",* http://www.arm.com/pdfs/IEM%20Flyer%200171-1.pdf.

Ashling (2003), *"LPC2000 Evaluation and Development Kits datasheet",* Ashling Microsystems, http://www.ashling.com/pdf_datasheets/DS266-V7U-EvKit2000.pdf.

Assaderaghi, F., Sinitsky, D., Parke, S. A., Bokor, J., Ko, P. K. and Hu, C. (1997), *"Dynamic threshold-voltage MOSFET (DTMOS) for ultra-low voltage VLSI",* IEEE Transactions on Electron Devices, Vol.44 (3), pp. 414-422.

Atkinson, J. (1990), *"Reference: Jitter, Bits, & Sound Quality"*, Stereophile, Vol. 13 (12), December, 1990.

ATMEL (2003), *"Guidelines to keep ADC resolution within specification: 8051 Microcontrollers application note",* ATMEL.

Audsley, N., Burns, A., Richardson, M., Tindell, K. and Wellings, A. J. (1993a), *"Applying new scheduling theory to static priority pre-emptive scheduling"*, Software Engineering Journal, Vol.8 (5), pp. 284-292.

Audsley, N., Tindell, K. and Burns, A. (1993b), *"The end of the line for static cyclic scheduling?"* Proceedings of the 5th Euromicro Workshop on Real-time Systems, Finland, pp. 36-41.

Austin, T., Blaauw, D., Mudge, T. and Flautner, K. (2004), *"Making typical silicon matter with Razor"*, Computer, Vol. 37 (3), pp. 57-65.

Avoim (2002), *"White Paper: Managing Jitter, Wander, and Latency"*, Avoim, http://www.aviom.com/LibraryDocs/WhitePapers/WP202_Jitter_3.0.pdf.

Aydin, H., Melhem, R., Mosse, D. and Mejia-Alvarez, P. (2001), *"Dynamic and Aggressive Scheduling Techniques for Power-Aware Real-Time Systems"*, Proceedings of the 22nd IEEE Real-Time Systems Symposium, pp. 95-105.

Aydin, H., Devadas, V. and Zhu, D. (2006), *"System-Level Energy Management for Periodic Real-Time Tasks"*, Proceedings of the 27th IEEE International Real-Time Systems Symposium, pp. 313-322.

Aziz, P. M., Sorensen, H. V. and Spiegel, J. V. d. (1996), *"An overview of sigma-delta converters: How a 1-bit ADC achieves more than 16-bit resolution"*, IEEE Signal Processing Magazine, Vol. 13 (1), September 1996, pp. 61-84.

Bai, P., Auth, C., Balakrishnan, S., Bost, M., Brain, R., Chikarmane, V., Heussner, R., Hussein, M., Hwang, J., Ingerly, D., James, R., Jeong, J., Kenyon, C., Lee, E., Lee, S.-H., Lindert, N., Liu, M., Ma, Z., Marieb, T., Murthy, A., Nagisetty, R., Natarajan, S., Neirynck, J., Ott, A., Parker, C., Sebastian, J., Shaheed, R., Sivakumar, S., Steigerwald, J., Tyagi, S., Weber, C., Woolery, B., Yeoh, A., Zhang, K. and Bohr, M. (2004), *"A 65nm logic technology featuring 35nm gate lengths, enhanced channel strain, 8 Cu interconnect layers, low-k ILD and 0.57 /spl mu/m/sup 2/ SRAM cell"*, Electron Devices Meeting, 2004., pp. 657-660.

Baker, T. P. and Shaw, A. (1989), *"The cyclic executive model and Ada"*, Real-Time Systems, Vol.1 (1), pp. 7-25.

Ball, S. R. (1996), *"Embedded Microprocessor Systems: Real World Design"*, Newnes, USA.

Bambha, N. K., Bhattacharyya, S. S., Teich, J. and Zitzler, E. (2001), *"Hybrid global/local search strategies for dynamic voltage scaling in embedded multiprocessors"*, International Conference on Hardware Software Codesign, Copenhagen, Denmark, pp. 243-248.

Barreiros, J., Costa, E., Fonseca, J. and Coutinho, F. (2000), *"Jitter reduction in a real-time message transmission system usinggenetic algorithms"*, Proceedings of the 2000 Congress on Evolutionary Computation, CA, USA, Vol. 2, pp. 1095-1101.

Baruah, S., Buttazzo, G., Gorinsky, S. and Lipari, G. (1999), *"Scheduling periodic task systems to minimize output jitter",* International Conference on Real-Time Computing Systems and Applications (RTCSA '99), Hong Kong, pp. 62-69.

Bate, I. J. (1998), *"Scheduling and Timing Analysis for Safety Critical Real-Time Systems",* PhD thesis, Department of Computer Science, University of York.

Benini, L., Castelli, G., Macii, A., Macii, E. and Scarsi, R. (2000), *"Battery-Driven Dynamic Power Management of Portable Systems",* 13th International Symposium on System Synthesis (ISSS'00), pp. 25-30.

Benini, L., Macii, A., Macii, E. and Poncino, M. (2002), *"Discharge Current Steering for Battery Lifetime Optimization",* Proceedings of the 2002 International Symposium on Low Power Electronics and Design, pp. 118-123.

Bennett, S. (1994), *"Real-time Computer Control: An introduction",* 2nd ed, Prentice Hall.

Bilas, A., Fritts, J. and Singh, J. P. (1997), *"Real-Time Parallel MPEG-2 Decoding in Software",* Proceedings of the 11th International Parallel Processing Symposium, pp. 197-203.

Bini, E., Buttazzo, G. and Lipari, G. (2005), *"Speed Modulation in Energy-Aware Real-Time Systems",* Proceedings of the 17th Euromicro Conference on Real-Time Systems (ECRTS'05), pp. 3-10.

Boardman, A., Schlindwein, F., Thakor, N., Kimura, T. and Geocadin, R. G. (2002), *"Detection of asphyxia using heart rate variability",* Medical and Biological Engineering and Computing, Vol.40 (6), pp. 618-624.

Bogatin, E. and Garat, G. (2004), *"Analysis of board layout helps cure jitter problems"*, EDN, 5 August 2004, pp. 77-80.

Brannon, B. and Barlow, A. (2006), *"Aperture uncertainty and ADC system performance: Application note (AN-501)"*, Analog Devices.

Buckwalter, J., Analui, B. and Hajimiri, A. (2004), *"Predicting data-dependent jitter",* IEEE Transactions on Circuits and Systems II: Express Briefs, Vol.51 (9), pp. 453-457.

Burd, T. D. and Brodersen, R. W. (1995), *"Energy Efficient CMOS Microprocessor Design",* Proc. 28th Hawaii Int'l Conf. On System Sciences, Vol. 1, pp. 288-297.

Burd, T. D., Pering, T. A., Stratakos, A. J. and Brodersen, R. W. (2000), *"A Dynamic Voltage Scaled Microprocessor System",* IEEE J. Solid-State Circuits, Vol.35 (11), pp. 1571-1579.

Burns, A., Hayes, N. and Richardson, M. F. (1995), *"Generating feasible cyclic schedules",* Control Engineering and Practice, Vol.3 (2), pp. 151-162.

Buttazzo, G. C. (2004), *"Hard Real-time Computing Systems: Predictable Scheduling Algorithms and Applications"*, 2nd ed, Springer.

Buttazzo, G. C. (2005), *"Rate monotonic vs. EDF: Judgement day"*, Real-Time Systems, Vol.29 pp. 5-26.

Butts, J. A. and Sohi, G. S. (2000), *"A static power model for Architects"*, The 33rd Annual International Symposium on Microarchitecture (MICRO-33), pp. 191-201.

Cai, Y., Schmitz, M. T., Al-Hashimi, B. M. and Reddy, S. M. (2006), *"Workload-ahead-driven online energy minimization techniques for battery-powered embedded systems with time-constraints"*, ACM Transactions on Design Automation of Electronic Systems (TODAES), Vol.12 (5), pp. 1084-4309.

Cervin, A., Henriksson, D., Lincoln, B., Eker, J. and Arzen, K.-E. (2003), *"How does control timing affect performance? Analysis and simulation of timing using Jitterbug and TrueTime"*, IEEE Control Systems Magazine, Vol.23 (3), pp. 16-30.

Cervin, A., Lincoln, B., Eker, J., Arzen, K.-E. and Buttazzo, G. (2004), *"The Jitter Margin and Its Application in the Design of Real-Time Control Systems"*, the 10th International Conference on Real-Time and Embedded Computing Systems and Applications, Goteborg, Sweden.

Chandrakasan, A., Bowhill, W. J. and Fox, F. (2000), *"Design of High-Performance Microprocessor Circuits"*, Wiley-IEEE Press.

Chandrakasan, A. P., Sheng, S. and Brodersen, R. W. (1992), *"Low Power CMOS Digital Design"*, IEEE journal of Solid-State Circuits, Vol.27 (4), pp. 473-484.

Cheng, H. and Goddard, S. (2005), *"Integrated Device Scheduling and Processor Voltage Scaling for System-wide Energy Conservation"*, 2nd International Workshop on Power-Aware Real-Time Computing (PARC 05). pp. 24-29.

Chew, B. (2002), *"Dynamic voltage scaling conserves portable power"*, EDN, January 10, pp. 65-68.

Chiasserini, C. and Rao, R. (1999a), *"A Traffic Control Scheme to Optimize the Battery Pulsed Discharge"*, Proc. of Milcom'99, Atlantic City, NJ.

Chiasserini, C. and Rao, R. (1999b), *"A Model for Battery Pulsed Discharge with Recovery Effect"*, WCNC'99, New Orleans, USA.

Chiasserini, C. F. and Rao, R. R. (2000), *"Routing protocols to maximizebattery efficiency"*, Proceedings of IEEE Military Communications Conference (MILCOM'00), Los Angeles, CA, USA, pp. 496-500.

Choi, S., Cha, H. and Ha, R. (2006), *"A selective DVS technique based on battery residual"*, Microprocessors and Microsystems, Vol.30 (1), pp. 33-42.

Chow, C. T., Tsui, L. S. M., Leong, P. H. W., Luk, W. and Wilton, S. J. E. (2005), *"Dynamic Voltage Scaling for Commercial FPGAs"*, Proceedings of the 2005 IEEE International Conference on Field-Programmable Technology, Singapore, pp. 173-180.

Chuang, C.-N. and Liu, S.-I. (2006), *"A 1 V Phase Locked Loop with Leakage Compensation in 0.13 um CMOS Technology"*, IEICE Transactions on Electronics, Vol.E89-C (3), pp. 295-299.

Clark, D. D., Shenker, S. and Zhang, L. (1992), *"Supporting Real-Time Applications in an Integrated Services Packet Network: Architecture and Mechanism"*, Proceedings of the SIGCOMM '92, pp. 14-26.

Cooling, J. (2003), *"Software Engineering for Real-time Systems"*, Addison-Wessley.

Cottet, F. and David, L. (1999), *"A solution to the time jitter removal in deadline based scheduling of real-time applications"*, 5th IEEE Real-Time Technology and Applications Symposium - WIP, Vancouver, Canada, pp. 33-38.

Coutinho, F., Barreiros, J., Fonseca, J. and Costa, E. (2000), *"Jitter minimization with genetic algorithms"*, IEEE International Workshop on Factory Communication Systems, Porto, Portugal, pp. 267-273.

Daintith, J. (2004), *"Oxford: A Dictionary of Computing"*, 5th ed, Oxford University Press.

Dancy, A. and Chandrakasan, A. P. (1997), *"Ultra low power control circuits for PWM converters"*, Proceedings of the IEEE Power Electronics Specialists Conference, pp. 21-27.

Delmar-Reynolds (2002), *"7-Day Continuous ECG Recording System"*, Del Mar Reynolds, http://www.delmarreynolds.com/innovation.php?country=8&uid=34&page=1.

Dhar, S., Maksimovic, D. and Kranzen, B. (2002), *"Closed-loop adaptive voltage scaling controller for standard-cell ASICs"*, International Symposium on Low Power Electronics and Design, California, pp. 103-107.

Digilent (2004), *"Spartan 3 Board"*, Digilent., http://www.digilentinc.com/Data/Products/S3BOARD/S3BOARD-rm.pdf.

Doyle, M. (1995), *"Design and simulation of litium rechargeable batteries"*, Ph.D. dissertation, University of California at Berkeley.

Eisenbeis, C. and Windheiser, D. (1993), *"Optimal Software Pipelining In Presence Of Resource Constraints"*, Proceedings of the International Conference on Parallel Architecture and Compiler Techniques, Obninsk, Russia.

Ejlali, A., Al-Hashimi, B. M., Schmitz, M. T., Rosinger, P. and Miremadi, S. G. (2006), *"Combined time and information redundancy for SEU-tolerance in energy-efficient real-time systems"*, IEEE Transactions on Very Large Scale Integration (VLSI) Systems, Vol.14 (4), pp. 323-335.

Erickson, R. and Maksimovic, D. (1995), *"High Efficiency DC-DC Converters for Battery-Operated Systems with Energy Management"*, Worldwide Wireless Communications, Annual Reviews on Telecommunications.

Ernst, D., Das, S., Lee, S., Blaauw, D., Austin, T., Mudge, T., Kim, N. S. and Flautner, K. (2004), *"Razor: Circuit-Level Correction of Timing Errors for Low-Power Operation"*, IEEE Micro, Vol. 24 (6), pp. 10-20.

Flautner, K., Flynn, D. and Rives, M. (2003), *"A Combined Hardware-Software Approach for Low-Power SoCs: Applying Adaptive Voltage Scaling and Intelligent Energy Management Software"*, System-on-Chip and ASIC Design Conference (DesignCon 2003).

Fourre, R. (1993), *"Reference: Jitter & the Digital Interface"*, Stereophile, Vol. 16 (10), October, 1993.

Fuller, T. F., Doyle, M. and Newman, J. (1994), *"Relaxation Phenomena in Lithium-Ion-Insertion Cells"*, Journal of the Electrochemical Society, Vol.141 (4), pp. 982-990.

Gannett, E. K., Day, E. C., Carter, H. J., Greer, S., Martynec, S., Cole, L., Jacob, A., Schneider, A. and Stephen, M. (1972), *"IEEE Standard Dictionary of Electrical and Electronics Terms"*, John Wiley & Sons.

Gartner (2004), *"Statistics: Worldwide sales of Mobile Terminals to End Users, 2002-2003"*, Gartner, http://www.dataquest.com/press_gartner/quickstats/phone .html.

Ge, R., Feng, X. and Cameron, K. W. (2005), *"Performance-constrained Distributed DVS Scheduling for Scientific Applications on Power-aware Clusters"*, Proceedings of the 17th IEEE/ACM High Performance Computing, Networking and Storage Conference (SC'05), Seattle, USA.

Govil, K., Chan, E. and Wasserman, H. (1995), *"Comparing Algorithms for Dynamic Speed-Setting of a Low-Power CPU"*, Proceeding of the first ACM International Conference on Mobile Computing and Networking (MOBICOM 95), pp. 13-25.

Gruian, F. (2001), *"Hard real-time scheduling for low-energy using stochastic data and DVS processors"*, International Symposium on Low Power Electronics and Design, California, pp. 46-51.

Gruian, F. (2002), *"Energy-Centric Scheduling for Real-Time Systems"*, PhD thesis, Department of Computer Science, Lund University.

Hageman, S. (1993), *"Simple PSpice models let you simulate common battery types"*, EDN, Vol. October 28, 1993, pp. 117-129.

Hajimiri, A. and Lee, T. H. (1998), *"A pGeneral Theory of Phase Noise in Electrical Oscillators"*, IEEE Journal of Solid-State Circuits, Vol.33 (2), pp. 179-194.

Hampton, J. R. (1998), *"The ECG made easy"*, 5th ed, Churchill Livingstone.

Hartman, M. and Dhar, S. (2004), *"On-Chip Power Management Utilizing an Embedded Hardware Controller and a Low-Power Serial Interface"*, embedded world 2004 Conference.

Hazucha, P. and Svensson, C. (2000), *"Impact of CMOS technology scaling on the atmospheric neutron soft error rate"*, IEEE Transactions on Nuclear Science, Vol.47 (6), pp. 2586-2594.

Heath, S. (1997), *"Embedded Systems Design"*, Newnes.

Hollabaugh, C. (2002), *"Embedded Linux: Hardware, Software, and Interfacing"*, Addison Wesley.

Hsu, C.-H. and Kremer, U. (2002), *"Compiler-Directed Dynamic Voltage Scaling for Memory-Bound Applications"*, Technical Report DCS-TR-498, Department of Computer Science, Rutgers University, August 2002.

Huang, Z., Kurokawa, A., Inoue, Y. and Mao, a. J. (2005), *"A Novel Model for Computing the Effective Capacitance of CMOS Gates with Interconnect Loads"*, IEICE Transactions of Fundamentals of Electronics, Communication and Computer Sciences, Vol.E88-A (10), pp. 2562-2569.

Hughes, Z. H. and Pont, M. J. (2004), *"Design and test of a task guardian for use in TTCS embedded systems"*, Proceedings of the UK Embedded Forum 2004, Birmingham, UK, pp. 16-25.

Hughes, Z. M. and Pont, M. J. (2005), *"Time-triggered co-operative hardware scheduler"*, patent application filed (UK), 9 September 2005.

Hughes, Z. M., Pont, M. J. and Ong, H. L. R. (2005), *"A soft embedded core for use in university research and teaching"*, Proceedings of the 2nd UK Embedded Forum, Birmingham, UK, pp. 224-245.

Intel (1994), *"MCS51 Microcontroller Family User's Manual"*, Intel Corporation.

Intel (2001), *"Intel StrongARM SA-1110 Microprocessor: Developer's Manual"*, Intel, http://www.intel.com/design/strong/manuals/278240.htm.

Intel (2006a), *"Intel Itanium 2 Processor"*, Intel, http://www.intel.com/products /processor/itanium2/index.htm.

Intel (2006b), *"The Intel 4004: A testimonial from Federico Faggin, its designer, on the first microprocessor's thirtieth birthday."* Intel, http://www.intel4004.com/.

Intersil (2005), *"82C54 CMOS Programmable Interval Timer",* Intersil, http://www.intersil.com/data/fn/fn2970.pdf.

Ishihara, T. and Yasuura, H. (1998a), *"Voltage Scheduling Problem for Dynamically Variable Voltage Processors",* The International Symposium on Low Power Electronics and Design, pp. 197-202.

Ishihara, T. and Yasuura, H. (1998b), *"Programmable Power Management Architecture for Power Reduction",* IEICE Trans. Electronics, Vol.E81-C (9).

Jan, C.-H., Bai, P., Choi, J., Curello, G., Jacobs, S., Jeong, J., Johnson, K., Jones, D., Klopcic, S., Lin, J., Lindert, N., Lio, A., Natarajan, S., Neirynck, J., Packan, P., Park, J., Post, I., Patel, M., Ramey, S., Reese, P., Rockford, L., Roskowski, A., Sacks, G., Turkot, B., Wang, Y., Wei, L., Yip, J., Young, I., Zhang, K., Zhang, Y., Bohr, M. and Holt, B. (2005), *"A 65nm Ultra Low Power Logic Platform Technology using Uni-axial Strained Cilicon Transistors",* IEEE International Electron Devices Meeting (IEDM).

Jarman, D. (1995), *"A brief introduction to sigma delta conversion: Application Note",* Intersil Corp.

Jeffay, K., Stanat, D. F. and Martel, C. U. (1991), *"On Non-Preemptive Scheduling of Periodic and Sporadic Tasks",* Proceedings of the 12 th IEEE Symposium on Real-Time Systems, pp. 129-139.

Jejurikar, R., Pereira, C. and Gupta, R. (2004), *"Leakage Aware Dynamic Voltage Scaling for Real-Time Embedded Systems",* Proceedings of the 41st annual conference on Design automation, pp. 275-280.

Jerraya, A. A. and Wolf, W. (2005), *"Multiprocessor Systems-on-Chips",* Elsevier, USA.

Jerri, A. J. (1997), *"The Shannon sampling theorem: its various extensions and applications a tutorial review",* Proc. of the IEEE, Vol. 65, pp. 1565-1596.

Jha, N. K.(2006), *"Low power system scheduling, synthesis and displays".* In: Al-Hashimi, B. M. (Ed.) System-on-Chip: Next Generation Electronics, Vol. 18, The IEE, London, pp. 361-386.

Johnson, H. (2002), *"Random and deterministic jitter"*, EDN, 27 June 2002, pp. 24.

Kao, C.-Y. and Lincoln, B. (2004), *"Simple stability criteria for systems with time-varying delays",* Automatica, Vol.40 (8), pp. 1429-1434.

Kao, J. T. and Chandrakasan, A. P. (2000), *"Dual-Threshold Voltage Techniques for Low-Power Digital Circuits",* IEEE Journal of Solid-State Circuits, Vol.35 (7), pp. 1009-1018.

Kawaguchi, H., Shin, Y. and Sakurai, T. (2001), *"Experimental Evaluation of Cooperative Voltage Scaling (CVS): A Case Study"*, Proceedings of IEEE Workshop on Power Management for Real-Time and Embedded Systems, pp. 17-23.

Keshavarzi, A., Ma, S., Narendra, S., Bloechel, B., Mistry, K., Ghani, T., Borkar, S. and De, V. (2001), *"Effectiveness of reverse body bias for leakage control in scaled dual Vt CMOS ICs"*, Proceedings of the 2001 international symposium on Low power electronics and design, pp. 207-212.

Kester, W. (2005), *"MT-007: Aperture Time, Aperture Jitter, Aperture Delay Time – Removing the Confusion"*, Analog Devices, http://www.analog.com/en/content /0,2886,760%255F788%255F91284,00.html.

Kim, M., Kiyono, A., Ichige, K. and Arau, H. (2005), *"Experimental Study of Jitter Effect on Digital Downconversion Receiver with Undersampling Scheme"*, IEICE Transactions on Infomation and Systems, Vol.E88-D (7), pp. 1430-1436.

Kim, W., Kim, J. and Min, S. L. (2002), *"A Dynamic Voltage Scaling Algorithm for Dynamic-Priority Hard Real-Time Systems Using Slack Time Analysis"*, Proceedings of Design Automation and Test in Europe, pp. 788-794.

Kim, W., Kim, J. and Min, S. L. (2003), *"Dynamic Voltage Scaling Algorithm for Fixed-Priority Real-Time Systems Using Work-Demand Analysis"*, Proceedings of the 2003 international symposium on Low power electronics and design, Seoul, Korea, pp. 396-401.

Kocher, P. (1997), *"Blowfish source code"*, Kocher, P., http://www.schneier.com /blowfish-download.html.

Kopetz, H. (1991), *"Event-triggered versus time-triggered real-time systems."* Technical Report 8/91, Technical University of Vienna, Austria,

Kopetz, H. (1997), *"Real-Time Systems: Design Principles For Distributed Embedded Applications"*, Kluwer Academic.

Kumar, V. (2006), *"Design with (low) power while limiting leakage"*, EDA Design Line, 18 June 2006.

Kurian, S. and Pont, M. J. (in press), *"The maintenance and evolution of resource-constrained embedded systems created using design patterns"*, Journal of Systems and Software.

Kwak, S. W., Choi, B. J. and Kim, B. K. (2001), *"An optimal checkpointing-strategy for real-time control systemsunder transient faults"*, IEEE Transactions on Reliability, Vol.50 (3), pp. 293-301.

Labrosse, J. J. (1998), *"MicroC/OS-II: The Real-Time Kernel"*, 2nd ed, CMP Books.

Lahiri, K., Raghunathan, A., Dey, S. and Panigrahi, D. (2002), *"Battery-driven system design: a new frontier in low power design"*, Proceedings of ASP-DAC 2002. 7th Asia and South Pacific and the 15th International Conference on VLSI Design, pp. 261-267.

Lahiri, K., Raghunathan, A. and Dey, S. (2004), *"Efficient power profiling for battery-driven embedded system design"*, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol.23 (6), pp. 919-932.

Lattice (2003), *"Controlling and Monitoring Power-One Bricks and SIPs with the Lattice ispPAC-POWR1208"*, Lattice Semiconductor Corp., http://www.power-one.com/technical/articles/Lattice_ispPAC.pdf.

Lattice (2004), *"ispClock 5600 Family"*, Lattice Semiconductor Corp, http://www.msc.de/e/produkte/ele_kom/lattice/files/ispclock5600.pdf.

Lattice (2005), *"In-System Programmable Clock Generator"*, Lattice Semiconductor Corporation, http://www.latticesemi.com/products/ispClock/index.cfm.

Lee, B., Nurvitadhi, E., Dixit, R., Yu, C. and Kim, M. (2005), *"Dynamic voltage scaling techniques for power efficient video decoding"*, Journal of Systems Architecture, Vol.51 (10-11), pp. 633-652.

Lee, S. and Sakurai, T. (2000), *"Run-time Voltage Hopping for Low-Power Real-Time Systems"*, Proceedings of Design Automation Conference, pp. 806-809.

Lee, S., Yoo, S. and Choi, K. (2002), *"An Intra-Task Dynamic Voltage Scaling Method for SoC Design with Hierarchical FSM and Synchronous Dataflow Model"*, International Symposium on Low Power Electronics and Design (ISLPED 2002), pp. 84-87.

Lee, Y.-H., Reddy, K. P. and Krishna, C. M. (2003), *"Scheduling Techniques for Reducing Leakage Power in Hard Real-Time Systems"*, 15th Euromicro Conference on Real-Time Systems (ECRTS'03), pp. 105-112.

Leen, G., Heffernan, D. and Dunne, A. (1999), *"Digital networks in the automotive vehicle"*, Computing and Control, Vol.10 pp. 257-266.

Li, Y.-T. S. and Malik, S. (1995), *"Performance Analysis of Embedded Software Using Implicit Path Enumeration"*, Proceedings of the 32nd ACM/IEEE Design Automation Conference, pp. 456-461.

Lin, H.-H. and Hsueh, C.-W. (2006), *"Applying pinwheel scheduling and compiler profiling for power-aware real-time scheduling"*, Real-Time Systems, Vol.34 (1), pp. 37-51.

Lin, K.-J. and Herkert, A. (1996), *"Jitter Control in Time-Triggered Systems"*, Proceedings of the 29th Hawaii International Conference on System Sciences, Maui, Hawaii, pp. 451-459.

Linden, D. and Reddy, T. B. (1995), *"Handbook of batteries"*, 3rd ed, McGraw-Hill, New York.

Liu, C. L. and Layland, J. W. (1973), *"Scheduling algorithms for multi-programming in a hard real-time environment"*, Journal of the AVM 20, Vol.1 pp. 40-61.

Liu, J. W. S. (2000), *"Real-time systems"*, Prentice Hall.

Locke, C. D. (1992), *"Software architecture for hard real-time applications: cyclic executives vs. fixed priority executives"*, Real-Time Systems, Vol.4 pp. 37-52.

Lorch, J. R. and Smith, A. J. (1997), *"Scheduling Techniques for Reducing Processor Energy Use in MacOS"*, Wireless Networks, Vol.3 (5), pp. 311-324.

Lorch, J. R. and Smith, A. J. (1998), *"Software Strategies for Portable Computer Energy Management"*, IEEE Personal Communications, June '98, pp. 60-73.

Lorch, J. R. and Smith, A. J. (2001), *"Improving Dynamic Voltage Scaling Algorithms with PACE"*, Proceedings of SIGMETRICS, pp. 50-61.

Lu, Y.-H., Simunic, T. and Micheli, G. D. (1999), *"Software Controlled Power Management"*, Proceedings of the Seventh International Workshop on Hardware Software Codesign, pp. 157-161.

Lu, Y.-H., Chung, E.-Y., Simuni, T., Benini, L. and Micheli, G. D. (2000), *"Quantitative Comparison of Power Management Algorithms"*, the conference on Design, automation and test in Europe, pp. 20-26.

Luo, J. and Jha, N. K. (2000), *"Power-conscious joint scheduling of periodic task graphs and aperiodic tasks in distributed real-time embedded systems"*, International Conference on Computer Aided Design, California, pp. 357-364.

Luo, J. and Jha, N. K. (2001), *"Battery-driven static scheduling for real-time distributed embedded systems"*, IEEE Design Automation Conference, Bangalore, India, pp. 444-449.

Maaita, A. and Pont, M. J.(2005), *"Using 'planned pre-emption' to reduce levels of task jitter in a time-triggered hybrid scheduler"*. In: Koelmans, A., A. Bystrov, M. J. Pont, R. Ong and A. Brown (Eds.), Proceedings of the 2nd UK Embedded Forum (Birmingham, 2005), pp. 18-35.

Macadie, D. (2004), *"Key design considerations for high quality audio ADC performance"*, Wolfson Microelectronics.

Mansuri, M. and Yang, C.-K. K. (2002), *"Jitter optimization based on phase-locked loop design parameters"*, IEEE Journal of Solid-State Circuits, Vol.37 (11), pp. 1375-1382.

Marculescu, R., Marculescu, D. and Pedram, M. (1994), *"Switching Activity Analysis Considering Spatiotemporal Correlations"*, IEEE/ACM International Conference on Computer-Aided Design, pp. 294-299.

Marti, P., Fuertes, J. M., Ramamritham, K. and Fohler, G. (2001a), *"Jitter Compensation for Real-Time Control Systems"*, 22nd IEEE Real-Time Systems Symposium (RTSS'01), London, England, pp. 39-48.

Marti, P., Fuertes, J. M., Villa, R. and Fohler, G. (2001b), *"On Real-Time Control Tasks Schedulability"*, European Control Conference (ECC01), Porto, Portugal, pp. 2227-2232.

Marti, P. (2002), *"Analysis and design of real-time control systems with varying control timing constraints"*, PhD thesis, Automatic Control Department, Technical University of Catalonia.

Martin, S. M., Flautner, K., Mudge, T. and Blaauw, D. (2002), *"Combined dynamic voltage scaling and adaptive body biasing for lower power microprocessors under dynamic workloads"*, Proceeding of International. Conference on Computer Aided Design, pp. 721-725.

Martin, T. L. (1999), *"Balancing Batteries, Power, and Performance: System Issues in CPU Speed-Setting for Mobile Computing"*, A Ph.D. dissertation, Electrical and Computer Engineering, Carnegie Mellon University, Pittsburgh, Pennsylvania.

Martin, T. L. and Siewiorek, D. P. (1999a), *"Non-ideal Battery Properties and Low Power Operation in Wearable Computing"*, International Symposium on Wearable Computers, pp. 101-106.

Martin, T. L. and Siewiorek, D. P. (1999b), *"The Impact of Battery Capacity and Memory Bandwidth on CPU Speed-Setting: A Case Study"*, Proceedings of the 1999 International Symposium on Low Power Electronics and Design, pp. 200-205.

Marwedel, P. (2003), *"Embedded System Design"*, Kluwer Academic Publishers, The Netherlands.

Maxim (2005), *"WCDMA Cellular Phone 600mA Buck Regulators"*, Maxim, http:// pdfserv.maxim-ic.com/en/ds/MAX1820-MAX1821X.pdf.

Maxim-Dallas (2003), *"Demystifying Sigma-Delta ADCs: Application Note 1870"*, Maxim-Dallas.

Mazzoni, L. (2003), *"Power aware design for embedded systems"*, The IEE: Electronics systems and software, Vol. 1 (5), Oct/Nov 2003, pp. 12-17.

Melhem, R., Mosse, D. and Elnozahy, E. (2004), *"The interplay of power management and fault recovery in real-time systems"*, IEEE Transactionon Computers, Vol.53 (2), pp. 217-231.

Mills, D. L. (1995), *"Improved Algorithms for Synchronizing Computer Network Clocks"*, IEEE/ACM Transactions on Networks, Vol.3 (3), pp. 245-254.

Min, R., Bhardwaj, M., Cho, S., Ickes, N., Shih, E. and Sinha, A. (2002), *"Energy-centric enabling technologies for wireless sensor networks"*, IEEE Wireless Communications, Vol. 9 (4), August 2002, pp. 28-39.

Mochocki, B. C., Hu, X. S., Racu, R. and Ernst, R. (2005), *"Dynamic Voltage Scaling for the Schedulability of Jitter-Constrained Real-Time Embedded Systems"*, International Conference on Computer Aided Design 2005, San Jose, CA., pp. 445-448.

Mouw, J. A. K., Langendoen, K. and Pouwelse, J. (2002), *"LART Lessons Learned: cpufreq"*, Proceeding of the Ottawa Linux Symposium, Ottawa, Canada, pp. 376-382.

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"*, In: Society of Automotive Engineers (Ed.): In-vehicle software and hardware systems, the Society of Automotive Engineers (SAE) World Congress, Detroit, Michigan, USA.

Neumann, P. G. (1995), *"Computer Related Risks"*, ACM Press.

Nguyen, D., Davare, A., Orshansky, M., Chinnery, D., Thompson, B. and Keutzer, K. (2003), *"Minimization of dynamic and static power through joint assignment of threshold voltages and sizing optimization"*, Proceedings of the 2003 international symposium on Low power electronics and design, Seoul, Korea, pp. 158-163.

NI (2000), *"NI PCI-6035E datasheet"*, National Instruments, http://www.ni.com/pdf /products/us/4daqsc202-204_ETCx2_212_213.pdf.

NI (2001), *"LabView: User Manual"*, National Instruments, http://www.ni.com/pdf /manuals/320999d.pdf.

Nolte, T., Hansson, H. and Norstrom, C. (2002), *"Minimizing CAN response-time jitter by message manipulation"*, IEEE Real Time Technology and Applications Symposium 2002, pp. 197-206.

Nolte, T. (2003), *"Reducing Pessimism and Increasing Flexibility in the Controller Area Network"*, PhD thesis, Department of Computer Science and Engineering, Malardalen University, Vasteras, SWEDEN.

Nurvitadhi, E., Lee, B., Yu, C. and Kim, M. (2003), *"A Comparative Study of Dynamic Voltage Scaling Techniques for Low-Power Video Decoding"*, International Conference on Embedded Systems and Applications.

Oen, J. and Schultz, C. (2006), *"EMI shield for reducing clock jitter of a transceiver"*, United States Patent 7092639, Intel Corporation, August 15, 2006.

Ong, C.-K., Hong, D., Cheng, K.-T. T. and Wang, L.-C. (2004), *"Jitter spectral extraction for multi-gigahertz signal"*, Asia and South Pacific Design Automation Conference (ASP-DAC '04), pp. 298-303.

Ou, N., Farahmand, T., Kuo, A., Tabatabaei, S. and Ivanov, I. (2004), *"Jitter Models for the Design and Test of Gbps-Speed Serial Interconnects",* IEEE Design and Test of Computers, Vol.21 (4), pp. 302-313.

Paleologo, G. A., Benini, L., Bogliolo, A. and Micheli, G. D. (1998), *"Policy optimization for dynamic power management",* Proceedings of the 35th annual conference on Design automation, San Francisco, CA, pp. 182-187.

Panigrahi, D., Chiasserini, C., Dey, S., Rao, R., Raghunathan, A. and Lahiri, K. (2001), *"Battery Life Estimation of Mobile Embedded Systems",* The 14th International Conference on VLSI Design, pp. 57-63.

Patterson, D. A. and Hennessy, J. L. (2004), *"Computer Organization and Design: The Hardware/Software Interface",* 3rd ed, Elsevier / Morgan-Kaufmann.

Pedreiras, P. and Almeida, L. (2002), *"EDF message scheduling on controller area network",* Computing & Control Engineering Journal, Vol.13 (4), pp. 163-170.

Pering, T. and Brodersen, R. (1998), *"Energy Efficient Voltage Scheduling for Real-Time Operating Systems",* Proceedings of the 4th IEEE Real-Time Technology and Applications Symposium RTAS'98.

Pering, T., Burd, T. and Brodersen, R. (1998a), *"The Simulation and Evaluation of Dynamic Voltage Scaling Algorithms",* Proceedings of the International Symposium on Low Power Electronics and Design, pp. 76-81.

Pering, T., Burd, T. and Brodersen, R. (1998b), *"Dynamic Voltage Scaling and the Design of a Low-Power Microprocessor Systems",* the Power-Driven Microarchitecture Workshop, Barcelona, Spain, June.

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, Vol.55 (2), pp. 113-124.

Philips (2003), *"LPC2104/2105/2106: Single-chip 32-bit microcontrollers",* Philips Semiconductors, http://www.semiconductors.philips.com/acrobat/datasheets /LPC2104_2105_2106-04.pdf.

Pillai, P. and Shin, K. G. (2001), *"Real-Time Dynamic Voltage Scaling for Low-Power Embedded Operating Systems",* ACM Symposium on Operating Systems Principles, pp. 89-102.

Pont, M. J. (2001), *"Patterns for time-triggered embedded systems: Building reliable applications with 8051 family of microcontrollers",* Addison-Wesley.

Pop, P., Eles, P. and Peng, Z. (2004), *"Analysis and Synthesis of Distributed Real-time Embedded Systems",* Kluwer Academic Publishers, Netherlands.

Pouwelse, J., Langendoen, K. and Sips, H. (2001a), *"Energy Priority Scheduling for Variable Voltage Processors"*, International Symposium on Low Power Electronics and Design, pp. 28-33.

Pouwelse, J., Langendoen, K. and Sips, H. (2001b), *"Dynamic voltage scaling on a low-power microprocessor"*, Proceedings of the 7th annual international conference on Mobile computing and networking, pp. 251-259.

Pouwelse, J., Langendoen, K. and Sips, H. (2001c), *"Power-Aware Video Decoding"*, Proceeding of 22nd Picture Coding Symposium, Seoul, Korea.

Proctor, F. M. and Shackleford, W. P. (2001), *"Real-time Operating System Timing Jitter and its Impact on Motor Control"*, proceedings of the 2001 SPIE Conference on Sensors and Controls for Intelligent Manufacturing II, Vol. 4563-02.

Rakhmatov, D., Vrudhula, S. and Wallach, D. A. (2002), *"Battery Lifetime Prediction for Energy-Aware Computing"*, Proceedings of the 2002 International Symposium on Low Power Electronics and Design, pp. 154-159.

Rakhmatov, D. and Vrudhula, S. (2003), *"Energy management for battery-powered embedded systems"*, ACM Transactions on Embedded Computing Systems, Vol.2 (3), pp. 277-324.

Rao, R. and Vrudhula, S. (2005), *"Battery optimization vs energy optimization: which to choose and when?"* Proceedings of the 2005 IEEE/ACM International conference on Computer-aided design, pp. 439-445.

Raskovic, D., Martin, T. and Jovanov, E. (2004), *"Medical Monitoring Applications for Wearable Computing"*, The Computer Journal, Vol.47 (4), pp. 495-504.

Schilling, D. L. and Belove, C. (1985), *"Electronic Circuits: Discrete and Integrated"*, 2nd ed, McGraw-Hill, Singapore.

Schmitz, M. T. and Al-Hashimi, B. M. (2000), *"Low Power Process Assignment for Distributed Embedded Systems using Dynamic Voltage Scaling"*, Proceedings of IEE Hardware-Software Co-Design, pp. 7/1-7/4.

Schmitz, M. T., Al-Hashimi, B. M. and Eles, P. (2004), *"Iterative schedule optimization for voltage scalable distributed embedded systems"*, ACM Transactions on Embedded Computing Systems, Vol.3 (1), pp. 182-217.

Schneier, B. (1993), *"Fast Software Encryption"*, Cambridge Security Workshop Proceedings, Springer-Verlag, pp. 191-204.

Schossmaier, K. and Weiss, B. (1999), *"An Algorithm for Fault-Tolerant Clock State&Rate Synchronization"*, Proc. of the 18th IEEE Symp. on Reliable Distributed Systems (SRDS '99), Lausanne, pp. 36-47.

Schurgers, C., Aberthorne, O. and Srivastava, M. B. (2001), *"Modulation scaling for Energy Aware Communication Systems",* Proceedings of the 2001 international symposium on Low power electronics and design, California, Vol. 96-99.

Schurgers, C., Raghunathan, V. and Srivastava, M. B. (2003), *"Power management for energy-aware communication systems",* ACM Transactions on Embedded Computing Systems (TECS), Vol.2 (3), pp. 431-447.

Seifert, N., Moyer, D., Leland, N. and Hokinson, R. (2001), *"Historical trend in alpha-particle induced soft error rates of theAlpha microprocessor",* Proceedings of 39th Annual IEEE International Reliability Physics Symposium, pp. 259-265.

Sha, L., Rajkumar, R. and Lehoczky, J. P. (1990), *"Priority inheritance protocols: an approach to real-timesynchronization",* IEEE Transactions on Computers, Vol.39 (9), pp. 1175-1185.

Shin, D., Kim, J. and Lee, S. (2001), *"Low-energy intra-task voltage scheduling using static timing analysis",* Proceedings of the 38th conference on Design automation, Las Vegas, Nevada, pp. 438-443.

Shin, Y. and Choi, K. (1999), *"Power Conscious Fixed Priority Scheduling for Hard Real-Time Systems",* Proceeding of Design Automation Conference, pp. 134-139.

Shin, Y., Choi, K. and Sakurai, T. (2000), *"Power optimization of real-time embedded systems on variable speed processors",* Proceedings of the 2000 IEEE/ACM international conference on Computer-aided design, pp. 365-368.

Shivakumar, P., Kistler, M., Keckler, S. W., Burger, D. and Alvisi, L. (2002), *"Modeling the Effect of Technology Trends on the Soft Error Rate of Combinational Logic",* Proceedings of the 2002 International Conference on Dependable Systems and Networks, pp. 389-398.

Sidiropoulos, S. and Horowitz, M. A. (1997), *"A semidigital dual delay-locked loop",* IEEE Journal of Solid-State Circuits, Vol.32 (11), pp. 1683-1692.

Simunic, T., Benini, L. and Micheli, G. D. (1999), *"Event-Driven Power Management of Portable Systems",* International Symposium on System Synthesis, pp. 18-23.

Simunic, T., Benini, L., Glynn, P. and Micheli, G. D. (2000), *"Dynamic Power Management for Portable Systems",* Proceedings of the sixth annual international conference on Mobile computing and networking, pp. 11-19.

Simunic, T., Benini, L., Acquaviva, A., Glynn, P. and Micheli, G. D. (2001), *"Dynamic Voltage Scaling for Portable Systems",* The seventh annual international conference on Mobile computing and networking, pp. 251-259.

Sinha, A. and Chandrakasan, A. P. (2001), *"JouleTrack - A Web Based Tool for Software Energy Profiling"*, the 38th Design Automation Conference, pp. 220-225.

Son, D., Yu, C. and Kim, H.-N. (2001), *"Dynamic Voltage Scaling on MPEG Decoding"*, International Conference of Parallel and Distributed System (ICPADS), pp. 633-640.

Spuri, M., Buttazzo, G. and Sensini, F. (1995), *"Robust Aperiodic Scheduling under Dynamic Priority Systems"*, 16 th IEEE Real-Time Systems Symposium, Pisa, Italy, pp. 210-219.

Spuri, M. and Buttazzo, G. (1996), *"Scheduling Aperiodic Tasks in Dynamic Priority Systems"*, Journal of Real-Time Systems, Vol.10 (2), pp. 179-210.

Srivastava, A. and Sylvester, D. (2004), *"Minimizing total power by simultaneuos Vdd/Vth assignment"*, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol.23 (5), pp. 665-677.

Stankovic, J. A. and Ramamritham, K. (1990), *"What is predictability for real-time systems?"* Real-Time Systems, Vol.2 (4), pp. 247-254.

Stankovic, J. A., Spuri, M., Natale, M. D. and Buttazzo, G. (1995), *"Implications of Classical Scheduling Results For Real-Time Systems"*, IEEE Computer, Vol.28 (6), pp. 16-25.

Story, M. (1998), *"Timing Errors and Jitter"*, dCS Ltd., http://www.dcsltd.co.uk /technical_papers/jitter.pdf.

Swaminathan, V. and Chakrabarty, K. (2001), *"Real-Time Task Scheduling for Energy-Aware Embedded Systems"*, Journal of the Franklin Institute, Vol.338 (6), pp. 729-750.

Swaminathan, V. and Chakrabarty, K. (2003), *"Energy-Conscious, Deterministic I/O Device Scheduling in Hard Real-Time Systems"*, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol.22 (7), pp. 847-858.

Swaminathan, V. and Chakrabarty, K. (2005), *"Pruning-based, energy-optimal, deterministic I/O device scheduling for hard real-time systems"*, ACM Transactions on Embedded Computing Systems (TECS), Vol.4 (1), pp. 141-167.

Tektronix (2002), *"Understanding and Characterizing Timing Jitter"*, Tektronix Inc.

Tektronix (2004), *"Characterize Phase-Locked Loop Systems Using Real Time Oscilloscopes: Application Note"*, Tektronix Inc.

Tindell, K. and Burns, A. (1994), *"Guaranteed Message Latencies For Distributed Safety-Critical Hard Real-Time Control Networks"*, Technical Report YCS229, Dept. of Computer Science, University of York, June 1994.

Torngren, M. (1998), *"Fundamentals of implementing real-time control applications in distributed computer systems"*, Real-Time Systems, Vol.14 pp. 219-250.

Tryfonas, C. and Varma, A. (1999), *"MPEG-2 Transport over ATM Networks"*, IEEE Communications Surveys, Vol. 2 (4), 4th Quarter 1999, pp. 24-33.

Valvano, J. W. (2000), *"Embedded Microcomputer Systems: Real Time Interfacing"*, Brooks/Cole, USA.

Viredaz, M. A. and Wallach, D. A. (2003), *"Power Evaluation of a Handheld Computer"*, IEEE Micro, Vol.23 (1), pp. 66-74.

Wavecrest (2001), *"Understanding Jitter: Getting Started"*, Wavecrest Corporation.

Wei, T., Mishra, P., Wu, K. and Liang, H. (2006), *"Online Task-Scheduling for Fault-Tolerant Low-Energy Real-Time Systems"*, IEEE/ACM International Conference on Computer-Aided Design, 2006. ICCAD '06, pp. 522-527.

Weiser, M., Welch, B., Demers, A. and Shenker, S. (1994), *"Scheduling for Reduced CPU Energy"*, Proceeding of the First Symposium on Operating Systems Design and Implementation, pp. 13-23.

Wu, D., Al-Hashimi, B. M. and Eles, P. (2004), *"Dynamic and Leakage Power-Composition Profile Driven Co-Synthesis for Energy and Cost Reduction"*, Proceedings of System-On-Chip Design, Test and Technology, Loughborough, UK.

Xilinx (2005), *"Spartan-3 FPGA Family: Complete Data Sheet"*, Xilinx, http://www.xilinx.com/bvdocs/publications/ds099.pdf.

Yang, Z., Yuan, Y., He, J. and Chen, W. (2005), *"Adaptive Modulation Scaling Scheme for Wireless Sensor Networks"*, IEICE Transactions on Communications, Vol.E88-B (3), pp. 882-889.

Yu, Y., Krishnamachari, B. and Prasanna, V. K. (2004), *"Energy-Latency Tradeoffs for Data Gathering in Wireless Sensor Networks"*, In Proceedings of IEEE Infocom 2004, Los Angeles.

Yuan, Y., Yang, Z. and He, J. (2005), *"An Adaptive Modulation Scaling Scheme for Quality of Services Ensurance in Wireless Sensor Networks"*, American Journal of Applied Sciences, Vol.2 (3), pp. 734-738.

Zhang, F. and Chanson, S. T. (2003), *"Processor Voltage Scheduling for Real-Time Tasks With Non-Preemptible Sections"*, Proceedings of IEEE Real-Time Systems Symposium, Austin, Texas, pp. 235-245.

Zhang, H. (1995), *"Service Disciplines For Guaranteed Performance Service in Packet-Switching Networks"*, Proceedings of the IEEE, Vol.83 (10), pp. 1374-1396.

Zhang, Y. and Chakrabarty, K. (2006), *"A unified approach for fault tolerance and dynamic power management in fixed-priority real-time embedded systems"*, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol.25 (1), pp. 111-125.

Zhu, D., Melhem, R. and Childers, B. R. (2003), *"Scheduling with dynamic voltage/speed adjustment using slack reclamation in multiprocessor real-time systems"*, IEEE Transactions on Parallel and Distributed Systems, Vol.14 (7), pp. 686-700.

Zhu, D., Melhem, R. and Mosse, D. (2004), *"effects of energy management on reliability in real-time embedded systems"*, Proceedings of the 2004 IEEE/ACM International conference on Computer-aided design, pp. 35-40.

Zhu, D. (2006), *"Reliability-Aware Dynamic Energy Management in Dependable Embedded Real-Time Systems"*, Proceeding of the IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'06), San Jose, pp. 397-407.

Zhu, Y. and Mueller, F. (2004), *"Feedback EDF Scheduling Exploiting Dynamic Voltage Scaling"*, IEEE Real-Time and Embedded Technology and Applications Symposium 2004, pp. 84-93.

Zhu, Y. and Mueller, F. (2005), *"Feedback EDF scheduling exploiting hardware-assisted asynchronous dynamic voltage scaling"*, Proceedings of the 2005 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems, Vol. 40, pp. 203-212.

Zhuo, J. and Chakrabarti, C. (2005), *"An efficient dynamic task scheduling algorithm for battery powered DVS systems"*, Proceedings of the 2005 conference on Asia South Pacific design automation, Shanghai, China, pp. 846-849.

Ziv, A. and Bruck, J. (1997), *"An On-Line Algorithm for Checkpoint Placement"*, IEEE Transactions on Computers, Vol.46 (9), pp. 976-985.