An Experimental Evaluation of Real-Time DVFS Scheduling Algorithms

Sonal Saha

Thesis submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfilment of the requirements for the degree of

Master of Science
in
Computer Engineering

Binoy Ravindran, Chair
Paul E. Plassmann
Robert P. Broadwater

September 9, 2011
Blacksburg, Virginia

Keywords: Dynamic Voltage and Frequency Scaling, Real-Time Linux
Copyright 2011, Sonal Saha

An Experimental Evaluation of Real-Time DVFS Scheduling Algorithms

Sonal Saha

(ABSTRACT)

Dynamic voltage and frequency scaling (DVFS) is an extensively studied energy management technique, which aims to reduce the energy consumption of computing platforms by dynamically scaling the CPU frequency. Real-Time DVFS (RT-DVFS) is a branch of DVFS, which reduces CPU energy consumption through DVFS, while at the same time ensures that task time constraints are satisfied by constructing appropriate real-time task schedules. The literature presents numerous RT-DVFS scheduling algorithms, which employ different techniques to utilize the CPU idle time to scale the frequency. Many of these algorithms have been experimentally studied through simulations, but have not been implemented on real hardware platforms. Though simulation-based experimental studies can provide a first-order understanding, implementation-based studies can reveal actual timeliness and energy consumption behaviours. This is particularly important, when it is difficult to devise accurate simulation models of hardware, which is increasingly the case with modern systems.

In this thesis, we study the timeliness and energy consumption behaviours of fourteen state-of-the-art RT-DVFS schedulers by implementing and evaluating them on two hardware platforms. The schedulers include CC-EDF, LA-EDF, REUA, DRA and AGR1 among others, and the hardware platforms include ASUS laptop with the Intel I5 processor and a motherboard with the AMD Zacate processor. We implemented these schedulers in the ChronOS real-time Linux kernel and measured their actual timeliness and energy behaviours under a range of workloads including CPU-intensive, memory-intensive, mutual exclusion lock-intensive, and processor-underloaded and overloaded workloads.

Our studies reveal that measuring the CPU power consumption as the cube of CPU frequency can lead to incorrect conclusions. In particular, it ignores the idle state CPU power consumption, which is orders of magnitude smaller than the active power consumption. Consequently, power savings obtained by exclusively optimizing active power consumption (i.e., RT-DVFS) may be offset by completing tasks sooner by running them at the highest frequency and transitioning to the idle state earlier (i.e., no DVFS). Thus, the active power consumption savings of the RT-DVFS techniques' that we report are orders of magnitude smaller than their simulation-based savings reported in the literature.

Dedication

I dedicate this thesis to my mother.

Acknowledgments

First and foremost, I would like to thank my advisor, Dr. Binoy Ravindran, for his help, guidance and encouragement. It has been an honour to work under him.

I would also like to thank Dr. Paul Plassmann and Dr. Robert Broadwater, for serving on my committee and providing me with their valuable suggestions and feedback. I also thank my Real-Time Systems lab mates, Matthew Dellinger, Piyush Garyali and Aaron Lindsay for their help and feedback.

Finally, I would like to thank my family and friends, without whose love, support and encouragement, this thesis wouldn't have been possible.

Contents

1	Introduction	1
1.1	Limitations of Past Real-Time DVFS Research	3
1.2	Research Contributions	5
1.3	Thesis Organization	7
2	Related Work	8
2.1	GP-DVFS	9
2.2	RT-DVFS	11
2.3	Statistical DVFS	13
2.4	Summary	14
3	Algorithms	16
3.1	Schedulers for Independent Underloaded task-sets	17
3.1.1	Base-EDF	17
3.1.2	Static-EDF	18
3.1.3	CC-EDF	19
3.1.4	LA-EDF	20
3.1.5	Snowdon-min	22
3.1.6	DRA	23
3.1.7	DRA-OTE	26
3.1.8	AGR1	27
3.1.9	AGR2	30

3.2	Schedulers for Dependent Underloaded task-sets	30
3.2.1	EUA	30
3.2.2	HS	32
3.2.3	DS	35
3.2.4	USFLEDF	37
3.3	Schedulers for Overloaded task-sets	38
3.3.1	REUA	38
4	Extending ChronOS with DVFS support	41
4.1	ChronOS	41
4.1.1	Introduction	41
4.1.2	ChronOS real-time Scheduler	41
4.1.3	Scheduling Events	42
4.1.4	System Calls provided by ChronOS	42
4.2	Adding DVFS support to ChronOS	43
4.2.1	The CPUfreq Subsystem	43
4.2.2	Changing the frequency using CPUfreq module	45
4.2.3	Working of the CPUfreq module	45
4.2.4	Integration of CPUfreq subsystem with ChronOS	46
4.2.5	Implementation of the RT-DVFS Schedulers	46
4.2.6	Modification of the real-time data structure for RT-DVFS support . .	48
4.2.6.1	Additional fields for RT-DVFS support	49
5	Experimental Methodology	50
5.1	Platform Specifications	50
5.2	Test Application	50
5.2.1	Modification to the Test Application for RT-DVFS	51
5.3	Real-time Measurements	51
5.4	Power Measurements	51

5.4.1	Performance and Idle states of CPU	52
5.4.2	System Power Measurements	52
5.4.3	Normalized and Actual CPU power measurement	53
5.4.4	Calculation of Normalized CPU Energy consumption	54
5.4.5	Calculation of Actual CPU Power	54
6	Experiments	55
6.1	Energy consumption results on Intel I5 laptop for schedulers designed for independent task-sets	55
6.1.1	Varying the CPU utilization at constant ACET	55
6.1.2	Varying ACET at a constant CPU utilization	77
6.1.3	DSR results on Intel I5 laptop	86
6.2	Energy and DSR results for the schedulers designed for dependent task-sets .	93
6.3	Energy consumption results on the AMD Zacate Board	108
6.4	System Power results on the AMD Zacate Board	113
6.5	Energy and DSR results for the schedulers designed for dependent task-sets .	121
7	Conclusions and Future Work	131
7.1	Conclusions	131
7.2	Future Work	133
7.2.1	DVFS in CMPs	133
7.2.2	Considering Real world Applications	134
7.2.3	DVFS with DPM	135
7.2.4	DVFS with Procrastination Scheduling	135
7.2.5	Reducing Dynamic Power Consumption of other devices	135
	Bibliography	135

List of Figures

1.1	Idle Time Example	2
1.2	Dynamic Slack Example	3
3.1	Base-EDF	18
3.2	Static-EDF	19
3.3	CC-EDF	21
3.4	LA-EDF	22
3.5	Snowdon-min	24
3.6	DRA	26
3.7	Step TUF	30
3.8	HS	34
3.9	DS	35
4.1	CPUfreq Subsystem	44
4.2	Working of CPUfreq module	46
4.3	RT-DVFS scheduling in ChronOS	47
6.1	Normalized CPU Energy vs. CPU utilization (Underloads, 0.1 WCET, 5T, NL)	56
6.2	Actual CPU Power vs. CPU utilization (Underload, 0.1 WCET, 5T, NL) . .	56
6.3	Normalized CPU Energy vs. CPU utilization (Overloads, 0.1 WCET, 5T, NL)	57
6.4	Actual CPU Power vs. CPU utilization (Overloads, 0.1 WCET, 5T, NL) . .	58
6.5	Normalized CPU Energy vs. CPU utilization (Underloads, 0.2 WCET, 5T, NL)	59
6.6	Actual CPU Power vs. CPU utilization (Underload, 0.2 WCET, 5T, NL) . .	59

6.7	Normalized CPU Energy vs. CPU utilization (Overloads, 0.2 WCET, 5T, NL)	60
6.8	Actual CPU Power vs. CPU utilization (Overloads, 0.2 WCET, 5T, NL) . .	60
6.9	Normalized CPU Energy vs. CPU utilization (Underloads, 0.3 WCET, 5T, NL)	61
6.10	Actual CPU Power vs. CPU utilization (Underload, 0.3 WCET, 5T, NL) . .	61
6.11	Normalized CPU Energy vs. CPU utilization (Overloads, 0.3 WCET, 5T, NL)	62
6.12	Actual CPU Power vs. CPU utilization (Overloads ,0.3 WCET, 5T, NL) . .	62
6.13	Normalized CPU Energy vs. CPU utilization (Underloads, 0.4 WCET, 5T, NL)	63
6.14	Actual CPU Power vs. CPU utilization (Underload, 0.4 WCET, 5T, NL) . .	63
6.15	Normalized CPU Energy vs. CPU utilization (Overloads, 0.4 WCET, 5T, NL)	64
6.16	Actual CPU Power vs. CPU utilization (Overloads, 0.4 WCET, 5T, NL) . .	65
6.17	Normalized CPU Energy vs. CPU utilization (Underload, 0.5 WCET, 5T, NL)	66
6.18	Actual CPU Power vs. CPU utilization (Underload,0.5 WCET, 5T, NL) . .	67
6.19	Normalized CPU Energy vs. CPU utilization (Overloads, 0.5 WCET, 5T, NL)	68
6.20	Actual CPU Power vs. CPU utilization (Overloads, 0.5 WCET, 5T, NL) . .	68
6.21	Normalized CPU Energy vs. CPU utilization (Underloads, 0.6 WCET, 5T, NL)	69
6.22	Actual CPU Power vs. CPU utilization (Underload, 0.6 WCET, 5T, NL) . .	69
6.23	Normalized CPU Energy vs. CPU utilization (Overloads, 0.6 WCET, 5T, NL)	70
6.24	Actual CPU Power vs. CPU utilization (Overloads, 0.6 WCET, 5T, NL) . .	70
6.25	Normalized CPU Energy vs. CPU utilization (Underload, 0.7 WCET, 5T, NL)	71
6.26	Actual CPU Power vs. CPU utilization (Underload, 0.7 WCET, 5T, NL) . .	71
6.27	Normalized CPU Energy vs. CPU utilization (Overloads, 0.7 WCET, 5T, NL)	72
6.28	Actual CPU Power vs. CPU utilization (Overloads, 0.7 WCET, 5T, NL) . .	72
6.29	Normalized CPU Energy vs. CPU utilization (Underload, 0.8 WCET, 5T, NL)	73
6.30	Actual CPU Power vs. CPU utilization (Underload, 0.8 WCET, 5T, NL) . .	73
6.31	Normalized CPU Energy vs. CPU utilization (Overloads, 0.8 WCET, 5T, NL)	74
6.32	Actual CPU Power vs. CPU utilization (Overloads, 0.8 WCET, 5T, NL) . .	74
6.33	Normalized CPU Energy vs. CPU utilization (Underload, 0.9 WCET, 5T, NL)	75
6.34	Actual CPU Power vs. CPU utilization (Underload, 0.9 WCET, 5T, NL) . .	75

6.35	Normalized CPU Energy vs. CPU utilization (Overloads, 0.9 WCET, 5T, NL)	76
6.36	Actual CPU Power vs. CPU utilization (Overloads, 0.9 WCET, 5T, NL) . .	77
6.37	Normalized CPU Energy vs. CPU utilization (Underload, 1 WCET, 5T, NL)	78
6.38	Actual CPU Power vs. CPU utilization (Underload, 1 WCET, 5T, NL) . . .	78
6.39	Normalized CPU Energy vs. CPU utilization (Overloads, 1.0 WCET, 5T, NL)	79
6.40	Actual CPU Power vs. CPU utilization (Overloads, 1 WCET, 5T, NL) . . .	79
6.41	Normalized CPU Energy vs. ACET (Underload, 50% CPU Utilisation, 5T, NL)	80
6.42	Actual CPU Power vs. ACET (Underload, 50% CPU Utilisation, 5T, NL) .	81
6.43	Normalized CPU Energy vs. ACET (Underload, 60% CPU Utilisation, 5T, NL)	82
6.44	Actual CPU Power vs. ACET (Underload, 60% CPU Utilisation, 5T, NL) .	82
6.45	Normalized CPU Energy vs. ACET (Underload, 70% CPU Utilisation, 5T, NL)	83
6.46	Actual CPU Power vs. ACET (Underload, 70% CPU Utilisation, 5T, NL) .	83
6.47	Normalized CPU Energy vs. ACET (Underload, 80% CPU Utilisation, 5T, NL)	84
6.48	Actual CPU Power vs. ACET (Underload, 80% CPU Utilisation, 5T, NL) .	84
6.49	Normalized CPU Energy vs. ACET (Underload, 90% CPU Utilisation, 5T, NL)	85
6.50	Actual CPU Power vs. ACET (Underload, 90% CPU Utilisation, 5T, NL) .	86
6.51	Normalized CPU Energy vs. ACET (Underload,100% CPU Utilisation , 5T, NL)	87
6.52	Actual CPU Power vs. ACET (Underload, 100% CPU Utilisation, 5T, NL) .	87
6.53	DSR vs. CPU Utilisation (0.1 WCET, 5T, NL)	88
6.54	DSR vs. CPU Utilisation (0.2 WCET, 5T, NL)	88
6.55	DSR vs. CPU Utilisation (0.3 WCET, 5T, NL)	89
6.56	DSR vs. CPU Utilisation (0.4 WCET, 5T, NL)	89
6.57	DSR vs. CPU Utilisation (0.5 WCET, 5T, NL)	90
6.58	DSR vs. CPU Utilisation (0.6 WCET, 5T, NL)	90
6.59	DSR vs. CPU Utilisation (0.7 WCET, 5T, NL)	91
6.60	DSR vs. CPU Utilisation (0.8 WCET, 5T, NL)	91
6.61	DSR vs. CPU Utilisation (0.9 WCET, 5T, NL)	92
6.62	DSR vs. CPU Utilisation (1 WCET, 5T, NL)	92

6.63	5% CS - Normalized CPU Energy vs. CPU utilization (5% CS, 10T, 1L)	94
6.64	10% CS - Normalized CPU Energy vs. CPU utilization (10% CS, 10T, 1L)	94
6.65	15% CS - Normalized CPU Energy vs. CPU utilization (15% CS, 10T, 1L)	95
6.66	20% CS - Normalized CPU Energy vs. CPU utilization (20% CS, 10T, 1L)	95
6.67	30% CS - Normalized CPU Energy vs. CPU utilization (30% CS, 10T, 1L)	96
6.68	40% CS - Normalized CPU Energy vs. CPU utilization (40% CS, 10T, 1L)	96
6.69	50% CS - Normalized CPU Energy vs. CPU utilization (50% CS, 10T, 1L)	97
6.70	60% CS - Normalized CPU Energy vs. CPU utilization (60% CS, 10T, 1L)	98
6.71	70% CS - Normalized CPU Energy vs. CPU utilization (70% CS, 10T, 1L)	99
6.72	5% CS - Actual CPU Power vs. CPU utilization (5% CS, 10T, 1L)	99
6.73	10% CS - Actual CPU Power vs. CPU utilization (10% CS, 10T, 1L)	100
6.74	15% CS - Actual CPU Power vs. CPU utilization (15% CS, 10T, 1L)	100
6.75	20% CS - Actual CPU Power vs. CPU utilization (20% CS, 10T, 1L)	101
6.76	30% CS - Actual CPU Power vs. CPU utilization (30% CS, 10T, 1L)	101
6.77	40% CS - Actual CPU Power vs. CPU utilization (40% CS, 10T, 1L)	102
6.78	50% CS - Actual CPU Power vs. CPU utilization (50% CS, 10T, 1L)	102
6.79	60% CS - Actual CPU Power vs. CPU utilization (60% CS, 10T, 1L)	103
6.80	70% CS - Actual CPU Power vs. CPU utilization (70% CS, 10T, 1L)	103
6.81	5% CS - DSR vs. CPU utilization (5% CS ,10T, 1L)	104
6.82	10% CS - DSR vs. CPU utilization (10% CS, 10T, 1L)	104
6.83	20% CS - DSR vs. CPU utilization (20% CS, 10T, 1L)	105
6.84	30% CS - DSR vs. CPU utilization (30% CS, 10T, 1L)	105
6.85	40% CS - DSR vs. CPU utilization (40% CS, 10T, 1L)	106
6.86	50% CS - DSR vs. CPU utilization (50% CS ,10T, 1L)	106
6.87	60% CS - DSR vs. CPU utilization (60% CS, 10T, 1L)	107
6.88	70% CS - DSR vs. CPU utilization (70% CS, 10T, 1L)	108
6.89	Normalized CPU Energy vs. CPU utilization (Underload, 0.1 WCET, 5T, NL)	109
6.90	Normalized CPU Energy vs. CPU utilization (Underload, 0.2 WCET, 5T, NL)	109

6.91	Normalized CPU Energy vs. CPU utilization (Underload, 0.3 WCET, 5T, NL)	110
6.92	Normalized CPU Energy vs. CPU utilization (Underload, 0.4 WCET, 5T, NL)	110
6.93	Normalized CPU Energy vs. CPU utilization (Underload, 0.5 WCET, 5T, NL)	111
6.94	Normalized CPU Energy vs. CPU utilization (Underload, 0.6 WCET, 5T, NL)	112
6.95	Normalized CPU Energy vs. CPU utilization (Underload, 0.7 WCET, 5T, NL)	112
6.96	Normalized CPU Energy vs. CPU utilization (Underload, 0.8 WCET, 5T, NL)	113
6.97	Normalized CPU Energy vs. CPU utilization (Underload, 0.9 WCET, 5T, NL)	114
6.98	Normalized CPU Energy vs. CPU utilization (Underload, 1 WCET, 5T, NL)	114
6.99	Normalized CPU Energy vs. ACET (Underload, 50% CPU Utilization 5T, NL)	115
6.100	Normalized CPU Energy vs. ACET (Underload, 60% CPU Utilization 5T, NL)	115
6.101	Normalized CPU Energy vs. ACET (Underload, 70% CPU Utilization 5T, NL)	116
6.102	Normalized CPU Energy vs. ACET (Underload, 80% CPU Utilization 5T, NL)	116
6.103	Normalized CPU Energy vs. ACET (Underload, 90% CPU Utilization, 5T, NL)	117
6.104	Normalized CPU Energy vs. ACET (Underload, 100% CPU Utilization, 5T, NL)	117
6.105	CPU Intensive Workload : System Power vs. CPU Utilization(0.3 WCET, 5T, NL)	118
6.106	CPU Intensive Workload : System Power vs. CPU Utilization(0.6 WCET, 5T, NL)	118
6.107	CPU Intensive Workload : System Power vs. CPU Utilization(0.9 WCET, 5T, NL)	119
6.108	Memory Intensive Workload : System Power vs. CPU Utilization(0.3 WCET, 5T, NL)	119
6.109	Memory Intensive Workload : System Power vs. CPU Utilization(0.6 WCET, 5T, NL)	120
6.110	Memory Intensive Workload : System Power vs. CPU Utilization(0.9 WCET, 5T, NL)	120
6.111	20% CS - Normalized CPU Energy vs. CPU utilization (20% CS, 10T, 1L)	121

6.112	30% CS - Normalized CPU Energy vs. CPU utilization (30% CS, 10T, 1L) .	122
6.113	40% CS - Normalized CPU Energy vs. CPU utilization (40% CS, 10T, 1L) .	122
6.114	50% CS - Normalized CPU Energy vs. CPU utilization (50% CS, 10T, 1L) .	123
6.115	60% CS - Normalized CPU Energy vs. CPU utilization (60% CS, 10T, 1L) .	124
6.116	70% CS - Normalized CPU Energy vs. CPU utilization (70% CS, 10T, 1L) .	124
6.117	80% CS - Normalized CPU Energy vs. CPU utilization (80% CS, 10T, 1L) .	125
6.118	90% CS - Normalized CPU Energy vs. CPU utilization (90% CS, 10T, 1L) .	125
6.119	100% CS - Normalized CPU Energy vs. CPU utilization (100% CS, 10T, 1L)	126
6.120	20% CS - DSR vs. CPU utilization (20% CS, 10T, 1L)	126
6.121	30% CS - DSR vs. CPU utilization (30% CS, 10T, 1L)	127
6.122	40% CS - DSR vs. CPU utilization (40% CS, 10T, 1L)	127
6.123	50% CS - DSR vs. CPU utilization (50% CS, 10T, 1L)	128
6.124	60% CS - DSR vs. CPU utilization (60% CS, 10T, 1L)	128
6.125	70% CS - DSR vs. CPU utilization (70% CS, 10T, 1L)	129
6.126	80% CS - DSR vs. CPU utilization (80% CS, 10T, 1L)	129
6.127	90% CS - DSR vs. CPU utilization (90% CS, 10T, 1L)	130
6.128	100% CS - DSR vs. CPU utilization (100% CS, 10T, 1L)	130

List of Algorithms

1	Static-EDF	18
2	CC-EDF	20
3	Defer Function	21
4	LA-EDF	22
5	Snowdon-min	23
6	DRA	25
7	DRA-OTE	27
8	AGR1	28
9	AGR1 (contd)	29
10	Defer_EUA Function	32
11	EUA	33
12	HS	34
13	DS	36
14	USFI_EDF	37
15	REUA	39

List of Tables

2.1	Actual Implementation of DVFS algorithms	15
3.1	Sample 3 task task-set	17
3.2	Sample 2 task task-set	34
3.3	Slack Estimation Techniques used in the Algorithms	40
5.1	P-States in Intel I5 processor	53
7.1	Normalized Energy savings of RT-DVFS algorithms	133
7.2	Actual Energy savings of RT-DVFS algorithms	134

Chapter 1

Introduction

Energy is an important resource for many computer systems. A significant amount of research has been devoted to computer system power management at various levels of abstraction. For example, techniques such as clock gating [49] and low-power flip-flops [32] are hardware-level techniques that reduce active power during normal operation and reduce leakage power during sleep mode. Dynamic Voltage and Frequency Scaling (DVFS) [26, 51] and Dynamic Power management (DPM) [47, 39] are example techniques that optimize power consumption at the operating system-level. While DVFS involves adjusting the CPU voltage and frequency dynamically to reduce power consumption, DPM involves transitioning devices, including the CPU, into to low-power/sleep states. Compiler-level power management techniques [27] include optimizing the code to reduce its execution time and memory accesses to save power. Application-level power management has also been studied. Examples include doing DVFS from the user space [36], and sending information about the the beginning and end of task from the user space to the OS, so that OS can do DVFS, while maintaining soft real time guarantees [54].

In this thesis, we focus on DVFS, which is based on the following idea:

In most of the CMOS based modern processors, the maximum frequency of operation is dependent on the supply voltage and is given by:

$$f = k \times \frac{(V_{dd} - V_t)^2}{V_{dd}} \quad (1.1)$$

Here, V_{dd} is the supply voltage, f is the clock frequency, V_t is the threshold voltage [59], and k is a constant. Equation 1.1 can be rewritten [19] as:

$$f = a \times V_{dd} \quad (1.2)$$

where a is constant. Thus, frequency has a linear relation with the supply voltage.

When the CPU operates at a frequency f , its active power consumption, denoted as P_{active} ,

is given by:

$$P_{active} = C_{ef} \times V_{dd}^2 \times f \quad (1.3)$$

Here C_{ef} is the effective switch capacitance, V_{dd} is the supply voltage, and f is the clock frequency. Thus, substituting Equation 1.2 in Equation 1.3 we get:

$$P_{active} = \frac{C_{ef}}{a^2} \times f^3, \quad (1.4)$$

which in turn ,is equivalent to

$$P_{active} = S_3 \times f^3, \quad (1.5)$$

where S_3 is constant. Thus, the dynamic power consumption of a CMOS processor is directly proportional to the cube of CPU frequency [24].

Real-Time DVFS(RT-DVFS) is a branch of DVFS, which involves reducing the CPU energy consumption by scaling the CPU frequency, while at the same time, ensuring that task time constraints are satisfied. Broadly, RT-DVFS techniques have two objectives: (i) Reduce energy consumption through DVFS; and (ii) Optimize task timeliness behavior through real-time resource management (i.e., real-time scheduling and synchronization). These two objectives may conflict, because reducing the frequency may increase task execution times, which is antagonistic to timeliness optimization. Most RT-DVFS scheduling algorithms consider satisfaction of time constraints as a “hard constraint.” They often utilize task slack times (i.e., time between task deadlines and worst-case task execution times) to reduce energy consumption, to the extent possible [44, 15, 54, 33].

Figure 1.1: Idle Time Example

Figure 1.1 shows a task T_1 with deadline $4ms$ and a worst-case execution time (WCET) of $2ms$ at a maximum frequency f_{max} (on a frequency-scalable processor). If T_1 runs at f_{max} , then it will complete after $2ms$ and the CPU will be idle for the rest of the $2ms$, as shown in Case 1. The $2ms$ slack time can be exploited toward reducing energy consumption, by

running T_1 at the reduced frequency of $f_{max}/2$, which will then complete the task after $4ms$. The task deadline is still met, but at the same time, energy is also saved. The energy saved is $((0.5^3) * 4) / ((1^3) * 2) = 25\%$.

The utilization of task slack time to minimize the idle time is the fundamental idea used by the vast majority of the RT-DVFS algorithms [44, 15, 54, 33].

There are two types of slack time which are exploited by RT-DVFS algorithms:

(i) Static Slack: This is the idle time interval due to low CPU demand of the application. When the total CPU demand of the application¹ is less than 100%, then there exist time intervals during which the CPU idles. This is the static slack, which can be utilized by RT-DVFS algorithms to scale the CPU frequency.

(ii) Dynamic Slack: This is the slack time that is available when a task completes earlier than its predicted worst-case execution time (WCET), as shown in the Figure 1.2. This slack time can only be obtained when the task completes—i.e., at run-time, in contrast to the static slack, which is known off-line, as task periods and WCETs are presumed to be known off-line for many real-time applications. Once the slack becomes available, it can then be distributed to the other eligible tasks by scaling their frequency, increasing their time budgets, and thus saving energy.

Figure 1.2: Dynamic Slack Example

Most of the RT-DVFS algorithms differ in their techniques used to estimate and utilize the static and dynamic slacks.

1.1 Limitations of Past Real-Time DVFS Research

A number of RT-DVFS algorithms [13, 31, 46, 17, 61, 21, 28, 60, 34, 35] have been developed in the past two decades. These algorithms have been extensively analyzed and their fundamental properties have been established – e.g., schedulable utilization bounds below which they meet all deadlines; conditions under which they satisfy energy budgets. These

¹For a periodic real-time application, this is the ratio of the task period to the worst-case task execution time, aggregated for all tasks.

algorithms have also been experimentally studied using simulations [15, 53, 52, 57, 30] where the primary focus is on understanding their normalized real-time and power consumption behaviours (e.g., normalized to no DVFS; normalized to highest frequency), and relative real-time/power trends. Only a very small fraction of these algorithms have been implemented and evaluated on real hardware platforms [44, 33, 54].

Simulation-based experimental studies have several advantages. First, it provides an effective way to evaluate the performance of an algorithm, especially to understand relative performance trends. Second, it is highly scalable and repeatable: a vast number of experimental settings can be used to evaluate algorithm behaviours, and deterministically repeated, all programmatically. Additionally, it hides platform-specific issues through an abstract (often, discrete-event) simulation model, which significantly reduces development time.

However, simulation-based studies, especially those for RT-DVFS, have drawbacks. Unlike simulators used in the computer architecture community [8, 12, 11, 10], there does not exist OS/platform simulators that have been rigorously evaluated and accepted by the (RT-DVFS) community as a whole. This has resulted in researchers developing their “home grown” simulators with many built-in assumptions (e.g., idle states of CPUs, number of idle states, transition overheads) that are not easy to verify against any particular hardware. Since, the power savings of RT-DVFS are highly platform- (and application-) dependent, this can potentially lead to incorrect conclusions. To illustrate this, we note that, many RT-DVFS works that have relied on simulation-based experimental studies have made the following assumptions: (i) *A CPU has a continuous range of frequencies.* [15, 53, 54] However, actual CPUs do not have a continuous range, but discrete frequency steps. Some processors have a rich frequency set (e.g., 10), such as the Intel I5, and some have a smaller set (e.g., AMD zacate has 3 steps; Via C7 has two steps). Modelling a CPU with continuous range of frequencies can give highly optimistic results, which may not hold on actual hardware.

(ii) *The CPU’s idle state power consumption is negligible.* [15, 53, 54] These works only consider CPU’s dynamic (or active) power consumption, which is assumed to be directly proportional to the cube of frequency (as illustrated by Equation 1.5).

In contrast, CPUs of most modern hardware have idle states (C states) and performance states (P states), and the power consumed by the CPU is the summation of the power consumed in both these states. [1] Abstracting away this detail can lead to significantly optimistic (and sometimes) erroneous power savings, as the thesis shows (see Chapter 6).

(iii) *Non-CPU power consumption is insignificant.*

In most of the RT-DVFS algorithms [15, 44, 54], only the CPU’s power consumption is considered and the system level power consumption is ignored. The interaction between CPU and other components (e.g., memory, bus, disk) is often ignored. This again can lead to erroneous power savings, as non-CPU devices’ power consumption are *DVFS-independent*. Consequently, the overall power savings depend on the power profile of such devices and application workload characteristics – e.g., DVFS may prolong the CPU active state, which

may potentially increase memory power consumption. In [14] Aydin *et al.* have shown that below a particular speed DVFS has a negative impact on the system-level energy consumption and in [48] Snowdon *et al.* show that the optimal voltage and frequency setting is dependent on the system characteristics as well as on application.

In the past, a small subset of RT-DVFS algorithms have been implemented and evaluated on real hardware platforms. Grunwald *et al.* [26] did an actual implementation of DVFS algorithms developed by Weiser [51] in the Linux kernel, running on the Itsy pocket computer. Their main aim was to understand whether the energy saving claims made by these algorithms on simulated platforms give similar results on real platforms as well. They measured the system power consumption and concluded that these algorithms did not give significant energy savings on real hardware platforms. However these DVFS algorithms were not real-time, and Grunwald *et al.* focused on system-level power measurement for the schedulers, which aim at the reduction of CPU power consumption.

Pillai and Shin [44] implemented their RT-DVFS schedulers on a laptop, and measured the power consumption. Although their algorithms aimed at the reduction of CPU power and not the system power, they presented the entire system power consumption of the laptop. In this thesis, we present an accurate method to measure the exact CPU power consumption. For this, we have used the CPUpower tool [5], which measures the CPU power consumption as the sum of the power consumed in the active and the idle state.

Snowdon *et al.* [33] designed and implemented a RT-DVFS scheduler in the OKL4 microkernel on the Gumstix platform. Even though they claim to have measured the system power, they have not reported any result on energy measurements in [33]. This makes it difficult to evaluate the performance of their scheduler.

Yuan *et al.* [54] designed and implemented a statistical DVFS scheduler for multimedia applications called the Grace-OS. However, they have not done real power measurements. Similar to the other simulation based implementations of DVFS algorithms, they have also assumed that CPU power is proportional to the cube of the frequency and have done the CPU energy calculations likewise. We show that this is an inaccurate way of measuring the CPU power. The actual CPU power consumption is very different and depends on the energy consumed in the active as well as the idle state.

1.2 Research Contributions

In this thesis, we study the timeliness and power consumption behaviour of fourteen RT-DVFS schedulers through implementation and actual measurements. The schedulers include Static Earliest Deadline First (Static-EDF), Cycle Conserving Earliest Deadline First (CC-EDF), Look-Ahead Earliest Deadline First (LA-EDF), Snowdon-minimum (Snowdon-min), Resource-constrained Energy-Efficient Utility Accrual Algorithm (REUA), Dynamic Reclaiming Algorithm (DRA) and Aggressive Speed Reduction Algorithm (AGR) among the

others. Static-EDF utilizes static slack, whereas CC-EDF, LA-EDF, DRA and AGR utilize dynamic slack as well. LA-EDF, AGR and DRA are more aggressive as compared to the other algorithms as they try to reduce the frequency as much as possible, while still satisfying task time constraints. We implemented the schedulers in a Linux-based real-time kernel called ChronOS [4], and measured their real-time/power behaviours on two representative hardware platforms including ASUS laptop with the Intel I5 processor and a motherboard with the AMD Zacate processor.

We used a synthetic application (similar to the studies in [22]) that allowed us to generate a broad range of workload conditions including CPU-intensive, memory-intensive, mutual exclusion lock-intensive, and processor-underloaded and overloaded workloads. We measured the actual CPU power by accounting for the power consumption in the active and idle states, and also the system power using a multimeter. We also measured the normalized CPU energy² consumption, where CPU power is considered to be proportional to the cube of the frequency, so as to compare with the simulated results of the algorithms.

We draw the following conclusions from our experimental study:

- (1) Our studies reveal that measuring the CPU power consumption as the cube of CPU frequency can lead to incorrect conclusions. In particular, it ignores the idle state CPU power consumption, which is orders of magnitude smaller than the active power consumption. Consequently, power savings obtained by exclusively optimizing active power consumption (i.e., RT-DVFS) may be offset by completing tasks sooner by running them at the highest frequency and transitioning to the idle state earlier (i.e., no DVFS). Thus, the active power consumption savings of the RT-DVFS techniques' that we report are orders of magnitude smaller than their simulation-based savings reported in the literature.
- (2) From our study, we observed that, algorithms such as Static-EDF, CC-EDF, and Snowdon-min, which have been reported to outperform Base-EDF (on power savings) do not actually do so. These algorithms outperform Base-EDF on normalized CPU energy. However, they consume only slightly lesser CPU power and system power as Base-EDF, or in some cases, even more.
- (3) Aggressive energy saving algorithms such as LA-EDF, DRA, DRA-OTE, AGR1, and AGR2 do consume less actual CPU power and system power than Base-EDF. But their energy savings are not as high as reported in past simulation-based studies.
- (4) We also observe, that in overloads, aggressive algorithms like AGR1, AGR2 and LA-EDF save the maximum power.
- (5) Lock based algorithms consume almost similar CPU power, with REUA performing slightly better.
- (6) Energy savings of a RT-DVFS algorithm is highly dependent on the number of frequency

²Normalized CPU Energy is the same as Normalized CPU power, as the time factor gets cancelled in both the numerator and the denominator.

steps available on the processor. The energy savings obtained on the Intel I5 platform with 10 frequency steps was much higher than the energy savings obtained on the AMD Zacate platform with 3 frequency steps.

To the best of our knowledge, this is the first implementation and actual real-time/power measurement-based experimental study of the aforementioned RT-DVFS schedulers, and constitutes the thesis contribution.

1.3 Thesis Organization

The rest of the thesis is organized as follows: In Chapter 2, we review the past DVFS algorithms which have been implemented on real hardware platforms as well as on simulated platforms. We also compare and contrast our work with those algorithms which overlap with the thesis problem space. In Chapter 3, we describe the RT-DVFS scheduling algorithms which we have implemented and evaluated. In Chapter 4, we describe ChronOS real-time Linux, and our DVFS extensions to it. Chapter 5 discusses the experimental methodology and Chapter 6 reports the results. The thesis concludes in Chapter 7.

Chapter 2

Related Work

Significant amount of research has been done in the field of DVFS. A large number of algorithms have been designed which can be broadly classified into three categories. This classification was made by Yuan et al in [55] as follows:

(i) *real-time DVFS (RT-DVFS)* : RT-DVFS algorithms are designed for real-time systems and aim at saving energy while maintaining hard real-time constraints. They scale the CPU frequency based on the worst case execution times of the real-time application. Most of the RT-DVFS algorithms differ in their techniques to utilize the static slack available due to the low CPU utilization of the application or dynamic slack available due to the actual execution time being much lesser than the worst case execution time of the real-time application.

(ii) *General Purpose DVFS (GP-DVFS)* : GP-DVFS algorithms designed for general purpose systems aim at saving energy without degrading performance and are mostly suited for best-effort applications. This class of algorithms scale CPU frequency based on the workload prediction which in turn depends on the average CPU utilization. GP-DVFS algorithms are based on two techniques - prediction and speed setting. Prediction involves predicting the future workload while speed setting involves deciding the speed at which to run. These are interval based schedulers where the prediction and speed setting decisions are made for every interval so as to minimise the idle time in that interval.

(iii) *Statistical DVFS* : Both GP-DVFS and RT-DVFS algorithms are not suited for multimedia applications as their run time demand varies. The interval based approach of GP-DVFS algorithms may not satisfy the timing constraints whereas the worst case approach based RT-DVFS algorithms might not be that energy efficient. Thus, for multimedia applications, Statistical DVFS algorithms have been designed to deal with run time demand variations. The core idea of Statistical DVFS is to change the CPU speed based on the demand distributions of the applications. These algorithms involve either online or off line profiling of the applications to get the probability distribution of the cycle demand of the applications and make scheduling and frequency scaling decisions based on that.

In the rest of the chapter, we will review the DVFS algorithms which have been implemented on real hardware platforms as well as on simulated platforms in the above mentioned three categories. We have an overlapping problem space with the algorithms that have been implemented in real-time, so we are going to compare and contrast only with them. But at the same time, we will also review the algorithms that have just been simulated for completeness.

2.1 GP-DVFS

The concept of DVFS was introduced by Weiser in [51]. He devised three interval based schedulers PAST, FUTURE and OPT which differ in their prediction technique. In PAST, the workload of the current interval is assumed to be same as that of the previous interval, in OPT, the exact workload is known for the entire duration and in FUTURE, the workload is known for a small interval in the future. As can be understood, OPT and FUTURE algorithms are impractical as the future workload can never be predicted. Nevertheless, this ground breaking paper led the way for the multitude of DVFS research which took place in the coming decades. They evaluated these algorithms by analysing the traces collected by running applications on UNIX based workstations.

Govilet *al.* [23] developed a few more dynamic clock scaling algorithms with different prediction and speed-setting techniques. They developed six new algorithms, where each algorithm employ different techniques to predict the future workload. For example, in FLAT, prediction is done based on global average of the computational load, in LONG_SHORT it is based on the mean of the global and local average of the workload, while in AGED_AVERAGES, weighted average is considered, in CYCLE, cyclic behaviour of CPU utilization is considered, in PATTERN, prediction is done on the basis of pattern matching with a previous occurrence, whereas in PEAK, prediction is based on the expected value of the utilization, considering narrow peaks in its pattern. The speed setting policy involved reducing the frequency, when the utilization is low and the CPU is idle and increasing it, when the utilization is high and the CPU is active. They used the same traces as Weiser to evaluate their algorithms and found that PEAK performed the best.

Both Weiser and Govil considered only the CPU power consumption and its linear relationship with the CPU frequency. On the contrary, Martin in [40] studied the relation between system level performance and CPU frequency scaling. He took into account non linear relation between the total system power and the CPU frequency, non ideal properties of the batteries, and the memory bandwidth to evaluate the performance of the clock scaling algorithms. He modified Weiser's PAST algorithm taking into account the above mentioned non-ideal characteristics and evaluated the algorithm using the same traces as Weiser did. He concluded that reduction of the energy consumption is a system level problem, and all the above mentioned factors should be considered for devising a clock scaling algorithm.

Instead of using post simulation traces to analyse the performance of the clock scaling algorithms, Pering *et al.* [43] used simulated systems to do so. They implemented the clock scaling algorithms devised by Govil [23] and Weiser [51] on a simulated platform and evaluated it by running the MPEG decoder with QoS requirements. These algorithms when simulated resulted in 46% energy savings.

Grunwald *et al.* [26] did an actual implementation of interval based GP-DVFS algorithms in the Linux kernel, running on the Itsy pocket computer with the StrongARM SA-1100 processor. They implemented the algorithms proposed by Weiser [51], such as PAST and variations of AVGn, and evaluated them by running realistic workloads for handheld systems. In these algorithms CPU scaling decisions are made based on the average utilization, with the goal to minimise the idle time.

Their main aim was to find whether the energy saving claims made by these algorithms on simulated platforms give similar results on real platforms as well. They measured the system power consumption, and concluded that these algorithms did not give significant energy savings when implemented on real hardware platforms. They also concluded that in order to make efficient energy management decisions, operating system has to be aware of the properties of the application.

They contemplate that the poor performance of these algorithms is due to the voltage scaling limitations of the platform used, as well as due to the poor efficacy of the algorithms. However, this poor performance could also be due to the minimisation of idle time. This is because if the processor consumes very low power in the idle state as compared to the active state then the energy consumption on running at a lower frequency for a longer time can be greater than the energy consumption on running at a higher frequency for a shorter time. This is exactly what we have observed from our study.

In PACE [37], the authors have modified the algorithms developed by Weiser and Govil to further reduce their energy consumption without affecting their performance. They have shown that by modifying the way the scaling algorithm schedules tasks, the notion of deadline can be introduced, and this deadline information can be used by these algorithms to further reduce the energy consumption. In PACE, a speed schedule is created, where the speed is increased as the deadline approaches. The authors claim that they obtained around 20% more energy savings than the algorithms which they improved. They have evaluated their modified algorithms by doing simulations using real workloads.

In [41], Miyoshi *et al.* introduced the concept of critical power slope. They have used it as a metric to show, that the relation between power and frequency is dependent on the architecture of the processor, particularly on the relative energy efficiency of the idle states and the active state. They have shown that on Pentium based systems it is more power efficient to run at highest frequency, while in PowerPC based systems it is more power efficient to run at the lowest frequency. The results of this paper is highly relevant to our work as this explains why we did not get similar energy savings on implementing RT-DVFS algorithms on pentium based platform as claimed by their simulations.

2.2 RT-DVFS

A multitude of RT-DVFS algorithms [13, 31, 46, 17, 61, 21, 28, 60, 34, 35] have been developed in the past two decades. We review a subset of them in this section.

Pillai and Shin devised 5 RT-DVFS algorithms and found that the EDF based schedulers outperform the RMA based ones. Among the EDF based ones, Static-EDF utilizes the static slack, whereas CC-EDF and LA-EDF reclaim the dynamic slack to scale the CPU frequency. These algorithms were one of the earliest ones to be implemented on a real hardware platform. As suggested by their class, these algorithms are designed to reduce the energy consumption while ensuring 100% deadline satisfaction ratio. By this work authors have demonstrated that RT-DVFS algorithms can indeed save energy not only on modelled near idle simulated platforms, but also on real hardware platforms. The platform used by them was a Hewlett-Packard N3350 notebook computer, with a AMD K6-2+ processor, which uses the powernow! technology to scale the voltage and the frequency on-the-fly. In order to implement these schedulers, they modified the Linux 2.2.16 kernel, particularly its real-time scheduler and task management services.

Based on their work they have made some interesting conclusions. They conclude that the energy savings of a RT-DVFS algorithm is highly dependent on the voltage and the frequency settings available on the hardware, and not much on the number of tasks in a task-set or on the energy efficiency of the idle states of the CPU. In contrast with their result the result obtained from our study is different. We have observed that the performance of a RT-DVFS algorithm is highly dependent on the energy efficiency of the idle states of the processor.

The scope of their work is limited to independent underloaded CPU-intensive workloads. In contrast we have implemented and evaluated schedulers designed for independent underloaded workloads, and for dependent overloaded and underloaded workloads. Additionally, we have also implemented these schedulers on two different hardware platforms, and tested them with both CPU-intensive and memory intensive workloads.

RT-DVFS algorithms designed in this paper aim at the reduction of CPU power and not the system power. Accordingly, CPU power consumption is the proper metric to evaluate the performance of these algorithms. However, in [44] the authors have measured the entire system power. This might lead to incorrect evaluation of these algorithms as the system power also includes the power consumed by the memory, I/O devices and other subsystems as well. We were able to measure the actual CPU power consumption of the CPU by using the CPUpower tool [5], and thus correctly evaluate and compare the performance of these algorithms.

Aaydin *et al.* in [15] devised DRA and AGR, which were based on priority based slack stealing [31]. While DRA reclaimed unused slack based on actual workload and reduced the CPU frequency, in addition to that, AGR used average workload information to predict the early completion of the future workloads and thus obtain extra slack to further reduce the CPU

energy. However, they have evaluated the performance of these algorithms just on the basis of simulations. In their simulations, they have modelled an ideal CPU having continuous range of frequencies and considered CPU power to be proportional to the cube of average frequency. Due to these limitations, the actual energy savings obtained by implementing them on a real hardware platform do not match up with the savings obtained on simulated platforms.

In [52], Haisang *et al.* developed a utility accrual RT-DVFS algorithm suited for underloaded as well as overloaded non-dependent CPU intensive applications, subjected to TUF time constraints. This algorithm aims at accruing maximum utility per unit of energy consumed and the reduction of total system power. During the underloads it accrues the maximum utility and scales the frequency in a similar way as LA-EDF [44], whereas in overloads it accrues the maximum utility possible by running the tasks at the maximum speed. In [53], they extended their work to include applications that share resources and are subjected to mutual exclusion constraints. REUA defaults to EUA when subjected to non-dependent workloads. They have used the priority inheritance protocol to resolve the deadlocks and bound the blocking time. This work is also based on simulations.

In [57], Zhang *et al.* developed RT-DVFS algorithms for task-sets with non-preemptible blocking sections. They have used Stack Resource Policy (SRP) [16] to bound the blocking time. In the static speed algorithm High Speed (HS) a constant speed is selected based on the static slack available, and on the feasibility of the task-set scheduled with EDF under SRP. Dual Speed Algorithm (DS) extends the high speed algorithm by operating at high speed as well as at low speed, whenever possible. In DSDR, the dynamic slack is reclaimed and redistributed to further reduce the energy. They have also simulated these algorithms and shown that the dynamic algorithms can save 80% more energy than the static algorithms.

In [30], Rajkumar *et al.* have devised RT-DVFS algorithms to support synchronization of tasks, for access to shared resources. These algorithms are based on the computation of slowdown factors for the tasks which need to synchronise. Any Resource Access Protocol like SRP, PCP or PIP can be used to bound the blocking time. This algorithm involves frequency inheritance where a low priority job can inherit the frequency of the highest priority job that it has blocked. They have done simulations and reported 25% energy gains over other slowdown techniques.

Snowdon *et al.* [33], designed and implemented a DVFS algorithm aiming at the reduction of not only the CPU power consumption, but also the power consumption of other subsystems particularly the memory. Further, their work is based on the fact that for efficient energy management the operating system should have the knowledge of the properties of the platform, as well as that of the application. Accordingly, they have developed a time energy model, which uses this knowledge to calculate the time and the energy required by a piece of software at a particular CPU, memory and the bus frequency. They integrated this time and energy model with the real-time dynamic slack based scheduler RBED, designed by Brand *et al.* [18]. This scheduler uses the the time-energy model to select a feasible set of frequencies

that minimizes the total energy consumption while maintaining hard real-time constraints. They have implemented this scheduler inside the OKL4 microkernel on the Gumstix platform with the XScale PXA255 processor. They have measured the system power consumption, which is appropriate in their case as they aim to reduce the entire system power.

One important conclusion from their work is that the implementation of their scheduler in a microkernel is not the best approach. This is because, the performance of a microkernel is dependent on the small overhead of context switches and a DVFS scheduler introduces large overheads due to the complexity of the algorithm.

Even though theirs is one of the earliest effort to do a real implementation of a DVFS scheduler that takes into account the memory power consumption along with the CPU power consumption, they have not reported any result on energy measurements or real-time measurements. This makes it difficult to evaluate the performance of this scheduler. Another limitation of their work is that, they have implemented their algorithms on the Gumstix platform which doesn't support voltage scaling. The energy savings of a DVFS algorithm is highly dependent on the scaling of the voltage. Frequency scaling just by itself cannot reduce the energy.

In this paragraph we will review some of the very recent works in the RT-DVFS space. In [17], Bini *et al.* have considered the discrete speed levels of CPU as well as switching overhead to devise their RT-DVFS algorithm. Whereas in [60] and [61], Zhuo *et al.* have modelled a system consisting of a DVFS processor and other devices and considered the reduction of both the CPU power and the device power. In [21], the authors have taken into account the non-linear relationship between frequency and system power, leakage power and intra-task overheads. In [58], Zhang *et al.* have developed a procrastination based DVFS algorithm in which they have used stochastic workloads. In [35] and [34], the authors have created RT-DVFS algorithms for energy harvesting systems. In [56] similar to Snowdon's work [33], Yun *et al.* have based their work on saving energy of various subsystems by scaling frequency of the CPU, system bus and memory. In [28], Hung *et al.* have also considered systems with DVFS and non-DVFS elements. In [50], Wang *et al.* have devised a preemptive DVFS technique, where a frequency value is scaled on every preemption. They have shown that it can save 24% more energy than the inter-task DVFS algorithms.

2.3 Statistical DVFS

Yuan *et al.* [54], designed and implemented a statistical DVFS scheduler for multimedia applications called the Grace-OS. This algorithm copes with the dynamic cycle demands of a multimedia application, providing soft real-time guarantees while saving energy. They have implemented this scheduler in the Linux kernel 2.4.18, and evaluated it on a HP Pavilion N5470 laptop with a AMD Athlon processor. The authors report energy saving of 7% to 72% as compared to deterministic scheduling.

Even though they have implemented their algorithm on a real hardware platform, they have not done real power measurements. Similar to the other simulation based implementations of DVFS algorithms, they have also assumed that CPU power is proportional to the cube of the frequency and have done the CPU energy calculations likewise. On the contrary, our work shows that this is an inaccurate way of measuring the CPU power. The actual CPU power consumption is very different and depends on the energy consumed in the active as well as the idle state.

In [45], Poulwese *et al.* implemented a power aware video decoder on a LART device [29] with the StrongARM SA-1100 processor. They modified the decoder such that, the decoder can change the frequency from the user-space depending on the decoding time of the frame. They showed that they obtained greater energy savings compared to fixed frequency and interval based schedulers.

Choi *et al.* [20] designed and implemented a DVFS algorithm for MPEG decoders, which aimed at maintaining the Quality of Service while reducing energy. In this, they compute the workload to decode the frame and accordingly scale the voltage and frequency. They have also implemented their algorithm on a StrongArm-1110 based evaluation board.

In [55] Yuan *et al.* have implemented another statistical DVFS algorithm for multimedia applications. However, this algorithm has been devised considering the characteristics of a non-ideal CPU such as discrete speed levels and the energy consumption of the CPU in the idle state. The algorithm aims to minimize the system energy consumption rather than that of the CPU only. This algorithm known as PDVS, has been implemented as a part of Grace-OS and been evaluated on a HP N540 laptop with an Athlon CPU running the Linux kernel. The authors claim that, PDVS reduces the total energy consumption of the laptop by 15%-38% as compared to non-DVFS algorithms and around 10% compared to other DVFS algorithms, which assume ideal CPU.

In [25], the authors have developed a statistical DVFS algorithm which uses stochastic data to make CPU scaling decisions. This algorithm saves energy while meeting all the deadlines, and thus it is well suited for hard real-time systems. However, the performance of this algorithm is evaluated just on the basis of simulations.

2.4 Summary

In the Table 2.1, we list those DVFS algorithms which have been implemented on a hardware platform, as these are the ones which are relevant to the scope of our work.

Table 2.1: Actual Implementation of DVFS algorithms

Name	Class	Task_Model	Workload	Platform	System/CPU power
<i>Static-EDf</i> , <i>CC-EDF</i> , <i>LA-EDF</i> [44]	RT-DVFS	No-Dependency	CPU Intensive	Hewlett-Packard N3350 notebook computer, with a AMD K6-2+ processor	CPU Power
<i>RBED-DVFS</i> [33]	RT-DVFS	No-Dependency	CPU and Memory Intensive	OKL4 microkernel on the Gumstix platform with the XScale PXA255 processor	System Power
<i>PAST</i> , <i>AVGn</i> [26]	GP-DVFS	No-Dependency	CPU Intensive	Itsy pocket computer with the StrongARM SA-1100 processor	CPU Power
<i>GRACE-OS</i> [54]	Statistical-DVFS	No-Dependency	Multimedia Applications	HP Pavilion N5470 laptop with a AMD Athlon processor	CPU Power
<i>Power aware video decoder</i> [45]	Statistical-DVFS	No-Dependency	Multimedia Applications	LART device [29] with the StrongARM SA-1100 processor	CPU Power
<i>PDVS</i> [55]	Statistical-DVFS	No-Dependency	Multimedia Applications	HP Pavilion N5470 laptop with a AMD Athlon processor	CPU Power

Chapter 3

Algorithms

In this thesis, we evaluate and compare the performance of 14 RT-DVFS schedulers on two representative hardware platforms. In this chapter we will describe these schedulers in details. These schedulers can be classified into the following categories: (i) *Schedulers for Independent Underloaded task-sets*, (ii) *Schedulers for Dependent Underloaded task-sets* and (iii) *Schedulers for Overloaded task-sets*

All RT-DVFS schedulers have to make two decisions- (i) which task to run and (ii) which frequency to run it at. Generally, all RT-DVFS schedulers are EDF based, i.e they schedule the task with the earliest deadline. They differ from each other in the way, they estimate the [31] slack to scale the frequency. There are two kinds of slacks available, static slack, which is available due to the characteristic of the task-set itself, such as less than 100% CPU utilization, and dynamic slack, which is available due to variations in the execution time. These two slack estimation techniques can be further classified. This classification was made by Kim *et al.* in [31] as follows:

(i) Static Slack Estimation:

(a) *Maximum Constant Speed*: In this the utilization of the task-set is taken into account to decide on a static speed, for all the tasks in the task-set such that the task-set is feasible under EDF.

(ii) **Dynamic Slack Estimation**: Dynamic Slack arises due to the variations in the execution times of the tasks in the task-set at run time.

(a) *Stretching to NTA*: The maximum constant speed is decided on the basis of the WCETs of the tasks in the task-set. But when the actual execution time of the task is much lesser than its WCET, then dynamic slack arises. One way to estimate the dynamic slack is to utilize the time till the arrival of the next task, to scale the frequency. The time of arrival of the next task is called NTA.

(b) *Priority based stack stealing*: In this technique, the dynamic slack obtained due to the

early execution of the higher priority task is transferred to the lower priority task, which is to be executed next.

(c) *Utilization Updating*: In this technique, the worst case utilization of the processor is updated taking into consideration the actual execution times of the task which got completed. This modified utilization value is then used for scaling the CPU frequency.

In Table 3.3 we summarize the various techniques used in the fourteen RT-DVFS algorithms described below.

Example task-set

We will describe the algorithms with the help of an example task-set. Let us consider a three task periodic task-set with tasks T_1, T_2 and T_3 whose characteristics are described by the Table 3.1.

Table 3.1: Sample 3 task task-set

Task	WCET	Actual Time	Period
T_1	2	1.6	5
T_2	1	0.8	5
T_3	3	2.4	15

3.1 Schedulers for Independent Underloaded task-sets

3.1.1 Base-EDF

This is the basic EDF scheduler from [44] which doesn't involve any frequency scaling and operates at the maximum frequency. The task with the earliest deadline is scheduled and all the task run at the maximum frequency which in the case of I5 is 2400 MHz. We have included this in our experiments for comparison. The Figure 3.1 shows the schedule of the example task-set under this algorithm.

¹

¹We will call the task with the earliest deadline as T_{best} in all the remaining algorithms unless specified otherwise.

Figure 3.1: Base-EDF

Algorithm 1: Static-EDF

```

1: WCET:  $C_1, C_2, \dots, C_N$ 
2: Period:  $P_1, P_2, \dots, P_N$ 
3: Frequencies:  $f_1, f_2, \dots, f_m$ 
4:
5: Utilization =  $C_1/P_1 + C_2/P_2 + \dots + C_n/P_n$ 
6: k =  $f_m \cdot \text{Utilization}$ 
7: Select  $f \supseteq f_1, f_2, \dots, f_m$  such that  $f \leq k$ 

```

3.1.2 Static-EDF

This is the Static-EDF scheduler from [44]. This uses the static slack estimation technique to scale the CPU frequency. In this, the frequency is scaled based on the static utilisation value of the task-set, as can be seen from the Algorithm 1. All the tasks in the task-set are run at the same frequency such that the utilization of the processor at the scaled frequency, becomes 1. Thus this algorithm ensures, that no deadlines gets missed, as the utilisation is still less than equal to 1. This algorithm aims to minimise the idle time as can be seen from the schedule in Figure 3.2.

From Algorithm 1, we see that the new frequency is decided by scaling the maximum frequency by the utilization value of the task-set in line 6. As a non-ideal processor has discrete frequencies, a frequency less than equal to k is selected to run the selected task.

Figure 3.2: Static-EDF

3.1.3 CC-EDF

Cycle Conserving EDF or CC-EDF utilizes the dynamic slack to scale the CPU frequency. When a task is released, then a conservative approach is taken and it is assumed that the task is going to take its WCET to execute, and the frequency is set accordingly. However, on completion, if the actual execution time is lesser, then the extra unused cycles can be transferred to the remaining tasks. As the remaining tasks now get more cycles than they require to execute, the frequency can now be scaled.

The working of the actual algorithm has been illustrated in the Figure 3.3.

In Algorithm 2 the pseudo code of CC-EDF is described.

When the task T_i begins, we take the conservative approach considering that it will take its WCET to execute, and set the U_i as C_i/P_i in line 9 and compute the total utilization using this U_i in line 12. This total utilization value is then used to decide the frequency in line 22. When the task completes its current invocation, then we compute task T_i 's load during that invocation using actual execution time. If the actual execution time is considered to be AC_i then $U_{load} = AC_i/P_i$. As T_i doesn't take more than AC_i to execute, AC_i can now be considered as the WCET of T_i and the utilization can be recomputed using U_{load} in 19. $AC_i \leq C_i$ results in $U_{load} \leq U_i$ and thus the total U decreases and the frequency gets scaled.

Algorithm 2: CC-EDF

```

1: WCET:  $C_1, C_2, \dots, C_N$ 
2: Period:  $P_1, P_2, \dots, P_N$ 
3: Frequencies:  $f_1, f_2, \dots, f_m$ 
4:
5:  $U = C_1/P_1 + C_2/P_2 + \dots + C_n/P_n$ ;
6: if Task  $T_i$  begins then
7:   if ( $U_{prev} == 0$ ) then
8:      $U_{prev} = U$ ;
9:      $U_i = C_i/P_i$ ;
10:   if ( $U_{prev} == 0$ ) then
11:      $U_{prev} = U_i$ ;
12:    $U = U_{prev} - U_{iprev} + U_i$ ;
13:    $U_{prev} = U$ ;
14:
15: if Task  $T_i$  ends then
16:   if ( $U_{prev} == 0$ ) then
17:      $U_{prev} = U$ ;
18:      $U_{load} = AC_i/P_i$ ;
19:      $U = U_{prev} - U_i + U_{load}$ ;
20:      $U_{iprev} = U_{load}$ ;
21:      $U_{prev} = U$ ;
22:  $k = f_m * U$ ;
23: Select  $f \supseteq f_1, f_2, \dots, f_m$  such that  $f \leq k$ 

```

3.1.4 LA-EDF

Look-Ahead EDF or LA-EDF is the most energy efficient RT-DVFS algorithm devised by Shin [44]. Similar to the previous scheduler, this scheduler is also based on the dynamic slack reclaiming and the utilization updation technique. This is an aggressive algorithm which tries to work at the lowest frequency possible. It tries to do minimum work before the earliest deadline by pushing as much work as it can beyond that deadline, but at the same time making sure that the future deadlines are met, even if it has to run at a higher frequency in future.

The Algorithm 4 describes the pseudo-code for LA-EDF.

Whenever the scheduler is invoked (i.e. on task release or completion) the defer function (described in Algorithm 3) is called. In this function, the time interval till the earliest deadline is considered and the minimum amount of work that each task has to do in this interval (which is denoted in time as x in our example), so as to prevent any future deadline misses, is calculated.

In the for loop [10-15], tasks are considered in the order of their decreasing deadlines and the minimum amount of work required to be done by each task in this interval is calculated

Figure 3.3: CC-EDF

Algorithm 3: Defer Function

```

1: Procedure:Defer ()
2:
3: Input:  $\sigma_T$  // List of released tasks
4: WCET:  $C_1, C_2, \dots, C_N$ 
5: Period:  $P_1, P_2, \dots, P_N$ 
6: Frequencies:  $f_1, f_2, \dots, f_m$ 
7:
8:  $U = C_1/P_1 + C_2/P_2 + \dots + C_n/P_n$  ;
9:  $s = 0$ ;
10: for each task  $T_i$  in  $\sigma_T$  in decreasing deadline order do
11:    $U = U - C_i/P_i$ ;
12:    $x = \max(0, \text{Cleft}_i - (1 - U)(D_i - D_n))$ ;
13:    $U = U + (\text{Cleft}_i - x)/(D_i - D_n)$ ;
14:    $s = s + x$ ;
15:
16:  $f = s/(D_n - \text{current\_time})$ ;

```

and summed to obtain the total work. So s becomes the total amount of minimum work, and $D - \text{current_time}$ becomes the interval in which the work has to be completed. Using these values the frequency is calculated in line 16.

The schedule using the example task-set is shown in the Figure 3.4. It can be seen that the tasks are run at a higher frequency in the later interval to prevent deadline misses.

Figure 3.4: LA-EDF

Algorithm 4: LA-EDF

```

1: WCET:  $C_1, C_2, \dots, C_N$ 
2: Period:  $P_1, P_2, \dots, P_N$ 
3: Frequencies:  $f_1, f_2, \dots, f_m$ 
4:
5:  $U = C_1/P_1 + C_2/P_2 + \dots + C_n/P_n$ ;
6: if Task  $T_i$  begins then
7:    $U_{prev} = U$ ;
8:    $Cleft = C_i$ ;
9:    $k = \text{Defer}()$ ;
10:
11: if Task  $T_i$  ends then
12:    $Cleft = 0$ ;
13:    $k = \text{Defer}()$ ;
14: During execution of  $T_i$  Decerement  $Cleft$ ;
15:  $k = f_m * U$ ;
16: Select  $f \supseteq f_1, f_2, \dots, f_m$  such that  $f \leq k$ 

```

3.1.5 Snowdon-min

This is the RT-DVFS scheduler implemented by Snowdon. We call this Snowdon-min as this is not the full version of Snowdon's RT-DVFS scheduler. Here we do not consider the scaling of the memory or the bus frequency, but only that of the CPU frequency. In this algorithm the amount of dynamic slack produced by the completed task is added to the budget of the next runnable task, and its frequency is scaled accordingly.

This algorithm is described in Algorithm 5.

Algorithm 5: Snowdon-min

```

1: WCET:  $C_1, C_2, \dots, C_N$ 
2: Period:  $P_1, P_2, \dots, P_N$ 
3: Budget:  $B_1, B_2, \dots, B_N$ 
4: Frequencies:  $f_1, f_2, \dots, f_m$ 
5: The next runnable task selected by EDF:  $T_{best}$ 
6:
7:  $U = C_1/P_1 + C_2/P_2 + \dots + C_n/P_n$  ;
8: if Task  $T_i$  begins then
9:    $B_i = C_i/S_{optimal}$  ;
10:
11: if Task  $T_i$  ends then
12:    $UC_i = B_i - AC_i$  ;
13:    $left_{best} = B_{best} - AR_{best}$  ;
14:    $B_{best} = B_{best} - AR_{best} + UC_i$  ;
15:    $k = left_{best}/B_{best}$  ;
16: Select  $f \supseteq f_1, f_2, \dots, f_m$  such that  $f \leq k$ 

```

Whenever a task T_i begins, it is allotted a budget (in time units) equal to $C_i/S_{optimal}$ as in line 8. This is the time T_i will take to complete when it is run at the speed, $S_{optimal}$ which is the constant speed decided by the Static-EDF algorithm. When the tasks completes its current invocation, the unused time is calculated as in line 12 where AC_i is the actual execution time of the task T_i . In line 13 the remaining WCET of T_{best} at the speed $S_{optimal}$ is calculated, where B_{best} is the time budget allotted to T_{best} and AR_{best} is the time for which T_{best} has already executed. In line 14 the unused time from T_i is added to the remaining budget of T_{best} . In line 15 the frequency is calculated by dividing the remaining WCET of T_{best} by its available budget.

The schedule for the example task-set is shown in the Figure 3.5.

3.1.6 DRA

Dynamic Reclaiming Algorithm (DRA) is based on dynamic slack reclaiming and has been devised by Aydin *et al.* in [15]. In this algorithm a data structure called α queue is maintained. Whenever a task arrives, it pushes its worst case execution time at $S_{optimal}$ in the α queue. Thus each item of the α queue is characterized by the deadline D_i of the task T_i , and the remaining worst case execution of T_i under the speed $S_{optimal}$ which is denoted as rem_i . This alpha queue is ordered according to EDF* priority. (EDF* is similar to EDF, except that, if two tasks have the same deadline, then among the two, the task which has arrived earlier will have a higher priority.) With the progress of time, the rem_i field of the head of the α queue is subtracted with the elapsed time since the last scheduling event. If rem_i is smaller than the time elapsed, then after updating the rem_i of the head, the head is deleted and the update continues with the new head, till the entire elapsed time is used

Figure 3.5: Snowdon-min

up. Now whenever a task is selected for execution, the α queue is checked and the rem_i field of all the α queue items having a deadline lesser than or equal to T_{best} is added to the rem_i of T_{best} . Thus in this way, the unused slack time of all those tasks which got completed earlier than the scheduled task is transferred to the scheduled task and the frequency is scaled accordingly.

We will discuss the DRA algorithm in details in the following sections.

Auxiliary Functions

In this section, we define the auxiliary functions that are used by DRA.

Initialize_α_member(D_i, rem_i)

Initialize the fields of a *alpha_member* with the deadline D_i and the remaining WCET under $S_{optimal}$ rem_i of the arrived task.

Insert_Into_α_Queue(α_j, α_queue)

Insert this α_j into the α_queue at the deadline position.

Update_α_Queue($time_diff, \alpha_queue$)

Let $time_diff$ be the time elapsed since the last scheduling event. This $time_diff$ is subtracted from the rem_i field of the head of the α queue. If the rem_i is smaller than $time_diff$, then after updating the rem_i of the head, the head is deleted and the update continues with the new head, till the entire elapsed time is used up.

Algorithm 6: DRA

```

1: WCET:  $C_1, C_2, \dots, C_N$ 
2: Period:  $P_1, P_2, \dots, P_N$ 
3: Frequencies:  $f_1, f_2, \dots, f_m$ 
4:
5:  $U = C_1/P_1 + C_2/P_2 + \dots + C_n/P_n$  ;
6: if Task  $T_i$  begins then
7:   Initialize_alpha_member( $D_i, rem_i$ );
8:   time_diff = Compute_Time_Elapsed();
9:   Update_alpha_Queue(time_diff,  $\alpha\_queue$ );
10:  Insert_Into_alpha_Queue( $\alpha_j, \alpha\_queue$ );
11:
12: if Task  $T_i$  ends then
13:   Compute_Time_Elapsed();
14:   Update_alpha_Queue(time_diff,  $\alpha\_queue$ );
15:  $E = \text{Calculate\_Dynamic\_Slack\_Available}(T_{best}, \alpha\_queue)$ ;
16:  $Rem\_WCET_{best} = \text{Calculate\_Remaining\_WCET}(T_{best})$  ;
17:  $k = Rem\_WCET_{best}/E$  ;
18: Select  $f \supseteq f_1, f_2, \dots, f_m$  such that  $f \leq k$ 

```

Compute_Time_Elapsed()

Return the time elapsed since the last scheduling event.

Calculate_Dynamic_Slack_Available(T_{best}, α_queue)

This function calculates dynamic slack available to the selected task T_{best} . For any task T_i the dynamic slack is given by $E_i = \sum_{d_j \leq d_i} rem_j$ i.e it is obtained by summing up the rem_j fields of all the α members whose deadline field has a value lesser than or equal to the deadline of T_i

Calculate_Remainig_WCET(T_i)

Calculate the remaining WCET rem_i of T_i under $S_{optimal}$

Calculate_NTA()

Return the time of arrival of the next task .

Algorithm 6 describes the pseudo-code for DRA. This pseudo-code uses auxiliary functions that have been described in 3.1.6.

When the task T_i arrives, an alpha member's deadline and rem_i fields are initialized with D_i and rem_i of T_i in line 7. In line 8 time elapsed since the last scheduling value is calculated and stored in the variable time_diff. Using this time difference the α queue is updated as explained above. The insertion of the initialized α member is done in line 10. When the task T_i ends, the time elapsed is calculated in line 13 and the α queue is updated in line 14.

In line 15 dynamic slack available to T_{best} is calculated as explained above . In line 16 the remaining WCET of T_{best} is determined. In line 17 remaining WCET of T_{best} is divided by

the slack time available to T_{best} and the frequency is scaled accordingly.

The schedule with the example task-set is shown in the Figure 3.6.

Figure 3.6: DRA

3.1.7 DRA-OTE

DRA-OTE (Dynamic Reclaiming Algorithm-One Task Technique) is an extension to the DRA algorithm. This algorithm tries to further reduce the frequency when some conditions are met, if there is only one task in the ready queue.

In order to explain this algorithm we will first discuss the concept of NTA or Next Task Arrival. The next arrival time of any task instance in a system is known as NTA [31]. If there is only one task in the ready queue, and if its rem_i at $S_{optimal}$ is less than the time available till the NTA, then the frequency of execution of this task can further be reduced to utilize the entire time till NTA.

The algorithm 7 presents the pseudo-code for DRA-OTE.

Till line 18 it is same as DRA. In line 20 The next arrival time is calculated and stored in the variable NTA. In line 21 it is checked if the task T_{best} is the only task in the ready queue. If it is then the time till the NTA is calculated from the current time and stored in the variable Z. In 22 the frequency k is further scaled by $(Rem_WCET_{best})/Z$ and the new frequency value is obtained.

Algorithm 7: DRA-OTE

```

1: WCET:  $C_1, C_2, \dots, C_N$ 
2: Period:  $P_1, P_2, \dots, P_N$ 
3: Frequencies:  $f_1, f_2, \dots, f_m$ 
4:
5:  $U = C_1/P_1 + C_2/P_2 + \dots + C_n/P_n$  ;
6: if Task  $T_i$  begins then
7:   Initialize_alpha_member( $D_i, rem_i$ );
8:    $time\_diff = \text{Compute\_Time\_Elapsed}()$ ;
9:   Update_alpha_Queue( $time\_diff, \alpha\_queue$ );
10:  Insert_Into_alpha_Queue( $\alpha_j, \alpha\_queue$ );
11:
12: if Task  $T_i$  ends then
13:   Compute_Time_Elapsed();
14:   Update_alpha_Queue( $time\_diff, \alpha\_queue$ );
15:
16:  $E = \text{Calculate\_Dynamic\_Slack\_Available}(T_{best}, \alpha\_queue)$ ;
17:  $Rem\_WCET_{best} = \text{Calculate\_Remaining\_WCET}(T_{best})$  ;
18:  $k = Rem\_WCET_{best}/E$  ;
19:
20:  $NTA = \text{Calculate\_NTA}()$ ;
21: if Task  $T_i$  is the only task in the run q then
22:    $Z = NTA - current\_time$  ;
23:    $k = k * Rem\_WCET_{best}/Z$  ;
24:
25: Select  $f \supseteq f_1, f_2, \dots, f_m$  such that  $f \leq k$ 

```

3.1.8 AGR1

AGR1 is also an extension of DRA, however, in addition to the dynamic speed reclaiming technique this algorithm uses average workload information to predict the early completion of the future workloads and thus obtain extra slack to further reduce the CPU frequency. Thus this algorithm reduces the frequency more aggressively than DRA and DRA-OTE, consequently saving more energy. This algorithm is based on the fact that whenever there is more than one task in the ready queue, and when all the tasks have to complete before NTA, then the CPU time can be transferred among these tasks.

Lets consider that there are 3 tasks T_1, T_2 and T_3 in the ready queue and all of them have to complete before the NTA. If T_1 is the task having the earliest deadline, then it can obtain CPU time from T_2 and T_3 , such that its speed can be reduced, but at the same time ensuring that all these tasks complete before NTA. This additional frequency scaling is done to the frequency which is obtained as the result of DRA.

Before describing the algorithm let us first understand the meaning of some terms used in the algorithm.

Algorithm 8: AGR1

```

1:  $T_{best}$  : the next runnable task selected with the earliest deadline
2:  $\sigma_T$  : List of tasks in alpha q that have completed and have unused computation time and the tasks in
   the ready q having a deadline greater than the selected task's deadline
3:  $\sigma_L$  ; List of tasks in alpha q and the ready task which have to complete before NTA but can provide
   CPU time to  $T_{best}$ 
4: WCET:  $C_1, C_2, \dots, C_N$ 
5: Period:  $P_1, P_2, \dots, P_N$ 
6: Frequencies:  $f_1, f_2, \dots, f_m$ 
7:
8:  $U = C_1/P_1 + C_2/P_2 + \dots + C_n/P_n$  ;
9: if Task  $T_i$  begins then
10:   Initialize_alpha_member( $D_i, rem_i$ );
11:   time_diff = Compute_Time_Elapsed();
12:   Update_alpha_Queue(time_diff, alpha_queue);
13:   Insert_Into_alpha_Queue( $\alpha_j, \alpha\_queue$ );
14:
15: if Task  $T_i$  ends then
16:   Compute_Time_Elapsed();
17:   Update_alpha_Queue(time_diff, alpha_queue);
18:
19:  $E = \text{CalculateDynamicSlackAvailable}(T_{best}, \alpha\_queue)$ ;
20:  $Rem\_WCET_{best} = \text{CalculateRemainingWCET}(T_{best})$  ;
21:  $k = Rem\_WCET_{best}/E$  ;
22:
23:  $NTA = \text{Calculate\_NTA}()$ ;
24: if Task  $T_i$  is the only task in the run q then
25:    $Z = NTA - \text{current\_time}$  ;
26:    $k = k * Rem\_WCET_{best}/Z$  ;
27:   goto end;

```

 S_{optavg}

This is the optimal speed considering the average workload of the task-set.

 Q

This is the total amount of time required to be transferred to the next runnable task so that it can operate at $\max(S_{min}, S_{optavg})$

 B

This is the amount of time actually transferred.

Algorithm 8 and Algorithm 9 describe this algorithm. ²

The first part of the algorithm which is Algorithm 8 is same as DRA-OTE. Lets consider the second part of the algorithm which is 9

²This algorithm has been shown in two parts for clear visibility.

Algorithm 9: AGR1 (contd)

```

1: if  $k \leq \max(S_{min}, S_{optavg})$  then
2:   goto end ;
3:  $Q = \lceil S_{optimal} / \max(S_{min}, S_{optavg}) - 1 \rceil * Rem\_WCET\_best$ ;
4: if  $NTA - current\_time - Rem\_WCET\_best < Q$  then
5:    $Q = NTA - current\_time - Rem\_WCET\_best$  ;
6:    $Q_{actual} = 0$  ;
7:    $Z = 0$  ;
8:   for each task  $T_j$  in  $\sigma_T$  in decreasing deadline order do
9:      $count\_ready\_tasks++$  ;
10:     $Z_{prev} = Z$  ;
11:     $Z = \sum_{i=j}^n Rem\_WCET_i$  ;
12:    if  $Z > Q$  then
13:       $count\_ready\_tasks--$  ;
14:       $Z = Z_{prev}$  ;
15:      break ;
16:   for each task  $T_j$  in  $\sigma_L$  in decreasing deadline order do
17:     if  $T_j$  is ready then
18:        $T_j\_speed\_prev = T_j\_speed$ ;
19:        $Requested\_time = Q - Q_{actual}$  ;
20:        $T_j\_speed = T_j\_speed\_prev * [Rem\_WCET_j / (Rem\_WCET_j - Requested\_time)]$  ;
21:        $B = \lceil (T_j\_speed / T_j\_speed\_prev) - 1 \rceil * Rem\_WCET_j$ ;
22:       if  $T_j$  is completed but is in the  $\alpha$  queue then
23:          $B = \min(Requested\_time, Rem\_WCET_j)$ 
24:        $Q_{actual} = Q_{actual} + B$ ;
25:    $k = k * Rem\_WCET\_best / (Rem\_WCET\_best + B)$ ;
26: end: Select  $f \supseteq f_1, f_2, \dots, f_m$  such that  $f \leq k$ 

```

If the frequency selected under DRA-OTE is lesser than S_{min} or S_{optavg} then there is no point in further reducing the frequency so we just goto end as in lines 1 and 2.

If that is not the case then we calculate the amount of CPU time that is required to be transferred to the ready task T_{best} , so that it can operate at the lowest frequency possible which is the $\max(S_{min}, S_{optavg})$ in line 3.

In lines 4 we check if the time till NTA is enough to incorporate the increase in the Rem_WCET_best by Q . If not, then Q is adjusted accordingly in 51.

Once Q that is the amount of CPU time to be transferred to T_{best} from the other ready tasks is decided, then in the for loop from 8 to 15, the ready tasks as well as completed tasks having unused computation time in the α queue which can contribute in this Q are figured out. Say this list of tasks is in the list σ_L .

In the for loop from 16 to 24 the individual transfer of B units of time from these tasks take place.

As they are giving away a part of their allotted CPU time, these tasks are left with lesser time for execution, and so their speed is increased accordingly in line 20.

B is calculated in line 21 and the frequency is scaled accordingly in 25.

3.1.9 AGR2

AGR2 is very similar to AGR1 except in 2 points which we state as follows:

- (i) If the frequency value available after DRA computation in line 21 of AGR1 is lesser than $0.9 * S_{optavg}$ further frequency reduction is not done.
- (ii) Instead of S_{optavg} , $0.9 * S_{optavg}$ is used for computing Q.

3.2 Schedulers for Dependent Underloaded task-sets

3.2.1 EUA

EUA is a utility accrual RT-DVFS algorithm devised by Haisang *et al.* in [52]. This algorithm aims at accruing maximum utility per unit of energy consumed while reducing the total system power. During the underloads it accrues the maximum utility and scales the frequency in a similar way as LA-EDF [44], whereas in overloads it accrues the maximum utility possible by running the tasks at the maximum speed.

Figure 3.7: Step TUF

The version of EUA implemented in ChronOS makes certain assumptions which are stated as follows:

- (i) The task-set under consideration is subjected to a step TUF function which is shown in 3.7. In a Step TUF function a task has a constant maximum utility till its deadline. After its deadline the utility value becomes zero.
- (ii) The actual version of EUA aims at the reduction of the total system power. In this implementation we have considered the CPU power reduction, assuming it to be directly proportional to the cube of frequency, as was assumed by Haisang [53]

(iii) Considering the above two points the UER of phase J_i is given by $Utility/(f_{min}^3 * Rem_WCET_j)$ where f_{min} is the minimum frequency supported by hardware.

Auxiliary Functions

The following functions have been used in the algorithm.

sortByUER(σ)

Sort the list σ according to their decreasing UER of the phases i.e. the phase with the highest UER will be at the head of the list.

calculateUER(J_k, t)

Calculate the UER as $Utility/Rem_WCET_k$, where $Utility$ is the utility of the phase J_k and Rem_WCET_k is the remaining WCET under $S_{optimal}$. We just calculate UER as the ratio of Utility and remaining WCET and do not include the minimum power (which would be proportional to the cube of f_{min}). This is because minimum power will be constant for all the phases, as we only consider CPU power consumption in this implementation. Utility and remaining WCET are the only two parameters which will be different for different phases.

feasible(σ_{dl})

For a list to be feasible the predicted completion time of all the phases in the list at the maximum frequency, should be less than their respective deadlines.

Insert($J_k, \sigma_{dl}, J_k.X$)

This function inserts the phase J_k in the ordered list σ_{dl} at the position indicated by the index $J_k.X$.

EUA is described using the Algorithm 11. It uses auxiliary functions described in the Section 3.2.1.

In the for loop from lines 7 to 10, for all the phases in the ready queue it is checked, whether the phases are still feasible, i.e. if they have not blown their deadline. If they have, then they are aborted as in line 9.

If not, then the UER of all the feasible tasks in the ready queue is calculated as explained above.

Then these tasks are sorted in the order of their UER and placed in a list say σ_{tmp} in line 11.

In the for loop from lines 12 to 18 the head of the σ_{tmp} is selected (this is the task with the highest UER) and inserted into the list σ_{dl} , at its deadline position in line 15. In line 16, the feasibility of the list is checked as explained above. If feasible, then σ_{dl} is copied into σ as in

Algorithm 10: Defer_EUA Function

```

1: Procedure:Defer ()
2:
3: Input:  $\sigma_T$  // List of released tasks
4: WCET:  $C_1, C_2, \dots, C_N$ 
5: Period:  $P_1, P_2, \dots, P_N$ 
6: Frequencies:  $f_1, f_2, \dots, f_m$ 
7:
8:  $U = C_1/P_1 + C_2/P_2 + \dots + C_n/P_n$  ;
9:  $s = 0$ ;
10: for each task  $T_i$  in  $\sigma_T$  in decreasing deadline order do
11:    $U = U - C_i/P_i$ ;
12:   if  $U \geq 1$  then
13:     break ;
14:    $x = \max(0, \text{Cleft}_i - (1 - U)(D_i - D_n))$ ;
15:    $U = U + (\text{Cleft}_i - x)/(D_i - D_n)$ ;
16:    $s = s + x$ ;
17:
18: if  $U \geq 1$  then
19:    $f = f_{\max}$ ;
20: else
21:    $f = s/(D_n - \text{current\_time})$ ;

```

line 17. If not then we break from the loop. The head of σ is selected as T_{best} , which is the next runnable task.

Thereafter, the algorithm is the same as LA-EDF, with the only difference being in the defer function. A modified version of defer called as defer_EUA is used. The only difference between defer and defer_EUA is that when $U \geq 100$ the max frequency is selected. Thus during underloads the frequency selection is same as LA-EDF whereas during overloads the max frequency is selected as the main aim then is to accrue the maximum utility.

3.2.2 HS

This is an RT_DVFS algorithm devised by Zhang *et al.* in [57], for task-sets with non preemptible blocking sections. In this algorithm a static speed known as high speed is calculated to ensure that no deadlines are missed when the tasks are scheduled with EDF even in the presence of non-preemptible blocking sections.

In this algorithm, the non-preemptible blocking sections have been modelled as a special case of SRP [16], where there is just one resource shared by all the tasks.

When the tasks are scheduled with EDF under SRP then feasibility is given by:

Algorithm 11: EUA

```

1: WCET:  $C_1, C_2, \dots, C_N$ 
2: Period:  $P_1, P_2, \dots, P_N$ 
3: Frequencies:  $f_1, f_2, \dots, f_m$ 
4:
5:  $U = C_1/P_1 + C_2/P_2 + \dots + C_n/P_n$ ;
6:
7: for  $\forall J_k \in \mathcal{J}_r$  do
8:   if  $feasible(J_k) = false$  then
9:      $abort(J_k)$ ;
10:   $J_k.UER := calculateUER(J_k, t)$ ;
11:  $\sigma_{tmp} := sortByUER(\mathcal{J}_r)$ ;
12: for  $\forall J_k \in \sigma_{tmp}$  from head to tail do
13:   if  $J_k.UER > 0$  then
14:      $\sigma_{dl} = \sigma$ ;
15:      $Insert(J_k, \sigma_{dl}, J_k.X)$ ;
16:     if  $feasible(\sigma_{dl})$  then
17:        $\sigma = \sigma_{dl}$ ;
18:   else break;
19:  $T_{best} = headOf(\sigma)$ ;
20: if Task  $T_i$  begins then
21:    $U_{prev} = U$ ;
22:    $k = Defer\_EUA()$ ;
23:
24: if Task  $T_i$  ends then
25:    $Cleft = 0$ ;
26:    $k = Defer\_EUA()$ ;
27: During execution of  $T_i$  Decerement  $Cleft$ ;
28: Select  $f \supsetneq f_1, f_2, \dots, f_m$  such that  $f \leq k$ 

```

$\forall k, 1 \leq k \leq n \sum_{i=1}^n C_i/D_i + B_k/D_k \leq 1$, where B_k is the maximum blocking time that a task T_i can be blocked.

So A static speed H can be selected as follows, which ensures that the deadlines will be met.

$$\forall k, 1 \leq k \leq n \sum_{i=1}^n C_i/D_i + B_k/D_k \leq H \dots (1)$$

The algorithm is based on computing this static speed based on the properties of the task-set.

Algorithm 12 gives the pseudo-code for this algorithm which is explained as follows:

The equation (1) can be broken down in the 3 steps within the for loop from lines 9 to 12. In line 8 the utilization of the task-set is computed. G_j is the maximum length of time for which task T_i can be blocked under SRP by the lower priority jobs whose periods and deadlines are greater than that of T_i , since this task-set is being scheduled by EDF.

The example task-set to be used for HS and DS is given in the Table 3.2.

Figure 3.8: HS

Algorithm 12: HS

```

1: Input:  $\sigma_T$  // List of released tasks
2: WCET:  $C_1, C_2, \dots, C_N$ 
3: Period:  $P_1, P_2, \dots, P_N$ 
4: Frequencies:  $f_1, f_2, \dots, f_m$ 
5:
6:  $U = C_1/P_1 + C_2/P_2 + \dots + C_n/P_n$  ;
7:  $H = 0$ ;
8:  $utilization = 0$  ;
9: for each task  $T_i$  in  $\sigma_T$  in the increasing order of the periods do
10:    $utilization += C_i/P_i$ ;
11:    $speed = utilization + \lceil \max (G_j | P_i \leq P_j) \rceil / P_i$ ;
12:    $H = \max(speed, H)$ ;
13: return H ;

```

The Figure 3.8 shows the schedule and the speed of operation under HS algorithm using the example task-set.

Table 3.2: Sample 2 task task-set

Task	WCET	Period
T_1	2	5
T_2	5	10

Figure 3.9: DS

3.2.3 DS

In the HS algorithm, the processor can just operate at one constant speed. Another algorithm has been devised by Zhang *et al.* in [57] which is an improvement over the HS algorithm, as it can save more energy by operating at two speeds. These two speeds are the static high speed calculated by the HS algorithm (H), and the $S_{optimal}$ speed calculated by the Static_EDF algorithm (L). In this algorithm, the processor operates at the high speed only if a task is blocked by a lower priority task, for a duration, which is till the completion time of the blocking task. At all the other times the processor operates at the low speed, thus saving considerable energy while making sure that no deadlines are missed.

The DS algorithm is presented in the Algorithm 13.

First and foremost in line 5 the variable `high_speed_interval` is initialized to 0 and declared as static so that its value doesn't change between the scheduler invocations. In the HS and the DS algorithm the High Speed is calculated in the user space, and is transferred to the kernel as an argument to the system call.

In line 13 the processor speed is set as L. In line 14 it is checked if the current time is lesser than the `high_speed_interval`. If it is then it means that the interval is not over, and the processor speed is set as H.

Again, in lines 17 to 19 if the resource that T_{best} is going to request is not free, then it is blocked by a lower priority task and the lower priority task becomes T_{best} or the next runnable task and the processor speed is set to H in this case as well. The `high_speed_interval` is set to be the max of the previous `high_speed_interval` and the deadline of the task which has blocked.

However, if the blocking task finishes before its deadline then the `high_speed_interval` also

Algorithm 13: DS

```

1: Input:  $\sigma_T$  // List of released tasks
2: WCET:  $C_1, C_2, \dots, C_N$ 
3: Period:  $P_1, P_2, \dots, P_N$ 
4: Frequencies:  $f_1, f_2, \dots, f_m$ 
5: static  $high\_speed\_interval = 0$ 
6:
7:  $H = 0$ ;
8:  $utilization = 0$ ;
9: for each task  $T_i$  in  $\sigma_T$  in the increasing order of the periods do
10:    $utilization += C_i/P_i$ ;
11:    $speed = utilization + \lceil maxG_j[P_i < P_j] \rceil / P_i$ ;
12:    $H = \max(speed, H)$ ;
13: Set processor speed as  $L$ ;
14: if Blocking task finishes before its deadline then
15:    $high\_speed\_interval = 0$ ;
16: if Current_time less than high_speed_interval then
17:    $k = H$ ;
18: if the resource which  $T_{best}$  is going to request is not free then
19:    $T_{best} = T_{owner}$ ;  $k = H$ ;
20:    $high\_speed\_interval = \max(high\_speed\_interval, deadline\_of\_T_{owner})$ ;
21: if End of high_speed_interval is reached then
22:    $high\_speed\_interval = 0$ ;
23: if Task  $T_i$  begins then
24:    $U_{prev} = U$ ;
25:    $C_{left} = C_i$ ;
26:    $k = \text{Defer}()$ ;
27:
28: if Task  $T_i$  ends then
29:    $C_{left} = 0$ ;
30:    $k = \text{Defer}()$ ;
31: During execution of  $T_i$  Decerement  $C_{left}$ ;
32: Select  $f \supsetneq f_1, f_2, \dots, f_m$  such that  $f \leq k$ 

```

gets over with the completion of the blocking task as shown in line 14.

Ultimately when the high speed interval is reached it is set to be 0 as in line 22. If above conditions are not true, then the processor operates at the low speed.

The schedule is shown in the Figure 3.2.3.

Algorithm 14: USFILEDF

```

1: Input:  $\sigma_T$  // List of released tasks
2: WCET:  $C_1, C_2, \dots, C_N$ 
3: Period:  $P_1, P_2, \dots, P_N$ 
4: Frequencies:  $f_1, f_2, \dots, f_m$ 
5:
6:  $U = C_1/P_1 + C_2/P_2 + \dots + C_n/P_n$  ;
7:  $H = 0$ ;
8:  $utilization = 0$  ;
9: while  $q \leq n$  do
10:   for each task  $T_i$  in  $\sigma_T$  in the increasing order of the periods do
11:      $\sum_{1 \leq r \leq q} 1/\eta_r[C_r/D_r] + 1/\eta_i[B_i/D_i + \sum_{q \leq p \leq r} C_p/D_p] = 1$ ;
12:      $\eta_m = \max_{i=q}^n \eta_i$ 
13:     for each task  $T_i$  in  $\sigma_T$  in the increasing order of the periods do
14:        $\eta_i = \eta_m$ ;
15:    $q = m + 1$ ;
16: if the resource which  $T_{best}$  is going to request is not free then
17:    $T_{best} = T_{owner}$ ;
18:    $\eta_{best} = \eta_{owner}$ ;
19:  $k = \eta_{best}$ ;
20: Select  $f \supseteq f_1, f_2, \dots, f_m$  such that  $f \leq k$ 

```

3.2.4 USFILEDF

This is an RT-DVFS algorithm designed by Rajkumar *et al.* in [30], which allows synchronization of tasks for access to shared resources while maintaining hard real-time constraints. In this algorithm the static slowdown factors for the synchronizing tasks are calculated, considering the feasibility of the task-set when scheduled with EDF. This algorithm also involves frequency inheritance where a low priority job can inherit the frequency of the highest priority job that it has blocked in order to prevent deadline misses. In this algorithm described by Rajkumar *et al.*, the programmer has been given the freedom to use any Resource Access Protocol. Since we use EDF for scheduling, we choose SRP [16] as the Resource Access Protocol, as it is suited for dynamic priority scheduling. In SRP, whenever a job attempts to preempt another job, it is checked if the resource it will require in future is available at the time of preemption. If it is not then it is blocked and not allowed to execute.

USFILEDF is presented in 14.

In the for loop from 9 to 15 the static slowdown factors are calculated using the equation :

$$\sum_{1 \leq r \leq q} 1/\eta_r[C_r/D_r] + 1/\eta_i[B_i/D_i + \sum_{q \leq p \leq r} C_p/D_p] = 1$$

In lines 17 and 18 frequency inheritance takes place where the blocking tasks inherits the slowdown factor of the blocked task.

As such under SRP, a job can be blocked at most for one critical section so the max blocking

time will be the length of the highest critical section [16]. Here B is the maximum blocking time under SRP.

3.3 Schedulers for Overloaded task-sets

3.3.1 REUA

This is an extension of the EUA algorithm that is devised for dependent task models. Thus in addition to acquiring maximum utility while saving energy, this algorithm includes mechanisms to bound the blocking time as well as for deadlock detection and resolution,

Auxiliary Functions

The following functions have been used in the algorithm.

sortByUER(σ)

Sort the list σ according to their decreasing UER of the phases i.e. the phase with the highest UER will be at the head of the list.

calculateUER(J_k, t)

This function calculates the UER of a phase as follows : The utility and the remaining WCET of the phase being considered, as well as of all the dependent phases in the dependency chain of the phase are summed and their ratio is calculated as $(\sum_{k=1}^n Utility_k) / (\sum_{k=1}^n Rem_WCET_k)$ where n is the total number of phases in the dependency chain including the phase being considered.

buildDep J_k

This functions builds the dependency chain for the phase J_k . This chain is created considering the owner of the resource requested by J_k say J_{owner} , and then again the owner of the resource requested by this J_{owner} and so on. If there is a loop in the chain, which means that if the owner of the resource requested by a phase in the chain itself requests a resource whose owner is also present in the chain, then this will create a deadlock. In order to resolve this one of the phases in the loop is aborted, This phase which is aborted has the lowest UER.

insertPhaseWithDep(J_k, σ)

This function inserts the phase into the list σ along with all its dependencies at the critical deadline position. The critical deadline of the phase is the earliest deadline of the phase among all the phases in its dependency chain.

Algorithm 15: REUA

```

1: WCET:  $C_1, C_2, \dots, C_N$ 
2: Period:  $P_1, P_2, \dots, P_N$ 
3: Frequencies:  $f_1, f_2, \dots, f_m$ 
4:
5:  $U = C_1/P_1 + C_2/P_2 + \dots + C_n/P_n$  ;
6: for  $\forall J_k \in \mathcal{J}_r$  do
7:   if  $\text{feasible}(J_k) = \text{false}$  then
8:      $\text{abort}(J_k)$ ;
9:   else
10:     $J_k.\text{Dep} := \text{buildDep}(J_k)$ ;
11: for  $\forall J_k \in \mathcal{J}_r$  do
12:    $J_k.\text{UER} := \text{calculateUER}(J_k, t)$ ;
13:  $\sigma_{tmp} := \text{sortByUER}(\mathcal{J}_r)$ ;
14: for  $\forall J_k \in \sigma_{tmp}$  from head to tail do
15:   if  $J_k.\text{UER} > 0$  then
16:      $\sigma := \text{insertPhaseWithDep}(J_k, \sigma)$ ;
17:   else break;
18:  $T_{best} := \text{headOf}(\sigma)$ ;
19: if Task Ti begins then
20:    $U_{prev} = U$ ;
21:    $k = \text{Defer\_EUA}()$ ;
22:
23: if Task Ti ends then
24:    $Cleft = 0$  ;
25:    $k = \text{Defer\_EUA}()$ ;
26: During execution of Ti Decerement Cleft ;
27: Select  $f \supsetneq f_1, f_2, \dots, f_m$  such that  $f \leq k$ 

```

The Algorithm 15 describes this algorithm and uses the functions described in 3.3.1.

Till line 8 this algorithm is same as EUA. In line 10 the dependency chain of each phase is calculated using the $\text{buildDep}()$ function as explained above. In line 12 the UER is calculated for each phase. Then these tasks are sorted in the order of their UER and placed in a list, σ_{tmp} . From 14 to 17 the phases are inserted into σ along with their dependencies in the critical deadline order. The head of σ is selected as T_{best} , which is the next runnable task. Thereafter the algorithm is same as EUA.

Table 3.3: Slack Estimation Techniques used in the Algorithms

Name	Max Const Speed	Stretching to NTA	Priority Based stack stealing	Utilization Updating
<i>Static-EDf</i>	✓			
<i>CC-EDF,</i> <i>LA-EDF</i>				✓
<i>DRA,</i> <i>DRA-OTE</i>	✓	✓	✓	
<i>AGR1,</i> <i>AGR2</i>		✓	✓	
<i>EUA, REUA</i>				✓
<i>HS, DS</i>	✓			

Chapter 4

Extending ChronOS with DVFS support

4.1 ChronOS

4.1.1 Introduction

In order to implement and evaluate our RT-DVFS schedulers we have used ChronOS, which is, a best-effort real-time Linux kernel developed by Matthew *et al.* in [22]. ChronOS is based on the 2.6.33.7 version of the Linux kernel, and has been enhanced with Ingo Molnar's PREEMPT_RT real-time patch [42]. ChronOS provides a number of APIs and a scheduler plugin infrastructure that can be used to implement various RT-DVFS algorithms.

The base Linux kernel provides soft real-time capabilities. In order to make this kernel hard real-time, it has to be made preemptible. To achieve this, the PREEMPT_RT patch has been used in ChronOS. With this patch the interrupt latencies improve and most of the parts of the kernel become preemptible, thus providing hard real-time properties to the Linux kernel.

4.1.2 ChronOS real-time Scheduler

The ChronOS scheduler is built on top of the Linux scheduler. The `O(1)` Linux scheduler [38] has a bitmap implementation of the priority levels. Consequently, the scheduling algorithms implemented in the kernel which are (`SCHED_NORMAL`, `SCHED_FIFO`, `SCHED_RR`) take constant time to complete. In the Linux kernel, every priority level has a Linux run-queue associated with it. In ChronOS, in addition to that, every priority level also has a ChronOS real-time run-queue (`CRT-RQ`). This run-queue contains the ChronOS real-time tasks. ChronOS real-time tasks are real-time segments defined in the user applications, using the system

calls provided by ChronOS. Real-time segments are those portions of a user application thread, which have to be executed with hard real-time constraints. Thus, when a ChronOS real-time task enters the system it is added to the **CRT-RQ**. On being called, based on the scheduling algorithm selected, the task from the **CRT-RQ** is selected by the ChronOS scheduler and returned to the Linux scheduler, which executes it.

4.1.3 Scheduling Events

ChronOS is an event based real-time kernel, in which the real-time scheduler is invoked when certain events occur. These events are known as scheduling events, which include:

- When a task enters the system
When a task begins its real-time segment the scheduler is invoked.
- When a task leaves the system
When a task completes its real-time segment the scheduler is invoked.
- When a resource is requested
In case of dependent task-sets, when a task requests a resource then also the scheduler is invoked.
- When a resource is released
Similarly, whenever a task releases the resource, the scheduler is invoked.

4.1.4 System Calls provided by ChronOS

As already mentioned, ChronOS provides system calls with which, real-time segments can be defined within the threads of the user application. These system calls are described as follows:

`begin_rt_seg()`

This system call is used by the application to start a real-time segment. Using this system call the application also informs the kernel, about the real-time properties of the real-time segment, such as its period, deadline and so on. In order to implement RT-DVFS we have modified this system call to inform the kernel about the utilisation of the task-set, `high_frequency`, S_{optavg} and the locks that will be requested by the tasks in future.

`end_rt_seg()`

By using this system call the application indicates the end of the scheduling segment to the kernel.

`chronos_mutex_lock()`

This system call is used to request a resource, such as a mutex.

`chronos_mutex_unlock()`

This system call is used to release the resource.

`set_scheduler()`

This system call is used for selecting a particular scheduling algorithm. The ChronOS scheduler selects a task from the **CRT-RQ** based on the scheduling algorithm set by this system call.

In order to implement RT-DVFS schedulers in ChronOS, they have to be implemented as Linux kernel modules, and thus can be loaded or unloaded from the running kernel using `modprobe`.

The sequence of operation is as follows:

The real-time application calls the `set_scheduler()` system call to select a RT-DVFS scheduler, say **LA-EDF**. If the **LA-EDF** kernel module is available, then ChronOS loads this module and **LA-EDF** becomes the ChronOS local scheduler. Consequently, `sched_la_edf()` is called by the ChronOS scheduler at every scheduling event, which then operates on the **CRT-RQ**, finds the task with the earliest deadline, sets the frequency of the CPU to the value decided by the frequency evaluating part of the algorithm, and returns the selected task to the Linux **SCHED_FIFO** Scheduler for execution.

4.2 Adding DVFS support to ChronOS

RT-DVFS schedulers need to change the voltage and frequency of the processor at every scheduler invocation. A framework known as **CPUFreq** subsystem [9] has been implemented in the Linux kernel, since the 2.6.0 kernel, with which, the processor frequencies can be dynamically scaled. In order to implement RT-DVFS schedulers in ChronOS we have used this **CPUFreq** subsystem, described in the Section 4.2.1.

4.2.1 The CPUfreq Subsystem

The Figure 4.1 [6] shows the high-level view of the **CPUFreq** subsystem.

It contains the following components:

- (i) **CPUfreq module**: This module abstracts the low-level frequency controlling driver interface from the high-level frequency controlling policies. It provides APIs to the high-level code, which enables them to change CPU frequency.

Figure 4.1: CPUfreq Subsystem

(ii) **CPU-specific drivers:** For DVFS to work, CPU itself should support dynamic voltage and frequency scaling. The modern CPUs are enhanced with technologies which support DVFS. For example, Intel processors come with Enhanced Intel SpeedStep technology [6], AMD processors come with the Powernow! technology [3]. There are certain CPU specific drivers which enable the change of the frequencies. For Intel processors both the drivers', acpi-cpufreq and speedstep-centrino can be used, whereas for AMD processors powernow-k8 driver is used.

(iii) **In-kernel governors :** These governors are built as kernel modules and they change the frequency of the processor depending on they the way they have been implemented, when selected from the user-space. The 5 governors implemented in Linux kernel 2.6.33.7 include: (a) *Performance*: This governor runs the CPU at the maximum frequency.

(b) *Powersave*: This one runs the CPU at the minimum frequency.

(c) *Ondemand*: This one decides the CPU frequency based on the current CPU usage.

(d) *Conservative*: This one too decides the frequency based on the CPU usage, but unlike the Ondemand governor it changes the frequency in steps rather than changing it drastically.

4.2.2 Changing the frequency using CPUfreq module

For implementing DVFS in Chronos, we have used the CPUfreq module to scale the frequency. This module provides a function called `cpufreq_driver_target()` which has been defined in the file `Linux/drivers/cpufreq/cpufreq.c`.

Its prototype is as follows:

```
_cpufreq_driver_target(struct cpufreq_policy *policy, unsigned int target_freq, unsigned int relation)
```

We will briefly describe each argument that this function takes.

policy

This argument is for providing the limits within which the CPU frequency can be set. The frequency which is to be set must lie between `policy->min` and `policy->max`.

target_freq

This is the frequency to which the CPU frequency is requested to be set.

relation

This argument specifies the relation between the `target_freq` requested to be set and the actual frequency set. This can be set to two values as follows:

CPUFREQ_REL_L: The actual frequency selected is higher than or equal to the target frequency.

CPUFREQ_REL_H: The actual frequency selected is lower than or equal to the target frequency.

For example, if we want to set the processor frequency to 1.33 GHz we can call this function as follows:

```
_cpufreq_driver_target(policy, 1330000, CPUFREQ_RELATION_H).
```

4.2.3 Working of the CPUfreq module

The Figure 4.2 shows the flowchart which explains the working of the CPUfreq module. When the frequency change requests comes to the CPUfreq module, then first of all it inquires with all the registered drivers, about the frequency range that they can handle. If the new CPU frequency is out of that range, then the CPUfreq subsystem adjusts it, so that it falls within that range. It then notifies these registered drivers, that the frequency is going to change, so that they can decide which parameters to change, in order to adjust with the new frequency. After that, the CPU frequency is changed by writing into appropriate hardware registers of the CPU. The registered drivers are notified about the CPU frequency change, so that now they can change those parameters on which they had decided earlier.

Figure 4.2: Working of CPUfreq module

4.2.4 Integration of CPUfreq subsystem with ChronOS

The Figure 4.3 shows the integration of CPUfreq subsystem with ChronOS. A RT-DVFS algorithm needs to select a task to execute and the CPU frequency to execute it at. The ChronOS local scheduler rearranges the CRT-RQ, based on the RT-DVFS scheduling algorithm chosen, and returns the head of the CRT-RQ to the Linux SCHED-FIFO scheduler for execution. At the same time it also determines the frequency of execution and uses the `cpufreq_driver_target` function provided by the CPUfreq module to change the CPU frequency. This CPUfreq module, in turn calls the CPU-specific driver, which is, `acpi_cpufreq` in case of Intel I5 and `powernow_k8` in case of AMD. This driver then writes into the appropriate hardware registers inside the CPU to change the voltage and the frequency.

4.2.5 Implementation of the RT-DVFS Schedulers

In almost all the RT-DVFS schedulers described in Chapter 2, the schedulers do important computations based on, if a task has arrived or if it has completed. Thus, figuring out how to determine whether a real-time task has arrived or completed, was an important part of implementing RT-DVFS schedulers in ChronOS. In order to do so, we have made use of the *global current pointer* which has been defined as, `per_cpu` variables in the `arch/x86/include/asm/current.h` file. The *current* variable gets updated in the `context_switch()` function, called inside the Linux scheduler function. As the ChronOS scheduler

Figure 4.3: RT-DVFS scheduling in ChronOS

is called before the context switch takes place, the current pointer always points to the task which has just arrived or completed. This is because a ChronOS scheduler is called only at the scheduling events, which in the case of independent task-sets involve only 2 events, one being, when the task enters the system, and other being, when the task leaves the system.

We add a new field to the real-time data structure inside ChronOS, called `flag_begin_end_check`. There are two system calls provided by ChronOS with which a real-time application can indicate the beginning or the end of a real-time segment, these being `begin_rt_seg` and `end_rt_seg`. So in `begin_rt_seg`, we specify `flag_begin_end_check` as 100 and in `end_rt_seg`, we specify it as 200 to differentiate between the two.

So in the ChronOS scheduler we just check the value of this field of the current pointer to determine, if a task has arrived or completed so as to do the computations accordingly.

4.2.6 Modification of the real-time data structure for RT-DVFS support

In order to add DVFS support to ChronOS we have also added some additional fields to real-time data structure inside ChronOS. The code listing 4.1 shows all the fields of the real-time data structure, however in Section 4.2.6.1, we will discuss only the additional fields that have been added to support RT-DVFS in ChronOS.

```

1  /* Structure attached to struct task_struct */
2  struct rt_info {
3      struct rt_info {
4          /* Real-Time information */
5          struct timespec deadline;          /* monotonic time */
6          struct timespec temp_deadline;     /* monotonic time */
7          struct timespec period;           /* relative time */
8          struct timespec left;             /* relative time */
9          unsigned long exec_time;          /* WCET, us */
10         int max_util;
11         long local_ivd;
12         long global_ivd;
13         unsigned int seg_start_us;
14
15         /* The following fields have been added for RT-DVFS support */
16         int util_task_set;
17         long load;
18         int current_frequency;
19         int flag_begin_end_check;
20         unsigned long budget;
21         struct mutex_data *mutex0;
22         struct mutex_data *mutex1;
23         int S_optavg;
24         int high_freq;
25         struct alpha_member a_member;
26
27
28
29         /* Lists FLXME: convert to array of list_heads*/
30         /* 0 is local, 1 is global... should fix to pound defines */
31         struct list_head task_list[2];
32         struct list_entry list[SCHED_LISTS];
33
34         /* DAG used by x-GUA class of algorithims */
35         struct rt_graph graph;
36
37         /* Lock information */
38         struct mutex_head *requested_resource;
39         struct rt_info *dep;
40
41         /* Abort information */
42         struct abort_info abortinfo;

```

```

43 |
44 |     /* Task state information */
45 |     unsigned char flags;
46 |     int cpu;
47 | };

```

Listing 4.1: The ChronOS main real-time data structure

4.2.6.1 Additional fields for RT-DVFS support

util_task-set

This is the total utilisation the task-set to which the task belongs and is calculated as $\sum_{k=1}^n C_i/P_i$ where, n is the total number of tasks in the task-set.

load

This is the ratio of actual execution time to the period of the task. This is given by AC_i/P_i .

current_frequency

This is the frequency at which the task was operating before its preemption.

budget

This is the time allocated to the task for execution. This field is required for the implementation of dynamic slack reclaiming algorithms such as Snowdon-min, DRA, AGR etc.

mutex0, mutex1

This field is to let the kernel know about the future resource requirement of the task. Required for the implementation of HS, DS and USFLEDF.

high_freq

This is the static high-speed calculated for the entire task-set in the presence of non-preemptible blocking sections. Required for the implementation of HS and DS.

S_{optavg}

This field specifies the average workload of the task-set to which the task belongs. Required for the implementation of AGR1 and AGR2.

flag_begin_end_check

This field is required to check, whether the scheduler invocation took place when a task arrived, or when it completed.

a_member

This structure is used for the α_{queue} implementation, required for the implementation of DRA and AGR algorithms.

Chapter 5

Experimental Methodology

5.1 Platform Specifications

We have implemented the fourteen RT-DVFS algorithms on two hardware platforms. The first platform is an ASUS laptop with the Intel I5 processor. This processor has a rich set of 10 frequencies as shown in the Table 5.1. As this laptop has an Intel processor, it uses the Enhanced Intel Speedstep technology [6] for scaling the processor voltage and frequency on-the-fly. The driver used is `acpi_cpufreq`.

The other board we have used is the AMD Zacate mini-ITX Motherboard called the *GA-E350N-USB3* [2]. This processor can operate at 3 frequencies which are 800 MHz, 1.28 GHz and 1.6 GHz. As this board has an AMD processor, it uses the `powernow!` technology for DVFS. The driver it uses is the `powernow_k8`.

5.2 Test Application

In order to test and evaluate our RT-DVFS algorithms we have used a real-time test application called `Sched_test_App` developed by Matthew *et al.* in [22]. This user-space application takes a task-set file as the input. This task-set file provides the WCET, period and the deadline values for each task to this test application. Users can specify the scheduling algorithm and the workload to be used, as well as parameters such as, the % of CPU usage, run time, the length of the critical section in case of dependent task-set and so on. Depending upon the number of tasks in the task-set, this application creates a thread for every task. Every periodic instance of the task uses the ChronOS API `begin_rt_seg` to enter its real-time segment. Depending upon the workload chosen, the real-time task either burns the CPU for its WCET (in case of CPU intensive workload) or does some heavy memory accesses for its WCET (in case of memory intensive workloads). By calling `end_rt_seg`, it ends its real-time

segment and sleeps until the start of its next period. Thus, this application provides a set of periodic real-time tasks to the ChronOS local scheduler, which then schedules these tasks depending upon the scheduling algorithm selected and the timing constraints the tasks are subjected to.

5.2.1 Modification to the Test Application for RT-DVFS

In order to test and evaluate the RT-DVFS algorithms, there is one more parameter which is required. This is the actual execution time (ACET) of the real-time task, which is lesser than its WCET. We have modified the Sched_test_App such that we can vary this parameter. This means, that even though the kernel is provided with the WCET of the task, the task is actually executed, i.e it burns the CPU, only for its actual execution time. Actual execution time is expressed as a ratio of WCET. In our experiments we have used the values for ACET/WCET ranging from 0.1 to 1, in steps of 0.1.

5.3 Real-time Measurements

The real-time measurements in our experiments include measuring the the Deadline Satisfaction Ratio (DSR) and the Accrued Utility Ratio (AUR) given by Equations 5.1 and 5.2 respectively.

$$DSR = \frac{\text{The number of tasks that met their deadlines}}{\text{Total tasks in the system}} \quad (5.1)$$

$$AUR = \frac{\text{Total accrued utility of the tasks that met their deadlines}}{\text{Total possible accrued utility}} \quad (5.2)$$

DSR is the ratio of the number of tasks that met their deadlines to the total number of tasks in the system. Similarly AUR is the ratio of the total accrued utility of the tasks that met their deadlines to the total possible utility that can be accrued in the system.

Note: We have measured AUR only for the utility accrual algorithms such as EUA and REUA.

5.4 Power Measurements

In this section we will discuss the techniques we have used for doing the power measurements. But before that, we discuss the idle and performance states of the CPU in the Section 5.4.1.

5.4.1 Performance and Idle states of CPU

The ACPI specifications [1] define the following idle and performance states of the CPU:

P state

This is the performance state of the CPU. The processor is active, when it is in this state. The number of P States supported by a CPU is CPU specific. The processor operates at a different frequency/voltage pair in different P states, and thus consumes different amount of power at each state. The lower the P state is, the higher the frequency the CPU operates in, and thus consumes more power. The P states, the frequency of operation and the power consumed in each P-state in the Intel I5 processor is given in the Table 5.1.

C0 state

This is the operating state of the CPU. CPU is operating in one of the P-states when in this state. The power consumed in this state is dependent on the P-state in which it is operating in.

C1 state

This is the first idle state. In this state, only the CPU main internal clocks are halted through software. In Intel I5, the CPU consumes 1000mW, when in this state.

C2 state

In this state, only the CPU main internal clocks are halted via hardware and the CPU takes longer time to wake up from this state. For Intel I5, the power consumed in this state is 500mW.

C3 state

In this state, most parts of the processor is stopped, such as caches. As a result, the processor is no longer cache coherent in this state. It takes longer time to wake up from this state. The power consumed in this state for the Intel I5 is 300 mW.

5.4.2 System Power Measurements

We could measure the total system power of the AMD Zacate board using the Fluke 289 RMS multimeter [7]. This is a high resolution multimeter, its resolution being is 0.5 mA. This multimeter has an averaging mechanism, where it can perform the average of the current measurements over an interval of time (while it is measuring). We made a slit in the cord of the power supply, and attached the multimeter in series, to measure the current. We obtained the rms value of the average current and multiplied it with the rms voltage which is 120 V to get the average power consumed in that time interval.

Table 5.1: P-States in Intel I5 processor

P-State	Frequency in MHz	Power in Watts
<i>P0</i>	2400	25000
<i>P1</i>	2399	25000
<i>P2</i>	2266	23316
<i>P3</i>	2133	21689
<i>P4</i>	1999	20116
<i>P5</i>	1866	18531
<i>P6</i>	1733	17021
<i>P7</i>	1599	15517
<i>P8</i>	1466	14068
<i>P9</i>	1333	12640
<i>P9</i>	1199	11250

5.4.3 Normalized and Actual CPU power measurement

We have used the CPUpower tool [5] to obtain the normalized and the actual CPU power measurements. When a user application is fed as input, this tool gives information about the average frequency in the active state as well as the percentage of time spent in respective performance (P states) and idle states (C states) for the duration, the application runs.

The output of the CPUpower tool is as follows:

```

|Nehalem          || Mperf          || Idle_Stats
CPU | C3   | C6   | PC3  | PC6  || C0   | Cx   | Freq || <nul | C1   | C2   | C3
0 |  3.56| 21.53|  3.32| 18.65|| 67.29| 32.71| 1200||  0.00|  5.97|  3.52| 23.26

```

We will discuss the relevant fields of this output in the following section.

Freq

This field provides the average frequency for the duration the application fed to this tool, runs. In the example provided, the value of this average frequency is 1200 MHz.

C0

This field specifies the percentage of time the CPU was running in the operating state. In this example it is 67.29%.

C1,C2 and C3

These fields specify the percentage of time spent by the processor in the C1,C2 and C3 idle states, explained in the section 5.4.1. For eg. the CPU spends 5.97% of the total time in C1 state, 3.52% of the total time in the C2 state and 23.26% of the total time in the C3 state.

Cx

This field specifies the total percentage of time spent by the CPU in all the idle states. It is the sum of the the percentage of time spent in the 3 idle states C1, C2 and C3.

5.4.4 Calculation of Normalized CPU Energy consumption

The normalized CPU energy is calculated according to the following equation:

$$\text{Normalized CPU Energy} = (f_{avg}/f_{max})^3$$

f_{avg} is the average frequency for the duration, the application runs and is obtained from the **Freq** field of the CPUpower tool output. Considering the Intel I5 processor, which has a maximum frequency of 2400MHz, the Normalized CPU energy for the above example is:

$$(f_{avg}/f_{max})^3 = (1200/2400)^3 = 0.125$$

5.4.5 Calculation of Actual CPU Power

Actual CPU Power is calculated as the sum of the power consumed in the idle state and the active state. Considering the same CPU power tool output the power consumed in the various idle state and the operating state of the Intel I5 processor can be calculated as follows:

- (i) *Active power* = (% of time spent in C0)*(Power consumed in the corresponding P-state)
- (ii) *C1 power* = (% of time spent in C1)*1000mW
- (iii) *C2 power* = (% of time spent in C2)*500mW
- (iv) *C3 power* = (% of time spent in C3)*300mW

Actual CPU power = Active power + C1 power + C2 power + C3 power The actual CPU power consumed in the above example is:

$$(0.6729*11250)mW + (0.0597*1000)mW + (0.0352*500)mW + (0.2326*300)mW = 7718mW$$

Chapter 6

Experiments

In this chapter we present the experimental results obtained by implementing the RT-DVFS schedulers on the two hardware platforms and testing them using the test application mentioned in chapter 5.

6.1 Energy consumption results on Intel I5 laptop for schedulers designed for independent task-sets

6.1.1 Varying the CPU utilization at constant ACET

In this section we present the Actual CPU power results and the Normalized CPU Energy results on Intel I5 laptop, by varying the CPU utilization value from 50% to 250% at different ACETs ranging from 0.1WCET to 1WCET. We have used the 5T task-set which uses 5 tasks with deadlines and periods lying in the range of [500ms5000ms], where the utilization load for each task is in the range of [0.1 0.4]. Figures 6.1 and 6.2 show the Actual CPU power and Normalized CPU Energy results for the case when the ACET is 0.1 WCET for underloads.

First of all, we observe from these figures, that the Normalized CPU Energy consumption is totally different from the Actual CPU power consumption. The relative energy savings of the schedulers (relative to Base-EDF) observed from the normalized CPU energy savings plot, is much higher than that observed from the Actual CPU power consumption plot. Even though the Normalized CPU Energy consumption of the schedulers is consistent with their theoretical results, the Actual CPU power consumption of these schedulers show very less energy savings as compared to Base-EDF. This is because, the total CPU power consumption depends upon the power consumed in the active and the idle state of the CPU. In case of Base-EDF the CPU runs at a higher frequency for a shorter duration, and in case of RT-DVFS algorithms such as LA-EDF, the CPU runs at a lower frequency, but for a shorter

Figure 6.1: Normalized CPU Energy vs. CPU utilization (Underloads, 0.1 WCET, 5T, NL)

Figure 6.2: Actual CPU Power vs. CPU utilization (Underload, 0.1 WCET, 5T, NL)

duration. If the energy efficiency of the idle states of the CPU is high, then the power saved by running at a lower frequency for a longer time, may be offset by running at higher frequency for a shorter time, and transitioning to the idle state soon. In case of I5 the idle state power consumption is in the range of 0.3 W to 1 W, whereas the active state power consumption is in the range of 11.25 W to 25 W. Due to the high energy efficiency of the idle states in the Intel I5 core, we do not obtain much energy savings from the RT-DVFS algorithms. From the Figure 6.2, we observe that Base-EDF, does consume the maximum CPU power for most of the utilization values, but Static-EDF overtakes it after the utilization value of 0.85. We also observe that for this particular case, DRA consumes the minimum power. All the other algorithms consume power between these two extremes, with their power consumption value being very similar to one another. Figures 6.3 and 6.4 show the actual CPU power and the normalized CPU energy plots for the same case, but for overloads. From Figure 6.4 we can observe that for the overload case, AGR2 performs the best by consuming the least power. Algorithms such as Static-EDF, Snowdon-min and CC-EDF consume as much power as Base-EDF does, whereas the aggressive algorithms such as AGR2 and LA-EDF consume lesser power.

Figure 6.3: Normalized CPU Energy vs. CPU utilization (Overloads, 0.1 WCET, 5T, NL)

Figures 6.1 through 6.38 show these 4 plots(i.e. normalized CPU Energy consumption and the Actual Power consumption for underloads and overloads) for ACET ranging from 0.2WCET to 1WCET. All these figures show the same contrast between the normalized CPU Energy consumption and the Actual CPU power consumption. From hereafter we will only analyse the Actual CPU power consumption, as the normalized CPU Energy is not a true

Figure 6.4: Actual CPU Power vs. CPU utilization (Overloads, 0.1 WCET, 5T, NL)

measure of a CPU's power consumption and thus not suitable for evaluating the performance of the schedulers.

Figure 6.5: Normalized CPU Energy vs. CPU utilization (Underloads, 0.2 WCET, 5T, NL)

Figure 6.6: Actual CPU Power vs. CPU utilization (Underload, 0.2 WCET, 5T, NL)

Figure 6.7: Normalized CPU Energy vs. CPU utilization (Overloads, 0.2 WCET, 5T, NL)

Figure 6.8: Actual CPU Power vs. CPU utilization (Overloads, 0.2 WCET, 5T, NL)

Figure 6.9: Normalized CPU Energy vs. CPU utilization (Underloads, 0.3 WCET, 5T, NL)

Figure 6.10: Actual CPU Power vs. CPU utilization (Underload, 0.3 WCET, 5T, NL)

Figure 6.11: Normalized CPU Energy vs. CPU utilization (Overloads, 0.3 WCET, 5T, NL)

Figure 6.12: Actual CPU Power vs. CPU utilization (Overloads, 0.3 WCET, 5T, NL)

Figure 6.13: Normalized CPU Energy vs. CPU utilization (Underloads, 0.4 WCET, 5T, NL)

Figure 6.14: Actual CPU Power vs. CPU utilization (Underload, 0.4 WCET, 5T, NL)

Figure 6.15: Normalized CPU Energy vs. CPU utilization (Overloads, 0.4 WCET, 5T, NL)

Figure 6.16: Actual CPU Power vs. CPU utilization (Overloads, 0.4 WCET, 5T, NL)

Figure 6.17: Normalized CPU Energy vs. CPU utilization (Underload, 0.5 WCET, 5T, NL)

Figure 6.18: Actual CPU Power vs. CPU utilization (Underload, 0.5 WCET, 5T, NL)

From Figures 6.6, 6.10 and 6.14 we observe that Base-EDF consumes the maximum power, for almost all the CPU utilization values. Static-EDF is the next most power consuming algorithm, followed by CC-EDF. As in the case of 0.1 WCET plots, even in these cases, the aggressive algorithms consume the least power. From Figures 6.8, 6.12 and 6.16, we observe that Static-EDF consumes the maximum power. From Figure 6.18, we notice that the Actual CPU power consumption of Static-EDF is higher than that of Base-EDF when the ACET = 0.5 WCET. Thus, we see that for some cases Static-EDF performs even worse than Base-EDF. The power consumption of the remaining schedulers follow the similar trend as noted from previous figures.

The power consumption of the schedulers for the cases with ACET ranging from 0.6 WCET to 0.9 WCET show trends similar to cases discussed above, which can be observed from the Figures 6.22, 6.24, 6.26, 6.28, 6.30, 6.32, 6.34 and 6.36. Moreover in the overload case, we observe that AGR1 and AGR2 consume the least power.

One important observation over here is that, as we increase the ACET, the difference in the power consumption between the Base-EDF and the other schedulers increases considerably. This is because, by increasing the ACET, we are keeping the processor active for a longer time. Since Base-EDF operates at the maximum frequency and thus has a high active state power consumption, it will consume more power relative to other schedulers if the time spent in active state increases.

Figure 6.19: Normalized CPU Energy vs. CPU utilization (Overloads, 0.5 WCET, 5T, NL)

Figure 6.20: Actual CPU Power vs. CPU utilization (Overloads, 0.5 WCET, 5T, NL)

Figure 6.21: Normalized CPU Energy vs. CPU utilization (Underloads, 0.6 WCET, 5T, NL)

Figure 6.22: Actual CPU Power vs. CPU utilization (Underload, 0.6 WCET, 5T, NL)

Figure 6.23: Normalized CPU Energy vs. CPU utilization (Overloads, 0.6 WCET, 5T, NL)

Figure 6.24: Actual CPU Power vs. CPU utilization (Overloads, 0.6 WCET, 5T, NL)

Figure 6.25: Normalized CPU Energy vs. CPU utilization (Underload, 0.7 WCET, 5T, NL)

Figure 6.26: Actual CPU Power vs. CPU utilization (Underload, 0.7 WCET, 5T, NL)

Figure 6.27: Normalized CPU Energy vs. CPU utilization (Overloads, 0.7 WCET, 5T, NL)

Figure 6.28: Actual CPU Power vs. CPU utilization (Overloads, 0.7 WCET, 5T, NL)

Figure 6.29: Normalized CPU Energy vs. CPU utilization (Underload, 0.8 WCET, 5T, NL)

Figure 6.30: Actual CPU Power vs. CPU utilization (Underload, 0.8 WCET, 5T, NL)

Figure 6.31: Normalized CPU Energy vs. CPU utilization (Overloads, 0.8 WCET, 5T, NL)

Figure 6.32: Actual CPU Power vs. CPU utilization (Overloads, 0.8 WCET, 5T, NL)

Figure 6.33: Normalized CPU Energy vs. CPU utilization (Underload, 0.9 WCET, 5T, NL)

Figure 6.34: Actual CPU Power vs. CPU utilization (Underload, 0.9 WCET, 5T, NL)

Figure 6.35: Normalized CPU Energy vs. CPU utilization (Overloads, 0.9 WCET, 5T, NL)

Figure 6.36: Actual CPU Power vs. CPU utilization (Overloads, 0.9 WCET, 5T, NL)

6.1.2 Varying ACET at a constant CPU utilization

In this section, we present the Actual CPU power results and the Normalized CPU Energy results on Intel I5 laptop by varying the ACET from 0.1WCET to 1WCET for different CPU utilization values ranging from 50% to 100% using the same 5T task-set mentioned in the Section 6.1.1.

The Figures 6.41 through 6.52 show the normalized CPU energy consumption and the Actual CPU power consumption for different ACET values for the CPU utilization ranging from 50-100%. From the Figure 6.41, we observe that for Base-EDF, the normalized CPU energy consumption is very high as compared to the other schedulers. This is because, 50% CPU utilization corresponds to the lowest frequency available on Intel I5, and consequently all the schedulers operate at the lowest frequency, whereas Base-EDF always operates at the highest frequency irrespective of the CPU utilization. As the utilization increases, different schedulers utilize different techniques to utilize the static and the dynamic slack and thus operate at different frequencies, and consequently, the (almost horizontal) lines which represent the schedulers spread out with increase in CPU utilization as can be seen from the Figures 6.41, 6.43, 6.45, 6.47, 6.49, 6.51. Static-EDF has a perfectly horizontal line as it only utilizes the static slack, and not the dynamic slack, which depends on the ACET. Although the normalized CPU energy plots do not indicate the actual power consumption, but nevertheless, from these plots we can learn, how well a scheduler utilizes the available dynamic

Figure 6.37: Normalized CPU Energy vs. CPU utilization (Underload, 1 WCET, 5T, NL)

Figure 6.38: Actual CPU Power vs. CPU utilization (Underload, 1 WCET, 5T, NL)

Figure 6.39: Normalized CPU Energy vs. CPU utilization (Overloads, 1.0 WCET, 5T, NL)

Figure 6.40: Actual CPU Power vs. CPU utilization (Overloads, 1 WCET, 5T, NL)

slack. If we observe the actual CPU power consumption of these schedulers for different ACET values, for CPU utilization ranging from 50-100%, in the Figures 6.42 through 6.52, we observe that the with the increase in ACET the actual power consumption increases much steeply than in the case of Normalized CPU Energy consumption. This is because, with the increase in ACET, dynamic power consumption increases due to the increase in frequency as well as due to the increase in the time spent in the active state. The normalized CPU energy plots capture the dynamic power increase due to the increase in frequency, but not due to the increase in time. Both these parameters are captured by the Actual CPU power consumption plots.

Figure 6.41: Normalized CPU Energy vs. ACET (Underload, 50% CPU Utilisation, 5T, NL)

Figure 6.42: Actual CPU Power vs. ACET (Underload, 50% CPU Utilisation, 5T, NL)

Figure 6.43: Normalized CPU Energy vs. ACET (Underload, 60% CPU Utilisation, 5T, NL)

Figure 6.44: Actual CPU Power vs. ACET (Underload, 60% CPU Utilisation, 5T, NL)

Figure 6.45: Normalized CPU Energy vs. ACET (Underload, 70% CPU Utilisation, 5T, NL)

Figure 6.46: Actual CPU Power vs. ACET (Underload, 70% CPU Utilisation, 5T, NL)

Figure 6.47: Normalized CPU Energy vs. ACET (Underload, 80% CPU Utilisation, 5T, NL)

Figure 6.48: Actual CPU Power vs. ACET (Underload, 80% CPU Utilisation, 5T, NL)

Figure 6.49: Normalized CPU Energy vs. ACET (Underload, 90% CPU Utilisation, 5T, NL)

Figure 6.50: Actual CPU Power vs. ACET (Underload, 90% CPU Utilisation, 5T, NL)

6.1.3 DSR results on Intel I5 laptop

Figures 6.53 through 6.62 show the DSR vs. CPU utilization results for ACET varying from 0.1WCET to 1WCET. In Figures 6.53, 6.54, 6.55, 6.56, 6.57, 6.58, 6.59, 6.60 and 6.61 we observe that all the scheduler meet all the deadlines during underloads. However, in the Figure 6.62 which is the DSR plot for the case of ACET=WCET, we observe that some schedulers like LA-EDF, AGR1, AGR2, Snowdon-min, DRA and DRA-OTE do miss some deadlines even in underloads. This might be due to the fact that ACET=1WCET represents the extreme case, where some deadlines can get missed considering the characteristics of real platforms such as cache misses etc. Pillai *et al* report in their paper [44] that LA-EDF misses deadlines when ACET is equal to WCET. During the overloads, we observe that the aggressive algorithms like AGR2 and REUA start missing deadlines for the case when ACET is 0.3 WCET, whereas LA-EDF starts missing deadlines at ACET=0.4WCET. We also observe that AGR1 and AGR2 miss deadlines more aggressively than others during overloads. The remaining algorithms start missing deadlines at ACET=0.5WCET, but, at a much higher CPU utilization as compared to AGR1, AGR2 and LA-EDF. REUA performs the best during the overloads and meets the maximum number of deadlines as can be seen from the Figures 6.57, 6.58, 6.59, 6.60, 6.61 and 6.62.

Figure 6.51: Normalized CPU Energy vs. ACET (Underload,100% CPU Utilisation , 5T, NL)

Figure 6.52: Actual CPU Power vs. ACET (Underload, 100% CPU Utilisation, 5T, NL)

Figure 6.53: DSR vs. CPU Utilisation (0.1 WCET, 5T, NL)

Figure 6.54: DSR vs. CPU Utilisation (0.2 WCET, 5T, NL)

Figure 6.55: DSR vs. CPU Utilisation (0.3 WCET, 5T, NL)

Figure 6.56: DSR vs. CPU Utilisation (0.4 WCET, 5T, NL)

Figure 6.57: DSR vs. CPU Utilisation (0.5 WCET, 5T, NL)

Figure 6.58: DSR vs. CPU Utilisation (0.6 WCET, 5T, NL)

Figure 6.59: DSR vs. CPU Utilisation (0.7 WCET, 5T, NL)

Figure 6.60: DSR vs. CPU Utilisation (0.8 WCET, 5T, NL)

Figure 6.61: DSR vs. CPU Utilisation (0.9 WCET, 5T, NL)

Figure 6.62: DSR vs. CPU Utilisation (1 WCET, 5T, NL)

6.2 Energy and DSR results for the schedulers designed for dependent task-sets

We implemented and evaluated four RT-DVFS schedulers for dependent task-sets. These are the High Speed (HS), Dual Speed (DS) Uniform Slowdown with frequency Inheritance-Earliest Deadline First(USFI-EDF) and Resource-constrained Energy-Efficient Utility Accrual Algorithm(REUA). We use the 10T task-set with one lock to evaluate the performance of these schedulers. 10T is a ten task task-set with deadlines and periods in the range of [400ms 20000ms] and the utilization load in the range of [0.01 0.4].¹

We measure the Actual CPU power, Normalized CPU energy and DSR by varying the CPU utilization for different values of critical section lengths (CS) ranging from 5 to 70% of WCET. Figures 6.63 through 6.71 show the normalized CPU Energy consumption and Figures 6.72 through 6.80 show the Actual CPU consumption of these schedulers, using the 10T 1L task-set for different critical section lengths varying from 5 to 70% of WCET. Similar to the case of schedulers for independent task-sets, the Actual CPU power consumption and the Normalized CPU Energy consumption is different for these schedulers as well. Normalized CPU Energy consumption of DS, USFI-EDF and REUA is very low as compared to HS, and REUA performs the best. However, from the Figures 6.72 through 6.80 we observe that the Actual CPU power consumption of these schedulers is similar to each other. We observe that for low values of CS, the energy consumption of DS, HS and USFI-EDF are similar, and maximum energy is saved by REUA. From Figures 6.79 and 6.80, we observe that for value of CS greater than 60% of WCET, DS saves the maximum energy.

HS and DS satisfy the timing constraints, i.e they do not miss deadlines, only when the high frequency calculated by HS is less than 100. In the Figures 6.63 to 6.71, the point where HS starts operating at the maximum frequency is the point from where it starts missing deadlines. But still for completeness, we have included all the utilization values.

Figures 6.81 through 6.88 show the DSR plots for these schedulers for varying CS lengths. HS and DS miss deadlines when the high frequency value calculated by HS (which is derived from the maximum length of the blocking section as well as from the deadlines and periods of the tasks in the task-set) becomes greater than 100. Similarly USFI-EDF also starts missing deadlines when the feasibility conditions as explained in the Section 3.2.4, are no longer satisfied.

¹We use one lock as, HS and DS are designed for task-sets with non-preemptible blocking sections, and in their paper [57] Zhang *et al.* have used one lock to implement the blocking section. We have followed a similar approach and evaluated all the schedulers with a single lock so as to make comparisons between them.

Figure 6.63: 5% CS - Normalized CPU Energy vs. CPU utilization (5% CS, 10T, 1L)

Figure 6.64: 10% CS - Normalized CPU Energy vs. CPU utilization (10% CS, 10T, 1L)

Figure 6.65: 15% CS - Normalized CPU Energy vs. CPU utilization (15% CS, 10T, 1L)

Figure 6.66: 20% CS - Normalized CPU Energy vs. CPU utilization (20% CS, 10T, 1L)

Figure 6.67: 30% CS - Normalized CPU Energy vs. CPU utilization (30% CS, 10T, 1L)

Figure 6.68: 40% CS - Normalized CPU Energy vs. CPU utilization (40% CS, 10T, 1L)

Figure 6.69: 50% CS - Normalized CPU Energy vs. CPU utilization (50% CS, 10T, 1L)

Figure 6.70: 60% CS - Normalized CPU Energy vs. CPU utilization (60% CS, 10T, 1L)

Figure 6.71: 70% CS - Normalized CPU Energy vs. CPU utilization (70% CS, 10T, 1L)

Figure 6.72: 5% CS - Actual CPU Power vs. CPU utilization (5% CS, 10T, 1L)

Figure 6.73: 10% CS - Actual CPU Power vs. CPU utilization (10% CS, 10T, 1L)

Figure 6.74: 15% CS - Actual CPU Power vs. CPU utilization (15% CS, 10T, 1L)

Figure 6.75: 20% CS - Actual CPU Power vs. CPU utilization (20% CS, 10T, 1L)

Figure 6.76: 30% CS - Actual CPU Power vs. CPU utilization (30% CS, 10T, 1L)

Figure 6.77: 40% CS - Actual CPU Power vs. CPU utilization (40% CS, 10T, 1L)

Figure 6.78: 50% CS - Actual CPU Power vs. CPU utilization (50% CS, 10T, 1L)

Figure 6.79: 60% CS - Actual CPU Power vs. CPU utilization (60% CS, 10T, 1L)

Figure 6.80: 70% CS - Actual CPU Power vs. CPU utilization (70% CS, 10T, 1L)

Figure 6.81: 5% CS - DSR vs. CPU utilization (5% CS ,10T, 1L)

Figure 6.82: 10% CS - DSR vs. CPU utilization (10% CS, 10T, 1L)

Figure 6.83: 20% CS - DSR vs. CPU utilization (20% CS, 10T, 1L)

Figure 6.84: 30% CS - DSR vs. CPU utilization (30% CS, 10T, 1L)

Figure 6.85: 40% CS - DSR vs. CPU utilization (40% CS, 10T, 1L)

Figure 6.86: 50% CS - DSR vs. CPU utilization (50% CS, 10T, 1L)

Figure 6.87: 60% CS - DSR vs. CPU utilization (60% CS, 10T, 1L)

Figure 6.88: 70% CS - DSR vs. CPU utilization (70% CS, 10T, 1L)

6.3 Energy consumption results on the AMD Zacate Board

We have done the Normalized CPU Energy measurements and the System power measurements on this board. Figures 6.89 through 6.98 show the normalized CPU Energy consumption of the different schedulers for the values of ACET ranging from 0.1WCET to 1WCET. From these figures, we observe that the curves in these plots are not that smooth as compared to the curves in the Normalized CPU Energy plot for I5 as shown in the Figures 6.1 through 6.37. This is due to the availability of lesser number of frequency steps in AMD Zacate. This processor has just three frequency steps compared to the ten frequency steps available in the Intel I5 processor. The relative normalized CPU energy savings of the schedulers (relative to Base-EDF) on this board is much lower than that obtained on the Intel I5 laptop, as can be observed from the Figures 6.25 and 6.95 due to the same reason. The variation in the Normalized CPU energy consumption with ACET for fixed CPU utilization value show results similar to the I5 case, as can be seen from the Figures 6.99 through 6.104, due to the similar reasons mentioned in 6.1.2, and so we do not repeat the discussion over here.

Figure 6.89: Normalized CPU Energy vs. CPU utilization (Underload, 0.1 WCET, 5T, NL)

Figure 6.90: Normalized CPU Energy vs. CPU utilization (Underload, 0.2 WCET, 5T, NL)

Figure 6.91: Normalized CPU Energy vs. CPU utilization (Underload, 0.3 WCET, 5T, NL)

Figure 6.92: Normalized CPU Energy vs. CPU utilization (Underload, 0.4 WCET, 5T, NL)

Figure 6.93: Normalized CPU Energy vs. CPU utilization (Underload, 0.5 WCET, 5T, NL)

Figure 6.94: Normalized CPU Energy vs. CPU utilization (Underload, 0.6 WCET, 5T, NL)

Figure 6.95: Normalized CPU Energy vs. CPU utilization (Underload, 0.7 WCET, 5T, NL)

Figure 6.96: Normalized CPU Energy vs. CPU utilization (Underload, 0.8 WCET, 5T, NL)

6.4 System Power results on the AMD Zacate Board

We have also measured the entire system power consumed by this board for ACET=0.3 WCET, 0.6 WCET and 0.9WCET. These measurements are shown in the Figures 6.105, 6.106 and 6.107 respectively. From these figures, we observe that Base-EDF consumes the maximum power and LA-EDF, the least. The power consumption of other schedulers lie between these two extremes. However, the difference between the power consumption of these schedulers is not much, which can be attributed to the high energy efficiency of the idle states of the AMD zacate processor. We have also tested this schedulers using memory intensive workloads. The Figures 6.108, 6.109 and 6.110 show the energy consumption of these schedulers when tested with memory intensive workloads. From these figures we can see that the power consumed by the memory intensive workloads is similar to the power consumed by CPU intensive workloads. From this we can conclude that the power consumption of memory is *DVFS-independent*.

Figure 6.97: Normalized CPU Energy vs. CPU utilization (Underload, 0.9 WCET, 5T, NL)

Figure 6.98: Normalized CPU Energy vs. CPU utilization (Underload, 1 WCET, 5T, NL)

Figure 6.99: Normalized CPU Energy vs. ACET (Underload, 50% CPU Utilization 5T, NL)

Figure 6.100: Normalized CPU Energy vs. ACET (Underload, 60% CPU Utilization 5T, NL)

Figure 6.101: Normalized CPU Energy vs. ACET (Underload, 70% CPU Utilization 5T, NL)

Figure 6.102: Normalized CPU Energy vs. ACET (Underload, 80% CPU Utilization 5T, NL)

Figure 6.103: Normalized CPU Energy vs. ACET (Underload, 90% CPU Utilization, 5T, NL)

Figure 6.104: Normalized CPU Energy vs. ACET (Underload, 100% CPU Utilization, 5T, NL)

Figure 6.105: CPU Intensive Workload : System Power vs. CPU Utilization(0.3 WCET, 5T, NL)

Figure 6.106: CPU Intensive Workload : System Power vs. CPU Utilization(0.6 WCET, 5T, NL)

Figure 6.107: CPU Intensive Workload : System Power vs. CPU Utilization(0.9 WCET, 5T, NL)

Figure 6.108: Memory Intensive Workload : System Power vs. CPU Utilization(0.3 WCET, 5T, NL)

Figure 6.109: Memory Intensive Workload : System Power vs. CPU Utilization(0.6 WCET, 5T, NL)

Figure 6.110: Memory Intensive Workload : System Power vs. CPU Utilization(0.9 WCET, 5T, NL)

6.5 Energy and DSR results for the schedulers designed for dependent task-sets

We have also implemented two schedulers for dependent task-set on this platform, which are HS and REUA using the same 10T-1L task-set mentioned in the Section 6.2. We measured the Normalized CPU energy and the DSR for these two schedulers. We measure the Normalized CPU energy consumption and DSR, by varying the CPU utilization for different values of critical section lengths(CS) ranging from 20 to 100% of WCET. From the Figures 6.111, 6.112, 6.113, 6.114, 6.115, 6.116, 6.117, 6.118, 6.119, which show the DSR vs. utilization plots for CS ranging from 20% of WCET to 100% of WCET, we observe that the normalized CPU energy consumption of HS is much higher than that of REUA. This is because HS takes a very conservative approach and runs at a high frequency, which is calculated based on the maximum blocking time, to ensure that no deadlines are missed. On the other hand, REUA is a very aggressive energy saving algorithm and doesnot consider critical sections or blocking time to calculate the frequency at which to run at. Consequently, with increase in the CS, REUA satisfies lesser deadlines as compared to HS as can be observed from the Figures 6.123, 6.124 6.125, 6.126 and 6.127.

Figure 6.111: 20% CS - Normalized CPU Energy vs. CPU utilization (20% CS, 10T, 1L)

Figure 6.112: 30% CS - Normalized CPU Energy vs. CPU utilization (30% CS, 10T, 1L)

Figure 6.113: 40% CS - Normalized CPU Energy vs. CPU utilization (40% CS, 10T, 1L)

Figure 6.114: 50% CS - Normalized CPU Energy vs. CPU utilization (50% CS, 10T, 1L)

Figure 6.115: 60% CS - Normalized CPU Energy vs. CPU utilization (60% CS, 10T, 1L)

Figure 6.116: 70% CS - Normalized CPU Energy vs. CPU utilization (70% CS, 10T, 1L)

Figure 6.117: 80% CS - Normalized CPU Energy vs. CPU utilization (80% CS, 10T, 1L)

Figure 6.118: 90% CS - Normalized CPU Energy vs. CPU utilization (90% CS, 10T, 1L)

Figure 6.119: 100% CS - Normalized CPU Energy vs. CPU utilization (100% CS, 10T, 1L)

Figure 6.120: 20% CS - DSR vs. CPU utilization (20% CS, 10T, 1L)

Figure 6.121: 30% CS - DSR vs. CPU utilization (30% CS, 10T, 1L)

Figure 6.122: 40% CS - DSR vs. CPU utilization (40% CS, 10T, 1L)

Figure 6.123: 50% CS - DSR vs. CPU utilization (50% CS, 10T, 1L)

Figure 6.124: 60% CS - DSR vs. CPU utilization (60% CS, 10T, 1L)

Figure 6.125: 70% CS - DSR vs. CPU utilization (70% CS, 10T, 1L)

Figure 6.126: 80% CS - DSR vs. CPU utilization (80% CS, 10T, 1L)

Figure 6.127: 90% CS - DSR vs. CPU utilization (90% CS, 10T, 1L)

Figure 6.128: 100% CS - DSR vs. CPU utilization (100% CS, 10T, 1L)

Chapter 7

Conclusions and Future Work

7.1 Conclusions

In this thesis, we evaluated the timeliness and energy consumption behaviours of fourteen RT-DVFS schedulers on two representative hardware platforms. The schedulers include Base-EDF, Static-EDF, CC-EDF, LA-EDF, EUA, REUA, DRA, DRA-OTE, AGR1, AGR2, HS, DS and USFLEDF, and the hardware platforms include ASUS laptop with the Intel I5 processor and a motherboard with the AMD Zacate processor. We implemented these schedulers in the ChronOS real-time Linux kernel and measured their actual timeliness and energy behaviours under a range of workloads including CPU-intensive, memory-intensive, mutual exclusion lock-intensive, and processor-underloaded and overloaded workloads. Such an extensive evaluation of the RT-DVFS algorithms on real hardware platforms has not been done previously. In the past, most of the RT-DVFS algorithms have only been studied through simulations.

Our studies reveal, that actual power savings of these RT-DVFS algorithms are orders of magnitude smaller than their simulation-based savings reported in the literature. We observed that, algorithms such as Static-EDF, CC-EDF, Snowdon-min which were supposed to perform much better than Base-EDF, do not actually do so. The actual energy savings of these algorithms lie in the range of 10-12%. The actual energy savings of aggressive algorithms such as LA-EDF, DRA, DRA-OTE, AGR1 and AGR2 are slightly better, being around 15%.¹ We also observe that the performance of a RT-DVFS algorithm is highly dependent on the number of frequency steps available on the processor. The energy savings on the Intel I5 laptop containing 10 frequency steps, was much higher than the energy savings on the AMD Zacate board, containing only 3 frequency steps.

In almost all the simulation based implementations of RT-DVFS algorithms, the CPU power

¹These results are for a particular case when the test application is subjected to 70% CPU utilization and 0.6 WCET. The energy savings for the other cases are low, but the exact percentage might differ

consumption has been measured as the cube of frequency. Based on our studies we conclude, that measuring the CPU power consumption as the cube of CPU frequency can lead to incorrect conclusions. This is because it ignores the idle state CPU power consumption, which is orders of magnitude smaller than the active state power consumption. In this thesis, we have been able to measure the actual CPU power consumption by using the CPUpowertool [5]. Actual CPU power is measured by considering the CPU power consumption in both the idle and the active states. Our studies show that the CPU power consumption measured as the cube of frequency is totally different from the actual power consumption. Consequently, conclusions made on the performance of these schedulers by their simulation-based implementation do not hold true for real hardware platforms.

We observe that the the actual power savings of the RT-DVFS techniques' that we report are orders of magnitude smaller than their simulation-based savings reported in the literature. This can be explained as follows: Energy consumption of a processor can be given as $E_{cpu} = P_{idle} * t_{idle} + P_{active} * t_{active}$. Here P_{idle} and P_{active} is the idle state and the active state power consumption of the CPU, whereas t_{idle} and t_{active} is the time spent in the idle and active states respectively. If $P_{idle} \ll P_{active}$ then it makes more sense to keep t_{idle} high and t_{active} low. However RT-DVFS algorithms always aim at the minimisation of the idle time. In modern processors, $P_{idle} \ll P_{active}$ is indeed the case. For example in the Intel I5 processor, the power consumed in the active state ranges from 11.25 W to 25 W, whereas the energy consumed in the idle state ranges from 0.3 W to 1 W. Thus by reducing the frequency of the CPU, and increasing t_{active} , the energy savings obtained by the reduction of active power consumption can be offset by completing tasks sooner, by running them at the highest frequency and transitioning to the idle state earlier, so that t_{active} decreases. Thus, we conclude that energy efficiency of the RT-DVFS algorithms is highly dependent on the relative power consumption of the active and the idle states, which in turn, is dependent on the characteristics of the platform used. In both the platforms used by us, the energy efficiency of idle states is very high and consequently, the power savings obtained, was not high. This might be the case with most the modern processors.

Table 7.2 and Table 7.1 show the actual CPU power consumption and the normalized CPU energy consumption (relative to Base-EDF) of the CPU intensive schedulers by running the test application at 70% CPU utilization for the cases when the actual execution time is 0.3, 0.6 and 0.9 of the WCET.

We summarize the conclusions as follows:

- (1) Evaluation of RT-DVFS algorithm by considering the CPU power consumption as the cube of frequency leads to incorrect conclusions. Lets consider the column of 0.6 WCET in the tables, Table 7.1 and Table 7.2. If we consider just the normalized CPU energy consumption of the algorithms such as Static-EDF, CC-EDF and Snowdon-min, we find that these algorithms perform much better than Base-EDF, the energy savings being in the range of 65% to 80%. However, from Table 7.2 we observe that the actual energy savings of these algorithms is just in the range of 10% to 12%.

(2) Aggressive energy saving algorithms such as LA-EDF, DRA, DRA-OTE, AGR1 and AGR2 do perform slightly better than the algorithms mentioned in the previous point. But again, from the tables, Table 7.1 and Table 7.2 (considering the 0.6 WCET column), we observe that actual savings is just around 15%, whereas the normalized CPU energy savings is around 84 to 87%.

(3) We also observe, that in overloads, aggressive algorithms like AGR1, AGR2 and LA-EDF save the maximum power.

(4) Lock based algorithms consume almost similar CPU power, with REUA performing slightly better.

(5) We also observe that the performance of a RT-DVFS algorithm is highly dependent on the number of frequency steps available on the processor. The energy savings on the Intel I5 laptop containing 10 frequency steps, was much higher than the energy savings on the AMD Zacate board, containing only 3 frequency steps.

Table 7.1: Normalized Energy savings of RT-DVFS algorithms

Name	0.3 WCET	0.6 WCET	0.9 WCET
<i>Static-EDf</i>	0.373	0.373	0.373
<i>CC-EDF</i>	0.250	0.250	0.328
<i>LA-EDF</i>	0.125	0.125	0.238
<i>Snowdon-min</i>	0.157	0.185	0.238
<i>DRA</i>	0.166	0.166	0.262
<i>DRA-OTE</i>	0.148	0.166	0.262
<i>AGR1</i>	0.140	0.166	0.262
<i>AGR2</i>	0.140	0.166	0.262

7.2 Future Work

7.2.1 DVFS in CMPs

In this thesis, we have implemented a wide range of single core RT-DVFS scheduling algorithms. This work can be extended by considering the implementation of RT-DVFS algorithms on chip multiprocessors (CMPs). However, implementation of DVFS on CMPs has certain limitations. For example, applying DVFS independently to individual cores, so that the different cores can operate at different voltages, is not possible in CMPs. This is because the cores share a common voltage. Enabling per-CPU DVFS in CMPs would require a lot

Table 7.2: Actual Energy savings of RT-DVFS algorithms

Name	0.3 WCET	0.6 WCET	0.9 WCET
<i>Static-EDf</i>	0.935	0.92	0.914
<i>CC-EDF</i>	0.912	0.89	0.906
<i>LA-EDF</i>	0.872	0.852	0.862
<i>Snowdon-min</i>	0.879	0.877	0.864
<i>DRA</i>	0.872	0.85	0.89
<i>DRA-OTE</i>	0.873	0.853	0.89
<i>AGR1</i>	0.87	0.854	0.863
<i>AGR2</i>	0.87	0.854	0.863

of extra circuitry and complicated designs. Thus, we have to consider CMPs with global frequency and voltage, where techniques such as load balancing, running the tasks on a limited number of cores, and putting other cores into off states can be utilized to reduce the energy consumption.

7.2.2 Considering Real world Applications

In this thesis, we have evaluated the performance of the schedulers using a test application. As the input is provided by the user, such applications enable us to test and evaluate the performance of the schedulers under a wide range of input parameters. However, these kind of dummy applications just burn the CPU and do not do any useful work. It will be really interesting to test the schedulers using real world applications. Particularly, those applications can be considered which are subjected to task time constraints and run on battery powered systems, where energy is a limited resource. One such application that can be considered, is the software which runs on Autonomous Underwater Vehicles (AUVs). These vehicles have sensors which take input from the surroundings, process this input and accordingly produce output commands to control other devices on the vehicle. Thus, the controller code has to satisfy timing constraints, by processing the input commands and producing the output control commands within a particular time frame. Moreover, energy is a critical resource for these systems as they are battery powered. Thus, it would be interesting to see, how these applications would be benefited from RT-DVFS scheduling.

7.2.3 DVFS with DPM

In this thesis, we have particularly considered CPU intensive and memory intensive workloads. Mixed workloads containing CPU intensive portions as well as memory and I/O intensive portions can also be considered. While the time taken by the CPU intensive portion will be dependent on the CPU frequency, the time taken by the remaining portions of the code will be independent of it. DVFS can be used with other energy management techniques, and can be applied to these kinds of mixed workloads. For example, DVFS can be integrated with Dynamic Power Management(DPM). DPM puts devices (for e.g. memory and I/O) to low power or sleep states, when they are not in use. The main challenge in this technique is to decide when and how often, should the devices be put in low power state, as this involves significant amount of time and energy overheads. Thus by integrating DVFS with DPM, DVFS can help to save the CPU power consumption, while DPM can save the power consumption of other devices in the system. In this way the system level energy consumption can be reduced.

7.2.4 DVFS with Procrastination Scheduling

CPU's power consumption includes static and dynamic power. Dynamic power is consumed due to the switching activity when the CPU is performing some operations. Static power consumption is due to the leakage current. With the decrease in the size of the transistors in each generation of technology, the static power consumption has increased manifold. DVFS only aims at the reduction of dynamic power consumption of the CPU. In order to reduce the static power consumption, procrastination scheduling can be considered. This technique utilizes the slack time to delay the task execution, so that the processor can remain in its idle state for a longer time. In the idle state, many parts of the processor are shut down, which results in a considerable reduction of the static power consumption. DVFS can be combined with procrastination scheduling, where the available slack time can be used both for slowdown, as well as for procrastination to maximize the energy savings.

7.2.5 Reducing Dynamic Power Consumption of other devices

Similar to DVFS, dynamic power consumption of other devices like I/O, memory, and the bus can be reduced by scaling their respective frequencies. This in turn, can be integrated with DPM to reduce both the dynamic as well as static power consumption of the devices. In order to satisfy the task time constraints, it is important to find out the time spent by the code while using these devices, so that the frequency can be scaled accordingly. In order to find out this time, performance monitoring counters(PMCs) can be read. PMCs are hardware registers which measure a number of events that take place inside the processor.

Bibliography

- [1] Advanced configuration and power interface specification. <http://www.acpi.info/spec.htm>.
- [2] Amd fusion mini-itx motherboard. <http://www.gigabyte.com/press-center/news-page.aspx?nid=963>.
- [3] Amd powernow! technology. <http://www.amd.com/us/products/technologies/amd-powernow-technology/Pages/amd-powernow-technology.aspx>.
- [4] Chronos - linux based real-time multiprocessor scheduling framework. <http://www.chronoslinux.org>.
- [5] Cpupowerutils. <http://lwn.net/Articles/433002/>.
- [6] Enhanced intel speedstep technology and demand-based switching on linux. <http://software.intel.com/en-us/articles/enhanced-intel-speedstepr-technology-and-demand-based-switching-on-linux/>.
- [7] Fluke 289 true-rms industrial logging multimeter. <http://www.fluke.com/fluke/usen/digital-multimeters/fluke-289.htm?PID=56061>.
- [8] An interactive java-based cpu simulator for use in introductory computer organization classes. <http://www.cs.colby.edu/djskrien/CPUSim/>.
- [9] Linux kernel cpufreq subsystem. <http://www.kernel.org/pub/linux/utils/kernel/cpufreq/cpufreq.html>.
- [10] Sesc: cycle accurate architectural simulator. <http://sesc.sourceforge.net/>.
- [11] Spim a mips32 simulator. <http://pages.cs.wisc.edu/~larus/spim.html>.
- [12] Wind river simics. <http://www.windriver.com/products/simics/>.
- [13] T. A. AlEnawy and H. Aydin. On energy-constrained real-time scheduling. In *Proceedings of the 16th Euromicro Conference on Real-Time Systems*, pages 165–174, Washington, DC, USA, 2004. IEEE Computer Society.

- [14] H. Aydin, V. Devadas, and D. Zhu. System-level energy management for periodic real-time tasks. In *Proceedings of the 27th IEEE International Real-Time Systems Symposium*, pages 313–322, Washington, DC, USA, 2006. IEEE Computer Society.
- [15] H. Aydin, R. Melhem, D. Mossé, and P. Mejía-Alvarez. Power-aware scheduling for periodic real-time tasks. *IEEE Trans. Comput.*, 53:584–600, May 2004.
- [16] T. P. Baker. A stack-based resource allocation policy for realtime processes. In *IEEE Real-Time Systems Symposium*, pages 191–200, 1990.
- [17] E. Bini, G. Buttazzo, and G. Lipari. Minimizing cpu energy in real-time systems with discrete speed management. *ACM Trans. Embed. Comput. Syst.*, 8:31:1–31:23, July 2009.
- [18] S. A. Brandt, S. Banachowski, C. Lin, and T. Bisson. Dynamic integrated scheduling of hard real-time soft real-time and non-real-time processes. In *Proceedings of the 24th IEEE International Real-Time Systems Symposium*, RTSS '03, pages 396–, Washington, DC, USA, 2003. IEEE Computer Society.
- [19] A. Chandrakasan, S. Sheng, and R. W. Brodersen. Low-power CMOS digital design. In *IEEE Journal of Solid-State Circuits*, volume 27, pages 473–484, April 1992.
- [20] K. Choi, K. Dantu, W.-C. Cheng, and M. Pedram. Frame-based dynamic voltage and frequency scaling for a mpeg decoder. In *Proceedings of the 2002 IEEE/ACM international conference on Computer-aided design*, ICCAD '02, pages 732–737, New York, NY, USA, 2002. ACM.
- [21] F. Dabiri, A. Vahdatpour, M. Potkonjak, and M. Sarrafzadeh. Energy minimization for real-time systems with non-convex and discrete operation modes. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '09, pages 1416–1421, 3001 Leuven, Belgium, Belgium, 2009. European Design and Automation Association.
- [22] M. Dellinger. An experimental evaluation of the scalability of real-time scheduling algorithms on large-scale multicore platforms. Master's thesis, Blacksburg, VA, USA, 2011.
- [23] K. Govil, E. Chan, and H. Wasserman. Comparing algorithm for dynamic speed-setting of a low-power cpu. In *Proceedings of the 1st annual international conference on Mobile computing and networking*, MobiCom '95, pages 13–25, New York, NY, USA, 1995. ACM.
- [24] R. Graybill and R. Melhem. *Power Aware Computing*. Kluwer Academic/Plenum Publishers, 2002.
- [25] F. Gruian. Hard real-time scheduling for low-energy using stochastic data and dvs processors. In *Proceedings of the 2001 international symposium on Low power electronics and design*, ISLPED '01, pages 46–51, New York, NY, USA, 2001. ACM.

- [26] D. Grunwald, C. B. Morrey, III, P. Levis, M. Neufeld, and K. I. Farkas. Policies for dynamic clock scheduling. In *Proceedings of the 4th conference on Symposium on Operating System Design & Implementation - Volume 4*, OSDI'00, pages 6–6, Berkeley, CA, USA, 2000. USENIX Association.
- [27] C.-H. Hsu and U. Kremer. The design, implementation, and evaluation of a compiler algorithm for cpu energy reduction. In *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, PLDI '03, pages 38–48, New York, NY, USA, 2003. ACM.
- [28] C.-M. Hung, J.-J. Chen, and T.-W. Kuo. Energy-efficient real-time task scheduling for a dvs system with a non-dvs processing element. In *Proceedings of the 27th IEEE International Real-Time Systems Symposium*, pages 303–312, Washington, DC, USA, 2006. IEEE Computer Society.
- [29] M. J. Jan-Derk Bakker, Erik Mouw and J. Pouwelse. The lart pages. <http://www.lartmaker.nl/projects/scaling/>.
- [30] R. Jejurikar and R. Gupta. Energy aware task scheduling with task synchronization for embedded real time systems. In *Proceedings of the 2002 international conference on Compilers, architecture, and synthesis for embedded systems*, CASES '02, pages 164–169, New York, NY, USA, 2002. ACM.
- [31] W. Kim, D. Shin, H.-S. Yun, J. Kim, and S. L. Min. Performance comparison of dynamic voltage scaling algorithms for hard real-time systems. In *Proceedings of the Eighth IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'02)*, RTAS '02, pages 219–, Washington, DC, USA, 2002. IEEE Computer Society.
- [32] U. Ko and P. T. Balsara. High-performance energy-efficient d-flip-flop circuits. *IEEE Trans. Very Large Scale Integr. Syst.*, 8:94–98, February 2000.
- [33] M. P. Lawitzky, D. C. Snowdon, and S. M. Petters. Integrating real time and power management in a real system. In *Proceedings of the 4th Workshop on Operating System Platforms for Embedded Real-Time Applications*, Prague, Czech Republic, Jul 2008.
- [34] S. Liu, Q. Qiu, and Q. Wu. Energy aware dynamic voltage and frequency selection for real-time systems with energy harvesting. In *Proceedings of the conference on Design, automation and test in Europe*, DATE '08, pages 236–241, New York, NY, USA, 2008. ACM.
- [35] S. Liu, Q. Wu, and Q. Qiu. An adaptive scheduling and voltage/frequency selection algorithm for real-time energy harvesting systems. In *Proceedings of the 46th Annual Design Automation Conference*, DAC '09, pages 782–787, New York, NY, USA, 2009. ACM.

- [36] X. Liu, P. Shenoy, and M. D. Corner. Chameleon: Application-level power management. *IEEE Transactions on Mobile Computing*, 7:995–1010, August 2008.
- [37] J. R. Lorch and A. J. Smith. Improving dynamic voltage scaling algorithms with pace. *SIGMETRICS Perform. Eval. Rev.*, 29:50–61, June 2001.
- [38] R. Love. *Linux Kernel Development*. Novell Press, 2005.
- [39] Y.-H. Lu, L. Benini, and G. De Micheli. Power-aware operating systems for interactive systems. *IEEE Trans. Very Large Scale Integr. Syst.*, 10:119–134, April 2002.
- [40] T. L. Martin. *Balancing batteries, power, and performance: system issues in cpu speed-setting for mobile computing*. PhD thesis, Pittsburgh, PA, USA, 1999. AAI9950023.
- [41] A. Miyoshi, C. Lefurgy, E. Van Hensbergen, R. Rajamony, and R. Rajkumar. Critical power slope: understanding the runtime effects of frequency scaling. In *Proceedings of the 16th international conference on Supercomputing*, ICS '02, pages 35–44, New York, NY, USA, 2002. ACM.
- [42] I. Molnar. Config preempt realtime, fully preemptible kernel, vp-2.6.9-rc4-mm1-t4. <http://lwn.net/Articles/105948/>.
- [43] T. Pering, T. Burd, and R. Brodersen. The simulation and evaluation of dynamic voltage scaling algorithms. In *Proceedings of the 1998 international symposium on Low power electronics and design*, ISLPED '98, pages 76–81, New York, NY, USA, 1998. ACM.
- [44] P. Pillai and K. G. Shin. Real-time dynamic voltage scaling for low-power embedded operating systems. In *Proceedings of the eighteenth ACM symposium on Operating systems principles*, SOSP '01, pages 89–102, New York, NY, USA, 2001. ACM.
- [45] J. Pouwelse, K. Langendoen, and H. Sips. Dynamic voltage scaling on a low-power microprocessor. In *Proceedings of the 7th annual international conference on Mobile computing and networking*, MobiCom '01, pages 251–259, New York, NY, USA, 2001. ACM.
- [46] D. Shin, J. Kim, and S. Lee. Low-energy intra-task voltage scheduling using static timing analysis. In *Proceedings of the 38th annual Design Automation Conference*, DAC '01, pages 438–443, New York, NY, USA, 2001. ACM.
- [47] T. Simunic, L. Benini, P. Glynn, and G. De Micheli. Dynamic power management for portable systems. In *Proceedings of the 6th annual international conference on Mobile computing and networking*, MobiCom '00, pages 11–19, New York, NY, USA, 2000. ACM.
- [48] D. Snowdon, S. Ruocco, and G. Heiser. Power management and dynamic voltage scaling: Myths and facts, 2005.

- [49] A. G. M. Strollo, E. Napoli, and D. De Caro. New clock-gating techniques for low-power flip-flops. In *Proceedings of the 2000 international symposium on Low power electronics and design*, ISLPED '00, pages 114–119, New York, NY, USA, 2000. ACM.
- [50] W. Wang and P. Mishra. Predvs: preemptive dynamic voltage scaling for real-time systems using approximation scheme. In *Proceedings of the 47th Design Automation Conference*, DAC '10, pages 705–710, New York, NY, USA, 2010. ACM.
- [51] M. Weiser, B. Welch, A. Demers, and S. Shenker. Scheduling for reduced cpu energy. In *Proceedings of the 1st USENIX conference on Operating Systems Design and Implementation*, OSDI '94, Berkeley, CA, USA, 1994. USENIX Association.
- [52] H. Wu, B. Ravindran, and E. D. Jensen. On bounding energy consumption in dynamic, embedded real-time systems. In *Proceedings of the 2006 ACM symposium on Applied computing*, SAC '06, pages 933–934, New York, NY, USA, 2006. ACM.
- [53] H. Wu, B. Ravindran, E. D. Jensen, and P. Li. Energy-efficient, utility accrual scheduling under resource constraints for mobile embedded systems. In *Proceedings of the 4th ACM international conference on Embedded software*, EMSOFT '04, pages 64–73, New York, NY, USA, 2004. ACM.
- [54] W. Yuan and K. Nahrstedt. Energy-efficient soft real-time cpu scheduling for mobile multimedia systems. pages 149–163. ACM Press, 2003.
- [55] W. Yuan and K. Nahrstedt. Practical voltage scaling for mobile multimedia devices. In *Proceedings of the 12th annual ACM international conference on Multimedia*, MULTIMEDIA '04, pages 924–931, New York, NY, USA, 2004. ACM.
- [56] H. Yun, P.-L. Wu, A. Arya, T. Abdelzaher, C. Kim, and L. Sha. System-wide energy optimization for multiple dvs components and real-time tasks. In *Proceedings of the 2010 22nd Euromicro Conference on Real-Time Systems*, ECRTS '10, pages 133–142, Washington, DC, USA, 2010. IEEE Computer Society.
- [57] F. Zhang and S. T. Chanson. Processor voltage scheduling for real-time tasks with non-preemptible sections. In *Proceedings of the 23rd IEEE Real-Time Systems Symposium*, RTSS '02, pages 235–, Washington, DC, USA, 2002. IEEE Computer Society.
- [58] Y. Zhang, Z. Lu, J. Lach, K. Skadron, and M. R. Stan. Optimal procrastinating voltage scheduling for hard real-time systems. In *Proceedings of the 42nd annual Design Automation Conference*, DAC '05, pages 905–908, New York, NY, USA, 2005. ACM.
- [59] D. Zhu, N. AbouGhazaleh, D. Mosse, and R. Melhem. Power aware scheduling for and/or graphs in multi-processor real-time systems. In *IEEE ICPP*, August 2002.
- [60] J. Zhuo and C. Chakrabarti. System-level energy-efficient dynamic task scheduling. In *Proceedings of the 42nd annual Design Automation Conference*, DAC '05, pages 628–631, New York, NY, USA, 2005. ACM.

- [61] J. Zhuo and C. Chakrabarti. Energy-efficient dynamic task scheduling algorithms for dvs systems. *ACM Trans. Embed. Comput. Syst.*, 7:17:1–17:25, January 2008.