FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Energy and Temperature Aware Real-Time Systems

Muhammad Ali Awan

DISSERTATION

Doctoral Program in Electrical and Computer Engineering

Supervisor: Stefan Markus Ernst Petters

September 16, 2014

Energy and Temperature Aware Real-Time Systems

Muhammad Ali Awan

Doctoral Program in Electrical and Computer Engineering

Approved by:

President: Dr. Jose Alfredo Ribeiro da Silva Matos
External Referee: Dr. Gerhard Fohler
External Referee: Dr. Marko Bertogna
FEUP Referee: Dr. Luis Miguel Pinho Almeida
FEUP Referee: Dr. Mario Jorge Rodrigues de Sousa
Supervisor: Dr. Stefan Markus Ernst Petters

September 16, 2014

Abstract

Modern embedded systems have increasingly penetrated our daily life, and have facilitated and accelerated our regular activities. Some of these systems are constrained with strict timing requirements, and have limited and/or intermittent power supply. One of the major challenges in the design process of such systems is to minimise their energy consumption and thus to increase the battery life and enhance their mobility. In order to address this objective, it is important to understand the current trends in the embedded systems industry. With progressing CMOS technology miniaturisation, the leakage power dissipation — once neglected — has become a major contributor to the overall power dissipation of modern embedded systems and as a matter of fact it has started to dominate its counterpart, the dynamic power dissipation. To cope with current trend of increasing leakage current, hardware vendors have equipped modern embedded processors with several sleep states and reduced the overhead (energy/time) of a sleep transition. Secondly, there is a trend towards an increased number of devices, as an ever increasing need for extra functionality in a single embedded system demands for extra Input/Output (I/O) devices, which are expensive in terms of energy consumption. Similar to processors, these devices are also equipped with low power sleep states to reduce their energy consumption. Thirdly, modern embedded processors have started to suffer from thermal issues due to increase in power density. It is essential to keep the temperature within recommended limits for the safe operation of the system and to increase the durability/reliability of hardware platforms. Finally, the CMOS industry experienced a paradigm shift in the last decade from single processor design to multicore hardware platforms as the clock frequency cannot be further increased efficiently to enhance the performance of the system. This is driven by the increase in performance per watt ratio that demands special packaging techniques to dissipate the generated heat at high frequencies.

This dissertation attempts to provide energy efficient solutions and techniques to cope with the aforementioned arising trends, while closing the gap between theoretical research and practice. In particular, it focuses at the operating-system-level power management and exploits the available sleep states to improve on energy efficiency while mainly concentrating on the leakage power dissipation. Uniprocessor power management has been widely explored in the last two decades. Several procrastination approaches has been proposed in the literature to deal with the leakage current. However, these solutions approximate the procrastination interval to ease the analysis and sub-optimally utilise the available resources to minimise energy consumption. Such approximation is eliminated in this dissertation with the optimal algorithm to maximise energy savings. A practical limitation of the procrastination scheduling algorithm is relaxed by eliminating the need for an external hardware to implement the power saving algorithm. These newly developed algorithms with low complexity save energy comparable to procrastination scheduling. Furthermore, this dissertation demonstrates that idealised dynamic voltage and frequency scaling, and the thermally constrained dynamic power management are equivalent in nature. Hence, existing solutions proposed for dynamic voltage and frequency scaling can be easily ported to increase energy efficiency in thermally constrained systems.

Intra-task I/O device scheduling was vastly ignored in the past due to an increased overhead of sleep transitions. A decrease in sleep transition overheads allows to explore this new paradigm of device scheduling. This solution not only minimises the pessimism involved in traditional device scheduling algorithms but also reduces the online overhead of scheduling algorithms and has the flexibility to scale easily with an increase in I/O devices. Finally, this dissertation addresses the power management in the context of multicore hardware platforms. Global scheduling algorithms have become an attractive choice to schedule applications on a homogeneous multicore platform. The proposed energy saving algorithm exploits the spare capacity in the schedule and exploits the sleep states available in homogeneous multicore platform to save energy consumption. Heterogeneous multicore platforms are famous in modern computing to perform specific tasks efficiently. Energy efficient mapping on heterogeneous multicore platforms addressed in the literature considers only dynamic power dissipation while assuming leakage power dissipation a constant factor. Opposed to the state-of-the-art, the proposed allocation heuristics in the thesis are divided into two phases to tackle both dynamic and leakage power dissipation. All the algorithms proposed in this dissertation are evaluated with extensive set of simulations for a variety of hardware platforms and workloads.

Resumo

É um facto constatado que os sistemas embebidos têm tomado um lugar relevante na nossa vida quotidiana, tendo facilitado e até acelerado as nossas actividades diárias. Alguns destes sistemas caracterizam-se por requisitos temporais bastante rigorosos e são alimentados por fontes de energia limitadas e/ou intermitentes. Um dos maiores desafios no projecto deste tipo de sistemas consiste em minimizar o seu consumo de energia e, consequentemente, aumentar a sua autonomia e mobilidade. De forma a atingir este objectivo, é fundamental compreender as tendências actuais na indústria dos sistemas embebidos. Com a progressiva miniaturização da tecnologia CMOS, a potência devida à corrente de fuga – anteriormente desprezável – tornou-se numa das principais contribuições para o total da potência dissipada. Na realidade, a potência da corrente de fuga consegue já ultrapassar em certos casos aquela que era a principal fonte de dissipação de potência nos circuitos CMOS: a potência dinâmica, associada à transição entre estados. Para lidar com esta crescente potência da corrente de fuga, os fabricantes de circuitos equiparam os actuais processadores embebidos com vários estados de latência (*sleep modes*) e reduziram os custos energéticos e temporais associados a uma transição por um estado latente. Adicionalmente, há a tendência de se aumentar o número de dispositivos incluídos num único sistema embebido, devido à crescente complexidade da funcionalidade exigida às aplicações embebidas, requerendo um maior número de dispositivos de entrada-saída (I/O), traduzindo-se na prática por um aumento do consumo energético. Tal como no caso dos processadores, estes dispositivos também estão equipados com estados de latência, de forma a reduzir o consumo de energia. Um outro ponto a ter em conta relaciona-se com os problemas térmicos, devidos ao aumento da densidade de potência, presentes nos actuais processadores embebidos. É fundamental manter a temperatura dentro dos limites especificados para a operação segura do sistema e aumentar da durabilidade/fiabilidade da plataforma computacional. Por último, o paradigma de fabrico CMOS evoluiu na última década, do projecto de sistemas com um único processador para plataformas com múltiplos núcleos de execução (*multi-core*), pois tornou-se impossível continuar a obter ganhos de desempenho através do aumento da frequência de relógio. Esta mudança é motivada pelo aumento da relação de desempenho por watt, através de técnicas especiais de desenho dos circuitos integrados que permitem dissipar o calor gerado a altas-frequências.

Esta dissertação apresenta um conjunto de novas soluções eficientes do ponto de vista energético para lidar com as tendências previamente referidas, estabelecendo simultaneamente a ponte entre a investigação teórica e a prática. Este trabalho centra-se em particular na gestão de energia ao nível do sistema operativo, e explora os estados de latência disponíveis para melhorar a eficiência energética, concentrando-se na dissipação de potência devida às correntes de fuga. A gestão de energia em sistemas uniprocessador foi largamente explorada nas últimas duas décadas. Neste período, publicaram-se várias abordagens baseadas na procrastinação de tarefas para lidar com o problema da corrente de fuga. No entanto, estas soluções estimam um valor aproximado do intervalo de procrastinação para facilitar a análise e utilizar de forma sub-óptima os recursos disponíveis para minimizar o consumo de energia. Este trabalho conseguiu eliminar a referida

aproximação com um algoritmo óptimo para maximização da poupança de energia. A limitação prática do algoritmo de escalonamento com procrastinação de tarefas é relaxado através da eliminação da utilização de *hardware* externo para implementar o algoritmo de poupança de energia. Estes novos algoritmos de baixa complexidade, desenvolvidos neste trabalho, atingem poupanças de energia comparáveis ao escalonamento com procrastinação de tarefas. Além disso, esta dissertação demonstra como a variação dinâmica ideal de tensão e frequência, e a gestão dinâmica de consumo de potência baseada em factores térmicos são, por natureza, equivalentes. Desta forma, as actuais soluções propostas para variação dinâmica de tensão e frequência podem ser facilmente convertidas para aumentar a eficiência energética em sistemas com restrições térmicas.

O escalonamento de dispositivos de entrada-saída ao nível da tarefa tem sido negligenciado devido aos custos elevados de transições por estados de latência. A diminuição desses custos permite explorar este novo paradigma de escalonamento de dispositivos. Esta solução não só minimiza o pessimismo relacionado com os algoritmos tradicionais de escalonamento de dispositivos como também reduz os custos de execução dos algoritmos de escalonamento, possuindo a flexibilidade necessária para facilmente acompanhar um número crescente de dispositivos de entrada-saída. Por fim, esta dissertação aborda a gestão de potência no contexto das plataformas baseadas em arquitecturas de processadores com múltiplos núcleos de execução (*multi-core*). Os algoritmos de escalonamento globais tornaram-se uma opção interessante para ordenar a execução de tarefas em plataformas cujos múltiplos são homogéneos. O algoritmo para poupança de energia proposto, explora a capacidade excedente do sistema decorrente do escalonamento, bem como os estados de latência disponíveis nestas plataformas de núcleos homogéneos, afim de reduzir o consumo de energia. As plataformas de núcleos heterogéneos são reconhecidas pela capacidade de realizar eficientemente tarefas específicas. Os processos de afectação de tarefas por núcleos de execução baseada em critérios de eficiência energética publicados até hoje, consideram apenas a dissipação dinâmica de potência assumindo um factor constante para a potência devida à corrente de fuga. Em oposição ao estado-da-arte actual, as heurísticas de afectação proposta nesta dissertação dividem-se em duas fases para abordar tanto a dissipação de potência dinâmica como a dissipação de potência de fuga. Todos os algoritmos propostos nesta dissertação são avaliados através de um extenso conjunto de simulações para uma variedade de plataformas computacionais submetidas a diversas cargas.

Acknowledgements

A PhD milestone demands dedication, hard work, patience, concentration, motivation and support from people around you. Many individuals made this challenging milestone easier for me and paved the way to my success. First of all, I would like to express my very great appreciation and gratitude to my supervisor Stefan M. Petters for his valuable ideas, constructive discussions, useful feedback and excellent guidance. He never let me down at any stage and kept my motivation alive throughout my PhD process. I am very grateful to him for providing me such an exciting opportunity to work on this interesting topic, sharing his vast experience in this domain and encouraging me to work on different problems of my choice within this topic. I have learned a lot during his extraordinary supervision. I would also like to thank Eduardo Tovar for providing us an ideal research environment in CISTER. He was always accessible to solve our issues. I am also thankful to Stefan and Eduardo for arranging funds to attend conferences and summer schools. Also, I am grateful to Ines Almeida, Sanda Almeida and Cristiana Barros for taking care of administrative stuff in CISTER and Portugal. Especially, I really appreciate the effort of Ines for solving our visa-related issues and helping us with the local authorities here in Portugal. My special thanks to the technical stuff for providing us an excellent working environment in CISTER and allowing us to use lab resources for our experiments.

I would like to express my gratitude to my colleague and a good friend Patrick Meumeu Yomsi for improving my theorem proving skills and sharing interesting research ideas in the last two years of my PhD that resulted in reputed conference publications. I am extremely thankful to Geoffrey Nelissen, whose ideas on global power management and partitioned allocation problem will hopefully result in potential good quality publications. I really enjoyed working with Patrick and Goeffrey because of their clear thoughts and pragmatic approach to solve problems. My lab mate and a very good friend, Borislav Nikolic, helped me to develop the SPARTS simulator used in this thesis to evaluate the performance of different algorithms. I enjoyed his company as a friend and as a colleague. I wish to acknowledge the help of Gurulingesh Raravi and Vikram Gupta in the initial phases of my study of partitioned multicore power management problem. The discussions we had really helped me to understand the nature of the problem. I am particularly grateful to Dakai Zhu and Jian-Jia Chen for providing useful comments in the initial phases of my thesis research plan. I would also like to thank Antonio Barros, Paulo Baltarejo Sousa and Joao Loureiro for translating the abstract of my thesis to Portuguese language. I feel myself very lucky to share the workspace with Dakshina Dasari, Hazim Ali, Borislav, Artem Burmyakov and Kostiantyn Berezovskyi. You people are a great company. With such people around you never feel bored at work. Thanks to Farhan, Mushtaq, Guru, Dakshina, Anuj, Kritika, Ganga, Sujit, Shashank, Vikram, Vincent, Geoffrey, Patrick, Arif, Saqlain, Bilal, Arsalan, Ajmal and Asif for great parties, gaming nights and delicious food. These people made my PhD journey memorable.

I would not be here without a support from my family. I would like to express my deepest gratitude to my parents Altaf Hussain Awan and Shamim Akhtar for their assistance and support throughout my life. They always encouraged me to follow my dreams and avail the opportunities

in my best interest. I know it was always hard for you to stay away from me but you allowed me to leave the country for my better future. Without your support this dream of getting PhD was almost impossible. I would like to thank my sisters Tahira Naz and Farhat Yasmeen for their support, encouragement, motivation and well wishes. My special thanks to my cousin and best friend Zahid Imran for taking care of my parents in my absence. Finally, I would like to thank my dear wife Kiran Ali, who supported and motivated me throughout my PhD process. She bare with me in-spite of long working hours and mood swings. I would like to avail this opportunity to thank her for the delicious food and a great company.

This work was supported by FCT (Portuguese Foundation for Science and Technology) and by ESF (European Social Fund) through POPH (Portuguese Human Potential Operational Program), under PhD grant SFRH/BD/70701/2010..

Muhammad Ali Awan

"Education is not the learning of facts,
but the training of the mind to think."

Albert Einstein

Contents

List of Figures

List of Tables

List of Algorithms

List of Acronyms

ABS	Anti-lock breaking system
ACU	Air-bag control unit
API	Application programming interface
ASIC	Application specific integrated circuit
BCET	Best-case execution time
BE	Best effort
BET	Break-even-time
ccEDF	Cycle-conservative earliest deadline first
CMOS	Complementary metal-oxide-semiconductor
COLORS	Composite low-power scheduling framework
Cons	Consumption
CPU	Central processing unit
DBF	Demand bound function
DBFP	Demand bound function based procrastination
DD	Density difference
DFA	Dynamic frequency allocation
DFA-LP	Dynamic frequency allocation with reduced pessimism
DFR-RMS	Device forbidden regions algorithm for rate monotonic schedulers
DIBL	Drain-induced barrier lowering
DJP	Dynamic job priority
DM	Deadline monotonic
DM-PM	Deadline monotonic with priority migration
Don	Donation
DPM	Dynamic power management
DTM	Dynamic thermal management
DVFS	Dynamic voltage and frequency scaling
DVS	Dynamic voltage scaling
ED	Energy density
EDF	Earliest deadline first
EDS	Energy-optimal device scheduler
EEC	Expected energy consumption
EEDS	Energy efficient device scheduling
ERTH	Enhanced race-to-halt
ESSR	Execution slack service register
FF	First-fit
FIFO	First-in-first-out
FJP	Fixed job priority
FPGA	Field programmable gate array

FRT	Firm real time
FTP	Fixed task priority
GEDF	Global earliest deadline first
GIDL	Gate-induced drain leakage
Global-EDF	Global earliest deadline first
GPM	Global power management
GPS	Global positioning system
HDMI	High-definition multimedia interface
HPW	High priority workload
HRT	Hard real-time
HyWGA	Hybrid worst-fit genetic algorithm
I/O	Input/Output
IC	Integrated circuit
ILP	Integer linear programming
IPW	Intermediate priority workload
IRTH	Improved race-to-halt
ISR	Interrupt service routine
ITRS	International technology roadmap for semiconductors
LBET	Low break-even-time
LC-DP	Leakage control dynamic priority
LC-EDF	Leakage control earliest deadline first
LCM	Least common multiple
LEDES	Low energy device scheduler
LLED	Least Lost energy density
LLED-SP	Least lost energy density and second phase
LLF	Least laxity first
LLREF	Largest local remaining execution first
LP	Linear programming
LPW	Low priority workload
LQS	Low-power quasi-dynamic scheduling
LRE-TL	Local remaining execution TL-Plane
LWRTH	Light-weight race-to-halt
MDO	Maximum device overlap
MM	Maximum minimum
MM-SP	Maximum minimum with second phase
MOSFET	Metal-oxide-semiconductor field-effect transistors
MT	Matrix
MUSCLES	Multi-state constrained low-energy scheduler
nMOS	n-type metal-oxide-semiconductor field-effect transistors
NS	Without sleep states
PARTPN	Power-aware real-time petri-nets
PDMS_HPTS	Partitioned deadline monotonic scheduling with highest priority task split
PF	Proportionate progress
PLL	Phase lock loop
pMOS	p-type metal-oxide-semiconductor field-effect transistors
PROC	Procrastination algorithm based on Jejurikar et al. [JPG04] method
PUB	Period upper bound
RAM	Random access memory

RBED	Rate-based earliest deadline first
REC	Reducible energy consumption
RM	Rate monotonic
ROM	Read only memory
RT	Real-time
RTH	Race-to-halt
RTOS	Real-time operating system
SBET	Small break-even-time
SBF	Supply bound function
SFA	Static frequency allocation
SMP	Symmetric multicore platform
SMS	Short message service
SP	Second phase
SPARTS	Simulator for power aware and real-time systems
SRA	Slack reclamation algorithm
SRT	Soft real-time
SSC	Static slack container
SSSR	Static slack service register
staticEDF	Static earliest deadline first
TCDPM	Thermally constrained dynamic power management
TE	Total energy
TTL	Transistor-transistor logic
USB	Universal serial bus
WCET	Worst-case execution time
WFD	Worst-fit decreasing

List of Symbols

Hardware Platform	**Symbols**
Hardware Platform	π
No of processor types	M
Processor index	m
Processor type m	π^m
Task-set allocated to a processor type m	τ^m
Active power of a processor	P_A^m
Idle power of a processor	P_I^m
Vector of sleep states	$\vec{\S}^m$
Sleep index	n
Number of sleep states	N
Sleep state of a processor	$\S_n^m$
Power of a sleep state	P_n^m
Transition delay of going into a sleep state	ts_n^m
Transition delay of going out of a sleep state	tw_n^m
Complete transition delay	tsw_n^m
Transition delay	tr_n^m
Power dissipated in transition phase	Ptr_n^m
Energy overhead of a sleep state	Es_n^m
Break-even-time of a sleep state	bet_n^m
Average sleep energy	$\bar{E}^{\S_n^m}$
Sleep threshold	Ψ
Vector of frequencies	$\vec{f}^m$
Frequency index	v
Number of frequencies	V^m
Frequency of a processor	f_v^m
Power dissipation at a frequency	$P_{f_v^m}^m$
Critical speed	f_{crit}^m
Dynamic power dissipation	P_{dyn}
Leakage power dissipation	P_{lkg}
Short circuit power dissipation	P_{short}
Total power dissipation	P_{total}
Energy	E
Expected energy consumption	EEC_v^m
Frequency Combination	Λ_i
Set of frequency combinations	Λ
Energy consumption per unit time in the idle mode	Sf_e
Total Energy	TE

Speed up factor	κ^m
Helper variable	ζ
Average capacity of the heterogeneous platform	U^{avg}
Effective utilisation of the hardware platform	U^{eff}

System Model	**Symbols**
Time	t
Task-set	τ
Task-set size	ℓ
Total utilisation	U
Task index	i
Task	τ_i
Worst-case execution time	C_i
Average-case execution time	$\bar{C}_i$
Relative deadline	D_i
Minimum inter-arrival time	T_i
Average minimum inter-arrival time	$\bar{T}_i$
Actual allocated budget	A_i
Individual Task utilisation	U_i
Job index	k
Job	$j_{i,k}$
Absolute deadline	$d_{i,k}$
Release time	$r_{i,k}$
Current budget	$a_{i,k}$
Actual execution time	$c_{i,k}$
Hyper-period	H
Task-set distribution	ξ
Sporadic delay limit	Γ
Best-case execution time limit	C^b
Sporadic delay limit of a task	Γ_i
Best-case execution time limit of a task	C_i^b
Vector of execution profiles of τ_i on different core types	$\vec{C}_i$
Vector of WCET of a task on a specific core type at different frequencies	$\vec{C}_i^m$
WCET at specific frequency	$C_{i,v}^m$
WCET at maximum frequency	C_i^m
Average execution time at maximum frequency	$\bar{C}_i^m$
Vector of average energy consumption profiles of τ_i on different core types	$\vec{\bar{E}}_i$
Vector of average energy consumption of a task on a specific core type at different frequencies	$\bar{E}_i^m$
Average energy consumption at specific frequency	$\bar{E}_{i,v}^m$
Average energy consumption at maximum frequency	$\bar{E}_i^m$
Utilisation of a core type m	U^m
Utilisation of a core type m at frequency v	$U^{m,v}$
Minimum idle interval or Static sleep interval	χ_{min}^m
Set of static sleep interval	χ^m
Shortest gap in the schedule	ρ
Minimum procrastination interval computed through LC-EDF	Q_{min}

Minimum procrastination interval computed through PROC	Z_{min}
Timer	ϖ
Individual utilisation of task on a core type m at frequency v	$U_{i,v}^{m}$
Characteristics factor to model task's behaviour	β
Density difference	DD_i^m
Energy density	ED_i^m
Group of tasks to enable specific sleep state	G_n^m
Local cost of migration of a task	$LC_{\tau_i}^m$

Device Model	**Symbols**
Number of devices	W
Set of Devices	λ
Device	λ_i
Active power dissipation of a device	$P_A^{\lambda_i}$
Vector of device sleep states	$\vec{\S}^{\lambda_i}$
Sleep state of a device	$\S_n^{\lambda_i}$
Power dissipation of a device in a sleep state	$P_n^{\lambda_i}$
Transition overhead of going into a device's sleep state	$ts_n^{\lambda_i}$
Wake-up transition overhead of a device's sleep state	$tw_n^{\lambda_i}$
Complete transition overhead of a device's sleep state	$tsw_n^{\lambda_i}$
Transition overhead of a device's sleep state	$tr_n^{\lambda_i}$
Power dissipated in the transition phase of a device's sleep state	$Ptr_n^{\lambda_i}$
Energy consumption in the transition phase of a device's sleep state	$Es_n^{\lambda_i}$
Break-even-time of a device's sleep state	$bet_n^{\lambda_i}$
Static slack container	SSC
Offline algorithm for multiple sleep states devices	SSC^{o}
Static slack container algorithm with multiples sleep states per devices	SSC^{m}
Aggressive static slack container algorithm with multiples sleep states per devices	SSC^{a}
Set of all sleep states of devices in the system	ϕ
Intra-task device compatible	Φ
Device budget	D_b
Pending high priority workload	Ξ_i^t
Device transition start time	λ_i^{start}
Device ready time	λ_i^{ready}
Device in transition phase	λ_t
Energy density function of a device	$\lambda_i^{ED_n}$
Device percentage usage time	Ω
Next utilisation time of a device	λ_i^{NUT}

Slack Sources	**Symbols**
Execution slack	S_e
Size of execution slack	S_e^{sz}
Deadline of execution slack	S_e^{dl}
Usable execution slack	S_e^{u}
Not usable execution slack	S_e^{nu}
Usable idle slack	S_i^{u}
Parallel execution slack	S_e^{p}

Set of next earliest release time	γ
Earliest release time of a task	γ_i
i^{th} element of sorted γ	$\gamma^{(i)}$

Thermal Aware Design	**Symbols**
Temperature	Tm
Available utilisation	U_{avail}
Requested utilisation	U_{req}
Time of cooling phase	t_c
Time of active phase	t_a
Start time of active phase	$\hat{t}$
Temperature at time instant $\hat{t}+t$	$T_{act}(\hat{t},t)$
Start time of cooling phase	$\check{t}$
Temperature at the end of the interval $(\check{t},\check{t}+t]$	$T_{dor}(\check{t},t)$
Current	I
Average current	$\bar{I}$
Voltage	V
Variability factor in hardware characteristics	α

Chapter 1

Introduction

1.1 Embedded Systems

The technology evolution has made embedded systems an integral part of our life. These systems perform a set of dedicated functions and interact with their environment. In fact most of the embedded systems are hidden from our eyes and thus make us forget their existence. These sophisticated systems are rapidly replacing complex jobs previously preformed by human evolving our society to the era of automation. These systems not only reduce the risk of failure as humans are prone to errors but also provide increased precision and high efficiency previously not possible with human interaction. Up to some extent, the credit goes to these systems that have raised our quality of life in this modern era of computing. Nowadays, embedded systems are deployed in various aspect of our life. Typical domains in which such systems are deployed includes consumer electronics, medical equipment, avionics, automotive industry, banking, and defence industry [Noe05, Nel11]. The list is not limited to the aforementioned domains. Despite their existence in a variety of different domains, the basic principles of their design tend to resemble. Before going into the details of embedded system design, trends, challenges and constraints, lets visit a definition of this term. The term "embedded system" is not rigorously defined in the literature. Experts in the field have come up with different meaning of this term corresponding to different properties, features and constraints of embedded systems. Some of the definitions from various experts in the domain are summarised by Raj Kamal [Kam03]. In the context of this thesis, an embedded system is defined as follows.

Definition 1. *An embedded system is a microprocessor-based system composed of hardware, software and/or mechanical components to perform a dedicated function or a range of functions.*

These dedicated functions vary from a simple task of toasting a slice of bread to an air traffic control system that involves numerous workstations, networks and radar sites. Nevertheless, an embedded system is still considered different from general purpose computer system designed to satisfy a variety of end-user requirements. A general purpose computer system provides a flexibility to craft the system according to the needs of a user and designed to run a variety of applications. The desired functionality of an embedded system is usually known at design time.

The information on dedicated function or a range of functions that an embedded system is desired to perform allows to design these systems with optimised software and hardware capabilities.

In general, an embedded system is designed to provide extra reliability over its counterpart general purpose computing system such as a personal computer. Some embedded systems are mission critical such as aircraft flight control and satellites, and any malfunction in such systems can risk human life, equipment damage, property loss and mission failure. Embedded systems deployed in avionics, automotive industry, industrial controllers and military equipments have to deal with vibration, shock, extreme heat, cold and radiations. Contrary to personal computers, the luxury of a software update is also sometimes trickier as these systems are embedded inside a big system and/or deployed in remote areas such as undersea applications or space voyagers. These system must have a mechanism to solve its issues remotely. On top of this, any faults that leads to a failure of the system can also destroy the reputation of a manufacturer. Therefore, such systems are exhaustively tested in their design phase to ensure their functional correctness. Such reliability in personal computers is hard to maintain due to the dynamic nature of applications designed by various third party companies with different tools and made compatible for a variety of hardware platforms available in the market.

Another strict requirement over the dimensions (weight and size) of an embedded system is usually dictated by aesthetics or a limitation to fit in interstices among mechanical parts. Users demand to increase the endurance also prompts a system designer to optimise the dimensions of embedded systems. The extra fuel cost in transportation system and space ventures is another factor that imposes size and weight constraint on embedded systems. Similar to other technology markets, embedded systems in the consumer electronics domain are sold in a very competitive market. The cost sensitivity is usually attached with the performance, precision and the quantity of items produced. For example, a management is less sensitive to a cost issue of a high end embedded system produced in a small quantity when compared to a system produced in an order of millions. Time-to-market is another important constraint that system designers has to cope with. The designers need to deliver systems on time to gain a maximum advantage out of their product and have to adopt very quickly according to new technology trends. One of the recent example is Nokia in the market of mobile systems. Nokia [Cor] has a dominating market share in the mobile phone industry in the last decade. Samsung [Gro] brought its smart-phones very quickly in the market and acquired a large share in the mobile industry.

The primary requirement of an embedded system is to correctly perform a desired functionality. There is a class of embedded systems that has an additional constraint of temporal requirement to be met on top of the functional correctness for the overall system to be considered correct. This class of embedded systems is named as real-time (RT) systems in the literature. Consider an example of an anti-lock breaking system (ABS) in cars. The RT or temporal constraint in this system requires to release breaks for a very short period of time before reaching the skidding point that may cause the car to get out of a driver's control. The timing is an important property of the system as a minor delay can cause a system failure. Stephan J. Young [You82] formally defined a RT system as follows.

Definition 2. *"Any information processing activity or system which has to respond to externally generated input stimuli within a finite and specified period." — Stephan J. Young[You82]*

Similarly, Oxford dictionary of computing [Wri] gives the following comprehensive definition of a RT system.

Definition 3. *"Any system in which the time at which output is produced is significant. This is usually because the input corresponds to some movement in the physical world, and the output has to relate to that some movement. The lag from input time to output time must be sufficiently small for acceptable timeliness." — Oxford Dictionary of Computing [Wri]*

These definitions cover a wide range of RT systems but fortunately, all these different RT systems can be classified into two main categories depending on the nature of timing requirement [BW09]. These two different categories of RT systems are given as follows.

- **Hard Real-Time Systems:** Hard real-time systems (HRT) are the class of embedded systems in which a desired operation violating the temporal constraint, i.e., completing after the predefined time interval, may cause catastrophic or irreversible consequences. In other words, it is imperative to meet the timing requirements regardless of a system's state. The constraint on the timing is commonly known as a deadline. These catastrophic or irreversible consequences may lead to a damage to the physical surrounding or threaten human life. The results obtained after a given time interval (or deadline) are considered useless in HRT systems. A simple example is an operation of an air-bag in our modern cars. The air-bag control unit (ACU) must inflate the fabric bag within 60-80 milliseconds after the first moment of a car's contact with the opposing object in case of an accident. ACU failing to meet this specification may even increase the risk of injury to the persons inside the car. Another example of a HRT system is an automatically controlled train. The train cannot stop immediately. In order to stop the train at some desired point say x, it must activate the break command a certain distance away from x. The controller of the train considers the safe deceleration rate and the speed of the train to compute the distance before x to apply breaks. Any delay in computation and/or activating the break command my cause disastrous consequences. Similarly, other examples of HRT systems are artificial heart pacemaker that regulate the beating of a heart patient, industrial process controllers, ABS, engine control system etc.

  This thesis focuses on HRT systems.

- **Soft Real-Time Systems:** Opposed to HRT systems, soft real-time systems (SRT) can tolerate occasional temporal violations, but the significance of the results degrades with the passage of time after their deadline. In literature, the usefulness of the results is sometimes referred to as tardiness. A desired function completing before or at its deadline has tardiness equal to zero. An operation failing to meet its deadline has a tardiness equal to the

difference between the completion time of an operation and its deadline. It is desirable to meet all deadlines and minimise tardiness if not all deadlines can be met. However, it does not cause dire consequences due to any misbehaviour in the timing constraint. For example, a degradation in the quality of electronics games is annoying but not life threatening. Similarly, a delay in the online transaction system will not cause the whole system to crash but can be extremely expensive. The degradation in the usefulness of the results can be demonstrated with the stock price quotation system [Liu00]. It is desirable to update the price of each stock as soon as its price changes. The delay in the price change reduces the usefulness of the results with time. Additionally, SRT systems in which the results are no more valuable after the deadline miss but such a situation does not have any catastrophic consequences (as in HRT systems deadline miss) are said to have firm deadlines. A delay in the video conferencing application causes a drop of frames after their deadline miss and people experience some glitches. Similarly, the quality of the voice in phone calls is another example. The validation of a SRT system is not as rigorous as it is performed in a HRT system and it allows system designers to focus on other performance metrics as well.

Many embedded devices are nomadic and have limited energy supply. Such energy constraints are induced by e.g., battery powered mobile devices or those with limited or intermittent power supply such as solar cells. Apart from limited power supply, some embedded systems also have thermal issues. Satellites are the prominent example of such systems. Reasons to reduce the energy consumption of an embedded system include the following.

1. The high requirement of the energy can lead to an increase in the size of an embedded system which is not desirable in many cases such as consumer electronics, avionics, automotive industry and military equipments.

2. A longer lasting battery is a market differentiator. Consumer always opts for a system that offers extra battery life with same functionality to avoid the hassle of recharging and increase its mobility. A system optimised for energy consumption is especially useful in scenarios where frequent battery replacements are very costly such as sensor networks deployed in remote areas.

3. High energy requirement causes thermal issues which in turn increase the packaging cost of an embedded system and/or demands efficient cooling systems. Thermal issues also affect the speed, power and reliability of the semiconductor chips [WA11].

4. Energy savings have positive impact on the environment. Batteries used in embedded systems are usually made from harmful chemical such as cadmium, lead and mercury [BET04]. These chemical can effect the living beings as batteries are usually dumped in fields. The lack of recycling and disposal sites is currently a major issue.

1.2 Basic Components of Real-Time Systems

A RT system may be viewed as three main components called applications, real-time operating system (RTOS) and hardware platform. The interaction between these components is demonstrated in Figure 1.1. Applications correspond to the dedicated functionality that a RT system is desired to perform on a given hardware platform. A real-time operating system sits in between a hardware platform and a given applications to provide hardware abstraction, perform scheduling and facilitate communication. It provides application programming interfaces (API) to allow the interaction of the application with the given hardware platform and the given application can access the different components of the hardware platform through available API's. Please note that a small scale RT system may not have an RTOS. The source code of the application is compiled and stored in a read only memory (ROM) to access the hardware platform. For example, a simple RT system that monitors the temperature of a room does not require a complex RTOS. Nevertheless, an RTOS is assumed to be a part of a RT system in the context of this dissertation. The hardware platform provides the physical layer that executes the given application. These basic components involved in the design of RT systems are discussed here providing us a base to explore the main topic of this dissertation, i.e., energy and thermal management.

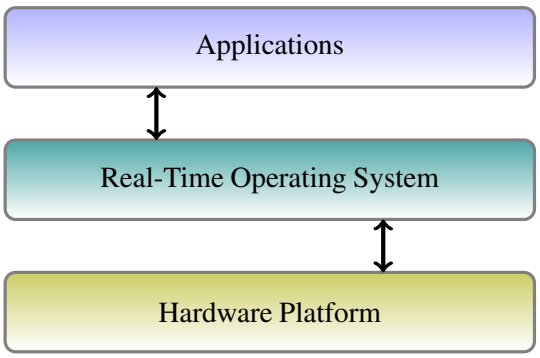


Figure 1.1: Different components of a RT system

1.2.1 Applications

Real-Time applications are usually represented by an abstract workload model that specifies the relevant characteristics of the workload generated by such applications when analysing a system. The functionality of a RT application can be modelled as a finite collections of simple, highly repetitive or abstract entities called real-time tasks [BG03]. These tasks are recurrent in nature. Each instance of a task is a basic unit of work that executes on the physical hardware platform and is called a RT job or in short a job [Liu00]. All jobs related to a particular task are semantically related. From now onwards, the functionality of a RT application is represented as a set of tasks called task-set. A frequency with which a task releases its jobs can be categorised into three types [IF00].

- **Periodic Tasks:** A task that releases its jobs periodically after a fixed time interval is defined as a periodic task. The fixed duration between the two consecutive jobs releases is called a period of a task.
- **Sporadic Tasks:** A task that releases its jobs at some arbitrary time instant but the two consecutive jobs of a task are always separated by at least a predefined time interval called minimum inter-arrival time.
- **Aperiodic Tasks:** Jobs of an aperiodic task is not constrained by a minimum inter-arrival time or a period, it can release jobs at any instant.

Within this work the focus is on sporadic tasks.

RT tasks are always constrained with a timing requirement. A task should complete its execution within a predefined time interval called the relative deadline of a task. A task failing to generate desired results within its relative deadline can jeopardise the whole system, environment or user's safety. A relative deadline of a task depends on the nature of an application. For example, the air-bag application installed in a car has a relative deadline of 60-80 milliseconds, while a room temperature monitoring application can have a relative deadline of a few seconds. A relative deadline of a periodic or a sporadic task can be categorised into three main classes.

- **Implicit Deadline Task:** An implicit deadline task has a relative deadline equal to its period or minimum inter-arrival time.
- **Constrained Deadline Task:** A constrained deadline task may have a relative deadline less than or equal to its period or minimum inter-arrival time.
- **Arbitrary Deadline Task:** As the name implies, an arbitrary deadline task has no relation with the period or minimum inter-arrival time of a task. It means that multiple jobs of the same task may be released with a difference of minimum inter-arrival time and coexist in the ready queue.

This work focuses on constrained deadline tasks.

The execution time of a task is another parameter that must be specified to characterise its temporal behaviour. Different jobs of a task exhibit variation in their execution time depending on the hardware characteristics, structure of the software, input data and different behaviour of the environment with which such job is interacting. In order to guarantee the temporal correctness, the upper bound on the execution time of a task is specified called worst-case execution time (WCET). The WCET of a task is the safe upper bound beyond or equal to the longest execution of any job released by such task. However, there is an assumption that execution times of the jobs are measured without any interruption. Any miscalculation in this parameter may cause a system failure. The term WCET is introduced formally in Definition 4. There are numerous methods and

techniques to compute the WCET of a task and the interested reader is directed to the following surveys of such techniques for further reference [PB00, WEE+08]. RT system designers consider the WCET of tasks while designing a system to guarantee the timing properties, however, different jobs of a task may execute for less than their WCET leaving behind unused computing resource. This bound must be pessimistic to be safe.

Definition 4 (WCET)**.** *Assume processor is in any legal state at the beginning of an execution of a task then the worst-case execution time of a task on a given hardware platform is the maximum length of its execution time, under worst-case input conditions without considering interference from other tasks.*

The nature of the application sometimes demands precedence constraints and data dependencies among tasks. For example, the inflate task in the air-bag system is dependent on the data from the sensor that provides information about the intensity of an impact in case of an accident. Similarly, the *authentication task* is performed before the *access tasks* in most of the banking systems. The type of tasks that needs to perform their execution in some order are said to have a precedence constraint. The tasks that can perform their execution without any order are called independent tasks. Such a task does not depended on the outcome of any other task or tasks to initiate their execution. For example, toast a slice of bread with the given temperature. Similarly, displaying the sensor reading of different parameters in the system on the monitor. The collection and display of data from a specific sensor can be performed independent of each other. Please note that the term task and job are used interchangeably in this dissertation. An execution of a task implicitly corresponds to the execution of its job.

This work focuses on independent tasks.

1.2.2 Real-Time Operating System

A real-time operating system is tailored for RT applications and designed to provide predictability and reliability in the system. The term predictability means the ability of the system to guarantee the timing properties at design time. The term reliability means “the ability of a system or component to perform its required functions under stated conditions for a specified period of time”[Dec98]. One of the main objectives of an RTOS is to provide an interface between RT tasks and resources available on the hardware platform. Furthermore, it provides an abstraction of the underlying hardware platform, and facilitates scheduling and communication. As the resources are usually limited in such platforms, therefore, it also coordinates and arbitrates their allocation among different tasks. Examples of an RTOS include VxWorks, RTEMS and PikeOS. The Mars reconnaissance orbiter and curiosity rover sent to the Mars used VxWorks as the operating system. RTEMS is commonly used in space applications, while PikeOS targets safety and security critical embedded systems. Similar to any other general purpose operating system, API’s of an RTOS relieves the programmer of a RT application to worry about the hardware details. These API’s

are optimised for different types of hardware platforms. Typically an RTOS performs many activities such as task management (scheduling), interrupt handling, memory management, inter-task communication and resource sharing. Nevertheless, the discussion in this section is limited to task management or scheduling.

A scheduler is a mechanism by which the RTOS allocates resources (such as processor) to tasks to perform their execution. It decides the time instant and the duration of execution for each task. Scheduling in RT systems has been widely studied in the literature. There exist numerous scheduling techniques for a vast variety of systems and task-models. Initially, scheduling techniques for a single processor were studied and later extended to the multiple processors case. Scheduling algorithms can be classified based on many factors. For example, scheduling algorithms can be divided into online and offline algorithms. In an online algorithm, the scheduling decisions are made based on the current state of a system, while in an offline scheduling algorithm, a precomputed schedule is determined offline. However, this section adopts the classification proposed by Jane Liu [Liu00]. She divides scheduling algorithms into following three main classes.

1) **Clock Driven Scheduling:** Clock driven scheduling approaches are also commonly known as time driven scheduling algorithms. In this category of algorithms, the scheduling decisions — which job executes at what time instant — are made at predefined time instances. Such decisions are made offline and stored in a memory to access online. The task parameters are usually fixed in this type of scheduling algorithms and a designer has complete knowledge available a-priori to derive a static schedule. Usually, the complete static schedule is divided into frames. The scheduling decisions are made at the boundaries of each frame. The size of a frame is selected consciously such that it minimises the scheduling overhead. The static schedule is repeated in a cyclic manner. A clock driven scheduling is a very simple approach. Its online complexity is very low as the schedule is precomputed. In this approach, scheduling tables can be easily replaced in different operating modes. The context switching overhead can be reduced by optimising the frame size. Many traditional RT systems are scheduled through this technique such as a traditional flight control systems or health care systems. These schedules are easy to validate, test and certify. The disadvantage of such a system includes its fixed nature. Any alteration in the task-set needs a redesign of a static schedule. Hence, it is suited for a fixed small embedded controller that rarely requires any changes.

2) **Round Robin Scheduling:** Round robin scheduling algorithms are suitable for time shared applications. Jobs in this strategy are placed in a first-in-first-out (FIFO) queue. Each job on the head of FIFO queue gets a same share of time. A job not completing in this share is pre-empted and added at the end of the FIFO queue. The time sharing slowly progresses the execution of all jobs. This algorithm is sometimes called a processor-sharing algorithm. One of the variation of such algorithm is a weighted round robin scheduling algorithm. Each job is allocated a specific share in FIFO order. The complete round of such algorithm is a summation of such weights allocated to different jobs in the FIFO queue. Weighed round robin is commonly used for RT traffic in high-speed switched networks [Liu00].

3) **Priority Driven Scheduling:** In a priority driven scheduling algorithm, tasks or jobs are allocated a priority and scheduled accordingly. The priorities can be allocated based on different criterion such as earliest deadline first, least laxity first, arrival rate of a task, shortest execution time first, shortest deadline first etc. The priorities of jobs or tasks can be allocated statically at design time or dynamically at run time. Most of the research effort is dedicated in this category of scheduling algorithm in a RT context. The pioneer work of Liu and Layland [LL73] on dynamic priority scheduling algorithm called earliest deadline first (EDF) scheduling algorithm and fixed priority scheduling algorithm, and the work of Mok [Mok83a] on least laxity first (LLF) are some examples of this class of scheduling algorithm. The priority driven scheduling approach can be further divided into three main categories.

   (a) **Fixed Task Priority (FTP):** In a fixed task priority scheduling algorithm, priorities are assigned to tasks. All the instances of a task (i.e., all its jobs) inherit the same priority. The priority of a job remains static through out the execution time. There are various priority assignment algorithms such as rate-monotonic (RM) [LL73] and deadline-monotonic (DM) [LW82]. Usually, the priority is assigned based on certain property of a task. In case of the DM priority assignment algorithm, a task with the shortest deadline is assigned the highest priority. Similarly, in the RM priority assignment algorithm, a task with smallest period is assigned the highest priority.

   (b) **Fixed Job Priority (FJP):** In this category of priority scheduling algorithm, priorities are assigned to jobs rather than their tasks. It means that different jobs of the same task may execute on a processor with different priorities. The priority of the certain job remains the same between its release time and deadline. There are many scheduling algorithms that falls in this category such as optimal EDF algorithm [LL73], earliest deadline Deferrable Portion (EDDP) [KY08] and EDF with $C = D$ [BDWZ12]. The priority of a job in this class of algorithms is usually assigned based on the fixed property of a job. For example, in case of EDF, the absolute deadline of a job is the fixed property that does not change throughout its active time.

   (c) **Dynamic Job Priority (DJP):** This is the most general form of a priority driven scheduling scheme. The priority of a job may change at any instant during its execution. One of the examples in this category is the LLF scheduling algorithm [Mok83a]. The priority of a job in LLF depends on the job's laxity (its deadline minus its remaining execution time). A job with the minimum laxity is allocated the highest priority and vice versa. The priority of a job varies with its execution on a processor. Such systems are difficult to design and may suffer from high number of pre-emptions. Other examples of such algorithms include proportionate progress (PF) [BCPV93], local remaining execution TL-Plane (LRE-TL) [Fun10] and largest local remaining execution first (LLREF) [CRJ06].

This work focuses on a rate-based scheduling approach with EDF **and in particular considers fixed job priority schedulers at its core.**

Most scheduling algorithms that belong to the priority scheduling class are work conserving in nature. A work conserving scheduling algorithm is defined as follows.

Definition 5 (Work conserving scheduler)**.** *A work conserving scheduler always executes a job if available in the ready queue and consequently does not allow a processor to get idle in the presence of ready jobs.*

A scheduler grants access of a processor to a job to perform its execution. The execution time of jobs may be interleaved and the scheduler can suspend a low priority job to execute a high priority job. If the execution of a job is interrupted in the middle by another job, this phenomenon is called a pre-emption (see Definition 6 for formal definition).

Definition 6 (Pre-emption)**.** *A pre-emption occurs when the execution of a job on a processor is suspended in order to execute another higher priority job.*

Some schedulers allow pre-emptions and are called pre-emptive schedulers. On the contrary, a class of schedulers that allows a job to complete its execution once started without any interruption are known as non-preemptive scheduling algorithms [Bar06]. The majority of scheduling algorithms belongs to the class of pre-emptive schedulers. Each pre-emption has an overhead associated to it as the pre-empted job has to save its status to resume its execution later in time. There has been some research [JCR07, LHS+98] in which the overhead of such pre-emptions is considered in the scheduling analysis.

This work focuses on pre-emptive schedulers.

A class of scheduling algorithms designed for the hardware platform having more than one processing element (processors) are usually divided into three main categories, i) global schedulers, ii) partitioned schedulers and iii) semi-partitioned schedulers. Before going into the details of such classification, the concept of migration is defined as follows.

Definition 7 (Migration)**.** *A migration occurs when the execution of a job is suspended from one processor and later resumed on another processor.*

1. **Global Scheduling Algorithms:** In global scheduling algorithms, all tasks are maintained in a single global ready queue and n high priority tasks in the ready queue are allocated to the n available processors. Tasks are not statically allocated to individual processors. A task may start it execution on one processor, can be pre-empted by a high priority task and later may resume its execution on another processor, i.e., migrations are allowed. Global-EDF [DL78] is a well known example of such a scheduling algorithm. Other examples include the work of Andersson et al. [ABJ01], Srinivasan and Baruah [SB02], Goossens et al. [GFB03] and Baker's [Bak05].

2. **Partitioned Scheduling Algorithms:** In contrast to a global scheduling, in partitioned scheduling algorithms, a given task-set is initially distributed among the processors based

on some criterion such as best-fit, first-fit, worst-fit, next-fit etc. The initial assignment is performed at design time. Such an assignment is static and tasks are not allowed to migrate from one processor to another at run time. After the task assignment phase, any uniprocessor scheduling algorithm can be applied over an individual processor to schedule the tasks allocated to it. The most important phase of such scheduling algorithm is the task to processor mapping. The research of Dhall and Liu [DL78], Oh and Son [OS95], and Burchard et al. [BLOS95] are pioneer works in partitioned schedulers.

3. **Semi-Partitioned Scheduling Algorithms:** Semi-partitioned scheduling algorithms are a mix of global and partitioned scheduling algorithms. A subset of tasks are initially allocated to specific processors and are migration-less at run time, while the rest of the tasks are allowed to migrate from one processor to another processor. Examples of such algorithms include EDF with task splitting and k processors in a group (EKG) [AT06], EDDP [KY08], deadline monotonic with priority migration (DM-PM) [KY09] and partitioned deadline monotonic scheduling with highest priority task split (PDMS_HPTS) [LRL09].

1.2.3 Hardware Platform

A hardware platform provides the physical components to execute the desired functionality of the given RT application. In a RT system, a typical hardware platform is composed of three main components, i) processor(s), ii) memory and iii) input/output (I/O) devices. These components are interconnected through buses. The structure of the buses depends on the architecture of the platform. Intuitively, all these components have an impact on the performance and the behaviour of the system. Figure 1.2 presents a block diagram of MPC8544E PowerQUICC III Processor (figure taken from [Fre14]). It is a typical example of an embedded hardware platform that includes processing elements, I/O devices and memory units. This platform is commonly used in multimedia and communication applications. A hardware platform is an active topic of research in academia and industry. Only the essential components of a hardware platform are briefly discussed in this dissertation to develop the basic understanding of the topic required for the main contents.

1.2.3.1 Embedded Processors

The terms central processing unit (CPU), processor or core represent the processing elements of a hardware platform. Note that these terms (CPU, core or processor) are used interchangeably throughout this document. Many embedded processors are cheap and less complex when compared to their counterparts general purpose processors. According to Barr Group's embedded systems glossary [Bar14] out of 10 billions processors produced last years, 9.8 billions processors were used in embedded systems ranging from toys, factories, weapon systems, nuclear power plants etc. These embedded processors span from 4-bit micro-controllers to 128-bit high end processors. Over the years, hardware vendors have increased the performance of these embedded processors borrowing the concepts from general purpose processors. One of the side effect of performance increasing tweaks such as pipelining, onchip memory, instruction prefetching etc, is

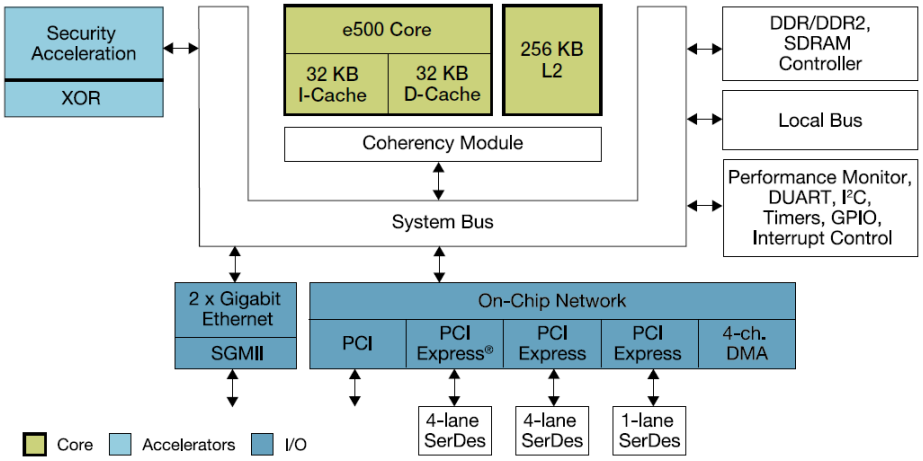


Figure 1.2: Block diagram of MPC8544E PowerQUICC III processor (source [Fre14])

the increase of unpredictability in the execution time of an application which needs to be analysed carefully in the RT context. The process of WCET estimation is challenging on these modern embedded processors resulting in pessimistic bounds. On top of this, there is a paradigm shift towards multicores in the design process of embedded processors. A multicore or multiprocessor hardware platform has more than one core or processor. These cores can resemble in properties or may be completely unrelated in design. Consequently, multicore platforms can be categorised into two main types based on the correlation between the available cores on a given platform.

1. **Homogeneous Multicores Platform:** Homogeneous multicore platforms are also commonly known as identical multicore platforms. All cores on identical multicore platforms have exactly the same properties in terms of computation and the cores are interchangeable. The execution time and the energy consumption of a task remains the same on all cores on such a platform. These multicore platforms are also sometimes called symmetric multiprocessor platforms (SMPs). Many multicore platforms manufactured and deployed today in embedded systems falls under this category. For example, Cortex-A17 [ARMb] from ARM (used in smart phones, tablets, smart TV's etc) has four identical cores on a same die.

2. **Heterogeneous Multicores Platform:** Heterogeneous multicore platforms can be further divided into two main classes.

   (a) **Uniform multicore platforms:** In a uniform multicore platform all the available cores have similar characteristics — same functional blocks, instruction set architecture etc — but the speed of the cores may differ from each other. The WCET and the energy consumption of a task may differ on different cores depending on the operating frequency of the cores. Such difference in frequency are either imposed intentional depending on the design requirement or can be caused due to the variation in the chip manufacturing process. big.LITTLE processing [ARMa] is the main example of such a platform used to optimise the energy consumption of the hardware platform. In

big.LITTLE processing, there are two processor types big and LITTLE processors. big processors provide high performance, while LITTLE processors have high energy efficiency. Both types of processors are architecturally compatible, i.e., they run the same instruction set. Depending on the online requirements, the workload can transparently switch its execution from big processors with high performance to energy efficient LITTLE processors and vice versa. Apart from its online adaptability, this architecture (big.LITTLE) also allows to run all the processors types simultaneously to fully utilise its computing potential.

(b) **Unrelated multicore platforms:** Processors or cores on an unrelated multicore platform have no relation among each other. They have the highest degree of heterogeneity. Usually, different cores have different instruction set architecture. The energy consumption and the WCET of a task vary substantially on these different cores. For example, a task-A may have a WCET of 2 and 5 on core-I and core-J respectively. It is equally possible that another task-B may have WCET of 10 and 1 on core-I and core-J respectively. Normally, unrelated multicore platforms are designed and tailored for the given application to execute its tasks efficiently. OMAP-5 from Texas Instruments [Tex], Tegra K1 from nvidia [nvi] and Aurix TC27xT from Infineon [Inf] are the common examples of such multicore platforms.

1.2.3.2 Memories

Memory is an essential components of an embedded system. Different types of memories exists in embedded systems, for example, on-chip cache/scratchpad memory and off-chip random access memories (RAM) or non-volatile read only memory (ROM). In order to reduce the latency in access time, the architecture of memories is an important design parameter. Most of the processors have an on-chip cache or scratchpad memory for fast access to data and instructions. In multicore platforms, these on-chip memories may be placed in a distributed or shared manner. In a distributed architecture all the processors have their private on-chip memory to store the instructions and their data, while in shared architecture, processors share on-chip memories among each other. In practice, a hierarchical memory architecture is common in multicore platforms. All processors have a layer of private memories followed by a layer of shared memories. One of the major issues in the memory architecture design is the coherency of the data in the private memories. This problem is out of scope of this dissertation and hence not discussed here.

1.2.3.3 I/O Devices

An embedded device usually communicates with the outside world and hence, require I/O devices. These I/O devices are used in different scenarios with different objectives. For example, in factory automation applications, different sensors provide the means to observe and manipulate the environment (temperature, pressure etc). Network devices (such as ethernet, WiFi, modem etc) allow to connect and provide communication among embedded devices. Human interface with

embedded devices is also performed via I/O devices (e.g., keypads, displays). I/O devices can trigger the mechanical components or completely electrical in nature. The number of I/O devices on modern embedded systems is increasing and results in a major portion of energy consumption. An operating system access the devices through device drivers. The operating frequency of an I/O device is usually very small when compared to a processor's frequency.

1.2.3.4 Integrated Circuits

Complementary metal-oxide-semiconductor (CMOS) is the most widely used device technology in fabricating integrated circuits such as microprocessor, memories and many other digital/analog devices. CMOS technology was developed by Frank Wanlass in 1963 but the first CMOS circuit was developed in 1968. The low power dissipation due to the low input currents is the major advantage of CMOS over the previously used technologies such as transistor-transistor logic (TTL). Another, advantage is its high noise immunity. It uses p-type and n-type metal-oxide-semiconductor field-effect transistors (MOSFETs) to implement logic gates, which in turn are used to develop digital integrated circuits (ICs). In CMOS logic gates, n-type MOSFETs (also called nMOS) are arranged in the pull-down network between the output and the low-voltage supply rail (commonly called ground). Similarly, the collection of p-type MOSFETs (also called pMOS) are arranged in the pull-up network between the high-voltage supply rail and the output. Connecting points of pull-up and pull-down networks provide the output that has an internal capacitance (capacitance is the ability of the component to store electric charge). The internal capacitance of the output is charged when the pathway between the high-voltage supply and the output (drain) in the pull-up network offers a low resistance. A circuit is said to be in the pull-up state. Similarly, it can be discharged by allowing a low resistance between the output and the low-voltage supply rail in the pull-down network. This state is called the pull-down state of a circuit. A p-type MOSFET has a low resistance between source and drain when a low gate voltage is applied and has a high resistance when a high gate voltage is applied. In case of a n-type MOSFET, a high gate voltage provides a low resistance between source and drain, and a low gate voltage offers a high resistance. A CMOS circuit has an important property of a duality in which the p-type MOSFET network (pull-up) is complementary to the n-type MOSFET network (pull-down) to enforce the activation of only one network (either pull-up or pull-down) at a time.

In order to demonstrate the aforementioned concepts, consider an example of a simple NOT logic gate (input-inverter). Other logic gates such as NOR, OR, AND, XOR, XNOR etc works on the similar principles. The diagram of a NOT logic gate is presented in Figure 1.3. When a low gate-level voltage is applied, a NOT gate transitions into a pull-up state and a pMOS transistor acts a low resistance between V_{dd} and drain, while a nMOS transistor behaves as a high resistance between V_{ss} and a drain. As a result, the internal capacitance of the output/drain is charged. A high gate-level voltage causes a pMOS transistor to act as a high resistance and a nMOS transistor to behave as a low resistance. The circuit in this situation is in pull-down state and it discharges the internal capacitance of the output/drain. Summarising its operation, a low gate-level voltage charges the internal capacitance of the output and a high gate-level voltage discharges it.

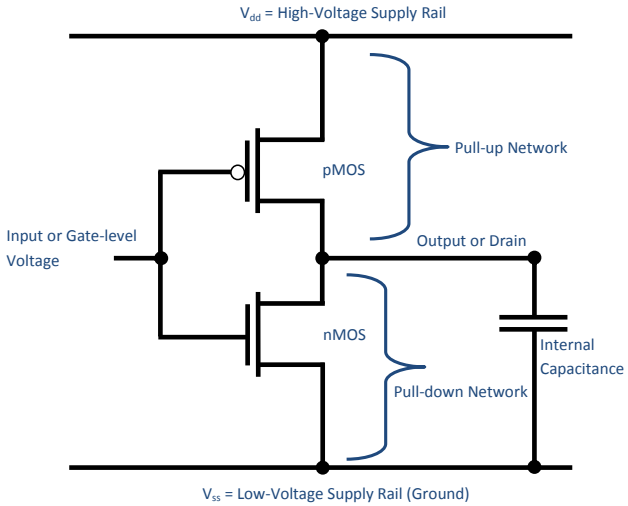


Figure 1.3: CMOS NOT logic gate (input-inverter)

1.2.3.5 Power Dissipation in CMOS Technology

In the early age of CMOS technology, the major focus of research was to increase its speed, reliability and cost [Che04]. The power dissipation was considered as a secondary issue. Hardware vendors over the years have followed Moore's law to integrate extra functionality on a single die. The need for extra computing capabilities, high speed, low cost and increased mobility has made the power dissipation a critical design metric. This section explores the basic sources of the power dissipation in CMOS technology that provides the basis to further explore this important design metric. The power dissipation in a digital CMOS circuit can be divided into three main types [Che04, RCN03]. The parametric equations and the contents in this section mostly summarises the work of Wai-Kai Chen[Che04] and Rabaey et al. [RCN03].

i) **Dynamic Power Dissipation:** The dynamic power dissipation (P_{dyn}) is the power component utilised to charge and discharge parasitic capacitance of all the nodes in CMOS circuits. This power is dissipated due to the switching current that flows when the circuit node switches from one logic state to another [Ins97]. For example, in case of a NOT logic gate presented in Figure 1.3, when a low gate-level voltage is applied, the pMOS transistor opens the path between V_{dd} and the output to charge the internal capacitance of the output load. Similarly, when a high gate-level logic is applied, a NOT gate discharges the stored energy at the output load through the nMOS transistor. The power dissipated in charging and discharging the output load in this process corresponds to the dynamic power dissipation. Assume C_L is the parasitic capacitance of the output load charged per cycle, f is the frequency of operation, α_s is the switching activity of the capacitive node C_L on each clock cycle, then the dynamic power dissipation for a single node (in this case input-inverter) can be defined as Equation 1.1 [Che04]. The same analysis can be easily extended to more than one node as presented in Equation 1.2 [Che04], where x is the number of nodes and $\alpha_{s,i}$ is the switching activity of a node i with a capacitance C_i. Assume, C_{eff} represents the average switching

capacitance per cycle, then the average dynamic power dissipation can be defined as given in Equation 1.3 [Che04, RCN03]. The dynamic power dissipation can be reduced through the techniques gate sizing, control synthesis, clock gating and voltage/frequency scaling (see [PSSG10] for the details of the aforementioned techniques).

$$P_{dyn} \stackrel{\text{def}}{=} C_L V_{dd}^2 f \alpha_s \tag{1.1}$$

$$P_{dyn}^{Total} \stackrel{\text{def}}{=} V_{dd}^2 f \sum_{i=1}^{x} (\alpha_{s,i} C_i) \tag{1.2}$$

$$P_{dyn}^{Avg} \stackrel{\text{def}}{=} V_{dd}^2 f C_{eff} \tag{1.3}$$

ii) **Short Circuit Power Dissipation:** The short circuit power dissipation is due to the current that flows from V_{dd} to ground, when the voltage level at the input of a logic gate is changing from one state (high/low voltage) to another state. This current is sometimes called the through current. In the analysis, it is assumed the transition time of pMOS and nMOS networks is zero and both networks are on simultaneously. In other words, it is assumed both networks acts as a high or a low resistance instantly when the gate-level voltage is changed from one state to another. In reality, both nMOS and pMOS devices are simultaneously conducting for a very short period of time in their transition phase causing the short circuit power dissipation. This kind of power dissipation depends on the switching rate and decreases with an increase in the switching rate. It is directly proportional to the rise time and the fall time of a gate [PSSG10]. The quantity of the through current is negligible when compared to the switching current that causes the dynamic power dissipation. Assume, I_{sc} is the through current that flows from V_{dd} and V_{ss} of a NOT logic gate, then its short power dissipation can be shown with Equation 1.4 [Che04].

$$P_{short} \stackrel{\text{def}}{=} I_{sc} V_{dd} \tag{1.4}$$

iii) **Static Power Dissipation:** The static power dissipation is caused by the leakage current that flows through a transistor even in the absence of a switching activity. This type of power dissipation is also terms as the leakage-power dissipation. The common sources of the leakage-power dissipation are given as follows [JNW10, RMMM03].

- Reversed biased pn junctions leakage current
- Subthreshold leakage current or weak inversion current
- Drain-induced barrier lowering (DIBL)
- Gate-induced drain leakage (GIDL)
- Channel punch-through leakage current
- Oxide leakage tunnelling
- Gate current due to hot carrier injection

The subthreshold leakage current (or weak inversion current), drain-induced barrier lowering and oxide leakage tunnelling are the major sources of the leakage-power dissipation [JNW10]. The interested reader is referred to the following books [Che04, RCN03, JNW10] for further in depth discussion on the different sources of leakage current. In the previous generation of CMOS technology, the leakage current is not considered as a considerable portion of the power dissipation. Technology scaling has increased the leakage-power dissipation to an extent that it has become a considerable portion of the overall power dissipation. Assume, I_{lkg} is the summation of the leakage current in a NOT gate, then its static power dissipation can be represented by Equation 1.5 [Che04]. There are various techniques proposed to reduce the leakage current that include multiple supply voltage, multiple threshold voltage, active body biasing, transistor stacking and power gating. The interested reader is referred to the work of Panda et al. [PSSG10] for further details of these techniques.

$$P_{lkg} \stackrel{\text{def}}{=} I_{lkg} V_{dd} \tag{1.5}$$

Combining all the components of the power dissipation of a NOT logic gate, Equation 1.7 gives its total power dissipation [Che04]. The same equation can easily be extended to compute the power dissipation of a complete CMOS circuit. All the parameters of Equation 1.7 play an important role in the design of low power digital CMOS circuits. These parameters are exploited in the power saving approaches at different abstraction levels.

$$P_{total} \stackrel{\text{def}}{=} P_{dyn} + P_{short} + P_{lkg} \tag{1.6}$$

$$P_{total} = V_{dd}\left(C_L V_{dd} f \alpha_s + I_{sc} + I_{lkg}\right) \tag{1.7}$$

1.2.3.6 Power Dissipation Vs Energy Consumption

The terms power dissipation, power consumption and energy consumption are often used interchangeably. Similarly, low-power and energy-efficiency are also perceived as a similar goals. The energy consumed by a circuit or a hardware platform is the amount of power dissipated for a certain period of time, i.e., $E \stackrel{\text{def}}{=} \int_0^t P(t)\,dt$. In other words, if the power dissipation is shown on y-axis and the computation time of a function is presented on x-axis, then the energy consumption is the area under the curve. The energy consumption to perform a specific function decreases, if the time to compute the function decreases and/or the power dissipation decreases. On the other hand, the power dissipation is the amount of energy consumed per unit of time. A decrease in the power dissipation not necessary means a decrease in energy consumption. For example, a decrease in the frequency or the voltage of a CMOS circuit reduces the power dissipation but at the same time also increases the computation time of the function as well. A power saving approach can either target to minimise the instantaneous power dissipation that impacts the power grid and the power supply design, or reduce the average power dissipation that increases the battery life and reduces the packaging cost of an embedded system.

1.3 Power Saving Techniques

After discussing the basic components involved in the design of RT systems, the power saving features available in embedded systems are discussed below. The power saving techniques in modern embedded systems are employed at different stages in the design process including application-level [LZ10, LSC08], system-level [SLSPH09, DA08a], architectural-level [STD94, MSV98], circuit-level [JKC10, VB08] and physical-level [YAY+07, JCS+10]. The system-level is the highest level of abstraction while the physical-level considers the processes involved in a transistor fabrication. The following list highlights several approaches to reduce the power dissipation corresponding to the different stages in the design process [Che04] of an embedded system. Two comprehensive surveys of different approaches on each stage of a system design are given by Luca et al. [BDMM01] and Chen [Che04]. These approaches consider different factors of Equation 1.7 in their optimisation process.

1. Application-level
2. System-level
    - Dynamic power management (DPM)
    - Dynamic voltage and frequency scaling (DVFS)
    - Instruction-level optimisation
    - Hardware-software codesign
    - Memory design techniques
3. Architectural-level
    - Parallelism and pipelining exploitation
    - Block-disabling techniques and clock gating
    - Intercommunication and interconnect optimisation
4. Logic gate-level
    - Path equalisation (lower V_{dd}, resizing)
    - Glitch avoidance and local transformations (re-factoring, remapping, phase assignment and ping swapping)
5. Circuit-level
    - Library cell design
    - Transistor sizing
    - Circuit design style
6. Physical-level

In the context of this thesis, only two system-level power saving techniques including dynamic power management and dynamic voltage and frequency scaling are discussed here. The major portion of the work in this thesis considers the dynamic power management and partially addresses frequency scaling.

1.3.1 Dynamic Power Management

Dynamic power management techniques allow a system or some functional blocks of a system to transition into a low-power sleep state (or low-power sleep mode) when a system is idle (inactive state). A sleep state achieves a low-power state by disabling certain part of the system. Modern hardware platforms offer several sleep states of different type. Different sleep states vary from just disabling a small part of the chip to shutting down the voltage supply of a circuit. Each sleep state has an overhead associated to each transition, i.e., a system has to pay the time and the energy penalty while disabling and enabling the functional block again, e.g., saving and restoring state. These sleep states are categorised based on their associated overheads (time/energy). The variation in their overheads depends on the technique used to initiate a sleep state and the area of a chip disabled. There are different techniques to disable a hardware or parts of a hardware such as clock gating and power gating. These techniques help to considerably reduce the dynamic and the static power dissipation of a system. Clock gating and power gating techniques are discussed in details here to understand the basics of different sleep states.

A major portion of modern hardware platforms is composed of synchronous CMOS circuits. It is a type of digital circuits in which all parts of a circuit are updated simultaneously and synchronised by the clock signal. The clock gating is commonly used mechanism in synchronous CMOS circuits to reduce the dynamic power dissipation. The clock signal is an input to the majority of circuit blocks and it switches the block activity on each cycle. The clock gating disables the clock input to these blocks and stops their switching activity on each clock. The clock gating mechanism identifies the group of flip-flops (basic storage element in a sequential logic) sharing a common enable signal. The common enable signal allows the new input to be fetched into the flip-flops on a clock cycle. This enable signal and the clock are combined using AND-gate to generate a gated clock. The clock gating can save up to $5-10\%$ of the dynamic power in synchronous circuits [PSSG10]. The granularity of the clock gating is an important parameter for designers. At coarse-grain level, the clock gating is usually managed by system-level software through a sleep state and it disables the whole functional block(s). For example, in modern mobile devices many functional blocks (such as display, radios, memory, processor) can be systematically disabled through clock gating to fit the mode of operation. The time overhead associated to clock gating is very small and usually in an order of few clock cycles. Therefore, sleep states based on this technique are well suited for short idle intervals.

Power gating is another technique commonly used in sleep states to shut-down hardware components. It is an effective approach against static power dissipation. It reduces the leakage current by cutting the power supply to the functional blocks. This techniques is implemented by adding a pMOS transistor between V_{dd} and the logic block, and the nMOS transistor between ground and

the logic block. These newly added pMOS and nMOS transistors are called power gate transistors. The size of these transistors is a major design challenge. The transition overheads (time/energy) to shut-down the logic blocks and bring it back to an active state through power gating is relatively high when compared to the clock gating. The transition overheads also depend on the granularity of the power gating. A fine-grained approach (adding switching transistor to each logic cell) increases the area overhead of power gate transistors but at the same time can decreases the static power dissipation up to 10 times [PSSG10]. On the other hand, a coarse-grained approach implements the power gating in the power distribution network rather than in the standard cells. This approach is less sensitive to process variation and has low area overhead.

Apart from the aforementioned techniques to implement the sleep state, the area of the chip disabled to reach a sleep state also plays an important role in the overhead of a sleep state. For example, a sleep state can disable a CPU, cache and other parts of a processor by turning it off completely. This process at the system-level is usually managed by an operating system. It is the responsibility of an operating system to save the processor's context (if allowed by the particular sleep state) including the saving of the cache contents and processor registers etc. These contents are brought back on transition-out phase of the sleep state. Such sleep states are useful if initiated for a longer time duration to compensate for the extra energy consumed in saving and loading the processor's context. There are some deeper sleep states that not only cut the power supply from the logic but also reduce the voltage of the supply as well to further decrease the leakage current.

As an example, consider Freescale PowerQUICC III Integrated Communications Processor MPC8536 [Sem] which has four sleep states named as doze, nap, sleep and deep sleep. In the doze mode, the instruction execution on the core is suspended but the snooping on the level-1 data-cache is still supported and its coherency is maintained. The nap mode turns-off all the clocks internal to the core except its timer facilities clock and the level-1 cache is also flushed. In the sleep mode all the internal clocks to the core including the clock to the timer are turned off. The clock that allows to turn-on the core itself is kept active. The deep sleep mode is more aggressive in power saving. It turns off the core, level-1 cache and level-2 cache by removing the power supply. The low-power sleep states of a system are usually managed by operating system calls. Most of the power saving algorithms in a non-RT setting are time based. A system transitions into a sleep state after a certain inactivity period. However, such techniques cannot be used in RT context as it might risk the temporal constraint due to the transition overhead associated to each sleep state.

1.3.2 Voltage and Frequency Scaling

It is evident from Equation 1.1 that the dynamic power dissipation has a quadratic relation with the supply voltage. The reduction in a supply voltage can help to save a considerable portion of the dynamic power dissipation. The frequency of a system is directly proportional to its supply voltage. Hence, a decrease in a supply voltage also allows to reduce the frequency of a system as well. Combining these two factors, the dynamic power dissipation has a cubic relation with frequency and voltage together. Though a voltage and frequency scaling can reduce the overall dynamic power dissipation but as a side effect, the performance of a system is also effected with

their scaling. The degradation in the performance increases the execution time of an application. Therefore, a trade-off exists between voltage and frequency scaling, and performance of the system. In most of the systems, the voltage and frequency is reduced such that it meets the system's temporal constraints. There exists a lower bound on the voltage and frequency scaling, where the energy saving in the dynamic power dissipation is smaller than the energy consumed by the static power dissipation due to an increase in the execution time. One method to recover the degradation in the performance at scaled voltages is to scale down the threshold voltage of a transistor (the minimum voltage applied on the gate of a transistor that turns it on) [PSSG10]. However, this approach increases the leakage-power dissipation and decreases noise margins.

Embedded systems exploiting frequency and voltage scaling are mostly designed with predefined voltage and frequency levels. System designers have the choice to select statically the voltage and frequency level for the given application based on a design time analysis or switch among different voltage and frequency levels at run time. The former approach is called the static voltage and frequency scaling and it is effective against an application having less dynamic behaviour at run time. The later is know as the dynamic voltage and frequency scaling (DVFS) and is suitable for more dynamic applications. The overhead of the switching among different voltage and frequency levels also plays an important role in the selection of a scaling strategy. While, DVFS saves more energy when compared to a static frequency and voltage allocation, it has an extra overhead of dealing with execution and energy models online. A hardware may have the ability to apply the voltage and frequency scaling on its some parts. The approach of having multiple voltage islands is very common in multicore platforms. In such a multicore platform, a group of cores have the flexibility to scale their voltage and frequency level (either statically or dynamically at run time). An independent supply voltage is required for each power domain that adds an additional challenge in the design of such platforms. The advantages of the voltage and frequency scaling is twofold, it reduces the dynamic power dissipation and also decreases the temperature of a system. The latter has an exponential impact on the leakage-power dissipation.

This work mainly focuses on the dynamic power management and also considers partially static voltage and frequency scaling.

1.4 Current Trends in Embedded Systems and their Impact on Energy Consumption

1.4.1 Non-negligible leakage-power Dissipation

CMOS technology is widely used in current hardware platforms and replaced the previous IC technologies because of its low power dissipation. In the beginning of this technology, the major source of power dissipation was switching activity and the leakage current was negligible. Therefore, DVFS was the major focus of research and the static power dissipation received little attention. CMOS technology miniaturisation following Moore's law reduced the transistor size

on every new technology generation. The scaling of CMOS transistors not just allowed to reduce the transistor size but also other parameters such as the power supply of a transistor [Hu10]. Each technology node reduces the capacitance due to the smaller size of a transistor and shorter interconnects. The reduction of capacitance and power supply are an effective means to reduce the dynamic power dissipation. However, on the other end, technology scaling has increased the leakage power proportion substantially. The increase is exponential as the process moves to finer technologies [ITR11]. In the early age of CMOS technology, CMOS circuits were operated at a high supply voltage when compared to the threshold voltage of the transistor. As mentioned previously, the reduction in the supply voltage of a circuit reduces the dynamic power dissipation but at the same time degrades the performance. The degradation in performance enhances as the supply voltage reaches closer to the threshold voltage of the transistor. The effect of degradation on the performance with voltage scaling can be compensated by reducing the threshold voltage of a transistor [Che04]. Unfortunately, a reduction in the threshold voltage exponentially increases the subthreshold leakage current [WRD00] as a transistor cannot be properly switched off at a low threshold voltage. Leakage-power dissipation and its variability has been identified as a major concern in the International Technology RoadMap For Semiconductors 2010 Update under special topics [ITR10]. A need to reduce the leakage current motivated hardware vendors to put in extra effort to equip modern embedded processors with several sleep states allowing a trade-off between the transition overhead and power dissipation in a sleep state. Moreover, the transition overhead of these sleep states is also reduced by several orders of magnitude.

1.4.2 Increased Number of I/O Devices

RT systems interact with their environment through the use of I/O devices. The technology miniaturisation has allowed to integrate additional functionality on a single chip. The currently observed trend in the increased number of on chip I/O devices can be attributed to the integration of previously isolated functionalities on to a single chip. Consider for an example a smart phone which includes several I/O devices such as global positioning system (GPS), gyroscope, cameras, high definition displays, high definition multimedia interface (HDMI), universal serial bus (USB), router etc. Energy consumption of CPUs has decreased considerably in modern embedded systems, while on the other hand, I/O devices are more power hungry relative to CPUs and consume a large portion of the system's energy [CH10]. Therefore, energy consumption of I/O devices are of particular concern in mobile systems and provide opportunities to reduce the overall energy consumption of a system. Nowadays, I/O devices are often equipped with power saving states to minimise their energy consumption. Similar to CPUs, energy saving is achieved by turning-off certain parts of the device. For example, a hard-disk in an idle mode can be spun-down to reduce its energy consumption. A device can only operate in an active mode, and its transition into and out of a low-power sleep state incurs both time and energy overheads. For instance, a hard-disk can only read/write in an active mode and it requires extra energy/time to spin-up from its power saving state.

1.4.3 Rising Thermal Issues

The increase in the power density of modern processors is another trend which demands efficient thermal management solutions to keep the temperature within given limits avoiding physical damage and also to increase the reliability of a chip. As mentioned previously, the leakage-power dissipation also increases exponentially with an increase in the temperature of a chip. Thermal management can be done at design time through sophisticated packaging and heat dissipation techniques, and at run time through dynamic thermal management (DTM). The techniques applied at design time through packaging and active heat dissipation are very expensive [TSR+98]. It has been predicted in the International Technology Roadmap for Semiconductor (ITRS2005) that the packaging solutions will become challenging in the near future due to an increase in the peak power dissipation and the high power density in an emerging system-in-package solutions. This trend motivates to explore DTM techniques for the wide variety of systems. The energy minimisation under thermal power constraint adds extra challenges to resolve.

1.4.4 Towards Multicore

Another observation is that Moore's law is no longer sustained by increasing clock frequencies, but rather by an addition of extra cores in multiprocessors. This is driven for example, by the performance per watt ratio, as higher clock ratios demand also higher supply voltages. Multicores have several tightly coupled processing cores to enhance the performance and the computation capacity by allowing parallel processing. The increase of computing capability of the processors takes place at a dramatic pace and is leading to a change towards multi-functional and multi-criticality embedded system. Besides symmetric multicore processors, homogeneous and heterogeneous multicores gain in popularity. The move beyond symmetric multicores is driven by the aim to use cores geared to perform specific tasks well and cheap.

1.4.5 Mixed Criticality

The increase in computing power also leads to a progressive integration of functionality into a single device. For example, a current mobile phone combines applications of soft real-time character (e.g., base station communication) with such of best-effort character (e.g., SMS). Additionally the different system components and software modules are potentially provided by different third party suppliers. Consequently such mixed criticality systems require temporal and functional isolation not only to protect critical applications from less critical ones, but also as a means to identify the offending application in the case of a misbehaving system and avoiding fault propagation.

1.5 Thesis Statement

Energy consumption of RT systems can be efficiently reduced with low online complexity using a system-level power-saving feature called sleep states. This applies to a large variety of modern hardware platforms while allowing temporal isolation between RT and BE type applications.

1.6 Focus of this Dissertation

The objective of this dissertation is to explore power saving strategies at system-level that mainly target the leakage-power dissipation in modern embedded systems while satisfying temporal constraints of RT applications. Modern embedded systems use various processor types (single core, homogeneous multicore, heterogeneous multicore etc) and have besides many components (I/O devices, memories etc) contributing to the power dissipation. The leakage-power dissipation of modern hardware platforms is increasing with the technology miniaturisation as discussed in Section 1.4.1 and has become one of the major challenges in CMOS technology scaling. In the context of battery powered mobile devices — where battery life is of utmost importance — it has become very challenging to continue to prolong the battery life with the current trend of increasing the leakage current. To overcome this issues, many efforts have been undertaken, spanning from the physical design of a transistor to operating system level optimisations.

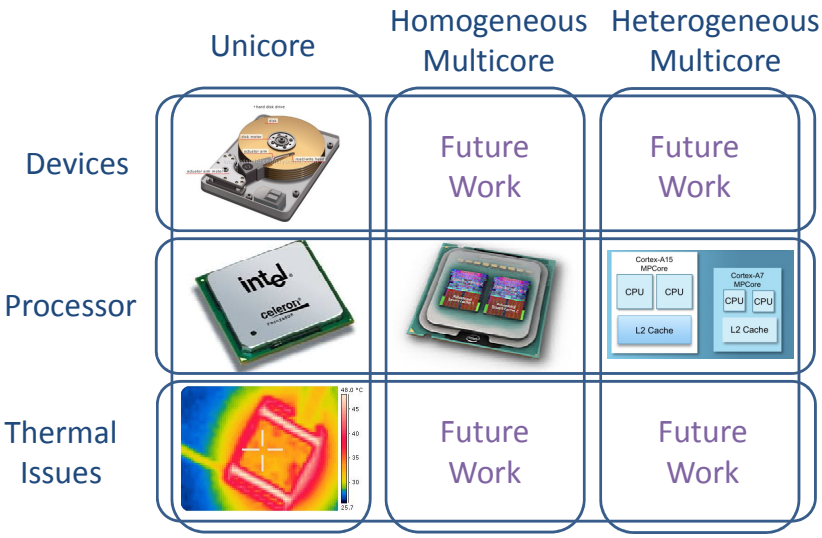


Figure 1.4: Highlighting the focus of this dissertation

This dissertation proposes system-level power saving strategies to prolong the battery life of embedded systems. In particular it considers the power dissipation of processors, I/O devices and also analyses the effect of temperature on the power dissipation. The proposed power saving approaches consider hardware platforms ranging from unicore to multicore architectures. Figure 1.4 highlights the different blocks of this domain considered in this work. From an RT perspective, this thesis considers an independent sporadic task-model scheduled with a variety of scheduling algorithms on different hardware platforms. Though there exist some approaches on leakage-aware power saving mechanisms in the literature, most consider simplistic assumptions that limit their practical relevance. One of the objectives of this dissertation is to relax the simplistic assumptions made in the state-of-the-art, and bridge the gap between theoretical research carried out in the domain of energy management and practice.

1.7 Thesis Organisation

This thesis is organised into eight chapters. After introduction, the related work and the model of computation are presented in Chapter 2 and Chapter 3 respectively. The power saving approaches for unicore platform, I/O devices, homogeneous platform and heterogeneous platform are discussed in Chapter 4, Chapter 5, Chapter 6 and Chapter 7 respectively. Chapter 8 concludes the work performed in this thesis and enlists future directions. The contents of these chapters are briefly outlined below.

- **Chapter 2:** The state-of-the-art presented in Chapter 2, elaborates on the shortcoming of the existing approaches. The literature survey is categorised into two main sections, i) unicore power management and ii) multicore power management. The former section addresses power management on a single processor platform, I/O device power management, thermally constrained energy management techniques, while the latter discusses homogeneous and heterogeneous multicore power management techniques.

- **Chapter 3:** The model of computation in this chapter is divided into two sections. The first section addresses the system model and the common terminologies used throughout the thesis. In particular, it considers the application model, temporal isolation, hardware model, slack sources and slack reclamation algorithms used in power saving algorithms. The latter section summarises the simulation framework used to evaluate different proposed techniques in later chapters.

- **Chapter 4:** This chapter discusses the optimality of the procrastination interval and proposes the optimal leakage-aware procrastination algorithm. The limitation of the external hardware of the existing leakage-aware procrastination algorithm is relaxed through proposed race-to-halt (RTH) algorithms. Afterwards, the effect of power saving algorithm on the number of pre-emptions is presented. Finally, it is shown that the thermally constrained dynamic power management is equivalent to the DVFS problem. It means, DVFS algorithms can be easily transformed to solve the thermally constrained dynamic power management problem.

- **Chapter 5:** In the new paradigm of intra-task device scheduling introduced in this chapter, a device is requested on demand without violating the timing guarantees rather than keeping it on through-out the execution time of a task. The proposed intra-task device scheduling is initially presented for devices with a single sleep state. Different techniques to collate the slack are also proposed to enhance the efficiency of the proposed algorithm. Later on, the assumption of a single sleep state per device is relaxed and three heuristics are proposed providing trade-off between energy efficiency and algorithm complexity.

- **Chapter 6:** The leakage-aware power saving algorithm is presented for the global scheduling algorithms on homogeneous multicore platforms. This is the first effort to reduce the

static power dissipation in the context of global scheduling algorithms. The proposed algorithm exploits the spare capacity of the schedule and can donate it among cores to prolong sleep states of the cores already in sleep mode.

- **Chapter 7:** Task-to-core mapping in partitioned scheduling is a NP-hard problem on a heterogeneous multicore platform. This chapter presents different heuristics to perform task-to-core mapping such that it reduces the dynamic and the static power dissipation. The proposed algorithm is divided into two phases. The first phase reduces the dynamic power dissipation, while the second phase trades the increased dynamic power dissipation with the reduced leakage-power dissipation in sleep states. Initially, algorithms are presented for hardware platforms without DVFS capabilities. Finally, this assumption is relaxed and DVFS enabled hardware platforms are integrated into the proposed heuristics.

- **Chapter 8:** Finally, the work presented in this dissertation is concluded, the results are summarised, author's perspective is highlighted and future directions are identified to extend the presented work to more general models.

1.8 Published Research in the Context of this Dissertation

This section presents the research papers generated as result of the research performed in the context of this dissertation. A brief description of each paper is presented here for the quick reference.

1.8.1 Conference Publications

1. M. A. Awan and S. M. Petters, "Enhanced race-to-halt: A leakage-aware energy management approach for dynamic priority systems", in Proceedings of the 23^{rd} Euromicro Conference on Real-Time Systems (ECRTS), pp. 92-101, July 2011.

   Abstract: This paper presented a race-to-halt algorithm (an alternative to leakage-aware procrastination scheduling) to reduce the total power dissipation of a unicore platform. It relaxes the need of an external hardware required in procrastination scheduling and has low complexity when compared to the state-of-the-art approaches.

2. M. A. Awan and S. M. Petters, "Online intra-task device scheduling for hard real-time systems", in Proceedings of the 7^{th} International Symposium on Industrial Embedded Systems (SIES), pp. 48-56, June 2012.

   Abstract: In this paper, a new paradigm of intra-task device scheduling is explored, in which a device is requested on demand. The spare capacity of the schedule is exploited to prolong the sleep state of a device and compensate for its transitional delays. Simulation results show a considerable energy saving especially in a system where a device is used for a very short period of time.

3. M. A. Awan and S. M. Petters, "Energy-aware partitioning of tasks onto a heterogeneous multi-core platform", in Proceedings of the 19^{th} IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), pp. 205-214, April 2013.

   Abstract: Task-to-core mapping on a heterogeneous multicore platform is a NP-hard problem. This paper presents different allocation heuristics to reduce the dynamic and the static power dissipation. This algorithm divides the allocation process into two phases. In the first phase, allocation is performed such that it reduces the dynamic power dissipation of a system. The second phase corrects the allocations performed in the first phase to use efficient sleep in each core, which in turn helps to reduce the static power dissipation of a system.

4. M. A. Awan and S. M. Petters, "On the Equivalence of Idealised DVFS and Thermally Constrained DPM in Real-Time Systems", in Proceedings of the 19^{th} IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA), pp. 346-351, August 2013.

   Abstract: In this paper, it is argued that from RT systems perspective, thermally constrained dynamic power management approaches behave very similar to idealised DVFS. Hence, existing DVFS solutions proposed for RT systems in the literature for periodic and sporadic task models can be applied to thermally constrained dynamic power management systems with moderate effort. This work presents the similarities along with the distinctive elements between two approaches and demonstrate the equivalence with the help of a case study.

5. M. A. Awan, P. M. Yomsi and S. M. Petters, "Optimal procrastination interval for constrained deadline sporadic tasks upon uniprocessors", in Proceedings of the 21^{st} International Conference on Real-Time Networks and Systems (RTNS), pp. 129-138, October 2013.

   Abstract: To deal with the leakage current, several procrastination approaches have been proposed in the past in order to reduce the energy consumption. These approaches approximate the procrastination interval for the ease of analysis and sub-optimally utilise the potential to reduce the energy consumption. This paper presents an optimal method to determine the procrastination interval of each task and generalise the task-model to cover the constrained deadline tasks. Analytical and experimental results show the superiority of the proposed technique.

6. B. Nikolic, M. A. Awan, and S. M. Petters, "SPARTS: Simulator for power aware and real-time systems", in Proceedings of the 8^{th} IEEE International Conference on Embedded Software and Systems (TrustCom), pp. 999-1004, November 2011.

   Abstract: Over the years, we are witnessing an ever increasing demand for functionality enhancements in RT systems. Along with the functionalities, the design itself grows more complex. Posed constraints, such as energy consumption, time, and space bounds, also require attention and proper handling. Additionally, efficient scheduling algorithms, as proven through analyses and simulations, often impose requirements that have significant run-time cost, specially in the context of multi-core systems. In order to further investigate

the behaviour of such systems to quantify and compare these overheads involved, SPARTS, a simulator of a generic RT system, has been developed. While the current implementation is primarily focused on our immediate needs in the area of power-aware scheduling, it is designed to be extensible to accommodate different task properties, scheduling algorithms and/or hardware models for the application in wide variety of simulations. The source code of SPARTS is available for download at [NAP11a].

7. D. Dasari, B. Akesson, V. Nelis, M. A. Awan and S. M. Petters, "Identifying the sources of unpredictability in COTS-based multicore systems", in Proceedings of the 19^{th} IEEE International Symposium on Industrial Embedded Systems (SIES), pp. 19-21, June 2013.

   Abstract: The underlying architecture of commercially available multicores is extremely complex and non-amenable to straight-forward timing analysis. In this paper, the architectural features are highlighted that lead to the temporal unpredictability, which mainly involve shared hardware resources, such as buses, caches, and memories. This paper discusses the existing work in timing analysis with respect to these features, identify their limitations, and present some un-addressed issues that must be dealt with to ensure safe deployment of RT systems.

1.8.2 Journals

1. M. A. Awan and S. M. Petters, "Intra-task device scheduling for real-time embedded systems", **(under submission)** in Journal of Systems Architecture, 2013.

   Abstract: This is an extension of a paper published in SIES 2011 titled "Online Intra-Task Device Scheduling for Hard Real-Time Systems". An Intra-Task Device Scheduling algorithm in original paper is complemented by an online device budget reclamation algorithm which recovers unused time allocations of devices in a system. Furthermore, an energy density function is developed to analyse the effect of the different sleep states of a device on the overall device energy consumption of a system. Using this energy density function, a single sleep state assumption is relaxed and three different algorithms for a generic power model, in which each device assumes more than one sleep states are proposed. The proposed algorithms are scalable with increasing I/O devices and have less complexity when compare to the state-of-the-art algorithms.

2. M. A. Awan and S. M. Petters, "Real-time race-to-halt energy saving strategies and their impact on the number of pre-emptions", **(under submission)** in Journal of Systems Architecture, 2014.

   Abstract: This work is an extended version of a paper published in ECRTS 2011 titled "Enhanced race-to-halt: A leakage-aware energy management approach for dynamic priority systems". It decreases the pessimism of the enhanced race-to-halt algorithm (ERTH) with an improved race-to-halt algorithm (IRTH) at the cost of extra complexity to predict

the future release information. Furthermore, a complexity-wise light-weight race-to-halt algorithm (LWRTH) is also proposed. The relation of sleep states with the pre-emption count is also studied that shows on average sleep states have positive impact on the number of pre-emptions.

3. M. A. Awan, G. Nelisson, P. M. Yomsi and S. M. Petters, "Energy-aware Task Mapping onto Heterogeneous Platforms Using DVFS and Sleep States", **(under submission)** in Journal of Real–Time Systems, 2014.

   Abstract: One of the challenges in heterogeneous multicore platforms is to optimise the energy consumption in the presence of temporal constraints. This paper addresses the problem of task-to-core allocation onto a heterogeneous multicore platform such that the overall energy consumption of a system is minimised. This article is an extension of the paper published in RTAS 2013 titled "Energy-aware partitioning of tasks onto a heterogeneous multi-core platform". The extension includes a task-to-mapping algorithms for DVFS enabled heterogeneous multicore platforms. Similar to the original publication, the approach for this general platform is also divided into two phases. In the first phase, tasks are allocated such that the dynamic energy dissipation is reduced. The second phase refines the allocation performed in the first phase to improve on the possible sleep states by trading off the dynamic power dissipation with reduction in the leakage-power dissipation. This hybrid approach considers core frequency set-points, tasks energy consumption and sleep states of the cores when performing allocation to reduce the energy consumption. Major value has been placed on a realistic power model which increases the practical relevance of the proposed approach.

4. M. A. Awan, G. Nelisson, P. M. Yomsi and S. M. Petters, "Online Slack Consolidation in global-EDF for Energy Consumption Minimisation", **(available as a technical report and under submission)** in Journal of Systems Architecture, 2014.

Abstract: With the current body of knowledge, an efficient selection of sleep states is a non-trivial problem for system designers assuming a global scheduling algorithm. In this work, a leakage-aware energy management algorithm is proposed for homogeneous multicore platforms using a global-EDF scheduler. Global-EDF is one of the most prominent scheduling policy upon homogeneous multicore platforms. The proposed algorithm: (i) exploits the spare capacity available in the schedule on each core to either initiate a sleep state on this core or prolong the sleep state of the cores already in a sleep state; and (ii) has a low complexity, thus making it practically feasible.

1.8.3 Workshops, Posters and Work-in-Progress

1. S. M. Petters and M. A. Awan, "Slow down or race to halt: Towards managing complexity of real-time energy management decisions", in Proceedings of the 12^{th} Brazilian Workshop on Real-Time and Embedded Systems, (Gramado/RS, Brazil), May 2010. Work-in-Progress Session.

2. M. A. Awan and S. M. Petters, "The roman conquered by delay: Reducing the number of pre-emptions using sleep states", in Proceedings of the Work-in-Progress session of the 17^{th} IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), (Chicago, IL, USA), April 2011.

3. M. A. Awan, B. Nikolic, and S. M. Petters, "Comparing the schedulers and power saving strategies with SPARTS", in the RTSS@Work, Open Demo Session of Real-Time Techniques and Technologies, Proceedings of the 32^{nd} IEEE Real-Time Systems Symposium (RTSS), (Vienna, Austria), November 2011.

4. M. A. Awan and S. M. Petters, "Device power management for real-time embedded systems", in the Proceedings 1^{st} PhD. Students Conference in Electrical and Computer Engineering (StudECE), (Porto, Portugal), June 2012.

Chapter 2

State of the art

There exists an extensive amount of work on power management that considers different aspects of embedded systems. The research effort in this domain ranges from the transistor level design [VZG+10] to the application level optimisation [LSC05]. The subject of this research is system level power management approaches in the context of RT embedded systems. The power management in embedded systems has been exhaustively explored at system level through well known tools of DVFS and sleep states. The RT community has explored these two major types of power saving features of the modern embedded systems and developed interesting results. Dynamic power dissipation was the main source of energy consumption in traditional hardware platforms. Therefore, DVFS was the major focus of research in the beginning of last decade and a large amount of work exists in the literature. Chen and Kuo [CK07a] presented a comprehensive survey of energy management techniques on DVFS enabled hardware platforms. In this chapter, the main focus of the literature review will emphasise on the power saving strategies based on sleep states to tackle the leakage-power dissipation in the context of RT systems. This chapter summarises the work consistent with the current industry trends and highlights the unexplored issues. The state-of-the-art in the power management domain can be categorised into two main categories of unicore and multicore systems.

2.1 Unicore Power Management

The unicore power management can be further divided into three parts, CPU power management, I/O device power management and temperature aware energy minimisation.

2.1.1 CPU Power management

To deal with an increase in leakage-power dissipation, Lee et al. [LRK03] addressed leakage-aware scheduling for periodic hard real-time systems. They proposed leakage control EDF (LC-EDF) and Leakage Control Dual Priority (LC-DP) algorithms for dynamic and static priority schemes respectively. The LC-EDF algorithm is an online algorithm that maximises the idle interval by delaying the busy period to increase the duration of the sleep state. Such mechanism is commonly

called procrastination scheduling. They assumed an external specialised hardware such as application specific integrated circuit (ASIC) or field programmable gate array (FPGA) to implement their algorithm. Baptiste [Bap06] did a theoretical study of a non-DVFS system with unit sized RT aperiodic tasks. He developed a polynomial time algorithm to minimise the energy consumption of static power and the sleep transition overhead of the system.

A combination of leakage-aware and dynamic voltage scheduling appears to be a promising way to reduce overall energy consumption. Irani et al. [ISG07] proposed a 3-competitive offline and constant competitive ratio online algorithms for power saving while considering shut-down in combination with DVFS. The competitive analysis is used to measure the performance of the proposed algorithm when compared to the clairvoyant optimal offline algorithm. The algorithm is termed as competitive if its competitive ratio, i.e., ratio between the performance of the algorithm and the optimal offline algorithm, is bounded by a constant number. It means their 3-competitive offline algorithm consumes energy within three times the energy consumption of optimal algorithm. Similarly, the energy consumption of their online algorithm is bounded by a constant competitive ratio. Although the combination of shut-down and DVFS has its merits fundamentally, their approach requires further work to relax the assumptions in terms of the used DVFS power model. Besides requiring external hardware to implement their shut-down algorithm, they assume a continuous spectrum of available frequencies and an inverse linear relation of frequency with execution time. Niu and Quan's [NQ04] scheduling technique also addressed the dynamic and leakage-power dissipation simultaneously on a DVFS enabled processor for hard-real time systems. They integrated DVFS and shut-down to minimise the overall energy consumption based on the latest arrival time of jobs, which is estimated by expanding the schedule to the hyper-period (least common multiple of the tasks minimum inter-arrival time). However, this algorithm cannot be used online due to the extensive analysis overhead. Previously, Jejurikar et al. [JPG04] integrated DVFS with the procrastination algorithm, to minimised the total power dissipation. The presented critical speed (a lower bound on frequency scaling and its formal definition is presented in Definition 11) determines the lower bound on the processor frequency to minimise the energy consumption per cycle. Moreover, they showed the procrastination interval determined by their algorithm is always greater than or equal to the procrastination interval estimated by LC-EDF. Nevertheless, they did not relax on the requirement of additional hardware to support their shut-down approach.

Soon after, Jejurikar et al. [JG04] showed that procrastination under LC-DP originally proposed by Lee et al. [LRK03] may cause some of the tasks to miss their deadlines. They proposed improvements in the original algorithm and also integrated their DVFS approach. However, they adopted the same assumptions of the previous work [LRK03, JPG04]. Later on, Chen and Kuo [CK06] showed that the procrastination approach proposed by Jejurikar et al. [JG04] still might lead to some tasks missing their deadlines. They proposed a two phase algorithm that estimates the execution speed and procrastination interval offline, and predicts turn off/on instances online but also rely on extra hardware. Further work of Jejurikar and Gupta [JG05] reclaims the execution slack generated due to the difference between WCET and actual execution

time (see Definition 13). They used procrastination scheduling and DVFS to minimise the overall energy consumption, and called their approach slack reclamation algorithm (SRA). The dynamically reclaimed slack is either used entirely for slowdown or distributed between slowdown and procrastination using slack distribution policy. This algorithm follows the same assumptions made by previous works [LRK03, JPG04, JG04].

Chen and Kuo [CK07b] developed a novel algorithm distinct to greedy procrastination algorithms [LRK03, ISG07, NQ04, JG05, JG04, CK06, JPG04] for procrastination interval determination. They showed that their algorithm can decrease the energy consumption by executing jobs at lower speeds than the previously mentioned critical speed, when the processor is decided not to be turned off in the procrastination interval. Chen and Thiele [CT08b] proposed leakage-aware DVFS scheduling, where tasks execute initially with decelerating frequencies to accumulate the slack (unused time) to initiate a sleep state. Towards the end the tasks execute with accelerating frequencies to reduce the dynamic power dissipation. However, the work of Chen et al. [CK07b, CT08b] still relies on continuous spectrum of available frequencies and external hardware. Considering previous history of events, predicting the future events using RT calculus [TCN00] and doing the scheduling analysis with RT interfaces [TWS06], Huang et al. [HSC+09, HSC+11] estimated the procrastination interval of a device to activate the shut-down.

Santinelli et al. [SMP+10] proposed energy-aware packet and task co-scheduling algorithm EAS for the distributed RT embedded system consisting of a set of wireless nodes. EAS generates the schedule till the next idle time in the schedule, and determines the frequency of the processor and the sleep interval such that the total energy consumption is minimised while efficiently utilising the reserved communication bandwidth. The online complexity of this algorithm is high as system has to compute the demand bound function [RGR08] online to determine the frequency and the switch off time. Wang et al. [WLL+11] determined the static schedule for the given set of dependent periodic tasks for homogeneous multiprocessors. In the first step they relax the dependencies of the tasks using coarse-grained task parallelisation algorithm RDAG. The second phase determines the static schedule using a genetic algorithm (gene evolution) to minimise the energy consumption assigning frequencies to the tasks and enforcing sleep intervals in the schedule. However, the work of Santinelli et al. [SMP+10] and Wang et al. [WLL+11] is proposed for different system models and hardware platforms compared to the one discussed in this dissertation.

One of the assumptions commonly made throughout the state-of-the-art is a requirement of the external specialised hardware to implement procrastination scheduling. A part of Chapter 4 addresses this issue and propose algorithms to optimise energy minimisation while relaxing these assumptions with a more general power model.

2.1.2 I/O Device Power Management

The demand for extra functionality has increased the number of I/O devices on modern platforms. These I/O devices consume a considerable amount of energy and provide a large potential to reduce the energy consumption of the platform. Hence, it has become an active research area in the embedded computing domain. Initially the device power management was extensively studied

in a non-RT setting. These techniques can be divided into three main categories, 1) time-out based, 2) predictive and 3) stochastic. Time-out based algorithms shut-down the devices when they are idle for the specified threshold. The device wake-up calls are made when it is requested again. Predictive techniques adapt themselves with the varying system's workload. Stochastic methods model the requests behaviour with different probabilistic distributions. The device shut-down times are estimated by solving the stochastic models such as Markov chains. For a detailed survey of device power management algorithms in a best-effort environment (non-RT systems), the reader is directed to the work of Benini et al. [BBDM00].

Swaminathan et al. [SCI01] explored the device scheduling in the context of RT systems. They proposed an offline method for dynamic I/O power management with hard RT constraints. Their low energy device scheduler (LEDES) is based on look-ahead information about the tasks future arrival-pattern to decide on the shut-down of devices. Later on, multi-state constrained low-energy scheduler (MUSCLES), an extension of LEDES for the multiple sleep state devices was proposed by Swaminathan and Chakrabarty [SC03]. MUSCLES generates the sequence of power states for every device given the precomputed task schedule with a per task device usage list. The LEDES and MUSCLES algorithms assume fixed offset strictly periodic tasks releases, which limits its applicability/extension to a sporadic task model and/or to a task model that allows variable task's execution time. The algorithms proposed in this thesis relax these assumptions.

The same authors also developed energy optimal device scheduler (EDS) [SC05]. EDS computes a schedule tree for all possible scheduled combination, and prune it based on the temporal and energy constraints. Due to high spatial requirement and temporal complexity of EDS, they provide a heuristic which clusters the requests of the same device to prolong the idle intervals. It is based on the work of Lu et al. [LBDM00] that was initially proposed for best-effort systems. Both heuristic and EDS are based on an inter-task scheduling mechanism. A device scheduling algorithm is called inter-task scheduling mechanism, if all the devices used by a task are kept active throughout its active time (i.e., between task's arrival and completion time). They are computationally expensive and are of limited utility for sporadic task models as they assume a-priori information of a task's release pattern.

A procrastination based I/O device scheduling algorithm is proposed by Cheng and Goddard [CG06]. The basic idea is to prolong the device's sleep interval by procrastination of the task's execution that requires this device. This method assumes inter-task device scheduling and has high online overhead. However, it can be applied to a sporadic task model with tasks having varying execution times. Later on, Devadas and Aydin [DA08b] proposed a device power management algorithm for static priority systems through device forbidden regions. The device forbidden regions enforces idle intervals in the schedule to prolong the sleep interval of devices. To preserve the schedulability, the bounds on the explicit idle intervals are computed using time bound analysis [LSD89]. Their algorithm is also based on inter-task device scheduling.

Chu et al. [CHT+09] proposed a composite low-power scheduling framework called COLORS, which is a Dynamic Voltage Scaling (DVS) assisted I/O device scheduling algorithm for periodic hard RT systems. They assume devices access intervals and their usage times are known

a-priori. The execution of the task is divided into computation and peripheral intervals. It uses both static and dynamic slack to extend the computation interval of a task by running it at low frequency to prolong the device shut-down time. A simplistic power-model and a-priori device usage information restrict is applicability to the majority of systems, where such information cannot be predicted a-priori. A similar slot-base algorithm was proposed by Kim and Ha [KH01]. The execution time of the task is split into CPU execution and peripheral usage time-slots. The frequency of CPU and the device shut-down period is adjusted such that the overall energy consumption is reduced. They assumed transition overhead of the device's sleep state is negligible. A genetic algorithm customised for the device power management is proposed by Tian and Arslan [TA03] for periodic RT systems. This algorithm assumes jobs execute for their WCET and try to find the near-optimal solution with the provided set of jobs and devices.

The low-power quasi-dynamic scheduling (LQS) proposed by Hsiung and Kao [HK05] determines the feasible schedule to reduce the device power dissipation. The system is modelled with power-aware real-time petri-nets (PARTPN). LQS uses the reachability tree constructed statically from the given PARTPNs models and finds the schedule that has the minimum total power dissipation. Their system model also assumes tasks execute for their WCET and other device usage information is known a-priori.

Isolation of device power management from CPU power management gives system-wise suboptimal solutions. Cheng and Goddard [CG05] integrated device scheduling, and DVFS. Their approach predicts the device usage times based on future release patterns and accordingly sets timers to initiate the wake-up procedure of the respective device. DVFS runs the tasks at low frequency to reduce the dynamic power dissipation. Consequently, it increases the execution time of tasks and also prolongs the active time of the devices. The approach aimed to select the processor's frequency that reduces the overall energy consumption. The proposed solution is based on an inter-task device scheduling and unnecessary prolongs the device's active time. The system-level power management algorithm developed by Devadas and Aydin [DA08a] for the frame-based systems (same period tasks) similarly addresses the interplay of DVFS and the device power management. Their work finds the optimal frequency set-point for the processor that minimises the energy consumption. While their approach is promising in principle, the restriction of frame-based (same period) tasks requires further work relaxing these assumptions.

Augustine et al. [AIS08] addressed the problem of selecting a sleep state of a device in a non-RT setting and, proposed offline and online power down strategies. Their proposed approaches for a single device have competitive ratio arbitrarily close to optimal. Huang et al. [HSC+11] proposed a device power management algorithm for hard RT system. The arrival curves used in their approach can model periodic, periodic with jitter (jitter is a delay in the release time of a task) and sporadic task models (event streams). However, such an approach cannot be extended to multiple devices in the system. Later on, Lampka et al. [LHC11] reduced the complexity of their algorithm through dynamic counters. Neukirchner et al. [NMA+12] addressed the arbitrary activation patters in RT systems that can also be used with the work of Huang et al. [HSC+11] to reduce the leakage energy consumption of the devices. While some work in device power

management in RT systems has been performed in a DVFS setting, this dissertation focuses on a sleep states and explores the different paradigm of intra-task device scheduling that addresses the shortcoming of the existing work by relaxing some of their assumptions.

2.1.3 Temperature-Aware Energy Minimisation

The state-of-the-art has mostly focused on the objective to reduce the peak temperature under performance constraints [BKP07, CHK07, CQ11, CHQ10, CWT09]. For instance, to reduce the peak temperature under performance constraint, Bansal et al. [BKP07] proposed speed scaling algorithms, Chen et al. [CHK07] presented approximation algorithm, while Chaturvedi and Quan [CQ11] used leakage conscious DVS scheduling. Chaturvedi et al. [CHQ10] developed a leakage-aware scheduling algorithm called m-oscillating for frame-based periodic hard RT systems to minimise the peak temperature. Given a 2-speed schedule, their m-oscillating algorithm divides the high speed interval and low speed interval into m sections, and run these sections alternatively. The maximum temperature decreases with an increase in m.

To explore temporal aspects and schedulability, Wang and Bettati [WB08] performed a delay analysis of the proposed reactive speed scheduling algorithm for the thermally-constrained RT system with identical-periodic tasks. The algorithm performs execution at maximum speed at low temperature and scales to the lower speed when it crosses some threshold to respect the temperature constraint. Later on, this work was extended for a more generic RT task model with FIFO and static-priority scheduling [WAB10]. Their thermal model does not consider the temperature-aware leakage current and does not perform energy minimisation. Quan and Chaturvedi [QC10] have done the feasibility analysis of the leakage-aware thermally-constrained periodic RT system. Chen et al. [CWT09] proposed two proactive speed scheduling algorithms under a thermal constraint for frame-based RT systems. In the first approach, the speed of the processor is estimated with an objective to minimise the response time of tasks (a time between its release and completion) under a given peak temperature constraint, while in the second approach, a speed schedule is determined to minimise the temperature at the beginning of the period under a given thermal and time constraints.

Another area of RT research in this domain is the energy minimisation under thermal constraint. For example, Wang et al. [WCST09] proposed a thermally constrained, energy efficient optimal proactive speed scheduling algorithm for frame-based RT tasks. They adopted an optimal control framework and executed tasks at higher speed in the beginning of the period and then gradually slow down the speed without violating the thermal constraint. Huang and Quan [HQ11] extended the m-oscillating algorithm [CHQ10] to reduce the energy consumption of the frame-based RT system. They derived the energy function in the form of m and obtained its optimal value with an exhaustive search under the given temperature constraint.

Recently, it has been shown that leakage-power dissipation is temperature dependent and increases rapidly with a rise in temperature [LHL05]. Yuan et al. [YLQ06] proposed the online temperature-aware leakage minimisation technique TALK for frame-based RT systems. The basic

idea is to execute workload when the processor is cool and postpone the workload at high temperature. A pattern based approach [YCTK10] reduces the energy consumption of the frame-based RT systems with a temperature dependent leakage-power dissipation. This approach divides the given frame (time horizon) into several equally-sized time-segments. The execution of the task is performed in the beginning of each time-segment and then the processor is cooled by using a low power sleep state. The required execution of the system and the idle time is equally divided among the time-segments. They developed a procedure to determine the optimal pattern that minimises the energy consumption.

The state-of-the-art corresponding to temperature aware energy minimisation though addresses the various aspects of RT systems under thermal constraints but make one or more of these assumptions: i) frame-based RT system, ii) leakage-power dissipation is independent of temperature, iii) do not consider energy consumption. The objective is to proposed leakage-aware thermally constrained energy minimisation approach for sporadic RT task model based on thermally constrained dynamic power management (TCDPM). This dissertation presents a detailed study on the equivalence of idealised DVFS with TCDPM. It shows that conventional idealised DVFS algorithms can be applied with minimal modifications to TCDPM to reduce the energy consumption of the system while relaxing the assumptions made in the literature.

2.2 Multicore Power Management

The multicores hardware platforms can be divided into two type, homogeneous and heterogeneous platforms. These two types of hardware platforms have been widely explored in the RT community. The work performed in these two type of platforms are summarised as follows.

2.2.1 Power Management in Homogeneous Platforms

In the context of homogeneous multicore RT systems, Chen and Kuo [CK07a] provided a comprehensive state-of-the-art survey regarding energy minimisation. Most of the achievements have been done in the context of partitioned schedulers, including DVFS and non-DVFS solutions. For instance, Alenawy and Aydin [AA05] compared the energy efficiency of the popular bin-packing heuristics (first-fit, best-fit, worst-fit and next-fit) for periodic real-time tasks assuming the rate-monotonic scheduler on each core. They considered different DVFS approaches and spotted that worst-fit is the winner in offline partitioning. Aydin and Yang [AY03] showed that worst-fit decreasing is a better choice in terms of energy consumption, but assuming EDF scheduler on each core. Kandhalu et al. [KKLR11] related the task period relationship in the allocation heuristics and proposed an energy efficient partitioned fixed-priority scheduling algorithm for the DVFS enabled chip multicores. Their work assumes a single voltage and clock frequency domain.

Chen et al. [CKYK07] studied energy-efficient task scheduling with task rejection on a platform with DVFS capability assuming a continuous spectrum of available frequencies. In this case, each rejected task was associated a penalty. They proposed an algorithm which aims at reducing

both the rejection penalty and the energy consumption. Chen et al. [CYLK08] derived approximation algorithms to partition an independent periodic task-set on a platform with DVFS capability to reduce the expected energy consumption. They considered a probabilistic distribution on the execution time requirement on the tasks and assumed that the leakage-power dissipation is a constant factor. Moreover, a number of sound algorithmic techniques have been developed in the literature to reclaim the unused resources upon multicore platforms. Using these techniques, a number of important theoretical results on the slack produced by the schedule of a task-set upon a target platform have been derived. For systems composed of both periodic and aperiodic tasks, a framework to accommodate the execution of aperiodic tasks in the slack left after the execution of the periodic tasks is available (see for example [PC08, CC89, Che08]).

Practical aspects (discrete speed, idle power, critical speed and task specific power characteristics) of DVFS for periodic task-model has been discussed by Zeng et al. [ZYTT09]. Their energy efficient scheduler assigns tasks with a first-fit strategy starting with the lowest frequency on each core and then gradually increases it to accommodate all the workload. Fu and Wang [FW11] proposed an online mechanism to reduce the energy consumption, but only for soft real-time systems. Their solutions monitors the utilisation of the cores to either consolidate the workload to shut-down or slow-down the frequency of the core.

Regarding semi-partitioned scheduling, Lu and Guo [LG11] integrated DVFS capabilities to existing semi-partitioned algorithms. The comparison of the energy saving is performed among different semi-partitioned algorithms, yet assuming a simplistic task scaling model where frequency and execution have a linear relation.

The class of global schedulers allows tasks to be dynamically assigned to the available processing cores at runtime and inherently provides support for load balancing among cores. The state-of-the-art of energy efficient systems assuming global schedulers is very limited and only few results exist. Anderson and Baruah [AB08] explored the trade-off between the energy consumption of RT tasks and the required number of cores on the multicore platform with an assumption that all the tasks run at the same frequency. Nelis et al. [NGDN08] proposed an energy saving algorithm for the well known global-EDF scheduler, assuming the sporadic constrained-deadline task-model. The offline core speed is computed while ensuring temporal constraints. The unused idle slots in the schedule are reclaimed by their online algorithm, called MOTE, to further reduce the core speed. Later, they proposed another slack reclamation algorithm, called MORA [NG09], which also exploits execution slack to reduce the frequency of the core.

Although this entire body of knowledge provides good insights on how to evaluate slack in a given schedule for the design of energy efficient systems, there appears to be little interest in the context of global scheduling and static power dissipation optimisation while executing sporadic tasks. The sporadic task model is a super-set of the classical periodic task model. For such a model, it is not possible to extend the existing techniques as neither the location nor the duration of the slack can be determined at system design-time, unfortunately. The major reasons for such a limited literature on energy-aware global scheduling in homogeneous multicores is inherent to the difficulty of predicting the impact of a decision taken on one core, to the scheduling on the

other cores. Since the scheduling decisions are globally taken, reducing the frequency of a specific core or sending it in a sleep state does not only affect that core but changes the overall platform schedule.

To the best of our knowledge, non-DVFS based power saving strategies tackling the leakage-power dissipation do not exist in global-EDF scheduling yet, and the proposed work in this thesis is the first effort to solve this issue. As a major difference with those previous works, the proposed framework does not change the frequency of the cores, which only reduces the dynamic power dissipation of the system, but instead allows us to send some cores to the sleep state, thus reducing the overall energy — static and dynamic — consumed by the system.

2.2.2 Power Management in Heterogeneous Platforms

The global and semi-partitioned schedulers are difficult to implement on heterogeneous multicore platforms, as different core types have different instruction set and migration becomes very expensive. Therefore, the focus of research in heterogeneous multicore platform is partitioned scheduling. Similar to the homogeneous multicore power management techniques, the state-of-the-art for partitioned heterogeneous multicores is limited in the non-DVFS setting. Yu and Prasanna [YP02] proposed the static allocation of the tasks in a RT system for the heterogeneous processing units under DVS. They formulated the problem as an Integer Linear Programming (ILP) and provided a linearisation heuristics. A pseudo polynomial time greedy algorithm [HTC07] is proposed by Huang et al. for the frame-based RT task model and heterogeneous systems. Furthermore, a greedy heuristics is provided to migrate the tasks from the overloaded processor to reduce energy consumption.

Given a library of heterogeneous processing unit and periodic task-set, Chen and Thiele [CT09] studied the selection of processing units to synthesis the energy efficient heterogeneous multicore platform while respecting the RT constraints. Saha et al. [SLD12] proposed the hybrid worst-fit genetic algorithm (HyWGA) to reduce the energy consumption of the heterogeneous multicore platform under given thermal constraint. The HyWGA algorithm integrates the worst-fit partitioning heuristics with a genetic algorithm. Watanabe et al. [WKI+07] presented a pipelined task scheduling method for the dependent task model to reduce the energy consumption of GALS MP-SoC under latency and throughput constraints. The problem is formulated as an Mixed-ILP and proposed a scheduling algorithm based on simulated annealing.

Luo and Jha addressed the tasks model with precedence constraints and proposed the list-scheduling strategy [LJ02] for the heterogeneous distributed systems. Chen and Thiele [CT08a] considered a case of 2 type heterogeneous processors and proposed a polynomial time approximation scheme based on the ratio of task execution times on the different processor types. Hsu et al. [HCK06a] addressed the synthesis problem of heterogeneous platform to schedule a set of RT tasks with a given energy constraint. They proposed approximation algorithm based on a rounding technique by applying a parametric relaxation on an ILP to minimise the processor cost under the given timing and energy cost. Hung et al. [HCK06b] considered a heterogeneous platform with 2 processing elements, one with DVS enabled core and second without DVS capability, with an

objective to reduce the overall energy consumption and maximise the energy saving in migration from DVS enabled core to non-DVS core. While DVS has its advantages, the state-of-the-art [YP02, HTC07, LJ02, CT08a, HCK06a, HCK06b] ignores the static power dissipation.

Yang et al. [YCKT09] proposed an approximation algorithm based on dynamic programming and provides polynomial-time solution when the number of processor types is a small constant. However, in the general case when the restriction over the number of processor types is relaxed, this scheme has exponential time/space complexity. They also assume static power dissipation of the system as a constant factor. The work of Chen et al. [CST09] presented a task assignment algorithm for periodic real-time tasks on heterogeneous platforms. The problem is formulated as an ILP problem. They relax some of the assumptions to adapt it into linear programming (LP) and solve it through extreme point theory [DT97]. The tasks assigned fractionally in the previous steps are reassigned through known heuristics such first-fit, best-fit, worst-fit or last-fit. They assume the static power dissipation of the system to be a constant factor and it cannot be reduced due to the significant overhead of sleep transitions [YCKT09, CST09]. This assumption does not hold for modern processors which contains several sleep states to reduce the static power dissipation of a system. Moreover, the static power dissipation has become a considerable part of the overall energy consumption. Therefore, the effect of the task allocation on the power dissipation in the sleep states should be considered to avoid suboptimal assignments.

In the context of heterogeneous multicores, the state-of-the-art assumes only dynamic power dissipation, ignores static power dissipation or considers it a constant factor while doing task allocation on such platforms. The objective of the research performed in this thesis is to relax the assumption of constant static power dissipation, and propose algorithms for heterogeneous platforms, while assuming a general power model and generic heterogeneous multicore platform.

Chapter 3

Model of Computation and Simulation Framework

A fundamental prerequisite of the work is the definition of the model of computation, as well as it's implementation for evaluation purposes, which are both introduced in this chapter. It includes the detailed description of the application model and different hardware aspects of the underlying platform. Later chapters of the thesis document require modifications and extensions of this model, which will be described in detail in the respective chapters.

3.1 Application Model

3.1.1 Task Model

This work assumes a traditional sporadic task model [Mok83b]. A task-set τ is composed of ℓ independent tasks $\tau \stackrel{\text{def}}{=} \{\tau_1, \tau_2, \cdots, \tau_\ell\}$. A task τ_i is described by a tuple $\tau_i \stackrel{\text{def}}{=} \langle C_i, D_i, T_i \rangle$, where D_i is the relative deadline, $T_i \geq D_i$ the minimum inter-arrival time between two consecutive jobs of τ_i and C_i the worst-case execution time. A sporadic task is allocated a budget of A_i and it releases an infinite sequence of jobs $j_{i,k}$ at run time separated by an interval of time greater than or equal to T_i. Figure 3.1 shows the specification of the tasks. The task's budget size and its allocation is discussed Section 3.1.2.

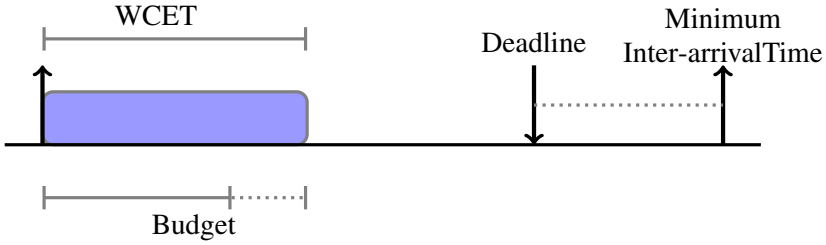


Figure 3.1: Task specifications

The k^{th} job $j_{i,k}$ of τ_i is defined as $j_{i,k} \stackrel{\text{def}}{=} \{r_{i,k}, c_{i,k}, d_{i,k}\}$, where $r_{i,k}$ is a release time, $c_{i,k} \leq C_i$ the actual execution time and $d_{i,k}$ the absolute deadline. The absolute deadline of a job $d_{i,k}$ cannot be determined before its release time and $d_{i,k} \stackrel{\text{def}}{=} r_{i,k} + D_i$. Job $j_{i,k}$ must complete before its absolute deadline $d_{i,k} \stackrel{\text{def}}{=} r_{i,k} + D_i$. Each job is associated a budget of $a_{i,k}$ and it decrements with the execution of the job. All the parameters mentioned above are real-valued. Job $j_{i,k}$ is said to be active at any time t if and only if $r_{i,k} \leq t$ and it is not completed yet. More precisely, an active job is said to be running at time t if it is allocated to a processor and is being executed. Otherwise, the active job is in the ready queue of the operating system and it is said to be ready. The subsets of active, running and ready jobs of τ at time t are denoted as active(τ,t), run(τ,t) and ready(τ,t), respectively. It holds that active$(\tau,t) =$ run$(\tau,t) \cup$ ready(τ,t).

As the tasks are considered independent, they do not share any resource except processor, holds no precedence and there is no communication or precedence constraint among them. The worst-case execution time of a task C_i is computed on the full speed of the processor (i.e., maximum frequency). Please note that this is only relevant for later parts of the thesis document dealing with DVFS. The hyper-period H of task-set τ is defined as the least common multiple of the tasks' minimum inter-arrival time T_i, i.e., H $\stackrel{\text{def}}{=}$ LCM$\{T_1, T_2, \ldots, T_\ell\}$. The notion of LCM is extended to real numbers as presented in Equation 3.1 (see [Bin09] for further details).

$$\text{LCM}(a,b) \stackrel{\text{def}}{=} \inf\{x \in \mathbb{R}_+ : \exists p,q \in \mathbb{N}_+, x = pa = qb\} \tag{3.1}$$

Definition 8 (Task's Utilisation)**.** *The individual utilisation of a task τ_i is the ratio between its worst-case execution C_i and the minimum inter-arrival time T_i.*

$$U_i \stackrel{\text{def}}{=} \frac{C_i}{T_i} \tag{3.2}$$

Definition 9 (Total System Utilisation)**.** *The total system utilisation U of the task-set τ is the summation of the individual utilisation of the tasks U_i in the system.*

$$U \stackrel{\text{def}}{=} \sum_{i=1}^{\ell} U_i \tag{3.3}$$

3.1.2 Temporal Isolation

A Constant Bandwidth Server [AB98] like algorithm is used in this work, and terminologies and concepts are borrowed from the rate-based earliest deadline first (RBED) framework [BBLB03], which provides temporal isolation by associating each task τ_i with an enforced budget A_i. At runtime the default value $a_{i,k}$ for a budget when releasing a job $j_{i,k}$ is A_i. However, the value for $a_{i,k}$ may be subject to manipulations including spare capacity assignment, borrowing from future releases of the same task or consumption of budget during execution.

The temporal isolation of the RBED framework allows for mixed-criticality workloads (hard, soft and best-effort type applications). The allocation of budget for SRT and best-effort (BE) tasks in the original work is less than or equal to WCET ($A_i \leq C_i$). For HRT tasks the budget is equal to

the WCET ($A_i = C_i$), to ensure the timely completion of all jobs. The scheduler pre-empts every job when it has used up its allocated budget $a_{i,k}$. Thus a job exceeding its budget cannot affect the overall schedulability of other tasks. In this work, HRT and SRT tasks are assumed to have a budget equal to their WCET and treated as RT tasks onwards, while a BE task may have budget less than or equal to its WCET time.

3.1.3 Hardware Model

3.1.3.1 Processor Model

A processor of a particular type m is defined as $\pi^m \stackrel{\text{def}}{=} \{P_A^m, P_I^m, \vec{\S}^m, \vec{f}^m\}$. It is characterised by unique power dissipation and execution capabilities, and consists of an active state, an idle state, a set of sleep states and in some cases a number of frequency set-points. A processor is said to have an idle state, if it is neither executing any task nor transitioning into a sleep state. The parameters of π^m are given with the following interpretation. P_A^m is the average-case power dissipation in the active state at maximum frequency set-point, P_I^m is the average-case power dissipation in the idle state, $\vec{\S}^m \stackrel{\text{def}}{=} (\S_1^m, \S_2^m, \ldots, \S_N^m)$, with $N \in \mathbb{N}^+$, is the vector of different sleep states (ranging from clock gating to shut-down of several chip sections) and $\vec{f}^m \stackrel{\text{def}}{=} (f_1^m, f_2^m, \ldots, f_{V^m}^m)$, with $V^m \in \mathbb{N}^+$, is the vector of frequency set-points available on processor π^m. The top speed or the maximum frequency of the processor is represented with f_1^m, while the $f_{V^m}^m$ corresponds to the slowest speed or lowest frequency of the processor.

A processor π^m has N sleep states in a vector $\vec{\S}^m$ and each sleep state $\S_n^m$ in $\vec{\S}^m$ is characterised by a quadruple $\S_n^m \stackrel{\text{def}}{=} \langle P_n^m, ts_n^m, tw_n^m, Es_n^m \rangle$, where P_n^m is the power dissipation in a sleep state, ts_n^m the transition delay of switching from active state to a sleep state, tw_n^m the wake up time from sleep state to an active state and Es_n^m the energy overhead of the complete sleep transition. The complete sleep transition overhead includes transition time from active to sleep state and wake up time from sleep state to active mode. It is denoted as $tsw_n^m = ts_n^m + tw_n^m$. For brevity of notation, it is assumed that transition delay of going into and out of a sleep state are equal and represented as tr_n^m (i.e., $tr_n^m = ts_n^m = tw_n^m$). Note that none of the proposed methods rely on this equality and can be easily adapted to work with different values for ts_n^m and tw_n^m. The transition overhead of the idle mode is considered negligible [LBC+03], i.e., a processor can instantly transition between active and idle mode.

The energy overhead Es_n^m associated to each sleep transition is caused by tuning of phase lock loop (PLL) and, loading and saving the system state or the contents of the registers, caches etc. In case the energy overhead Es_n^m of the sleep state is not given, a constant power dissipation is assumed during the transition phase which will be denoted as Ptr_n^m. A state transition is only initiated in a stable state, i.e., active or sleep state, in other words the system has to complete a transition once it is initiated. Each sleep state $\S_n^m$ has a break-even-time (BET) bet_n^m associated to it. Depending on the hardware characteristics, the sleep state parameters can be used to determine the break-even-time through any known techniques [ANP11, DA08a, CG05], however, for simplicity sake within this research the one set in Definition 10 is used.

Definition 10 (Break-even-time)**.** *The break-even-time bet_n^m of the sleep state $\S_n^m$ is the minimum time interval for which entering a sleep state is more efficient (energy-wise) when compared to any shallower sleep state, despite an extra overhead (time/energy) associated to this sleep transition.*

Where DVFS is used in the thesis document, we assume that all frequency points are always larger than or equal to a critical speed f_{crit}^m of a processor described in Definition 11. The leakage energy consumption always dominates the dynamic energy consumption below critical speed f_{crit}^m. It is the lower bound on processor speed as the execution below this speed increases execution time along with energy consumption [JPG04]. This bound can be computed for a processor considering its total energy consumption (dynamic+static).

Definition 11 (Critical Speed)**.** *The critical speed f_{crit}^m is the lower bound on processor speed (frequency reduction in DVFS) that minimises the total energy per processor cycle considering both dynamic and static (leakage) energy consumption.*

In the context of uniprocessor scheduling, the superscript that indicates the processor type is dropped in all the parameters of processor model. In case the DVFS is not considered as a power saving strategy, the vector of different frequencies in the system $\vec{f^m}$ is removed from the processor characteristics.

3.1.3.2 Device Model

Assume, W denotes the number of devices in the system. A set that collects these devices is defined as $\lambda \stackrel{\text{def}}{=} \{\lambda_1, \lambda_2, \ldots, \lambda_W\}$. Each device $\lambda_i \stackrel{\text{def}}{=} \{P_A^{\lambda_i}, \vec{\S^{\lambda_i}}\}$ is characterised by it active power dissipation $P_A^{\lambda_i}$ and a vector of sleep states $\vec{\S^{\lambda_i}}$. It is assumed, a device has no idle state, so it either stays in active mode or transition into a sleep state. Similar to a processor power model, a device λ_i may have N sleep states, i.e., $\vec{\S^{\lambda_i}} \stackrel{\text{def}}{=} \{\S_1^{\lambda_i}, \S_2^{\lambda_i}, \ldots, \S_N^{\lambda_i}\}$. Any sleep state of a device $\S_n^{\lambda_i}$ is characterised by a quadruple $\S_n^{\lambda_i} \stackrel{\text{def}}{=} \langle P_n^{\lambda_i}, ts_n^{\lambda_i}, tw_n^{\lambda_i}, Es_n^{\lambda_i} \rangle$, where $P_n^{\lambda_i}$ is the sleep state power dissipation, $ts_n^{\lambda_i}$ the transition delay of switching from active to sleep mode, $tw_n^{\lambda_i}$ the wake up time needed to transition out of a sleep sleep state and $Es_n^{\lambda_i}$ the extra overhead of energy consumption during the complete sleep transition phase. Similar to the processor model, it is assumed $ts_n^{\lambda_i} = tw_n^{\lambda_i}$ and represented as $tr_n^{\lambda_i}$. A complete sleep transition-phase delay of $\S_n^{\lambda_i}$ (i.e., from active to sleep state and sleep state to back in an active mode) is denoted as $tsw_n^{\lambda_i} = 2tr_n^{\lambda_i} = ts_n^{\lambda_i} + tw_n^{\lambda_i}$. A state transition may occur only from active to sleep mode or vice versa. Similar to a processor, in a devices, a state transition can only be initiated in a stable state, i.e., active or sleep state. The break-even-time of a device's sleep state is denoted as $bet_n^{\lambda_i}$ and follows the same definition as given in Definition 10. The above mentioned parameters of the device's sleep state $\S_n^{\lambda_i}$ can be used to estimate its BET. The measurement technique used for $bet_n^{\lambda_i}$ follows from the work of Cheng and Goddard [CG06, CG05]. The selected device power model is generic in a sense that each device can have multiple sleep states with different parameters.

3.1.4 Slack Sources

The processing time not used in a system is called slack. System slack can be categorised in two types, static and dynamic slack. The static slack exists due to spare capacity available in the system schedule. This spare capacity occurs as the system is loaded less than what can be guaranteed by the schedulability tests.

Definition 12 (Static Slack)**.** *The static slack is the spare capacity available in the schedule even if the tasks execute using the maximum processing resource specified (minimum inter arrival, and WCET).*

The dynamic slack occurs due to difference between worst-case assumptions made in the offline analysis and the actual online behaviour of the system. It is further divided into two components based on two different worst-case assumptions. The first assumption is that each job of a task will execute for its WCET C_i. Due to the inherent pessimism in all WCET approaches [WEE+08], most if not all of the jobs in a real scenario finish their execution earlier than their C_i and by the chosen budget A_i, and thus generate slack. This kind of slack is termed as execution slack S_e, and it is quantified by the difference in C_i and actual execution time. The execution slack S_e available in the system at time instant t is represented by the duple $S_e = \langle S_e^{sz}, S_e^{dl} \rangle$, where S_e^{sz} corresponds to effective slack size and S_e^{dl} corresponds to the absolute deadline of the slack.

Definition 13 (Execution Slack)**.** *Dynamic slack generated by the difference of WCET C_i and actual execution time of tasks is called execution slack.*

Similarly, the system is analysed with the second worst-case assumption that each job of a sporadic task will be released as soon as possible i.e., released periodically with the minimum inter-arrival time. However, for truly sporadic tasks this rarely occurs in HRT systems. Jobs of a sporadic tasks are released with a variable delay bounded by the minimum inter-arrival time. Such sporadic delay can potentially generate a slack in the system termed as sporadic slack. However, it is not necessary a sporadic delay will always generate a sporadic slack as demonstrated with the following example.

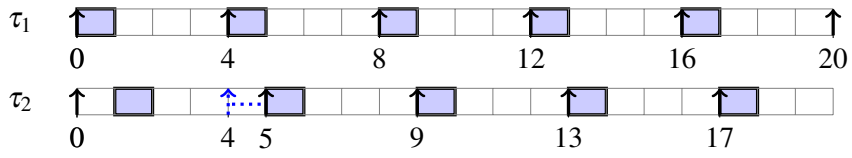


Figure 3.2: Sporadic slack example

Example 1. *Consider a task-set composed of two tasks $\tau_1 = \langle 1,4,4 \rangle$ and $\tau_2 = \langle 1,4,4 \rangle$. Assume the second instance of τ_2 is delayed by 1 time unit and arrives at time instant $t = 5$ as shown in Figure 3.2. Afterwards, all the other instances of τ_2 arrive exactly after T_2. In this example, it is evident that a sporadic slack is not generated. Hence, sporadic delay can potentially generate a slack but it is not a necessary condition.*

Definition 14 (Sporadic Slack)**.** *Dynamic slack generated by the delays in the task arrival after their minimum inter-arrival time is called sporadic slack.*

Naturally, the dynamic slack is generated online and can only be identified at run-time.

3.1.5 Slack Management Algorithm

There are number of execution slack reclamation algorithms exists in the literature [AMMM01, ZC02, JG05]. These approaches are efficient and exhaustively collate the execution slack. To further reduce the complexity of these approaches, a new execution slack reclamation algorithms is presented here. The proposed approach is based on the basic principles of [LB05]. The basic idea is to keep S_e received from previously completed jobs at time instant t in a central container. In contrast to traditional execution slack management algorithms [JG05, ZC02, AMMM01], this approach uses a single slack container and reduces the extra overhead of keeping multiple slack containers at different priority levels. In idle mode the system consumes available execution slack [PLHE09].

Algorithm 1 Slack Management

1: **if** (*Ready Queue Empty*) **then**
2: Consume execution slack S_e
3: **end if**

4: **Slack Collection Phase**
5: **Slack Update On Job** $j_{i,k}$ **Completion**
6: $S_e^{sz} += a_{i,k}$
7: $S_e^{dl} = \max\{S_e^{dl}, d_{i,k}\}$

8: **Slack Preservation Phase**
9: **Method 1 [Adding Slack in the Jobs Budget]**
10: **On Every Scheduling Event**
11: **if** ($S_e^{dl} \leq d_{i,k}$) **then**
12: $a_{i,k} += S_e^{sz}$
13: $S_e^{dl} = 0$
14: $S_e^{sz} = 0$
15: **end if**
16: **Method 2 [Extending Slack Deadline on Job Arrival]**
17: **On Every Scheduling Event**
18: **if** ($S_e^{sz} > 0$) **then**
19: $S_e^{dl} = \max\{S_e^{dl}, d_{i,k}\}$
20: **end if**

The pseudo-code of the execution slack management algorithm is given in Algorithm 1. It has two phases, slack collection and slack preservation. These two phases are explained as follows.

i) **Slack Collection:** This phase is invoked when any job generates execution slack. Assume a job $j_{i,k}$ executes for less than its WCET and generates the execution slack S_e of size X.

The size S_e^{sz} of execution slack is incremented by X (i.e., $S_e^{sz} += X$). If the deadline of the job $j_{i,k}$ that generated the execution slack of size X is greater than S_e^{dl} (i.e., $S_e^{dl} < d_{i,k}$), then the deadline of S_e is extended to the deadline of $j_{i,k}$, i.e., $S_e^{dl} = d_{i,k}$, otherwise its previous deadline is maintained. In general, the deadline of S_e is updated by the expression $S_e^{dl} = \max\{S_e^{dl}, d_{i,k}\}$ [LB05].

ii) **Slack Preservation:** One of the objective of the proposed algorithm is to preserve the available slack in the system without violating the temporal constraints. In order to maintain the schedulability of the system with EDF, the highest priority workload (workload with earliest deadline) should be executed first. Assume at time instant t, a slack container has an execution slack of size S_e^{sz} with a deadline S_e^{dl}. A job $j_{i,k}$ starts its execution at time t and has an absolute deadline greater than the deadline of the execution slack, i.e., $(S_e^{dl} < d_{i,k})$. In this case, maintaining the execution slack with the same deadline during the execution of $j_{i,k}$ means we are changing the schedule and it may miss a deadline (higher priority workload should execute/consume first). In order to solve this issues, two different methods are proposed.

- **Method 1 [Adding Slack in the Jobs Budget]:** On every scheduling event, the priority of the execution slack is compared against the priority of the current job $j_{i,k}$. If the priority of the execution slack S_e is greater than or equal to the priority the current job $j_{i,k}$ to be executed, the actual budget $a_{i,k}$ of the current job $j_{i,k}$ is incremented by S_e^{sz}; i.e., $a_{i,k} += S_e^{sz}$. When a slack S_e is allocated to a job $j_{i,k}$, the slack container is reset to zero. This method is presented in Algorithm 1. Moreover, if the slack has a lower priority compared to the job $j_{i,k}$, then the slack cannot be passed or added to the job's budget and it can only be maintained in the slack container.
- **Method 2 [Extending Slack Deadline on Job Arrival]:** The another way to solve the same issue presented above is to extend the deadline of the execution slack instead of adding it to the budget of the job. The deadline of the execution slack is updated when the job $j_{i,k}$ resumes/starts its execution in the presence of execution slack ($S_e^{sz} > 0$). In this case, the deadline of the slack is updated to $S_e^{dl} = \max\{S_e^{dl}, d_{i,k}\}$. This is method is also equivalent to the donation of the execution slack S_e to the job $j_{i,k}$ that will complete its execution S_e^{sz} early upon donation.

Theorem 15. *Algorithm 1 does not affect the correctness of the schedule produced by the* EDF *scheduler.*

Proof. It can be proved through the rational mentioned in the original RBED work [BBLB03]. Algorithm 1 effectively extends the deadline of jobs completing earlier than their worst-case execution time. The sustainability property of the EDF scheduler [BB06] states that a task-set schedulable with the EDF scheduling policy on a unicore platform will preserve its schedulability when jobs extend their deadline and/or execute for less than their worst-case execution time. Hence, the theorem follows directly from the sustainability property of EDF. □

The advantage of this approach is that slack generated at different priority levels are eventually accumulated implicitly with a very simplistic and transparent approach using just one container to hold the slack. The disadvantage of such an approach is the temporary unavailability of the slack in the presence of long period tasks. The slack generated with long deadline will decrease the priority of the already available slack and restrict the high priority jobs to use it.

3.2 Simulation Framework

A simulator for power aware and real-time systems (SPARTS) [NAP11b] is developed to evaluate the effectiveness of the proposed algorithms in this thesis. SPARTS is an open source simulator of a generic real-time device and its source code is available at the following link [NAP11a]. It is built as a slot-based execution environment and provides extensive flexibility in task-set generation for different objectives and scenarios. The modular structure of SPARTS allows easy development and integration of new scheduling algorithms for both, single and multi-core systems. It performs the simulation in event-driven manner. Rather than doing cycle-step execution, SPARTS works by looking backward into the interval between two consecutive job releases and calculates the execution without unnecessary cycle-level granularity. This approach allows to save computation and yet provide correct execution modelling. This allows to perform the simulations of large task-sets for long periods of time with high temporal efficiency.

SPARTS allows to generate task-sets from a large number of fine grained small tasks to a small number of coarse grained tasks to cover a wide range of different systems. The share distributions ξ_i provides the percentage share of RT and BE tasks in the overall system utilisation and the number of tasks. For example, a share distribution $\xi_i = \langle RT, BE \rangle = \langle 40\%, 60\% \rangle$ divides the task-set such that it has 40% RT and 60% BE tasks. Similarly, 40% of the system utilisation is distributed among RT tasks and 60% among BE Tasks. The utilisation allocated to a specific task class (RT or BE) is distributed randomly among the tasks of this class. The actual individual utilisation per task is generated such that the target share for each scheduling class is achieved. The minimum and the maximum limits are provided for the task classes (RT and BE) to chose their T_i time. Starting from the utilisation U_i and T_i, the WCET of each task is deemed to be $C_i = U_i \times T_i$. It has to be noted that due to numerical rounding in the parameters used in the SPARTS simulator to generate the task-set with a target utilisation of x has a resulting utilisation of $x - \varepsilon$, where ε is a very small number indicating the rounding error. It remains within 0.6% of the total utilisation for most of the experiments conduced in this dissertation.

Beyond those initial settings a two level approach is used to generate a wide variety of different tasks and subsequently varying jobs. Tasks are further annotated with a limit on the sporadic delay Γ_i in the interval $[0, \Gamma \times T_i]$ and on BCET C_i^b in the interval $[C^b \times C_i, C_i]$. The varying behaviour of different jobs of the same task depends on the system's state and input parameters. It is modelled by assigning each $j_{i,k}$ an actual sporadic delay in $[0, \Gamma_i]$ interval and an actual execution time in $[C_i^b, C_i]$ interval. All random numbers are taken from a uniform distribution and unless explicit values are given, random numbers are used for all assignments. For each task-set

of a particular configuration the seed value of the random number generator is varied from one to hundred. The results of these hundred values are averaged to get one set point presented in graphs of all experiments.

Chapter 4

Unicore Power Management

In this era of multicore platforms, a single processor is still the most commonly used designed choice in RT systems to avoid the complexity involved while ensuring the temporal correctness. Researchers have been studying the RT uniprocessor embedded systems that consists of a finite number of recurring tasks for more than forty years now. Over this period of time, they have come up with a number of very important results, developed some useful algorithmic techniques and built up an entire body of intuitions. Taken together, these results, techniques and intuitions have allowed system designers to come up with a very good understanding of the manner in which RT embedded uniprocessor systems behave.

The emerging application requirements in the embedded systems arena have increased dramatically over the past years in terms of computing demands. Following Moore's law [Moo98], CMOS chip manufacturers have successfully minimised the size, decreased power and increased performance of transistors. The transistor-technology miniaturisation has allowed the semiconductor industry to place more functionality on the same chip area. One of the side effect of the technology scaling is an increase in leakage current — especially in deep submicron technology nodes (65nm and below) — contributes to 30-50% of the total power dissipation. The exponential increase in leakage current requires more attention when it comes to power management approaches. On the other side, tremendous amount of work exists in DVFS ranging from theoretical results to practical approaches. Therefore, this chapter assuming sporadic task model initially extends the existing CPU power management approaches to yield the optimal energy savings and then propose new approaches to reduce the practical limitations of the existing work in leakage-power dissipation. Finally, the effect of temperature on the leakage-power dissipation is explored in the RT context.

The sleep states available at system level can be used in different ways to reduce the leakage current. Some approaches extend the sleep interval of the processor already in sleep state online, while other approaches execute the workload as soon as possible and initiate sleep state for the pre-determined sleep interval avoiding any online processing in it. In both cases, the objective is to minimise the transition overhead associated to each sleep transition and maximise the energy savings. This is a non-trivial issue assuming multiple sleep states in the sporadic task-model. A

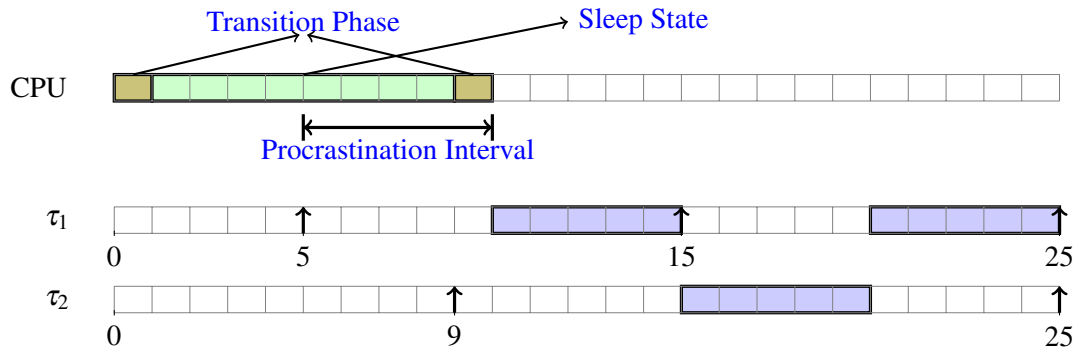


Figure 4.1: Schedule with $\tau_1 = \langle 5, 10, 10 \rangle$, $\tau_2 = \langle 5, 16, 16 \rangle$ and $tr_n = 1$

shallower sleep state has lower transition overhead but consumes more energy when compared to a deeper sleep state and vice versa. Irrespective of the strategy used to initiate a sleep state, a sleep interval is computed based on the minimum inter-arrival time and WCET of the jobs due to their dynamic behaviour — in arrival sequence and execution time — to ensure the temporal correctness of a schedule.

4.1 Procrastination Scheduling

4.1.1 Basics

Procrastination scheduling is commonly used at system level to reduce the leakage-power dissipation. In this technique the execution of the processor already in sleep state is delayed as much as possible while ensuring the timing constraints of all tasks are met. This is demonstrated with the help of an example given below.

Example 2. *Consider a system with two tasks* $\tau_1 = \langle 5, 10, 10 \rangle$ *and* $\tau_2 = \langle 5, 16, 16 \rangle$ *as given in Figure 4.1. Assume the processor is idle at time instant* 0 *and transitions into a sleep state with a transition delay of* $tr_n = 1$*. A processor will stay in the sleep state unless there is a job arrival. In this example the first job arrives at time instant* $t = 5$*. At this moment the scheduler computes how much further it can delay the current transition out of the sleep state such that all jobs meet their deadlines. The longest duration of such an interval is desired to reduce the energy consumption, "both by using deeper sleep states and making less transitions into sleep states (less overhead)". In the optimal case, it can be delayed for* 5 *time units.*

As the scheduler has to compute the procrastination interval during the sleep state, it requires extra hardware to perform such computation. The need for external hardware is one of the limitation of procrastination scheduling approaches. This external hardware increases the design/integration effort, communication complexity, energy consumption and cost of chip. Formally, the procrastination interval is defined as follows.

Definition 16 (Procrastination Interval)**.** *The procrastination interval is the maximum time interval allowed to delay the execution of ready tasks without violating any timing constraints of the system.*

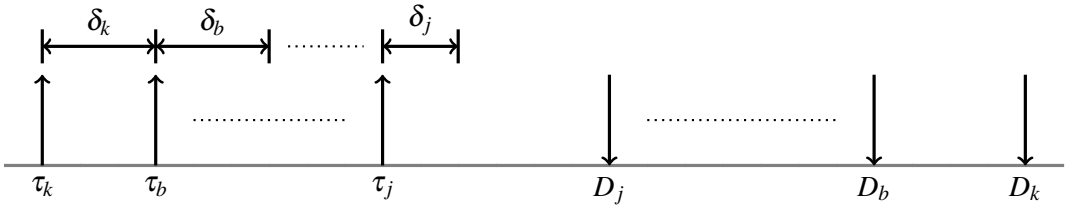


Figure 4.2: "Accumulated delays under EDF scheduling [LRK03]"

There exists different algorithms [LRK03, JPG04, JG04] to compute the procrastination interval used for power saving. These algorithms which are based on procrastination scheduling approximate the procrastination interval of tasks leading to sub-optimal energy savings. The procrastination algorithm proposed in this section computes the optimal procrastination interval and fills the gap in the related work. Before going into the details of the proposed procrastination algorithm, let us identify the pessimism involved in the state-of-the-art when computing the procrastination interval. Initially, implicit deadline task model is assumed, i.e., $D_i = T_i$, $\forall \tau_i \in \tau$, to compare against the state-of-the-art which assumes this model. Later in Section 4.1.5, this restriction is relaxed to a more general case, i.e., the constrained deadline task model, where tasks may have deadlines less than their periods ($D_i \leq T_i$).

Lee et al. [LRK03] initially proposed the online leakage-aware procrastination scheduling mechanism called LC-EDF. To understand the basic principle behind this algorithm, consider an example given in Figure 4.2 (this figure is taken from the work of Lee et al. [LRK03] and each arrival represents an instance of a task). Assume an instance of a task τ_k is the first arrival in a sleep state. The procrastination interval Q_k of this instance of a task τ_k is computed with the condition $\sum_{\forall \tau_i \in \tau : i \neq k} \frac{C_i}{T_i} + \frac{C_k + Q_k}{T_k} = 1$. Suppose t is the current time then the timer is initialised with $t + Q_k - tr_n$ to wake-up the system, where tr_n is the transition-out delay of the sleep state. After the timer initialisation, a procrastination interval is only recomputed when a new arrival has the absolute deadline smaller than the previous arrivals in the ready queue. For instance, after $\delta_k \leq Q_k - tr_n$ time units, instance of a task τ_b arrives with an absolute deadline less than the absolute deadline of τ_k's instance; a new procrastination interval Q_b is determined with Equation 4.1.

$$\sum_{\forall \tau_i \in \tau : i \notin \{k,b\}} \frac{C_i}{T_i} + \frac{C_k + \delta_k}{T_k} + \frac{C_b + Q_b}{T_b} = 1 \tag{4.1}$$

The wake-up timer is reset to $t + Q_b - tr_n$. Similarly, if an instance of any other task τ_j with the highest priority arrives, the procrastination interval Q_j in the sleep state of a processor is determined by using Equation 4.2, where $lp(j)$ is the set of indices of arrivals before τ_j's instance and with a deadlines longer than the deadline of τ_j's instance. In this equation, δ_i is the interval between an arrival of task τ_i's instance (having highest priority at that instant) and any next arrival having priority higher than the job of τ_i in the system's sleep state. The limitations of LC-EDF are the increased online complexity to maintain a track of δ_i and considering the utilisation of the low

priority tasks while computing the procrastination interval.

$$\sum_{\forall \tau_i \in \tau : i \notin lp(j), i \neq j} \frac{C_i}{T_i} + \sum_{i \in lp(j)} \frac{C_i + \delta_i}{T_i} + \frac{C_j + Q_j}{T_j} = 1 \tag{4.2}$$

Jejurikar et al. [JPG04] proposed an offline method based on Theorem 17 to compute the *procrastination interval of each task*, where X_k is the normalised frequency of the processor while executing a task τ_k. For the ease of presentation, the value of X_k is set to 1, i.e., maximum frequency. Their algorithm is represented as PROC hereafter. PROC reduces the online complexity of LC-EDF as the procrastination interval of each task is computed offline. Similar to LC-EDF (Figure 4.2), in the online phase of PROC, the tasks are scheduled with EDF scheduling. The system transitions into a sleep state when idle. The selection of the sleep state is based on the minimum procrastination interval in the task-set i.e., $\min_{\forall \tau_i \in \tau}(Z_i)$. A sleep state that has a break-even-time bet_n greater than $\min_{\forall \tau_i \in \tau}(Z_i)$ and consume minimum energy for this interval is selected for the system. The first task that arrives in sleep mode initialises the wake-up timer ϖ with its procrastination interval minus the transition-out delay. If another task (say τ_n) arrives before the timer expires, the timer value is adjusted as follows: $\varpi \leftarrow \min(\varpi, t + Z_n - tr_n)$, where t is the current time and Z_n is the procrastination interval of τ_n. It is proved that their derived technique is superior to LC-EDF to compute the procrastination intervals.

Theorem 17. *[JPG04] Given tasks in τ are ordered in non-decreasing order of their periods, the procrastination algorithm guarantees all task deadlines if the procrastination interval Z_i of each task τ_i satisfies the following two conditions:*

$$\forall \tau_i \in \tau, \qquad \frac{Z_i}{T_i} + \sum_{\forall \tau_k \in \tau : k \leq i} \frac{1}{X_k} \frac{C_k}{T_k} \leq 1 \tag{4.3}$$

$$\textit{and} \qquad \forall k < i, \qquad Z_k \leq Z_i \tag{4.4}$$

While computing the procrastination interval of a task τ_i, PROC [JPG04] only considers the utilisation of the tasks having priority greater than or equal to τ_i including the utilisation of τ_i (assuming a synchronous release of all tasks also known as *critical instant* in literature). Moreover, if any of the low priority task produce a low procrastination interval when compared to the high priority tasks, the procrastination interval of all the high priority tasks are readjusted by considering Equation 4.4. This latter equation is driven by the online phase of PROC (see [JPG04] for details). The proposed method has its merits as it reduces the set of tasks considered for the procrastination of each task and requires simple hardware to implement the algorithm. However, it has two main limitations. Firstly, it *approximates* the procrastination intervals by considering tasks utilisations and secondly, it cannot be effectively extended to the constrained deadline model. The first shortcoming is demonstrated with the help of the following example.

Example 3. *Assume a task-set consists of three tasks $\tau_1 = \langle 2,4,4 \rangle$, $\tau_2 = \langle 3,7,7 \rangle$ and $\tau_3 = \langle 0.25,14,14 \rangle$. Rearranging Equation 4.3, Z_i can be computed with Equation 4.5 as given below.*

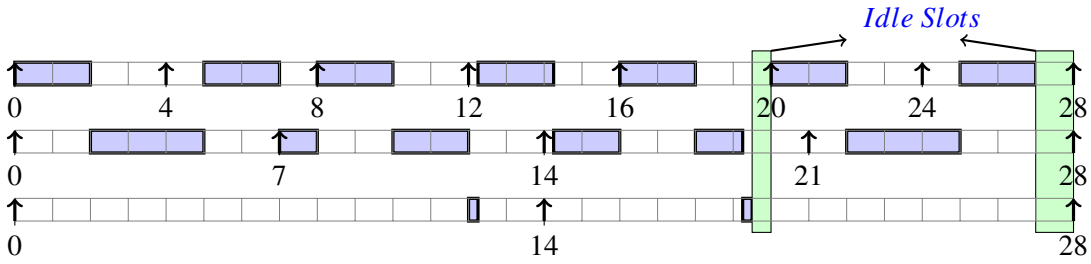


Figure 4.3: Schedule with $\tau_1 = \langle 2,4,4\rangle, \tau_2 = \langle 3,7,7\rangle$ and $\tau_3 = \langle 0.25,14,14\rangle$

$$
\begin{aligned}
Z_i &= \left(1 - \sum_{\forall \tau_k \in \tau : k \leq i} \frac{C_k}{T_k}\right) T_i \qquad (4.5)\\
Z_1 &= (1 - \frac{2}{4})4 = 2\\
Z_2 &= (1 - \frac{2}{4} - \frac{3}{7})7 = 0.5\\
Z_3 &= (1 - \frac{2}{4} - \frac{3}{7} - \frac{0.25}{14})14 = 0.75
\end{aligned}
$$

Final values after applying Equation 4.4 are $Z_1 = 0.5$, $Z_2 = 0.5$ *and* $Z_3 = 0.75$. *Figure 4.3 shows the schedule for the aforementioned example. With a careful observation it can be seen that the procrastination interval of* τ_1, τ_2 *and* τ_3 *can be extended to* $1, 1$ *and* 1.5 *time units respectively without causing any deadline miss, which represents* 50% *gain over* PROC.

This example illustrates that substantial energy gains can be achieved by improving the method to compute the procrastination interval of each task. The algorithm presented in Section 4.1.2 shows that the demand bound function used to estimate the procrastination interval of each task not only eliminates the sub-optimally in the related work but can effectively be extended to the constrained deadline model.

4.1.2 Demand Bound Function Based Procrastination (DBFP)

The demand bound function (DBF) [BRH90, PL05] is an abstraction of the computation requirements of tasks which has been observed to correlate very closely with schedulability property of the task-set. It is defined as follows.

Definition 18 (Demand Bound Function [BRH90])**.** *The demand for any constrained deadline task* τ_i *and positive time* t*, denoted by* $\mathrm{DBF}(\tau_i, t)$*, is the maximum cumulative execution requirement of jobs of task* τ_i *in any interval of length* t*. Formally,* $\mathrm{DBF}(\tau_i, t)$ *is presented in Equation 4.6.*

$$\forall t \geq 0, \qquad \mathrm{DBF}(\tau_i, t) \stackrel{\text{def}}{=} \left(\left\lfloor \frac{t - D_i}{T_i} \right\rfloor + 1 \right) \cdot C_i \qquad (4.6)$$

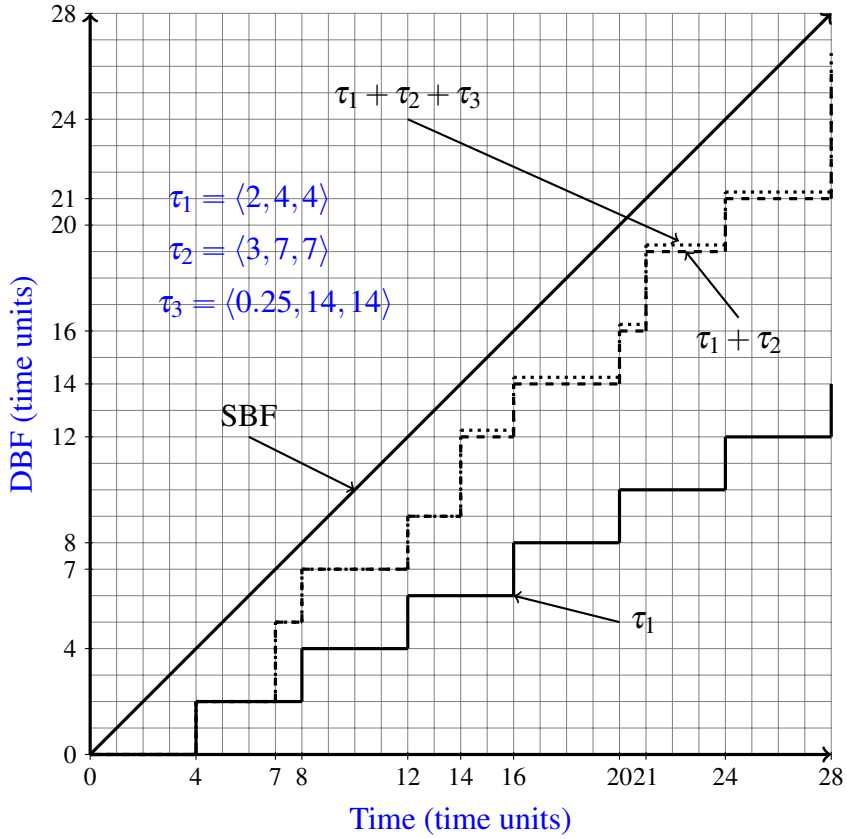


Figure 4.4: Demand bound function with tasks $\tau_1 = \langle 2,4,4 \rangle$, $\tau_2 = \langle 3,7,7 \rangle$ and $\tau_3 = \langle 0.25,14,14 \rangle$

Equation 4.6 shows that $\text{DBF}(\tau_i,t)$ is a step-case function in t with first step occurring at time $t = D_i$ and subsequent steps separated by *exactly* T_i time units. In case of implicit deadline task model (i.e., $D_i = T_i$), the $\text{DBF}(\tau_i,t)$ of task τ_i presented in Equation 4.6 can be rewritten as shown in Equation 4.7.

$$\text{DBF}_I(\tau_i,t) = \left\lfloor \frac{t}{T_i} \right\rfloor C_i \qquad \text{as} \qquad t \geq 0 \tag{4.7}$$

The demand of the whole task-set $\text{DBF}(\tau,t)$ at time instant t is the summation of the demands from the individual tasks as defined in Equation 4.8.

$$\text{DBF}(\tau,t) \stackrel{\text{def}}{=} \sum_{\tau_i \in \tau} \text{DBF}(\tau_i,t) \tag{4.8}$$

The demand bound function can be used to compute the procrastination interval of each task in the context of uniprocessor scheduling. The proposed algorithm DBFP uses the same logic as the one given in Theorem 17 but computes the procrastination interval of a task with DBF instead of considering tasks utilisations. The proposed algorithm has four steps and is demonstrated with the help of a running example given in Figure 4.3.

1. The tasks are sorted in a non-decreasing order with respect to their relative deadlines.

2. For each task τ_i, a function sums up the demand of a task τ_i along with the tasks having relative deadlines less than the relative deadline of τ_i. In the given example, three stair case functions $\mathrm{DBF}(\tau_1, \mathrm{H})$, $\mathrm{DBF}(\tau_1 + \tau_2, \mathrm{H})$ and $\mathrm{DBF}(\tau_1 + \tau_2 + \tau_3, \mathrm{H})$ are computed for the corresponding tasks τ_1, τ_2 and τ_3 respectively. These functions are presented in Figure 4.4 for the given task-set.

3. In the third step, the corresponding function of τ_i obtained in step 2 is subtracted from the supply bound function SBF. SBF is the supply provided by the processor and in uniprocessor case it is a straight line with a slope of 1 passing through origin as presented in Figure 4.4. Due to the stair-case property of the DBF, it is sufficient to compute the difference at the deadlines. It has to be noted that this difference is computed at all deadlines between the first deadline of a task τ_i till the end of the hyper-period (the reason is explained in Theorem 19). The minimum of these differences gives the maximum procrastination interval of a task τ_i. Let χ_i represent the minimum difference then for the given example, $\chi_1 = 2$, $\chi_2 = 1$ and $\chi_3 = 1.5$.

4. In the last step, a condition $\forall k < i, \chi_k \leq \chi_i$ is applied on the procrastination intervals of all tasks. In the given example, the procrastination interval of τ_1 is greater than the procrastination interval of τ_2. Therefore, the value of χ_1 is scaled down to 1 and the final procrastination values are $\chi_1 = 1$, $\chi_2 = 1$ and $\chi_3 = 1.5$.

The DBF based procrastination (DBFP) scheme achieves extended sleep intervals for the given task-set. Indeed when $D_i \leq T_i$ the utilisation is no longer a good metric for the computation requirement of the tasks and may cause deadline violations, whereas the DBFP approach is easily extensible. Similar to PROC, DBFP computes the procrastination interval for each task. Therefore, the online phase of PROC can be applied to DBFP. Another online algorithm [JG05] that incorporates the slack management in the work of Jejurikar et al. [JPG04] can also be used with DBFP. However, the work presented in this section emphasises on the computation of the procrastination interval rather than online methods to utilise it. Theorem 19 shows the proof of correctness of the schedulability concerns of DBFP with the implicit deadline task model.

Theorem 19. *Given tasks in τ are ordered in a non-decreasing order of their relative deadlines, the* DBFP *algorithm preserves all task deadlines with* EDF*, if the maximum procrastination interval of a task τ_i, denoted by χ_i, is computed with Equation 4.9 while respecting the condition given in Equation 4.10.*

$$\chi_i \stackrel{\text{def}}{=} \min_{\forall \tau_j \in \tau : j \leq i, \forall t \geq 0} \left\{ t - \sum_{\forall \tau_k \in \tau : k \leq i} \mathrm{DBF}_I(\tau_k, t) \right\} \tag{4.9}$$

$$= \min_{\forall \tau_j \in \tau : j \leq i, \forall t \in M(i,j)} \left\{ t - \sum_{\forall \tau_k \in \tau : k \leq i} \left\lfloor \frac{t}{T_k} \right\rfloor C_k \right\}$$

$$\textit{where} \qquad M(i,j) = \left\{ n_j T_j : \left\lceil \frac{T_i}{T_j} \right\rceil \leq n_j \leq \left\lfloor \frac{\mathrm{H}}{T_j} \right\rfloor \right\}$$

$$\forall k < i, \chi_k \leq \chi_i \tag{4.10}$$

Proof Sketch. Suppose a task τ_i arrives while the processor is in a sleep state. The timer is set to the procrastination interval computed with Equation 4.9 respecting the condition given in Equation 4.10. The time interval to wake up the system can only be decreased with an arrival of new task. This procrastination interval can be seen as an additional task τ_{proc} with a priority equal to the highest priority task, execution time equal to the wake-up sleep interval and it executes before the next busy period. Equation 4.10 ensures that all the tasks with deadlines greater than or equal to τ_i will have procrastination interval greater than or equal to χ_i. Therefore, τ_{proc} will not increase the system demand beyond the SBF in the presence of low priority tasks. Furthermore, the higher priority tasks can only shorten the execution time of τ_{proc} (i.e., procrastination interval) on their arrival to respect their deadlines and the deadlines of the other tasks. Thus the sleep interval is bounded by the procrastination interval of the first task and it only decreases with the new arrivals, therefore, based on the previous logic it will not affect the schedulability of any high priority task. Moreover, it is sufficient to consider the deadlines in the interval $[D_i, \mathrm{H}]$ as the procrastination interval of a task is only considered when it has the highest priority on its arrival in the ready queue. As none of the tasks miss their deadline, therefore, the theorem holds. □

4.1.3 Analytical Analysis of Procrastination Interval of each Task

The best known maximum procrastination interval is the one derived with PROC method for each task in the state-of-the-art. This is obtained by considering the worst-case scenario i.e., critical instant. This section shows that the procrastination interval computed for any task through DBFP will always be greater than or equal to Z_i (see Lemma 20).

Lemma 20. *Given tasks in τ are ordered in a non-decreasing order of their relative deadlines, the procrastination interval χ_i for any task τ_i computed with* DBFP *scheme is always greater than or equal to the procrastination interval Z_i computed* PROC*, i.e.,*

$$\forall \tau_j \in \tau : j \leq i, \forall t \in M(i,j) \min \left\{ t - \sum_{\forall \tau_k \in \tau : k \leq i} \left\lfloor \frac{t}{T_k} \right\rfloor C_k \right\} \geq \left(1 - \sum_{\forall \tau_k \in \tau : k \leq i} \frac{C_k}{T_k} \right) T_i \tag{4.11}$$

$$\text{where} \qquad M(i,j) = \left\{ n_j T_j : \left\lceil \frac{T_i}{T_j} \right\rceil \leq n_j \leq \left\lfloor \frac{\mathrm{H}}{T_j} \right\rfloor \right\}$$

Proof. The inequality given in Equation 4.11 can be proven by showing the procrastination interval computed with DBFP is greater than or equal to Z_i at all the deadlines between the first deadline of a task τ_i and the hyper-period H. PROC computes Z_i on the deadline of the task under consideration. To compare these two approaches, their functions are interpolated for all points in the demand bound function. To further illustrate this, consider the task-set of an example depicted in Figure 4.3. The interpolation is achieved through a straight line between two points $A(T_i, T_i \sum_{\forall \tau_k \in \tau : k \leq i} \frac{C_k}{T_k})$ and $B(\mathrm{H}, \mathrm{H} \sum_{\forall \tau_k \in \tau : k \leq i} \frac{C_k}{T_k})$ as shown in Figure 4.5. This figure illustrates the approximation by the straight line while the actual demand with the staircase function. Note that

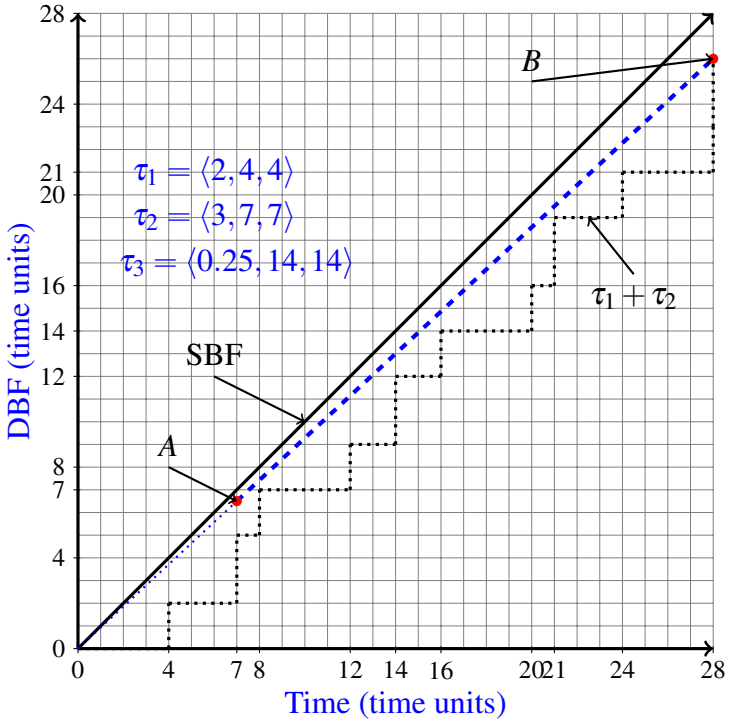


Figure 4.5: Procrastination interval for τ_2

Figure 4.5 only shows it for χ_2 and Z_2. The slope of this line is equal to $\sum_{\forall \tau_k \in \tau : k \leq i} U_k$. To demonstrate that $\chi_i \geq Z_i$, it is sufficient to prove this inequality in $[T_i, \mathrm{H}]$ (see Theorem 19). This interval is divided into two cases.

a) At time instances T_i and H, i.e., the deadline of τ_i and the hyper-period respectively.

b) An interval between time instant T_i and H, i.e., (A, B).

Case a) At the first time instant T_i, Equation 4.12 compares the two approaches.

$$T_i - \sum_{\forall \tau_k \in \tau : k \leq i} \left\lfloor \frac{T_i}{T_k} \right\rfloor C_k \geq \left(1 - \sum_{\forall \tau_k \in \tau : k \leq i} \frac{C_k}{T_k} \right) T_i \tag{4.12}$$

$$\Leftarrow \quad - \sum_{\forall \tau_k \in \tau : k \leq i} \left\lfloor \frac{T_i}{T_k} \right\rfloor C_k \geq - \sum_{\forall \tau_k \in \tau : k \leq i} \frac{C_k}{T_k} T_i$$

$$\Leftarrow \quad \cancel{\sum_{\forall \tau_k \in \tau : k \leq i}} \left\lfloor \frac{T_i}{T_k} \right\rfloor \cancel{C_k} \leq \cancel{\sum_{\forall \tau_k \in \tau : k \leq i}} \frac{T_i}{T_k} \cancel{C_k}$$

$$\Leftarrow \quad \left\lfloor \frac{T_i}{T_k} \right\rfloor \leq \left(\frac{T_i}{T_k} \right) \tag{4.13}$$

Equation 4.13 shows that at time instant T_i, Equation 4.11 holds. The same reasoning can be applied at time instant H (i.e., by replacing the T_i with H in Equation 4.12).

Case b: As already mentioned in the beginning of this proof, the demand of PROC in an interval (A,B) is computed with a straight line of slope $\sum_{\forall \tau_k \in \tau : k \leq i} U_k$ and is compared against DBF at all deadlines. The equation of the line is $y = mx + c$, where m is a slope and c is a *y-intercept*. The *y-intercept* is zero (i.e., $c = 0$) as the line passes through the origin. Hence, the demand determined through the PROC is given in Equation 4.14.

$$y = x \sum_{\forall \tau_k \in \tau : k \leq i} \frac{C_k}{T_k} \tag{4.14}$$

Now consider any deadline that lies in between T_i and H and then compare its *y-coordinate* to show that the demand of such deadlines lies below or on the line as the one given in Equation 4.14. Assume $t \in M(i,j) = \{n_j T_j : \left\lceil \frac{T_i}{T_j} \right\rceil < n_j < \left\lfloor \frac{\mathrm{H}}{T_j} \right\rfloor\}$. $M(i,j)$ describes the set of all the deadlines between T_i and H. As such $n_j T_j$ will be a deadline in an interval (A,B) and its demand is $\sum_{\forall \tau_k \in \tau : k \leq i} \left\lfloor \frac{n_j T_j}{T_k} \right\rfloor C_k$ (Equation 4.7). The deadline $n_j T_j$ is put in the *x-coordinate* of Equation 4.14 to get the resulting demand of PROC and compared against $\sum_{\forall \tau_k \in \tau : k \leq i} \left\lfloor \frac{n_j T_j}{T_k} \right\rfloor C_k$ as given in Equation 4.15.

$$y = n_j T_j \sum_{\forall \tau_k \in \tau : k \leq i} \frac{C_k}{T_k} \geq \sum_{\forall \tau_k \in \tau : k \leq i} \left\lfloor \frac{n_j T_j}{T_k} \right\rfloor C_k \tag{4.15}$$

$$\Leftarrow \quad \sum_{\forall \tau_k \in \tau : k \leq i} \frac{n_j T_j}{T_k} C_k \geq \sum_{\forall \tau_k \in \tau : k \leq i} \left\lfloor \frac{n_j T_j}{T_k} \right\rfloor C_k$$

$$\Leftarrow \quad \frac{n_j T_j}{T_k} \geq \left\lfloor \frac{n_j T_j}{T_k} \right\rfloor \tag{4.16}$$

Equation 4.16 is always true as $x \geq \lfloor x \rfloor, \forall x$. Thus, the curve of DBF is always below or on the line for all the deadlines in any interval (A,B).

As the demand of PROC for all deadlines in the interval $[A,B]$ (case a and b) are greater than or equal to DBF, the lemma follows. □

4.1.4 Improvements in Minimum Idle interval (Static Sleep Interval)

Definition 21 (Minimum Idle Interval or Static Sleep Interval[1])**.** *The minimum idle interval or static sleep interval is the bound on the length of the shortest possible idle interval in the schedule.*

All the idle intervals in the schedule are greater than or equal to this bound. The minimum bound on the idle period in the schedule is an important metric in procrastination scheduling to select the most efficient sleep state S_n offline. To reduce the online complexity, a processor can choose its sleep state based on this interval that minimises the energy consumption in the sleep state while respecting the temporal constraint. The system increase the chance to use better sleep

[1]The minimum idle interval and static sleep interval are used interchangeably throughout this thesis.

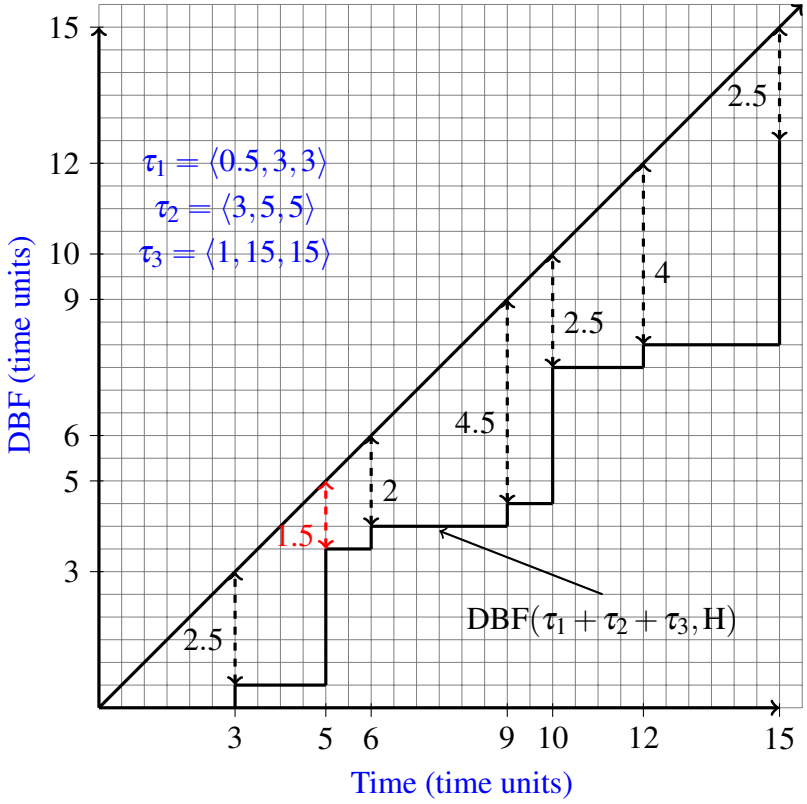


Figure 4.6: Static sleep interval with tasks $\tau_1 = \langle 0.5, 3, 3\rangle$, $\tau_2 = \langle 3, 5, 5\rangle$ and $\tau_3 = \langle 1, 15, 15\rangle$

states (multiple sleep states [AP11]) by maximising the minimum bound on the idle period, which in turn reduces the energy consumption. Therefore, it is also important to maximise this bound.

The state-of-the-art algorithms compute the minimum idle interval in slightly different ways. The minimum idle interval Q_{min} computed by LC-EDF [LRK03] is given by Theorem 22. Similarly, Jejurikar et al. [JPG04] also identified a static sleep interval Z_{min} given in Theorem 23. They proved in their work that $Z_{min} \geq Q_{min}$. To compare χ_{min}, Z_{min} and Q_{min}, consider a task-set composed of three tasks with the following parameters: $\tau_1 = \langle 0.5, 3, 3\rangle$, $\tau_2 = \langle 3, 5, 5\rangle$ and $\tau_3 = \langle 1, 15, 15\rangle$. The DBF of the given task-set is illustrated in Figure 4.6. For this example, $\chi_{min} = 1.5$ time units, $Z_{min} = 1.167$ and $Q_{min} = 0.5$, consequently, $\chi_{min} \geq Z_{min} \geq Q_{min}$.

Theorem 22 ([LRK03]). *Any idle period in the* LC-EDF *algorithm [LRK03] is greater than or equal to* $Q_{min} = \min_{k=1}^{\ell} \left\{ Q_k = (1 - \sum_{i=1}^{\ell} \frac{C_i}{T_i}) T_k \right\}$.

Theorem 23 ([JPG04]). *The minimum idle period* (Z_{min}) *identified by* PROC *algorithm [JPG04] is given as* $Z_{min} = \min_{i=1}^{\ell} \{ Z_i = (1 - \sum_{k=1}^{i} \frac{C_k}{T_k}) T_i \}$.

The objective of this section is to show that the static sleep interval determined through DBFP is greater than or equal to Z_{min}. Lemma 24 proves that $\chi_{min} \geq Z_{min}$.

Lemma 24. *Given tasks in τ are ordered in a non-decreasing order of their relative deadlines, the minimum idle period guaranteed by the* DBFP *scheme is always greater than or equal to the minimum idle period guaranteed by* PROC.

Proof. Assume all the tasks are sorted in a non-decreasing order of their periods/deadlines. The minimum idle interval Z_{min} determined through PROC algorithm is equal to $Z_{min} = \min_{\forall \tau_i \in \tau} Z_i$. Similarly, the minimum idle interval guaranteed by the DBFP scheme $\chi_{min} = \min_{\forall \tau_i \in \tau} \chi_i$. Formally, Lemma 24 can be represented with the inequality given in Equation 4.17.

$$\min_{\forall \tau_j \in \tau, \forall t \in M(i,j)} \left\{ t - \sum_{\forall \tau_k \in \tau : k \leq i} \left\lfloor \frac{t}{T_k} \right\rfloor C_k \right\} \geq \min_{\forall \tau_i \in \tau} \left(1 - \sum_{\forall \tau_k \in \tau : k \leq i} \frac{C_k}{T_k} \right) T_i \tag{4.17}$$

$$\text{where} \qquad M(i,j) = \left\{ n_j T_j : \left\lceil \frac{\min_{\forall \tau_i \in \tau} T_i}{T_j} \right\rceil \leq n_j \leq \left\lfloor \frac{\mathrm{H}}{T_j} \right\rfloor \right\}$$

In order to prove this inequality, we have to show that at any time $t \leq \mathrm{H}$ the demand of the given task-set will remain below or will be equal to the demand computed by the PROC method, where $t \in M(i,j) = \left\{ n_j T_j : 1 \leq n_j \leq \left\lfloor \frac{\mathrm{H}}{T_j} \right\rfloor \right\}$. In other words, all the deadlines are checked for the difference. To interpolate the demand computed by PROC, the demand on the neighbouring deadlines of a task are connected with a straight line. Finally, the demand beyond the last period is extended with a line having a slope equal to the system utilisation. To illustrate this consider the example given in Figure 4.4. Figure 4.7 shows the demand of this task-set with both DBF and PROC. The demand of the tasks in PROC is computed on their first deadline are represented with A, B and C points. Points A and B are connected with a straight line to compare against all the deadlines in the DBF happening in between these two points. Similarly, B and C points are connected, and the demand beyond C for procrastination algorithm is extended with a line having a slope equal to the utilisation of the task-set.

Since the DBF needs to be checked at more instances than A, B and C in the procrastination algorithm, we need to consider constraints. The objective is to find the minimal distance between the supply bound function SBF and the demand. For all intervals between successive points A, B and C, it is true that the smallest gap between the SBF and the demand within these intervals can be found either of the two delimiting points (for example, for interval $[A, B]$, the smallest gap can either occur at A or B). Since $U \leq 1$, it is evident that beyond the largest period, the largest gap can be found at the largest period point. In order to demonstrate that the gap computed with the DBF based value is always greater than or equal to that of PROC it is sufficient to show that the DBF test dominates in the following cases.

a) First deadline of every task

b) The demand computed by the DBF is always smaller than the connecting lines of the first deadline of all tasks.

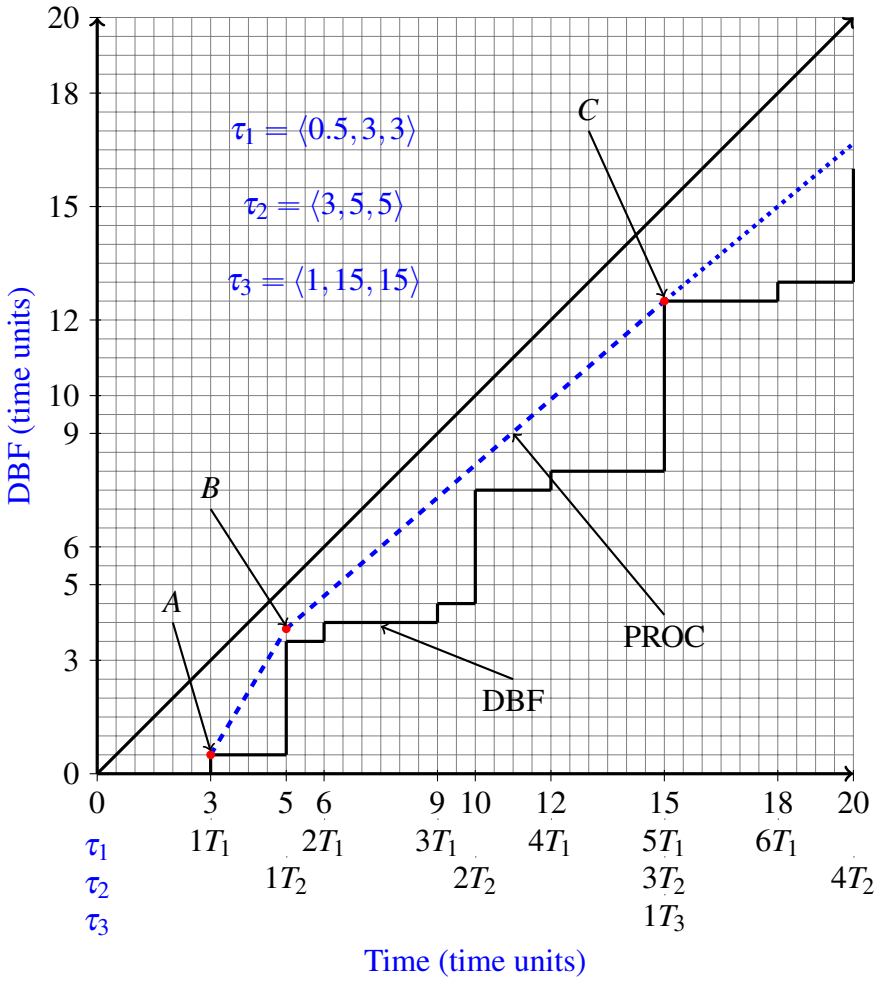


Figure 4.7: DBF vs SRA

Case a) To get the first deadline of every task, we set $t = T_i$ in Equation 4.17,

$$
\begin{aligned}
& \min_{\forall \tau_i \in \tau, t = T_i} \left\{ T_i - \sum_{\forall \tau_k \in \tau} \left\lfloor \frac{T_i}{T_k} \right\rfloor C_k \right\} \geq \min_{\forall \tau_i \in \tau} \left\{ T_i - \sum_{\forall \tau_k \in \tau : k \leq i} \left(\frac{T_i}{T_k} \right) C_k \right\} \\
\Leftarrow \quad & - \sum_{\forall \tau_k \in \tau} \left\lfloor \frac{T_i}{T_k} \right\rfloor C_k \geq - \sum_{\forall \tau_k \in \tau : k \leq i} \left(\frac{T_i}{T_k} \right) C_k \\
\Leftarrow \quad & \sum_{\forall \tau_k \in \tau : k > i} \left\lfloor \frac{T_i}{T_k} \right\rfloor C_k + \sum_{\forall \tau_k \in \tau : k \leq i} \left\lfloor \frac{T_i}{T_k} \right\rfloor C_k \leq \sum_{\forall \tau_k \in \tau : k \leq i} \left(\frac{T_i}{T_k} \right) C_k \\
\Leftarrow \quad & 0 + \cancel{\sum_{\forall \tau_k \in \tau : k \leq i}} \left\lfloor \frac{T_i}{T_k} \right\rfloor \cancel{C_k} \leq \cancel{\sum_{\forall \tau_k \in \tau : k \leq i}} \left(\frac{T_i}{T_k} \right) \cancel{C_k} \\
& \qquad \text{as } \left\lfloor \frac{T_i}{T_k} \right\rfloor = 0, \forall\, T_i < T_k \\
\Leftarrow \quad & \left\lfloor \frac{T_i}{T_k} \right\rfloor \leq \left(\frac{T_i}{T_k} \right)
\end{aligned}
\tag{4.18}
$$

Equation 4.18 shows that Equation 4.17 holds for the first case.

Case b) Suppose that τ_{i-1} is the task preceding τ_i. This case checks Equation 4.17 for all the deadlines that exist between T_{i-1} and T_i, i.e.,

$$t \in M(i,j) = \left\{ n_j T_j : \left\lceil \frac{T_{i-1}}{T_j} \right\rceil < n_j < \left\lfloor \frac{T_i}{T_j} \right\rfloor, \forall \tau_i \in \tau \right\}.$$

Equation 4.19 is the general form of the equation of a line between two points (x_1, y_1) and (x_2, y_2). In the representation of the DBF, the *x-axis* and *y-axis* represent the time and the demand, respectively. Let us assume the coordinates at the deadlines of τ_{i-1} and τ_i are $(x_1, y_1) = \left(T_{i-1}, \sum_{\forall \tau_k \in \tau : k \le i-1} (\frac{C_k}{T_k}) T_{i-1} \right)$ and $(x_2, y_2) = \left(T_i, \sum_{\forall \tau_k \in \tau : k \le i} (\frac{C_k}{T_k}) T_i \right)$, respectively. To find the equation between these two points, substitute their coordinates into Equation 4.19 correspondingly as shown in Equation 4.20.

$$y = \frac{y_2 - y_1}{x_2 - x_1}(x - x_1) + y_1 \tag{4.19}$$

$$= \frac{\left(\sum_{\forall \tau_k \in \tau : k \le i} \frac{C_k}{T_k} T_i - \sum_{\forall \tau_k \in \tau : k \le i-1} \frac{C_k}{T_k} T_{i-1} \right)}{T_i - T_{i-1}} (x - T_{i-1}) + \sum_{\forall \tau_k \in \tau : k \le i-1} \frac{C_k}{T_k} T_{i-1} \tag{4.20}$$

$$= \frac{\left(\left(\sum_{\forall \tau_k \in \tau : k \le i-1} \frac{C_k}{T_k} + \frac{C_i}{T_i} \right) T_i - \sum_{\forall \tau_k \in \tau : k \le i-1} \frac{C_k}{T_k} T_{i-1} \right)}{T_i - T_{i-1}} (x - T_{i-1}) + \sum_{\forall \tau_k \in \tau : k \le i-1} \frac{C_k}{T_k} T_{i-1}$$

$$= \left(\sum_{\forall \tau_k \in \tau : k \le i-1} \frac{C_k}{T_k} + \frac{C_i}{T_i - T_{i-1}} \right) (x - T_{i-1}) + \sum_{\forall \tau_k \in \tau : k \le i-1} \frac{C_k}{T_k} T_{i-1}$$

$$= \left(\sum_{\forall \tau_k \in \tau : k \le i-1} \frac{C_k}{T_k} + \frac{C_i}{T_i - T_{i-1}} \right) x - \frac{C_i T_{i-1}}{T_i - T_{i-1}} \tag{4.21}$$

Now consider any deadline that lies in between the deadlines of τ_{i-1} and τ_i (i.e., between (x_1, y_1) and (x_2, y_2)). It is shown that the demand (*y-coordinate*) of such deadlines will be below or on the line given in Equation 4.21. To this end, let us say that any deadline between (x_1, y_1) and (x_2, y_2) is specified by $(x_m, y_m) \stackrel{\text{def}}{=} \left(n_j T_j, \sum_{\forall \tau_k \in \tau, t = n_j T_j} \left\lfloor \frac{n_j T_j}{T_k} \right\rfloor C_k \right)$. Substitute the *x-coordinate* of this selected point (x_m, y_m) into Equation 4.21 and compare the resulting value of the *y-coordinate* with its y_m. If it is greater than or equal to y_m then the DBF is below or on the line. The resulting expression is shown in Equation 4.22.

$$n_j T_j \left(\sum_{\forall \tau_k \in \tau : k \le i-1} \frac{C_k}{T_k} + \frac{C_i}{T_i - T_{i-1}} \right) - \frac{C_i T_{i-1}}{T_i - T_{i-1}} \ge \sum_{\forall \tau_k \in \tau, t = n_j T_j} \left\lfloor \frac{n_j T_j}{T_k} \right\rfloor C_k \tag{4.22}$$

Point (x_m, y_m) is in between T_{i-1} and T_i, therefore the factor $\sum_{\forall \tau_k \in \tau, t = n_j T_j} \left\lfloor \frac{n_j T_j}{T_k} \right\rfloor C_k$ can be rewritten as $\sum_{\forall \tau_k \in \tau : k \leq i-1} \left\lfloor \frac{n_j T_j}{T_k} \right\rfloor C_k$. Hence, it follows that

$$n_j T_j \left(\sum_{\forall \tau_k \in \tau : k \leq i-1} \frac{C_k}{T_k} + \frac{C_i}{T_i - T_{i-1}} \right) - \frac{C_i T_{i-1}}{T_i - T_{i-1}} \geq \sum_{\forall \tau_k \in \tau : k \leq i-1} \left\lfloor \frac{n_j T_j}{T_k} \right\rfloor C_k \tag{4.23}$$

$$\Longleftarrow \sum_{\forall \tau_k \in \tau : k \leq i-1} \frac{n_j T_j}{T_k} C_k + \frac{n_j T_j C_i}{T_i - T_{i-1}} - \frac{C_i T_{i-1}}{T_i - T_{i-1}} \geq \sum_{\forall \tau_k \in \tau : k \leq i-1} \left\lfloor \frac{n_j T_j}{T_k} \right\rfloor C_k$$

$$\Longleftarrow \sum_{\forall \tau_k \in \tau : k \leq i-1} \frac{n_j T_j}{T_k} C_k + \frac{C_i}{T_i - T_{i-1}} (n_j T_j - T_{i-1}) \geq \sum_{\forall \tau_k \in \tau : k \leq i-1} \left\lfloor \frac{n_j T_j}{T_k} \right\rfloor C_k \tag{4.24}$$

Obviously, $n_j T_j - T_{i-1}$ is greater than 0 as $n_j T_j > T_{i-1}$. Hence, all the deadlines such that

$$\forall \tau_k \in \tau, t \in M(i,k) = \{T_{i-1} \leq n_k T_k \leq T_i\}$$

lie below the line represented by Equation 4.21.

As the difference computed between the supply SBF and the demand for all deadlines (case a and b) are greater than or equal to their corresponding difference computed through the PROC algorithm, therefore, the lemma follows. □

4.1.5 Extending DBFP to the Constrained Deadline Task Model and its Optimality

The state-of-the-art procrastination algorithms [LRK03, JPG04] cannot be effectively extended to the constrained deadline task model ($D_i \leq T_i$) in their current form. The use of densities (i.e., $\frac{D_i}{T_i}$) will degrade their performance substantially. One of the advantages of the DBFP approach is its straight forward extension to this model. For the constrained deadline task model, Equation 4.9 can be rewritten in its general form by replacing $\text{DBF}_I(\tau_k, t)$ with $\text{DBF}(\tau_k, t)$ as given Equation 4.25, where the set $M(i,j)$ is substituted by

$$M_1(i,j) = \left\{ n_j D_j : \left\lfloor \frac{T_i - D_i}{T_j} \right\rfloor + 1 \leq n_j \leq \left\lfloor \frac{\text{H} - D_i}{T_j} \right\rfloor + 1 \right\}.$$

Similarly, the minimum idle interval for constrained deadline task model is given in Equation 4.26, where

$$M_2(i,j) = \left\{ n_j D_j : 1 \leq n_j \leq \left\lfloor \frac{\text{H}}{T_j} \right\rfloor \right\}.$$

$$\chi_i = \min_{\forall \tau_j \in \tau : j \leq i, \forall t \in M_1(i,j)} \left\{ t - \sum_{\forall \tau_k \in \tau : k \leq i} \text{DBF}(\tau_k, t) \right\} \tag{4.25}$$

$$\chi_{min} = \min_{\forall \tau_j \in \tau, \forall t \in M_2(i,j)} \left\{ t - \sum_{\forall \tau_k \in \tau : k \leq i} \text{DBF}(\tau_k, t) \right\} \tag{4.26}$$

Equation 4.26 aligns with the results provided by Chetto et al. [CC89, Sil99, Che08] on the slack time estimation to schedule the aperiodic task in the presence of periodic task-set.

The optimality of the procrastination interval of each task χ_i and the minimum idle interval χ_{min} (i.e., *maximal without violating any temporal constraint*) can be easily inferred through the results borrowed from the sensitivity analysis framework [GH09] or Chetto et al. [Che08]. Nevertheless, short proofs based on the DBF-based analysis are provided here for completeness. The interested readers are directed to the technical report [AYP13] for a formal proof using the sensitivity analysis or the work of Chetto et al. [CC89, Sil99, Che08].

Theorem 25. *The minimum idle interval χ_{min} determined by the* DBFP *approach for a constrained deadline task-set is optimal.*

Proof. Since sleep transitions are taken in idle intervals, only the critical instant has to be considered. Lemma 24 demonstrates that $\chi_{min} \geq Z_{min}$ and the chosen sleep interval is safe i.e., no deadline is missed in the resulting schedule. Hence, χ_{min} is not optimistic. At the same time the DBF based analysis demonstrates a concrete scheduling scenario. Thus, χ_{min} is clearly not pessimistic, as the derived value by the DBFP approach can actually occur. Since the derived sleep interval χ_{min} is at the same time neither pessimistic nor optimistic, it is safe and optimal, thus the theorem follows. □

Theorem 26. *The procrastination interval determined by the* DBFP *approach for individual task χ_i in a constrained deadline task model is optimal.*

Proof. In this case, instead of considering the whole task-set τ, only the set of tasks with a priority greater than or equal the current one are taken into account. Theorem 19 shows that it is sufficient to consider only the set of deadlines after the first deadline of the task under analysis, including the first deadline of the task as well. Afterwards, the proof follows the same principle as that of Theorem 25 where the given procrastination interval has been shown neither optimistic nor pessimistic. □

4.2 Alternative Real-Time Race-To-Halt Algorithms

The procrastination scheduling in general can achieve long sleep intervals to save energy. The need for external hardware is the major limitation that restrict its practical value. One of the way to circumvent this issue is to determine the safe sleep interval such that it avoids any deadline misses when used online to initiate a sleep state. This bound ensures that no matter how many tasks arrive during the sleep state, the system will still meet all deadlines. The minimum idle interval or static sleep interval χ_{min} is the optimal safe bound on a sleep state that respects all the temporal constraints of EDF and can be used for such purpose. The following lemmas demonstrate that the schedulability of the system is preserved when the processor initiates a sleep state for static sleep interval χ_{min}.

Lemma 27 ([RGR08]). *A synchronous periodic task-set τ is schedulable under* EDF *if and only if, $\forall L \leq L^*$, $\mathrm{DBF}(L) \leq L$, where L is an absolute deadline and L^* is the first idle time in the schedule.*

Lemma 28. *Initiating a sleep state for the static sleep interval χ_{min} does not violate the* EDF *schedule if and only if*

$$\forall L \leq L^*, \qquad \mathrm{DBF}(L) + \chi_{min} \leq L$$

Where L is an absolute deadline and L^* is the first idle time in the schedule.

Proof. Let t be a time instant when processor transitions into a sleep state. To maximise the system workload, assume a critical instant at time t where all the tasks release their jobs and arrive as soon as possible. The static sleep interval χ_{min} can be modelled as the highest priority job $j_{h,p}$. In an EDF scheduled system it is equivalent to a job with a deadline equal to the shortest absolute deadline of any job, i.e., $d_{h,p} = t + \min_{\forall \tau_i \in \tau} D_i$. The job $j_{h,p}$ is co-scheduled with τ at time t. Assume, $\mathrm{DBF}^*(L)$ is the new demand and it is equal to $\mathrm{DBF}(L) + \chi_{min}$, i.e.,increased over $\mathrm{DBF}(L)$ by χ_{min}. The definition of $\chi_{min} = \min_{\forall \tau_i \tau} \chi_i$ can be rewritten as $\chi_{min} = \min_{\forall L \leq L^*} (L - \mathrm{DBF}(L))$. The new definition of χ_{min} implies that $\mathrm{DBF}^*(L) = \mathrm{DBF}(L) + \min_{\forall L \leq L^*} (L - \mathrm{DBF}(L))$ and it will not cross the supply bound function L, i.e., $\mathrm{DBF}^*(L) \leq L$. It follows from Lemma 27 and $\mathrm{DBF}^*(L) \leq L$ that the schedulability of the system is always preserved. □

The set of real-time race-to-halt algorithms presented in this section collects the available resources (slack) and as the size of such resources becomes equal to or greater than the predefined sleep interval, the processor transitions into a sleep state. Contrary, to procrastination scheduling algorithm, in this case the sleep state is fixed and will not be extended during the sleep mode. Three different energy management algorithms (enhanced race-to-halt algorithm, improved race-to-halt algorithm and light-weight race-to-halt algorithm) are proposed to increase the energy efficiency of embedded systems using alternative race-to-halt strategy followed by a sleep state.

4.2.1 Enhanced Race-To-Halt Algorithm (ERTH)

It is a server based techniques based on RBED framework [BBLB03]. The RBED framework allows temporal isolation between different task types (RT and BE tasks). This algorithm considers both RT and BE tasks. The execution slack and static slack are managed explicitly in ERTH. Nevertheless, the effect of the sporadic slack is considered implicitly. The slack management algorithm presented in Section 3.1.5 is used to collate the execution slack. In this slack management algorithm, a second method is used in the slack preservation phase, i.e., on every scheduling event, if the deadline of the execution slack is less than or equal to the deadline of the job to be scheduled then the available execution slack can be added to the job's budget. However, all the energy management algorithms proposed in this section do not depend on our slack management algorithm presented in Algorithm 1. Any existing slack management algorithm can be integrated with minimal effort into these algorithms. Nevertheless, the proposed slack management algorithm has low overhead (spatial/temporal) that makes it an attractive alternative. In the proposed ERTH algorithm, the state of the processor is divided into three types, 1) a processor is idle, 2) a

Algorithm 2 Enhanced Race-To-Halt Algorithm (ERTH)

1: **Offline**
2: Compute χ_{min}
3: Find most efficient sleep state $\S_n$ for χ_{min},
$\forall$ Sleep States $N : \chi_{min} \geq bet_n$,
Minimise $\{(\chi_{min} \times P_n) + (tsw_n \times (P_A - P_n))\}$
4: Let σ be the sleep interval such that $\sigma = \chi_{min} - tw_n$

5: **Online**
6: **if** (System is Idle) **then**
7: Manage Slack (χ_{min})
8: $\varpi = \sigma$
9: Mask-record interrupts and initiate Sleep
10: **else if** (Get Slack($j_{i,k}$)$\geq \chi_{min}$) **then**
11: **if** (RT Task) **then**
12: Manage Slack(χ_{min})
13: $\varpi = \sigma$
14: Mask-record interrupts and initiate Sleep
15: **else if** (BE Task) **then**
16: Compute φ
17: Manage Slack(φ)
18: Set Sleep Time(φ);
19: **end if**
20: **else**
21: Race-To-Halt
22: **end if**

23: **When Timer Expires**
24: Unmask interrupts
25: **if** (Interrupts) **then**
26: Service the interrupts (Schedule the tasks arrived during sleep state)
27: **else**
28: $\varpi = \sigma$
29: Mask-record interrupts and initiate a sleep state
30: **end if**

processor is executing the RT tasks or 3) a processor is executing the BE tasks. ERTH associates three different principles corresponding to each state of the processor. These principles consider the state of the processor (i.e., either the processor is idle or executing RT/BE task) and the capacity of the available slack in the system to transition the processor into a sleep state. This algorithm does not initiate a sleep state for less than the static sleep interval χ_{min} to minimise the transition overheads. The complete pseudo code of ERTH is presented in Algorithm 2. The commons subroutines (such as Manage Slack, Get Slack, Set Sleep Time and Get Next Release Time) shared with other algorithms (improved race-to-halt algorithm and light-weight race-to-halt algorithm) are given in Algorithm 3.

Algorithm 3 Common Routines for ERTH, IRTH and LWRTH

1: **Set Sleep Time(η)**
2: $\forall$ Sleep States N:$\eta \geq bet_n$
Minimise $\{(\eta \times P_n) + (tsw_n \times (P_A - P_n))\}$
3: $\varpi = \eta - tw_n$
4: Mask-record interrupts and initiate sleep state

5: **Get Slack($j_{i,k}$)**
6: **if** $(d_{i,k} \geq S_e^{dl})$ **then**
7: return S_e^{sz}
8: **else**
9: return 0
10: **end if**

11: **Manage Slack(η)**
12: **if** $(\eta \leq S_e^{sz})$ **then**
13: $S_e^{sz} -= \eta$
14: **else**
15: $S_e^{sz} = 0$
16: $S_e^{dl} = 0$
17: **end if**

18: **Get Next Release Time(γ)**
19: $\forall \tau_i \in \tau$
20: return min$\{\gamma_i\}$

4.2.1.1 Principle 1 [Executing RT Tasks]:

The principle one applies on the RT (SRT or HRT) task type. If any job of a RT task is at the head of the ready queue having deadline greater than or equal to the execution slack and $S_e^{sz} \geq \chi_{min}$, a timer ϖ is initialised with the static sleep interval minus the wake-up transition-overhead time (i.e., $\varpi = \chi_{min} - tw_n$) and the processor transitions into a sleep state until the timer expires. In case the available execution slack size S_e^{sz} is less than χ_{min}, it is added to the budget of the job. The processor executes the job at full speed with an expectation of collecting more slack in future sufficient enough (i.e., greater than or equal to χ_{min}) to initiate a sleep state.

Theorem 29. *If the next job $j_{i,k}$ to execute in the ready queue at time instant t is of type RT (HRT or SRT), while the execution slack has a size greater than or equal to the static sleep interval $(S_e^{sz} \geq \chi_{min})$ and the slack deadline is less than or equal to the absolute deadline $d_{i,k}$ of the RT job $j_{i,k}$ $\left(S_e^{dl} \leq d_{i,k}\right)$, then the processor can initiate a sleep state for a static sleep interval of χ_{min} without violating* EDF *schedulability.*

Proof. Assume, the available execution slack S_e at time t is modelled as a fake job $j_{f,k}$ with a budget and deadline equal to χ_{min} and S_e^{dl} respectively. The job $j_{f,k}$ is co-scheduled with τ. We need to prove $j_{f,k}$ is schedulable without causing a deadline miss in the schedule. Therefore, the potentially affected jobs of τ are split into two parts which are addressed separately: 1) Jobs not

released yet, 2) Jobs in the read queue.
Case 1 (Jobs not released yet): This can be proven by contradiction. Assume, $j_{f,k}$ is scheduled at time t and there is a synchronous arrival of jobs corresponding to each task not yet released, and any job misses its deadline. Lemma 28 states that all jobs released at critical instant can be delayed for an interval of χ_{min} without any deadline miss, which is a contradiction. Therefore, all the jobs not released yet can meet their respective deadlines.
Case 2 (Jobs in the ready queue): The principle 1 imposes a condition that a job $j_{f,k}$ (execution slack) has an earliest deadline when compared to the jobs in the ready queue at time t. Hence, the job $j_{f,k}$ can be scheduled first and will not affect any job in the ready queue.

As tasks in both cases do not violate the schedule, thus the theorem follows. □

4.2.1.2 Principle 2 [Executing BE Tasks]:

The principle 2 deals with the case, when the next job to execute at time t is of type BE, then Equation 4.27 is used to evaluate the length of a sleep interval. This equation (Equation 4.27) computes the maximum feasible sleep interval between time t and the deadline of the execution slack. It is the minimum of the available execution slack and the shortest gap ρ. The latter (ρ given in Equation 4.28) is computed as follows. The procrastination interval of a job $j_{i,k}$ (or maximum delay in $j_{i,k}$'s execution) at time t is the difference of its absolute deadline $d_{i,k}$ and $t+X$, (i.e., $d_{i,k}-t-X$), where X is the remaining execution time of jobs having priority greater than or equal to $j_{i,k}$ (including the execution time of $j_{i,k}$ itself). This procrastination interval of each job in an interval $[t, S_e^{dl}]$ having deadline $t \leq d_{i,k} \leq S_e^{dl}$ is computed and minimum of them gives a shortest gap ρ. The shortest gap ρ computed with Equation 4.28 assumes a synchronous release of all the jobs not released yet. This equation ensures that a schedule at time t can be delayed for ρ time units without violating any deadline in an interval $[t, S_e^{dl}]$. However, it does not guarantee about the schedulability of jobs at any time $t' > S_e^{dl}$. The shortest gap ρ may contain the processing time reserved for low priority jobs having deadlines greater than S_e^{dl}. The amount of execution slack available in the system gives us an upper bound on the sleep duration. To avoid more complex schedulability checks, it is assumed a sleep interval is always less than or equal to the available execution slack even if the available gap ρ is greater than the execution slack i.e., $\rho \geq S_e^{sz}$. Conversely, if $\rho < S_e^{sz}$, a sleep interval is obviously equal to the duration of ρ to ensure the schedulability of the higher priority jobs when compare to execution slack. Therefore, φ finds the minimum between the available execution slack and the shortest-gap identified by ρ ensuring the overall system's schedulability. Once a gap φ in the schedule is identified, the timer ϖ is set for an interval of $\varphi - tw_n$.

$$\varphi = \min\left(S_e^{sz}, \rho\right) \tag{4.27}$$

$$\text{Where,} \qquad \rho = \min_{k,m \in V(S_e^{dl})} \left\{ g_{k,m} - \sum_{j \in V(d_{k,m})} \left\lfloor \frac{g_{k,m}}{T_j} \right\rfloor \times C_j \right\} \tag{4.28}$$

$$g_{k,m} = d_{k,m} - t \tag{4.29}$$

$$\mathbf{V}(\mathbf{x}) = \{i : r_{i,k} \geq t \wedge d_{i,k} \leq x\} \tag{4.30}$$

This approach of computing the sleep interval through *Equation* 4.28 has the following advantage. The sleep interval estimated φ is always greater than or equal to χ_{min} (i.e., $\varphi \geq \chi_{min}$) as it is assumed that φ is computed when $S_e^{sz} \geq \chi_{min}$. It is especially useful, when χ_{min} is very small (i.e., high system utilisation). Let us demonstrate with the help of an example that $\varphi \geq \chi_{min}$.

Example 4. *Assume a task-set is composed of four tasks* $\tau_1 = \langle 2,8,8\rangle, \tau_2 = \langle 1,9,9\rangle, \tau_3 = \langle 5,12,12\rangle$ *and* $\tau_4 = \langle 3,14,14\rangle$. τ_4 *is a BE task while all others are RT tasks. The utilisation of the task-set is* 0.9921 *and* $\chi_{min} = 1$ *time unit. Figure 4.8 demonstrate a concrete schedule. Assume,* τ_3 *and* τ_4 *release their jobs* $j_{3,p}$ *and* $j_{4,q}$ *at time instant* 3. $j_{3,p}$ *executes in an interval* $[3,5]$ *and generates an execution slack of* 3 *time units at time instant* 5. *The execution slack of size* 3 *can be store in the slack container with its deadline* 15. *At time instant* 5, $j_{4,q}$ *is ready to execute but the execution slack is sufficient enough to initiate a sleep state as* $S_e^{sz} > \chi_{min}$ *and* $d_{4,q} > S_e^{dl}$. *As the knowledge about the previous release times of tasks is not assumed, therefore, a worst-case situation is assessed by assuming a synchronous release of all tasks not released yet. The procrastination interval of all the jobs having deadlines* $5 \leq d_{i,k} \leq 15$ *is computed. In this example, this interval for a job* $j_{1,x}$ *is equal to* $d_{1,x} - (t+X) = 13 - (5+2) = 6$, *similarly, a job* $j_{2,y}$ *has procrastination interval equal to* $d_{2,y} - (t+X) = 14 - (5+2+1) = 6$. *Hence,* $\rho = \min\{6,6\} = 6$ *and the sleep interval* $\varphi = \min\{3,6\} = 3$, *which is greater than the static sleep interval* χ_{min}.

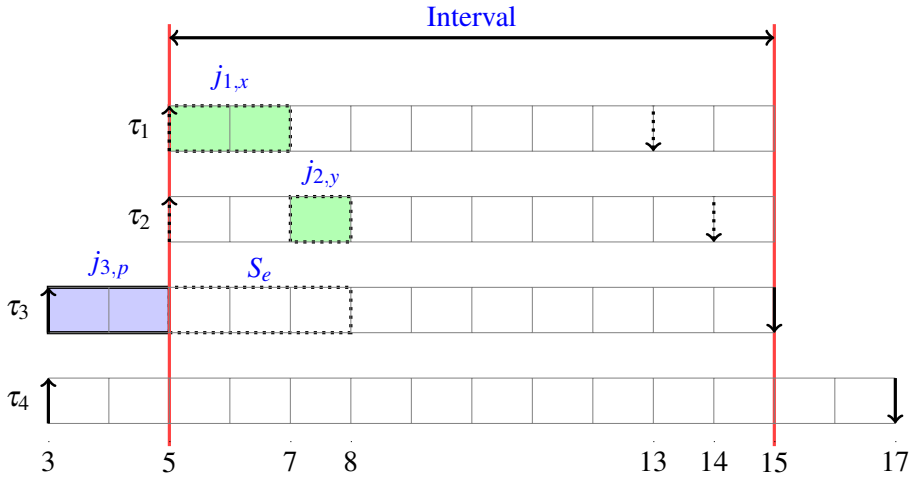


Figure 4.8: Example to illustrate that $\varphi \geq \chi_{min}$ with a task-set composed of $\tau_1 = \langle 2,8,8\rangle$, $\tau_2 = \langle 1,9,9\rangle$, $\tau_3 = \langle 5,12,12\rangle$, $\tau_4 = \langle 3,14,14\rangle$ and $\chi_{min} = 1$

Another way to visualise the working of Equation 4.28 is through DBF. Assume a synchronous arrival of all higher priority tasks at time instant t not released yet and compute the demand bound function including the demand of the tasks in the ready queue within an interval of $\left[t, S_e^{dl}\right]$. The

minimum gap ρ is the shortest distance between the demand and the supply bound function (SBF). One can also use Equation 4.27 to compute the sleep interval for principle 1. However, it is avoided due to two reasons. Firstly, to reduce the complexity of the scheduler. Secondly, the extra time taken in the computation of Equation 4.27 can be borrowed from the BE task's budget. In this work, it is assumed that computation time of Equation 4.27 is negligible.

Theorem 30. *If the job to execute in the ready queue is of BE type and the execution slack is greater than or equal to the static sleep interval* $(S_e^{sz} \geq \chi_{min})$ *with a deadline less than or equal to the absolute deadline of the BE job* $\left(S_e^{dl} \leq d_{i,k}\right)$*, then the processor can initiate a sleep state for* φ *without violating any deadlines under* EDF.

Proof. In this case the available sleep interval is not defined offline, rather computed online. To prove that a processor can initiate a sleep state for an interval of φ without violating any deadline, τ is segregated into four parts. The schedulability of each part is proven individually.

1) $\forall j_{i,k}$ has not yet been released and $d_{i,k} \leq S_e^{dl}$

2) $\forall j_{i,k}$ has not been released and $d_{i,k} > S_e^{dl}$

3) $\forall j_{i,k}$ is in the ready queue

4) $\forall j_{i,k}$ has already completed

Let ρ define the maximum available interval by which the higher priority jobs can be delayed at the current instant t. ρ is computed by Equation 4.28 considering each deadline within an interval of $\left[t, S_e^{dl}\right]$. In principle, it performs a limited demand-bound analysis for the defined interval to calculate the shortest gap ρ. Since there is a possibility to get a delay larger than the available S_e^{sz}, Equation 4.27 guarantees that a processor is not delayed more than available execution slack. Let us model a sleep interval as a fake job $j_{f,k}$ with a deadline equal to S_e^{dl}. Equation 4.28 implies scheduling $j_{f,k}$ for not more than ρ does not affect the schedule of any job $j_{i,k}$ that is not yet released and has a $d_{i,k} \leq S_e^{dl}$. Moreover, the execution of $j_{f,k}$ is restricted to S_e^{sz} with Equation 4.27. This ensures that any job $j_{i,k}$ not released yet with $d_{i,k} > S_e^{dl}$ will not be affected. Equation 4.30 excludes all jobs $j_{i,k}$ such that $d_{i,k} \leq t$. Similar to Theorem 29, the schedulability of all jobs in the ready queue is not affected as well, as they have a deadline later than that of $j_{f,k}$. Any jobs already completed, are obviously unaffected. As none of the task in τ miss their deadline, hence the theorem holds. □

4.2.1.3 Principle 3 [System is Idle]:

In the idle mode, a processor initiates a sleep state for a duration of $\chi_{min} - tw_n$. As χ_{min} is computed offline considering the worst-case scenario in the schedule, therefore, initiating a sleep state for $\chi_{min} - tw_n$ during idle mode will not affect the schedulability in any circumstance. The processor is not allowed to prolong the sleep state beyond the static sleep interval to preserve the schedulability. While, φ can also be used to increase the sleep interval but it will also substantially increase the complexity of the algorithm.

Theorem 31. *If the system is idle and the available execution slack size S_e^{sz} that may be less than the static sleep interval χ_{min} is consumed first, then the processor can initiate a sleep state for the static sleep interval χ_{min} without violating the* EDF *schedulability.*

Proof. The proof of the Theorem 31 follows the same reasoning of Lemma 28. □

Practical Issues: Now we discuss about some practical issues relevant to this algorithm. Let t_{sleep} is the time interval selected for the processor to initiate a sleep state through any of the principles explained above. The timer value is set to $\varpi = t_{sleep} - tw_n$. The sleep state initiated through any of these principles restricts the processor to wake up until the timer expires. It is assumed that all interrupts bar the timer interrupt are disabled on initiating a sleep state and re-enabled on the completion of the sleep. In many CPUs separate interrupt sources can be used for this. As usual with such disabled interrupts, events occurring during the sleep interval are to be flagged in the interrupt controller for processing after the interrupts are re-enabled.

Pessimism involved in ERTH**:** The ERTH algorithm is agnostic to the future release pattern of the tasks. It assumes a critical instant on each sleep transition. As such, each sleep state interval is estimated assuming a synchronous release of all higher priority tasks. For instant, in calculation of φ, the scheduler assumes synchronous release of all those tasks having deadlines earlier than the current deadline of the execution slack. Similarly, in principle 3 (idle mode), a synchronous release of all tasks is assumed at the instant of sleep transition. The critical instant occurs rarely, if ever, in reality. However, ERTH has to consider this pessimistic condition to guarantee the schedulability of the HRT tasks. This pessimism results in a sub-optimal sleep interval.

4.2.2 Improved Race-To-Halt Algorithm (IRTH)

All the pessimism in ERTH can be reduced by knowing the future release information of the task-set. As sporadic task model is assumed, it is not possible to predict exact future releases of all tasks. In the sporadic task model, a new job of a task can only arrive after T_i. Therefore, the future release time can be approximated by storing the past release information. While, this method can partially reduce the pessimism due to the nature of the sporadic task model introduced in ERTH but cannot eliminate it entirely. The IRTH algorithm maintains an array of future release information γ with a size equal to the number of tasks ℓ in the task-set. On every job arrival, it updates its future release time γ_i by adding T_i in its current job release time $r_{i,k}$ (i.e., $\gamma_i = r_{i,k} + T_i$).

The pseudo code of the IRTH algorithm is presented in Algorithm 4. The three basic principles corresponding to the different states of the processor (idle, executing RT or BE) stay the same. However, the sleep interval estimated in principle 2 (executing BE task) and principle 3 (idle mode) improves over the ERTH algorithm. In idle mode, the scheduler finds the next earliest release γ_{next} in the future from its future release information array γ. This information assures that there is no release in an interval $[t,\ \gamma_{next})$, hence, a sleep interval can be extended from χ_{min} to $\chi_{min} + \gamma_{next} - t$ without violating any deadline.

Algorithm 4 Improved Race-To-Halt Algorithm (IRTH)

Offline
Compute χ_{min}
Find most efficient sleep state $\S_n$ for χ_{min}:
$\forall$ Sleep States $N : \chi_{min} \geq bet_n$
Minimise $\{(\chi_{min} \times P_n) + (tsw_n \times (P_A - P_n))\}$
Let σ be the sleep interval such that $\sigma = \chi_{min} - tw_n$

Online
if (System is Idle) **then**
    Manage Slack(χ_{min})
    γ_{next} = Get Next Release Time(γ)
    Set Sleep Time($\gamma_{next} - t + \chi_{min}$)
else if ($GetSlack(j_{i,k}) \geq \chi_{min}$) **then**
    if (RT Task) **then**
        Manage Slack(χ_{min})
        $Timer = \sigma$
        Mask-record interrupts and initiate a sleep state
    else if (BE Task) **then**
        Compute ω
        Manage Slack(ω)
        Set Sleep Time(ω)
    end if
else
    Race-To-Halt
end if

On release of τ_i
Update τ_i's next predicted arrival time in the future release array γ i.e., $\gamma_i = r_{i,k} + T_i$

When Timer Expires
Unmask interrupts
if (Interrupts) **then**
    Service the interrupts (schedule the tasks arrived during sleep state)
else
    $Timer = \sigma$
    Mask-record interrupts and initiate a sleep state
end if

Theorem 32. *If the processor is in idle mode at time instant t and the earliest possible releases of all tasks γ after t is available, then the processor can initiate a sleep state for a duration of $\chi_{min} + \gamma_{next} - t$, without violating any deadline under* EDF *scheduling algorithm.*

Proof. Consider γ_{next} is the next release of any task in τ. γ_{next} is assumed to be the critical instant that leads to the longest busy interval (assuming synchronous releases) in the schedule (though that may or may not occur at this point). As there are no releases in the interval between t and γ_{next}, the schedulability of the system is not affected, however, scheduler needs to check for the

schedulability of the task-set τ for any time $t' \geq \gamma_{next}$. The schedulability of the system for t' follows directly from Lemma 28. Hence the schedulability of the overall schedule will be preserved under this sleep condition. □

The sleep interval in principle 2 (executing BE task) can also be improved by exploiting the future release information. Equation 4.28 used in ERTH to compute the sleep interval in principle 2 is equivalent to the limited demand bound function. It assumes a critical instant at time t with a synchronous release of all jobs not released yet having deadline less than or equal to S_e^{dl}. The jobs of the tasks awaiting in the ready queue are not are included in this analysis because by the definition of principle 2, they have deadlines later than S_e^{dl}. The offset of $\gamma_i - t$ can be safely added to the first job of all those tasks having future releases and deadlines in an interval $[t, S_e^{dl}]$. The offset is only added if the γ_i corresponding to τ_i's instance is greater than t, otherwise, it is assumed to be 0. The offsets greater than 0 shifts the jobs deadlines accordingly – which may or may not shift the last job deadline of some tasks outside the interval $[t, S_e^{dl}]$. If some of the jobs deadlines move outside the interval, the demand requested by the system in the interval $[t, S_e^{dl}]$ is decreased. Which in turn increases the possibility to get larger sleep interval compared to the conservative approach used in ERTH. The schedulability of the system in principle 2 with this new amendment is proved in Theorem 33. The IRTH algorithm is promising but it also has an extra online overhead when compared to ERTH (online/offline overheads are discussed in Section 4.4.1).

Theorem 33. *If the task to execute in the ready queue is of BE type and the available execution slack is greater than or equal to the static sleep interval $S_e^{sz} \geq \chi_{min}$ with a deadline less than or equal to the BE job $S_e^{dl} \leq d_{i,k}$ and the earliest estimated future release of all tasks γ is known at the time of initiating a sleep state, then the sleep state can be initiated for a time interval ω without violating any deadline under the* EDF *scheduling algorithm.*

$$\omega = \min(S_e^{sz}, \vartheta) \tag{4.31}$$

$$\textit{Where}, \qquad \vartheta = \min_{k \in V(S_e^{dl})} \left\{ g_{k,m} - \sum_{j \in V(d_{k,m})} \left\lfloor \frac{g_{k,m} - \gamma_j}{T_j} \right\rfloor \times C_j \right\} \tag{4.32}$$

$$g_{k,m} = d_{k,m} - t \tag{4.33}$$

$$\mathbf{V}(\mathbf{x}) = i : r_{i,k} \geq t \wedge d_{i,k} \leq x \tag{4.34}$$

Proof. In order to prove the schedulability, τ is segragated into following six parts. The schedulability of each part is proved individually.

1) $\forall j_{i,k}$ already completed

2) $\forall j_{i,k}$ released earlier than t and has $S_e^{dl} > d_{i,k} > t$

3) $\forall j_{i,k}$ released earlier than t and has $d_{i,k} > S_e^{dl}$

4) $\forall j_{i,k}$ that will be released after t with an initial offset of γ_j and has $d_{i,k} \leq S_e^{dl}$

5) $\forall j_{i,k}$ that will be released after t with an initial offset of γ_j and has $d_{i,k} > S_e^{dl}$

6) $\forall j_{i,k}$ that will be released after S_e^{dl}

The term ϑ given by Equation 4.32 estimates the worst-case response-time of all jobs in an interval of $[t, S_e^{dl}]$ having release times in an interval of $[t, S_e^{dl})$ and deadlines in an interval of $(t, S_e^{dl}]$. Moreover, it returns the feasible interval for the sleep state at time instant t. Assume, a sleep state is modelled as a fake job $j_{f,k}$ with a deadline S_e^{dl} and budget S_e^{sz}. The sleep state cannot be initiated for more than S_e^{sz}, as it might jeopardise the schedulability of the low priority tasks. With this restriction, scheduling the fake job $j_{f,k}$ will not affect the jobs of category $1, 3, 5$ and 6 considering the EDF algorithm. Equation 4.34 eliminate all these jobs from the analysis. The principle 2 is only invoked when the task to execute in the ready queue has $d_{i,k} \geq S_e^{dl}$. This restriction is imposed by the slack management algorithm. Hence, jobs with a category of 2 do not exist and are thus removed with this restriction from the analysis. The schedulability of the jobs in the category 4 is individually ensured with the Equation 4.32. Equation 4.32 can produce a sleep interval larger than S_e^{sz}, but it is assumed that the budget of a fake sleep job $j_{f,k}$ is equal to S_e^{sz}. Therefore, its size is restricted to S_e^{sz} with the use of Equation 4.31. As none of the tasks in τ misses its deadline, the theorem holds. □

Algorithm 5 Light-Weight Race-To-Halt Algorithm (LWRTH)

```
1: Online
2: if (System is Idle) then
3:     γ_next =Get Next Release Time(γ)
4:     Set Sleep Time(γ_next − t + χ_min)
5: else
6:     Race-To-Halt
7: end if

8: On release of τ_i's instance
9: Update τ_i next predicted arrival time in the future release array γ i.e., γ_i = r_{i,k} + T_i

10: When Timer Expires
11: Unmask interrupts
12: if (Interrupts) then
13:     Service the interrupts (schedule the tasks arrived during sleep duration)
14: else
15:     Set Sleep Time(χ_min)
16:     Initiate a sleep state
17: end if
```

4.2.3 Light-Weight Race-To-Halt Algorithm (LWRTH)

The light-weight race-to-halt algorithm (LWRTH) proposed in this section only initiates a sleep state in the idle mode of the processor. A sleep state is initiated using principle 3 of the IRTH algorithm. This algorithm does not require any slack management scheme but still performs slightly better than ERTH and marginally lower than IRTH. LWRTH has lower online complexity when compared to IRTH but is inferior (performance-wise) against it at high utilisations. Nevertheless, LWRTH needs to maintain a list for the predicted future release information of tasks that adds extra online overhead and does not give us full control over the sleep transition which might not be helpful in a thermal-aware system design. For instance, if the system crosses the maximum temperature threshold in the middle of the execution phase, LWRTH has no way to slow down or stop its execution. Furthermore, LWRTH performs worse compare to ERTH when it comes to the number of pre-emptions for small task-set sizes (discussed in Section 4.4.4). The pseudo code of LWRTH is given in Algorithm 5.

4.3 Effect of Sleep-States on the Number of Pre-emptions

A pre-emption is counted when the execution of the job is suspended by a higher priority job. Assume, a synchronous releases of one higher and one lower priority jobs. In this scenario, the higher priority job executes first and the pre-emption is not counted as the lower priority job has not yet started its execution. The number of pre-emptions poses a substantial overhead (time/energy) on the running system. For instance, on resumption of a job the system has to pay the penalty to reload the cache content displaced by a pre-emption. Access to off-chip memory is generally very expensive when compared to on-chip caches or scratch-pad memory. Therefore, the system designer has to reserve time for pre-emption related delays, which in turn also decreases the useful system utilisation. A decrease in pre-emption count not only increases the usable system utilisation but also reduces the energy consumption.

The power management algorithms discussed previously in this chapter affect the release behaviour of the system and subsequently the pre-emption relations between jobs. In this section, the change in the behaviour of the system in terms of number of pre-emptions of jobs at runtime is investigated. Jobs releases during sleep interval give rise to two conflicting scenarios given below.

1. On one side, the execution of job releases during the sleep-state interval are postponed and constrained to a smaller window for execution. One could easily perceive that the number of pre-emptions will rise, as delaying the execution of jobs increase the likelihood of higher priority job releases. Therefore, in the presence of low priority jobs, the higher priority jobs cause more pre-emptions.

2. On the other side, the interrupts that occur throughout the sleep state interval are served on completion of the sleep interval. It is assumed a job release is triggered by an interrupt. Therefore, job releases during sleep interval are collated and scheduled after the sleep state

in priority order. Thus delaying new job arrivals and waiting for the higher priority job releases during the sleep interval decreases the number of pre-emptions.

These two conflicting scenarios indicate positive or negative changes in the number of pre-emptions. Considering the overhead of pre-emptions on the energy consumption and the system utilisation, it is indeed an important issue to resolve which approach performs better. If the number of pre-emptions decreases, the overall energy consumption actually decreases more than just the energy saved with sleep transition, as the overhead of pre-emptions is also reduced. Through extensive simulations, it is shown in the results that on average-case, sleep states have a positive effective on the number of pre-emptions. The number of pre-emptions of different proposed algorithms are analysed and compared against the state-of-the-art in results section.

4.4 Evaluation of CPU Power Management Algorithms

Initially, the complexity comparison of all the algorithms is presented in this section. An extensive study is performed to compare the proposed algorithms against the state-of-the-art on different parameters. This section only summarises and highlights the interesting results to increase the readability of the thesis. The detailed discussion of the results is presented in Section A.3 for interested readers to further explore the evaluation of the proposed algorithms.

4.4.1 Overhead Analysis

The complexity of the proposed algorithms is compared with LC-EDF, PROC and SRA, as they are with their use of dynamic priorities closest to this work. As it has been discussed in Section 4.1.1, all these algorithms (LC-EDF, PROC and SRA) initiate a sleep state in the idle mode. The LC-EDF algorithm has a smaller number of sleep states when compared to EDF as it combines several small idle intervals to initiate a sleep state for a long period of time. While in the sleep state, on each higher priority (shorter deadline) task arrival, the LC-EDF algorithm recomputes the new procrastination interval for that task, unless the schedule does not allow further procrastination. The online overhead of the LC-EDF algorithm depends on two main factors, 1) Number of times a sleep state is initiated, 2) The overhead of each sleep transition. The first factor depends on the total number of idle intervals in the schedule as LC-EDF initiates a sleep in idle mode. However, the overhead of each sleep transition depends on the task-set size. The complexity of each sleep transition in LC-EDF is $O(\ell^2)$.

The PROC method has an offline complexity of $O(\ell^2)$. The DBFP approach has an offline complexity of $O(\ell \times x)$, where $x = \sum_{\forall \tau_i \in \tau} \frac{\mathrm{H}}{T_i}$ is the number of jobs in the hyper-period H. The online complexity of the DBFP approach and the PROC method is the same and equals to $O(\ell)$. The SRA algorithm [JG05] reclaims the execution slack from the schedule and uses it to further procrastinate the sleep interval. In a nutshell, on every release of a task during the sleep interval, the scheduler computes the available execution slack and compares it with the offline computed procrastination

interval of that task. The maximum of these two values is considered while deciding on the reinitialisation of the timer. DBFP or PROC can be used in the offline phase of the SRA algorithm to compute the procrastination interval. The complexity to determine the available execution slack for a task is $O(\ell)$. As both PROC and DBFP has an online complexity of $O(\ell)$, therefore, combined with slack reclamation, the online complexity of the SRA algorithm is same as LC-EDF i.e., $O(\ell^2)$.

The alternative race-to-halt algorithms do not require any external hardware. The χ_{min} is used in all alternative race-to-halt algorithms and the offline complexity of its computation is same as presented for DBFP. The online complexity of ERTH can be divided into three different categories based on its three different principles.

- Firstly, if the sleep transition is initiated through principle 1, it requires just one comparison against the offline computed static sleep interval χ_{min}, i.e., $O(1)$.
- Secondly, a sleep states initiated with principle 2 require the computation of φ in order to obtain the maximum feasible sleep interval. The major overhead lies in the computation of ρ that could be obtained either offline or online. Offline Method: The interval for computing ρ offline is no more than the longest T_i in the task-set. Therefore, the maximum available gap can be computed offline for each deadline and sorted in an increasing order by time. The runtime overhead is to search the sorted array of maximum available gaps for each given interval, which can be done in $O(ln(p))$, where p is the number of intervals. Online Method: The online complexity to compute ρ depends on the number of jobs in an interval. The former method is used to compute ρ.
- Thirdly, in idle mode (principle 3), sleep state is initiated for χ_{min} interval without any check. Thus, sleep states initiated in idle mode do not have any online overhead.

Apart from its low complexity, the second advantage of ERTH is the existence of the fixed sleep-interval at the sleep-state initialisation instant. Once the processor initiates the sleep transition, no matter how many tasks arrive during the sleep mode, it will wake up after a defined limit (when the timer expires). The presented schedulability tests ensure that all jobs will meet their deadlines. This mechanism simplifies the system implementation and eliminates a need for external hardware to run the algorithm. Which in turn further reduce the complexity of the design, as external hardware requires extra communication overhead and increases integration issues.

The online overhead of IRTH is similarly divided into three categories. If the sleep state is initiated by a RT task (principle 1), its overhead is same as in ERTH principle 1, i.e., $O(1)$. However, in idle mode (principle 3), its complexity increases, as the algorithm has to search for the earliest possible future release in an array of γ. There are two ways to manage it. Firstly, a sorted array of γ can be stored and its first value can be used when the processor initiates a sleep transition. Thus the complexity of maintaining the array on each job arrival is $O(ln(\ell))$. However, when the processor initiates a sleep the overhead is low i.e., $O(1)$. Secondly, γ can be stored with respect to the task-ID and on each sleep invocation the algorithm traverses γ to find the minimum value. In this case complexity to update an array of γ on each job invocation is $O(1)$, however, each

sleep transition has a complexity of $O(\ell)$. It is observed that the number of sleep transitions are fewer when compared to the number of jobs invocations. Therefore, the second approach is used. Thus the complexity of each sleep transition in IRTH through principle 3 is $O(\ell)$. The principle 2 of IRTH exploits the future release information (γ). Therefore, it is difficult to find the sleep interval offline, and hence, estimated online on each sleep invocation. To compute the complexity of a sleep transition in principle 2, it is assumed $\Theta = \frac{T_{max}}{T_{min}}$, where T_{max} is the maximum and T_{min} is the minimum inter-arrival time in the task-set. Then the complexity of each sleep transition in principle 2 is $O(\Theta \times \ell)$, as in worst-case the scheduler has to check the all possible job releases within T_{max}.

The online complexity LWRTH is low when compared to IRTH. LWRTH only initiates a sleep state transition in idle mode. It relies on future release information array to maximise the energy efficiency. Similar to IRTH, tasks are stored with respect to their ID's and on each sleep invocation the algorithm traverses γ. Therefore, each sleep transition happening in LWRTH has a complexity of $O(\ell)$. This algorithm does not need any slack management algorithm, and moreover, its online complexity to initiate a sleep transition is also low when compared to ERTH and IRTH. A system designer needs to perform a careful evaluation, while selecting among the available algorithms. IRTH clearly has the highest complexity when compared to ERTH and LWRTH but provides the best energy efficiency among them. The complexity comparison of ERTH and LWRTH is difficult. On one side, ERTH does not require to maintain a list of future release information, while LWRTH requires information which needs to be updated on every task's release. On the other hand, LWRTH has lower sleep transition overhead when compared to ERTH and does not exploit the execution slack generated from the slack management algorithm.

Parameters	Values
Task-set sizes $\lvert\tau\rvert \in$	$\{10, 20, \ldots, \underline{50}, \ldots, 100\}$
$T_{min} \in$	$\{\underline{30}, 40, \ldots, 100\}$
PUB $\in$	$\{1.1, 1.2, \ldots, \underline{1.5}, \ldots, 5\}$
BCET limit $C^b \in$	$\{0.2, 0.25, \ldots, \underline{1}\}$
Sporadic delay limit $\Gamma \in$	$\{\underline{0}, 0.05, \ldots, 1\}$

Table 4.1: Overview of simulator parameters used to evaluate demand bound function based procrastination

4.4.2 Simulation Results of the DBFP Algorithm

The discrete event simulator SPARTS (simulator for power aware and real-time systems) [NAP11a, NAP11b] discussed in Section 3.2 is used to evaluate the effectiveness of the DBFP approach. SPARTS is used with the parameters mentioned in Table 4.1, where underlined values are the default values if not mentioned otherwise in the description of the experiment. The parameters C^b and Γ are used to generate wide variety of different tasks and their subsequent varying jobs. The periods of both BE and RT tasks are chosen from an interval, $T_{min}[1, \text{PUB}]$, where T_{min} is the lower bound and PUB (Period Upper Bound) is the variable used to define the upper bound on the

interval. Each task-set with different parameters mentioned in Table 4.1 is simulated for 100 times with different seed values to the random number generator and averaged. The simulation time of each task-set is 100 seconds. All the tasks are assumed to have implicit deadlines ($D_i = T_i$).

The SRA algorithm [JG05] is an energy saving approach that takes procrastination intervals of the tasks determined through Jejurikar's method as an input. For a fair comparison, the same algorithm is used by just replacing the input phase with DBFP determined procrastination intervals. For simplicity sake, it is assumed that all the slack in the schedule (spare capacity) is reserved for the shut-down of the processor. Both variations of SRA are implemented in SPARTS and their sleep state is selected offline based on their respective minimum idle interval. It has already been shown in the state-of-the-art that SRA performs better than LC-EDF, hence, this section only considers SRA for the comparison.

No.	Power Mode	tr_n (μs)	bet_n (μs)	P_n ($Watts$)	Es_n ($\mu Joules$)
1.	Doze	5	225	3.7	42
2.	Nap	100	450	2.6	950
3.	Sleep	200	800	2.2	1980
4.	Deep Sleep	500	1400	0.6	5750
5.	Typical	0	0	4.7	0
6.	Maximum	-	-	12.1	-

Table 4.2: Different sleep states parameters

The power model used for simulations is based on the Freescale PowerQUICC III Integrated Communications Processor MPC8536 [Sem]. The power dissipation values are taken from its data sheet for different modes (Maximum, Typical, Doze, Nap, Sleep, Deep Sleep). The core frequency of 1500 MHz and core voltage of 1.1 V is used for all the experiments. The transition overheads are not mentioned in their data sheet, therefore, assumed values are used for four different sleep states. The transition overhead of the typical mode that corresponds to the idle state in our system model is considered negligible. The power values given in Table 4.2 sum up core power and platform power dissipation. More details are available in the reference manual [Sem].

Figure 4.9 presents the gain of DBFP over SRA with respect to average sleep interval for different values of U and PUB. The average sleep interval is computed by accumulating the idle time in the scheduling and dividing it by the number of sleep intervals. The gain of DBFP increases with an increase in system utilisation and PUB. At low utilisation DBFP and SRA have enough slack to initiate long sleep intervals. However, an increase in system utilisation decreases the slack and the procrastination-intervals lengths. Therefore, SRA starts to lose efficient sleep states at high utilisation, causing its frequent switching. In the best case, increase in the average sleep interval is approximately 75%. The gain in average sleep interval is also computed by varying the utilisation against the BCET Limit C^b as shown in Figure 4.10. Mostly, the gain occurs due to an increase in system utilisation, while the variation in C^b has a negligible effect as both algorithms use the same mechanism to manage the slack. Similarly, the change in sporadic delay limit Γ has been investigated against different values of U. The effect of Γ is negligible as well.

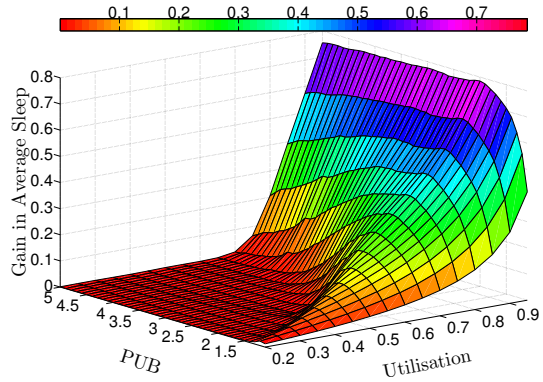


Figure 4.9: Variation in T_{max} (sleep interval)

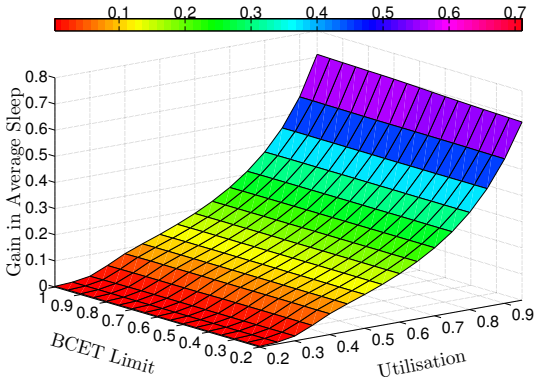


Figure 4.10: Variation in C^b (sleep interval)

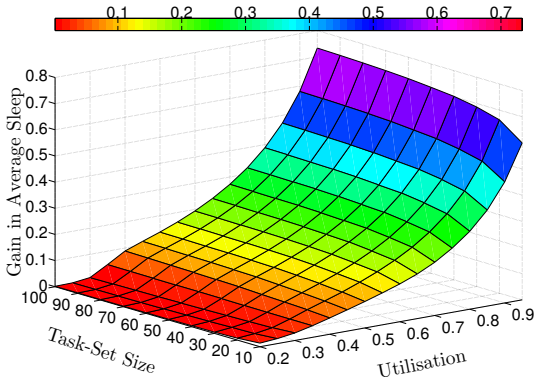


Figure 4.11: Variation in $|\tau|$ (sleep interval)

The variation in task-set size is demonstrated in Figure 4.11 against different values of U. The gain in average sleep interval increase with an increase in task-set size. In the best case (i.e., $|\tau| = 100$), the gain reaches 75%. The procrastination interval of a high priority task is always bounded by the low priority tasks in the given task-set. The difference between the procrastination intervals of different tasks between DBFP and SRA has a cascading effect. For instance, a low priority task τ_i having a procrastination interval Z_i smaller than that of a high priority task will have its Z_i scaled down due to Equation 4.4. If $Z_i < \chi_i$, then not only the difference exists at level τ_i but also $\forall \tau_k : k < i$. A large task-set has high probability to get this cascading effect. The gain in energy consumption of the processor is also analysed in idle mode. All the findings are similar to the trends presented here (see Section A.2.3 for further details).

4.4.3 Simulation Results of ERTH, IRTH and LWRTH Algorithms

The proposed alternative race-to-halt algorithms (ERTH, IRTH, LWRTH) are implemented in SPARTS and compared against the state-of-the-art (SRA and LC-EDF). The LC-EDF algorithm is included in this comparison as it has some interesting properties. The SPARTS simulator is used with the parameters specified in Table 4.3. Though not a fundamental requirement of the proposed algorithms, implicit deadlines $D_i = T_i$ are assumed for evaluation purposes. It is obvious

Parameters	Values
Task-set sizes $\|\tau\|$	$\{10, 50, 200\}$
Share of RT/BE tasks $\xi = \{\xi_1, \xi_2\}$	$\{\langle 40\%, 60\% \rangle, \langle 60\%, 40\% \rangle\}$
Inter-arrival time T_i for RT tasks	$[30ms, 50ms]$
Inter-arrival time T_i for BE tasks	$[50ms, 1sec]$
Sporadic delay limit $\Gamma \in$	$\{0.1, 0.2\}$
BCET limit C^b	0.2
Sleep threshold Ψ_x in	$\{\underline{1}, 2, 5, 10, 20\}$

Table 4.3: Overview of simulator parameters used to evaluate alternative race-to-halt algorithms

that $D_i > T_i$ leads to greater saving opportunities, but does not provide greater insights. In total, 1020 different task-sets configurations ($C^b, \Gamma, U_i, \cdots$ etc) are generated. The same power model (based on the Freescale PowerQUICC III Integrated Communications Processor MPC8536 [Sem]) specified in Table 4.2 is used in these simulations.

A vast variety of CPUs are available in the market. They have diverse hardware architectures and consequently different power characteristics. In order to observe the effect of different types of hardware platforms on the proposed alternative race-to-halt algorithms, different power parameters of the processor are generated. In the system model, active and idle time of the CPU remain constant for a specific task-set. As the total energy consumption is normed, the factor among the power model parameters that affects the energy gain of an algorithm is the overhead of the sleep transitions. However, the overhead of the sleep transition is modelled by the break-even-time of the sleep state. Therefore, the power model parameters are altered to generate a distinct BET such that it is a multiple of the original BET by a factor of x. The different break-even-times are represented with Ψ_x called sleep threshold (Table 4.3). The sleep threshold with a value of $x = 1$ denotes the BET of the original power model and this is a default value.

The overhead of all the algorithms including procrastination algorithms (LC-EDF and SRA) is considered negligible. This is obviously a favourable treatment for LC-EDF and SRA, as the time/energy overhead of the external specialised hardware is substantial. SPARTS takes into account the effect of the sleep state transition delays and its energy/time overhead is included in the power model. Each point in the figure present results averaged over 100 runs with different respective seed values as well as all different free parameters. As baseline, ERTH is simulated without the use of sleep states (NS) — processor uses typical power in idle mode — and all the results are normalised to the corresponding results of NS. The RBED framework is used for the integration of applications with different criticality levels and this framework allows an overrunning job to borrow from its future invocations [LB05]. Two different scenarios are explored.

4.4.3.1 Scenario 1 ($A_i = C_i$, $\forall$ task types)

In this scenario both task classes (RT and BE task) are assumed to have $A_i = C_i$. Moreover, $\Gamma_{0.1}$ is assumed for all experiments as the difference is marginal when compared to $\Gamma_{0.2}$. The total

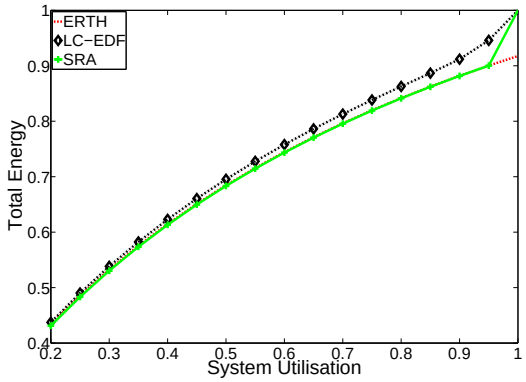


Figure 4.12: Normalised total energy consumption (ξ_1 and $|\tau| = 200$)

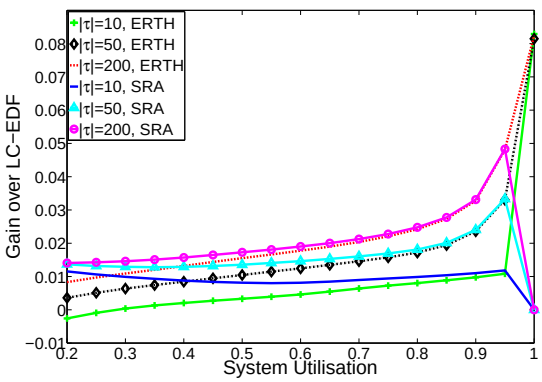


Figure 4.13: Gain of ERTH and SRA over LC-EDF for different task-set sizes

energy consumption of ERTH is compared against LC-EDF and SRA for a task-set size of 200 and a task distribution of ξ_1 in Figure 4.12. Overall, ERTH outperforms LC-EDF, particularly at high utilisations as the maximum feasible idle interval (procrastination interval) computed by the LC-EDF algorithm shrinks restricting the use of efficient sleep state. Both SRA and ERTH exploit execution slack. SRA performs comparable to ERTH except at high utilisations where the savings of ERTH are larger when compared to SRA. This is motivated by the following observations. Firstly, the resulting utilisation is less than the target utilisation by a very small factor of ε due to numerical rounding of the parameters used to generate a task-set. The secondary effect is the diversity in periods of task-set that rarely aligns and as a result the hyper-period of the given task-set is very long. Therefore, at high utilisations, the use of the demand bound function yields an actually usable χ_{min} due to the disparity of periods and deadlines. The same experiment is repeated for a distribution of ξ_2 and the processor consumes approximately 1% more energy when compared to ξ_1. In ERTH, it is due to the lesser usage of principle 2, as the system has fewer BE tasks in ξ_2. The LC-EDF and SRA algorithms depend on the period of the tasks. Extra tasks with long periods result in greater opportunities to save energy, therefore, ξ_2 consumes slightly more energy when compared to ξ_1.

An interesting observation may be noticed in the total energy consumption of LC-EDF: fine-grained large task-sets consume more energy when compared to the coarse-grained small task-sets at the same utilisation. Each procrastination interval computation shortens the sleep interval in LC-EDF and the large task-set increases this probability (for detailed analysis see Section A.3.2.2). Oppose to LC-EDF, the task-set variation does not affect the total energy consumption ERTH, IRTH, LWRTH and SRA. The overall-gain of ERTH and SRA over LC-EDF for three different task-set sizes with a distribution of ξ_1 is depicted in Figure 4.13. In general, SRA and ERTH save more energy compared to LC-EDF. There is one exception at $U \leq 0.2$ for $|\tau| = 10$ in which the energy saving of LC-EDF negligibly dominates ERTH. SRA saves approximately 1% more energy when compared to ERTH at low utilisations and its performance degrades towards high utilisations. If the energy consumption of the external hardware is more than 1% of the saving, then ERTH is still a better approach in terms of energy saving due to lower complexity when compared

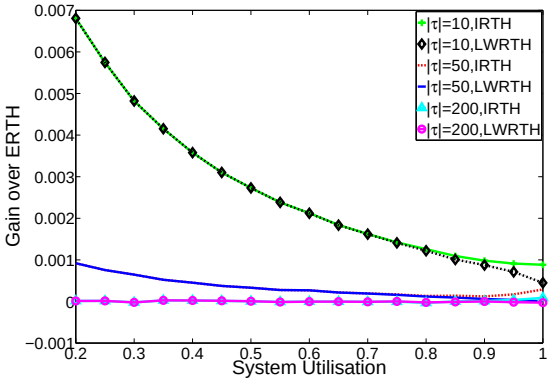


Figure 4.14: Overall-gain of IRTH and LWRTH over ERTH (ξ_1)

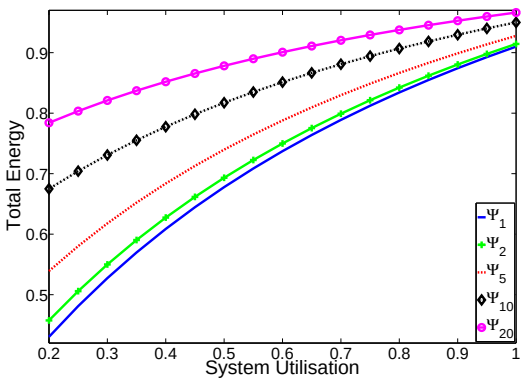


Figure 4.15: Sleep threshold effect on total energy of ERTH ($|\tau| = 50$ and ξ_1)

to SRA. For all three different task-set sizes, the gain of ξ_2 dominates ξ_1 and this difference is negligibly small. Similarly, the gain in idle interval of these algorithms is also analysed, which leads to the same conclusions discussed here (see Section A.3.2.4 for detailed discussion).

The improved slack management approach used in SRA is also integrated with ERTH for the fair comparison. The gain of the ERTH algorithm with improved slack management over ERTH with simplistic slack management approach proposed in this work in the current experimental set-up is negligible. The reason behind such a behaviour is the fact that better slack distribution plays an important role for DVFS based algorithms, where the slack distribution among different tasks is important. However, when it comes to race-to-halt algorithms, slack accumulation is important than better slack distribution. The overall-gain of IRTH and LWRTH over ERTH is illustrated in Figure 4.14 with a ξ_1 and $\Gamma_{0.1}$. In general, the gain of IRTH and LWRTH decreases with in increase in task-set size and system utilisation as the future release information becomes less effective. The average sleep-interval ERTH, IRTH and LWRTH is also analysed against SRA and LC-EDF, and all the results corresponds to the aforementioned findings (see Section A.3.2.7).

To analyse the effect of different types of hardware platforms, the effect of a high sleep threshold Ψ that indicates the scaled value of bet_n obtained by altering the power model parameters is studied for ERTH, IRTH, LWRTH, SRA and LC-EDF for two different distributions of ξ_1 and ξ_2. Figure 4.15 presents the total energy consumption of ERTH for different values of Ψ with $|\tau| = 50$ and ξ_1. Naturally, an increase in bet_n is also reflected in higher overall energy consumption as depicted in Figure 4.15. IRTH, LWRTH , SRA and LC-EDF have the similar results (with some scaling) for the different values of Ψ. A comprehensive analysis is presented Section A.3.2.8 that discusses all the details of variation in sleep threshold among different algorithms. Moreover, the effect of sleep threshold on different task-set sizes is also analysed for all algorithms. High sleep threshold manages to split the energy consumption of different task-set sizes at same utilisation (further details are available in Section A.3.2.8). Similarly, the effect of Ψ is also analysed for a distribution ξ_2 with all task-set sizes and it results in an increase in the total energy consumption due to reduced share of BE tasks.

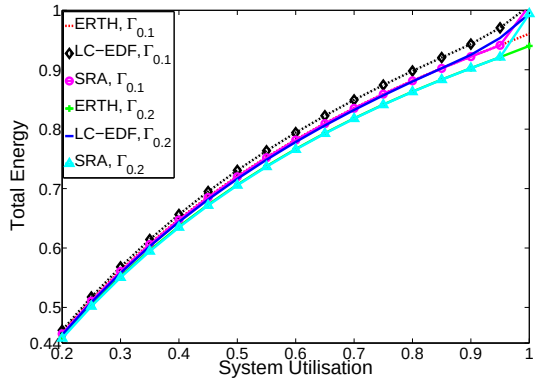


Figure 4.16: Normalised total energy consumption with $|\tau| = 200$ and ξ_1

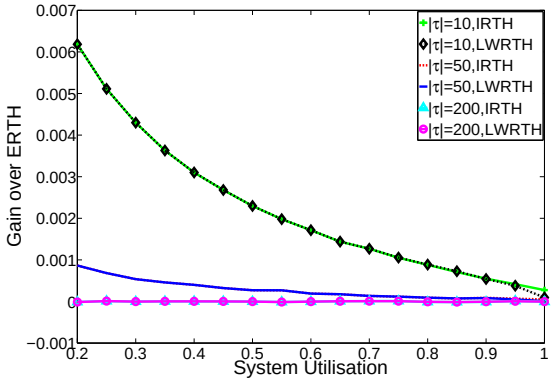


Figure 4.17: Overall-gain of IRTH and LWRTH over ERTH (ξ_1 and $\Gamma_{0.1}$)

4.4.3.2 Scenario 2 ($RT \Rightarrow (A_i = C_i), BE \Rightarrow (A_i \leq C_i)$)

In this scenario, BE tasks are allowed to occasionally require more than their allocated budget A_i. The mean of the BE tasks actual-execution-time distribution is set to 85% of A_i. All algorithms (ERTH, IRTH, LWRTH, SRA and LC-EDF) have been extended and allowed to borrow from the budget of future job releases of the same task. While it was of little consequence in scenario 1, it has to be noted that in scenario 2, ERTH and IRTH do not allocate execution slack to BE tasks. BE jobs usually overrun their budget and borrow from their future jobs, and hence, they are likely to consume the slack. However, the execution slack is only retained for energy management purposes. Thus, if the next job to execute is of BE type, the execution slack is maintained in the slack container and its deadline is updated as follows: $S_e^{dl} = \max\{S_e^{dl}, d_{i,k}\}$, where $d_{i,k}$ is the absolute deadline of the BE job under consideration.

The total energy consumption of ERTH, SRA and LC-EDF in this scenario is analysed for two different sporadic delay limits ($\Gamma_{0.1}, \Gamma_{0.2}$) and two different distributions (ξ_1,ξ_2) with $|\tau| = 200$. Figure 4.16 demonstrates the effect of a variation in the sporadic delay limit for a distribution of ξ_1. For the sake of clear representation, all the values of Figure 4.16 are normalised to the corresponding results of NS with a distribution of $\Gamma_{0.1}$. $\Gamma_{0.1}$ and $\Gamma_{0.2}$ define an interval of 10% and 20% of T_i respectively for the sporadic delay to maneuver for a task T_i. The expansion of this interval means extra sporadic slack in the system when compared to the nominal utilisation. The sporadic slack is dealt implicitly in the proposed algorithms. Therefore, energy consumption is less with $\Gamma_{0.2}$ when compared to $\Gamma_{0.1}$ as shown in Figure 4.16. Similarly, SRA and LC-EDF also have more room to initiate a sleep state as well. The same experiment is repeated with ξ_2, where the energy consumption of all algorithms decreases, the reason of which is explained in conjunction with the next experiment.

The energy consumption of two distributions (ξ_1,ξ_2) is studied with a fixed task-set size of $|\tau| = 200$ and $\Gamma_{0.1}$. The resulting figure (not shown here) has a similar shape when compared to Figure 4.16 but the different between ξ_1 and ξ_2 is slightly more pronounced. The energy consumption of SRA, LC-EDF and ERTH is reduced for ξ_2 when compared to ξ_1. The percentage

of BE tasks in ξ_2 is reduced to 40% that results in less borrowing and consequently, ξ_2 consumes less energy when compared to ξ_1. Nevertheless, ERTH outperforms LC-EDF and comparable to SRA in both distributions (ξ_1, ξ_2), even with the borrowing mechanism integrated. The energy consumption of all algorithms decreases, when the same experiment is performed with $\Gamma_{0.2}$ due to extra sporadic slack in the system. Moreover, it is also observed that the energy consumption of IRTH and LWRTH is similar to ERTH for the above mentioned two experiments as the borrowing effect dominates the total energy consumption.

The overall energy consumption gain of ERTH and SRA over LC-EDF is analysed in this scenario for three task-set sizes ($|\tau| \in \{10, 50, 200\}$) with two different distributions (ξ_1 and ξ_2) and sporadic delay limits ($\Gamma_{0.1}$ and $\Gamma_{0.2}$). The graphs are similar to the one presented in Figure 4.13. The gain with $\Gamma_{0.2}$ is greater than $\Gamma_{0.1}$ especially at high utilisations. Similarly, the gain with ξ_1 is less than ξ_2. In general, the overall-gain of ERTH and SRA over LC-EDF in this scenario is less than the overall-gain in scenario 1 at higher utilisation but approximately the same at lower utilisations. The overall energy gain of IRTH and LWRTH over ERTH is depicted in Figure 4.17 for ξ_1 and $\Gamma_{0.1}$. Compared to Figure 4.14, the overall gain has reduced in scenario 2. Moreover, IRTH and LWRTH behave identical when borrowing is enabled. Main reason is the extra execution requested by the BE task through borrowing i.e., an increase in effective utilisation. The normalised sleep state energy consumption of scenario 2 is similar to scenario 1. Moreover, the higher sleep threshold effect in scenario 2 is also identical to scenario 1 for IRTH, LWRTH, LC-EDF, SRA and ERTH with just one difference, i.e., energy consumption of the system increases in scenario 2. To summarise, for different combinations of ξ and Γ, an increase in gain occurs in the following ascending order $(\xi_2, \Gamma_{0.2})$, $(\xi_2, \Gamma_{0.1})$, $(\xi_1, \Gamma_{0.2})$ and $(\xi_1, \Gamma_{0.1})$.

4.4.4 Pre-emptions Related Results

A side effect of the use of the sleep states is a change in the number of pre-emptions. In order to find the sleep state relation with the number of pre-emptions, the pre-emptions for all algorithms (ERTH, IRTH, LWRTH, SRA and LC-EDF) are counted for different parameters. The DBFP is not included in this evaluation as it is easier to get the trend based on the results of LC-EDF and SRA. The experimental setup defined for alternative race-to-halt algorithms and the parameters defined in Table 4.3 remain the same except some alterations in best-case execution-time limit C^b and sporadic delay limit Γ. The best-case execution-time limit C^b is varied from 0.25 to 1 with an increment of 0.25 (i.e., $C^b \in \{0.25, 0.5, 0.75, 1\}$). Similarly, the sporadic delay limit Γ is varied from 0 to 0.6 with an increment of 0.2 (i.e., $\Gamma \in \{0, 0.2, 0.4, 0.6\}$). For the representation purposes, only the two corner values for $\Gamma = (0, 0.6)$ and $C^b = (0.25, 1)$ are plotted, as the results for the other two values lies in between these two curves and scales linearly. All the values in the following experiments are normalised to the number of pre-emptions with earliest deadline first algorithm (EDF). The results shows that the pre-emption count for LWRTH is virtually identical to IRTH, therefore, for presentation purposes only results of IRTH are shown hereafter.

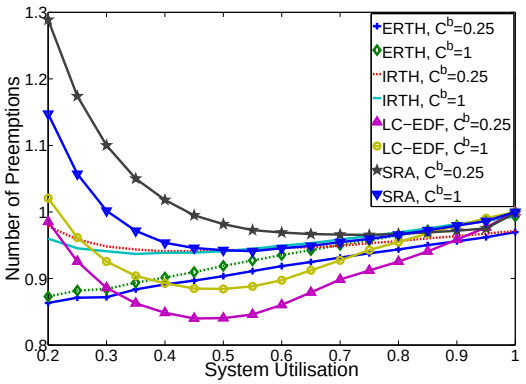


Figure 4.18: Variation in C^b for $|\tau| = 10$ $(\Gamma_{0.2}, \xi_1)$

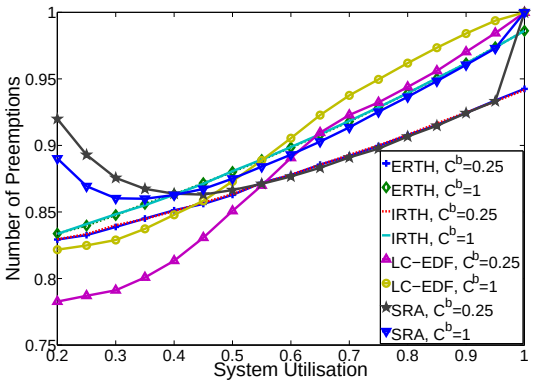


Figure 4.19: Variation in C^b for $|\tau| = 50$ $(\Gamma_{0.2}, \xi_1)$

4.4.4.1 Scenario 1

In this scenario, it is assumed all the tasks have $A_i = C_i$. The effect of best-case execution-time limit variation for task-set sizes of 10 and 50 are presented in Figure 4.18 and Figure 4.19 respectively with $\Gamma_{0.2}$ and ξ_1. Overall all scheduling algorithms showed a positive impact of sleep states on the number of pre-emptions, except for one case in LC-EDF at $U = 0.2$ and in SRA at $U \leq 0.45$ for a small task-set size of 10. With a small task-set size, jobs releases are anyway dispersed at low utilisation. SRA and LC-EDF initiate a sleep state in idle mode, start estimating the delay interval on the next job release and extend it as much as possible. This behaviour causes widely spread low priority jobs at low utilisation to come closer to high priority jobs and hence increase the pre-emption count. Moreover, at low utilisation, in EDF the number of pre-emptions are small and the use of sleep states cannot help much to reduce them. For $|\tau| = 50$, LC-EDF, ERTH and IRTH have fewer number of pre-emptions with $C^b = 0.25$ when compared to $C^b = 1$ at all utilisations, while this observation only holds at high utilisations for SRA. Another observation for a small task-set size of 10 is the positive impact of $C^b = 0.25$ over $C^b = 1$ that holds at all utilisations for LC-EDF and ERTH, and only at high utilisations for SRA and IRTH. A small value of C^b has the high potential to generate execution slack and increases the chance to initiate sleep states leading to a reduced number of pre-emptions.

IRTH (in Figure 4.18) and SRA (in Figure 4.18 and Figure 4.19) show an oddity at low utilisations, as $C^b = 1$ has fewer pre-emptions compared to $C^b = 0.25$. In IRTH algorithm, sleep states are increased by utilising predicted future release information. Future release information is very useful especially to prolong the sleep interval for a small task-set size at low utilisations. It can be easily motivated by the curve of Figure 4.14 that IRTH saves more energy at low utilisations for a task-set size of 10 due to extensively long sleep intervals. Similarly, SRA sleep intervals are even greater than or equal to all the algorithms. As a side effect of long sleep intervals, they assemble a large amount of work for later execution. This delayed execution later on encounters high priority tasks and causes additional pre-emptions. However, if the encountered high priority tasks execute for their C_i, the chances are higher that it might accumulate other tasks having priority higher than

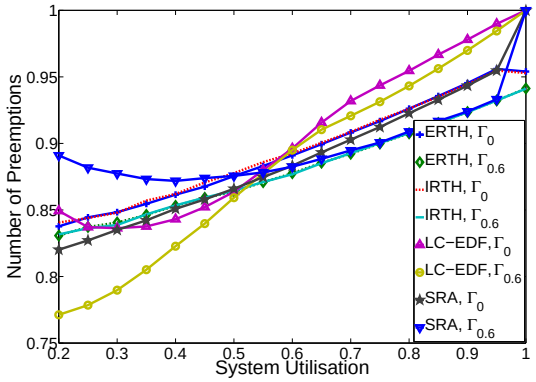


Figure 4.20: Variation in Γ for $|\tau| = 50$ (ξ_1)

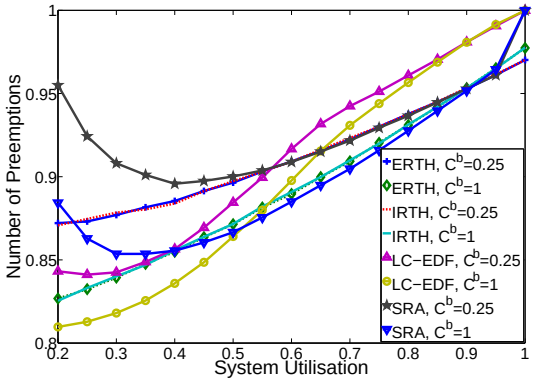


Figure 4.21: Variation in C^b in scenario 2 for $|\tau| = 50$ ($\Gamma_{0.2}, \xi_1$)

the backlog and less than the encountered high priority tasks. These intermediate priority tasks will not cause pre-emptions to a backlog. This effect causes the flip of $C^b = 0.25$ over $C^b = 1$ for low utilisations.

The effect of variation in sporadic delay limit Γ is illustrated in Figure 4.20 for a task-set $|\tau| = 50$ and a distribution of ξ_1. All algorithms consume sporadic slack implicitly. An increase in the sporadic delay limit causes an increase in sporadic slack and that can increase the number and/or prolong the sleep transitions. Similar to the execution slack, sporadic slack also helps to decrease the number of pre-emptions for all algorithms at all utilisations except for SRA at low utilisations. As mentioned previously, the widely spread out jobs in the EDF schedule are unlikely to preempt each other but SRA brings these jobs close to such a degree that they result in an increased number of pre-emptions at low utilisation. In general, it has been observed with our experimental setup that whenever there is a possibility to increase the length of a sleep state (either through execution slack or sporadic slack) at low utilisations, SRA increases the number of pre-emptions. The same experiment is repeated for a task-set size of 10 and it leads to the same findings. The effect of variation in the distribution ξ is also studied for $|\tau| \in \{10, 50\}$, $\Gamma_{0.2}$ and $C^b = 0.5$. In general, ξ_2 saves more pre-emptions when compared to ξ_1 for all the algorithms. BE tasks are more vulnerable to pre-emptions as they have longer periods along with their execution. Therefore, ξ_1 having more BE tasks results in more pre-emptions, when compared to ξ_2.

4.4.4.2 Scenario 2

In this scenario, BE jobs occasionally require more than their respective budget and borrow from their future job releases. The effect of variation in the best-case execution time limit C^b is investigated for $|\tau| \in \{10, 50\}$, $\Gamma_{0.2}$ and ξ_1. Figure 4.21 depicts the results only for a $|\tau| = 50$. One of the interesting observation that holds for all algorithms for all task-set sizes is that with borrowing $C^b = 1$ offers fewer pre-emptions when compared to $C^b = 0.25$. Because of the borrowing, BE tasks add a great deal of backlog in addition to a backlog assembled due to sleep transitions. Therefore, it increases the probability to encounter higher priority tasks. Similar to the case explained

for IRTH ($U \leq 0.5$) in Figure 4.18, if the encountered higher priority tasks execute for their C_i, chances are higher that they will collect some of the tasks having priority in between backlog and the higher priority executing jobs. Thus $C^b = 1$ offers fewer pre-emptions compared to $C^b = 0.25$. A further experiment explores a variation in the sporadic delay limit Γ for all task-set sizes. The results show an increase at low utilisations when compared to a system without borrowing. Moreover, SRA with borrowing in the system saves more pre-emptions with an increase in the amount of sporadic slack. Thus, the number of pre-emptions is higher for Γ_0 when compared to $\Gamma_{0.6}$. The variation in the distribution of task-set ξ also increase the number of pre-emption when the borrowing is allowed in the system. BE tasks that overrun demand extra execution and hence more pre-emptions compared to the normal system without borrowing.

Finally, it is observed, when it comes to number of pre-emptions, ERTH performs superior to IRTH, LWRTH and SRA for small task-set sizes. Nevertheless, it equally performs comparable to IRTH, SRA and LWRTH if not better for large task-set sizes. Though SRA performs better energy-wise but has the highest number of pre-emptions at low utilisations and sometimes it even exceeds those by plain EDF scheduler. The overhead associated to the number of pre-emptions saved through the use of sleep states can help to reduce the worst-case execution time of the tasks. This effect further extends the slack in the system and consequently provide an extra opportunity to save energy in the system or increase the system utilisation.

4.5 Thermal-Aware Energy Management

The increase in power density of modern processors demands efficient thermal management solutions to keep the temperature within given limits in order to avoid physical damage and also to increase the reliability of the chip. Thermal management can be done at design time through sophisticated packaging and heat dissipation techniques, and at run time through DTM. However, the packaging and the active heat dissipation solutions are very expensive [TSR+98]. It has been predicted in the International Technology Roadmap for Semiconductor (ITRS2005) that the packaging solutions will become more challenging in the near future due to an increase in peak power and the high power density in emerging system-in-packages [ITR05]. This trend motivates to explore DTM techniques for a wide variety of systems. The DTM techniques can be coupled with energy minimisation objective. Energy efficiency has the objective to reduce the cumulative power dissipation, while DTM techniques aim to keep the peak temperatures of the processor below the critical limit. The commonly used DTM approaches in RT systems to handle the thermal constraint along with energy and temporal restrictions are speed scheduling and TCDPM.

1. **Speed Scheduling:** The frequency of the processor is reduced to decrease the temperature and the dynamic power dissipation of the system.

2. **TCDPM:** The processor executes the workload at full speed and switches off when the peak temperature is reached to cool down the system.

This research effort only deals with TCDPM approaches. It is demonstrated that the TCDPM approach behave very similar to idealised dynamic voltage and frequency scaling in the context of RT systems. Therefore, any existing dynamic voltage and frequency scaling solution proposed for periodic/sporadic task models can be transformed to develop a new TCDPM approach with moderate effort. A detailed discussion given below identifies the similarities along with the distinctive elements between two approaches (TCDPM and Idealised DVFS).

4.5.1 Extension in the System Model

The system model used in this work is slightly different than the one presented in Chapter 3. The extensions in the system model are discussed below.

4.5.1.1 Workload Model

This work assumes a HRT system, where a system cannot afford to miss any deadline, therefore, BE and SRT tasks are treated as HRT tasks. The task-set, tasks and jobs have the similar characteristics as mentioned Section 3.1.1. This work can be extended for constrained deadline model ($D_i \leq T_i$), however, an implicit deadline model ($D_i = T_i$) is assumed for the ease of presentation. The optimal uniprocessor earliest-deadline-first (EDF) dynamic priority scheduling algorithm is used to schedule a task-set τ.

4.5.1.2 Power Model

The power and the thermal model used in this work are adopted from the work of Yang et al. [YCTK10]. The leakage-current is considered to be temperature dependent. The average leakage current $\bar{I}(Tm, V_{dd})$ at temperature Tm and supply voltage V_{dd} is modelled by Liao et al. [LHL05] as given in Equation 4.35,

$$\bar{I}(Tm, V_{dd}) = \bar{I}(Tm_0, V_0)\left(ATm^2 e^{(\frac{\mathscr{X}V_{dd}+\mathscr{Y}}{Tm})} + Be^{(\mathscr{Z}V_{dd}+\mathscr{W})}\right) \tag{4.35}$$

where $\mathscr{X}, \mathscr{Y}, \mathscr{Z}, \mathscr{W}, A$ and B are empirical constants for different circuit types, technology and designs. These empirical constants are obtained through curve fitting on the power dissipation of different circuit types at multiple temperatures using SPICE simulations [LHL05]. $\bar{I}(Tm_0, V_0)$ is a reference leakage current on temperature Tm_0 with a reference supply voltage of V_0. The unit of the temperature is in Kelvin (K). It is based on the curve fitting of the power dissipation of the different circuit types at different temperatures with SPICE simulations. Yang et al. [YCTK10] found a good approximation of such modelling in a quadratic form as shown in Equation 4.36,

$$\bar{I}(Tm, V_{dd}) = \hat{A}Tm^2 + \hat{B} \tag{4.36}$$

$$\hat{A} = \frac{\bar{I}(Tm_H, V_{dd}) - \bar{I}(Tm_L, V_{dd})}{Tm_H^2 - Tm_L^2} \tag{4.37}$$

$$\hat{B} = \bar{I}(Tm_L, V_{dd}) - \hat{A}Tm_L^2 \tag{4.38}$$

where $\hat{A}$ and $\hat{B}$ are constants, while Tm_H and Tm_L define the operating temperature range of the chip. They showed difference of this approximation is negligible when compared to average leakage current modelled by Laio et al. [LHL05] (Equation 4.35).

The processor assumed in this work has two modes: *active* and *sleep state*. The execution of tasks is performed in the active mode and P_A denotes its power dissipation. It has two components: a) dynamic power dissipation P_{dyn} and b) static or leakage-power dissipation P_{lkg}. The dynamic power dissipation of the processor is considered constant in active mode, while the static power dissipation is modelled as $P_{lkg} = \mathscr{A}Tm^2 + \mathscr{B}$, where $\mathscr{A}$ and $\mathscr{B}$ are $N_{gate}\hat{A}V_{dd}$ and $N_{gate}\hat{B}V_{dd}$ respectively. N_{gate} is a constant that depends on the circuit characteristics (for more details refer to [YCTK10, LHL05]). Only a single sleep state $\S_1$ is assumed in this work. The system can transition to a sleep state for two different purposes: 1) to cool down the processor and 2) to reduce the energy consumption. Each sleep transition has energy and delay cost associated to it. The transition time of going into and out of sleep state is denoted as ts_1 and tw_1 respectively. The extra energy consumed during a transition phase is denoted as Es_1. The processor has to complete its transition into and out of a sleep state once initiated. When the processor is in sleep state, it has a constant power dissipation of P_1. The processor assumed in this model runs at top speed in the active mode and does not support DVFS.

4.5.1.3 Thermal Model

A widely adopted [YCTK10, YLQ06] thermal RC model is used to characterise the temperature behaviour of the processor and expressed as a differential equation (Equation 4.39), where C_{th}, R_{th}, P_W, Tm and Tm_{amb} are the thermal capacitance ($Joule/K$), thermal resistance ($K/Watts$), processor's power dissipation ($Watts$), processor's temperature (K) and the ambient temperature (K) respectively.

$$\frac{dTm}{dt} = \frac{1}{C_{th}}P_W - \frac{1}{R_{th}C_{th}}(Tm - Tm_{amb}) = \hat{\alpha}P_W - \hat{\beta}(Tm - Tm_{amb}) \tag{4.39}$$

In the active mode the temperature of the processor increases as the power dissipation of the processor is converted into heat. This conversion ratio is modelled by a factor $\hat{\alpha} = \frac{1}{C_{th}}$. Usually, a processor has a heat sink to remove this heat. The temperature degradation is influenced by the difference in the processor's temperature Tm and the ambient temperature Tm_{amb}. The ability of the hardware to decrease the temperature is modelled with a factor $\hat{\beta} = \frac{1}{R_{th}C_{th}}$.

Yang et al. [YCTK10] solved this differential equation (Equation 4.39) and derived temperature as a function of time for both active (Equation 4.40) and sleep state (Equation 4.41) modes. The same notations are used here for consistency.

$$T_{act}(\hat{t},t) = \frac{-(k\theta_1 e^{(\theta_1-\theta_2)t}+\theta_2)}{a\left(ke^{(\theta_1-\theta_2)t}+1\right)} \tag{4.40}$$

$$T_{dor}(\check{t},t) = (1-e^{-\hat{\beta}t})\eta + T_{dor}(\check{t},0)e^{-\hat{\beta}t} \tag{4.41}$$

Assume, a processor starts its execution at time instant $\hat{t}$ and remains in an active state for an interval of $(\hat{t},\hat{t}+t]$, then $T_{act}(\hat{t},t)$ is the temperature at time instant $\hat{t}+t$. Similarly, $T_{dor}(\check{t},t)$ is a temperature at the end of the interval $(\check{t},\check{t}+t]$ assuming a processor started its sleep state at time instant $\check{t}$. Hence, $T_{act}(\hat{t},0)$ and $T_{dor}(\check{t},0)$ are temperatures at time instance $\hat{t}$ and $\check{t}$ respectively. The parameters $\theta_1 = \frac{b+\sqrt{b^2-4ac}}{2}$, $\theta_2 = \frac{b-\sqrt{b^2-4ac}}{2}$, $k = \frac{-(aT_{act}(\hat{t},0)+\theta_2)}{(aT_{act}(\hat{t},0)+\theta_1)}$, $\eta = (Tm_{amb} + \frac{\hat{\alpha}}{\hat{\beta}}P_s)$, $a = \hat{\alpha}\mathscr{A}$, $b = -\hat{\beta}$ and $c = \hat{\alpha}(P_A+\mathscr{B})+\hat{\beta}Tm_{amb}$. Let Tm_{cri} be the maximum allowed temperature for the safe operation of the chip. Equation 4.40 and Equation 4.41 can be rewritten in terms of temperature and their corresponding equations are given in Equation 4.42 and Equation 4.43 respectively. With Equation 4.42 and Equation 4.43, one can compute the time units system takes to move from one temperature to another both in active and sleep modes respectively.

$$t_a = \frac{1}{\theta_1-\theta_2}\ln\left(\frac{-(\theta_2+T_{act}(\hat{t},t)a)}{k(\theta_1+T_{act}(\hat{t},t)a)}\right) \tag{4.42}$$

$$t_c = \frac{1}{-\hat{\beta}}\ln\left(\frac{\eta-T_{dor}(\check{t},t)}{\eta-T_{dor}(\check{t},0)}\right) \tag{4.43}$$

The energy consumption in a sleep state for an interval of $[t_1,t_2]$ is $E_s = P_1(t_2-t_1)$. The active energy consumption E_a is computed by integrating P_A [YCTK10] as given in Equation 4.44.

$$\begin{aligned} E_a &= \int_{t_1}^{t_2} P_A dt = \int_{t_1}^{t_2}(P_{dyn}+\mathscr{A}T_{act}(t_1,t_2-t_1)^2+\mathscr{B})dt \\ &= (P_{dyn}+\mathscr{B})t\Big|_{t_1}^{t_2} + \frac{\mathscr{A}}{a^2}\left[\theta_2^2 t + (\theta_1-\theta_2)\ln\left(ke^{(\theta_1-\theta_2)t}+1\right) + \frac{(\theta_1-\theta_2)}{ke^{(\theta_1-\theta_2)t}+1}\right]\Bigg|_{t_1}^{t_2} \end{aligned} \tag{4.44}$$

4.5.2 Preliminaries

The concepts needed to explain the equivalence of TCDPM and idealised DVFS are presented here.

4.5.2.1 Available Utilisation

The execution of a workload on a processor increases its temperature. A processor triggers a cooling phase, when its temperature reaches the thermal threshold. The fundamental design decision

in such systems is to define the length of the cooling and active phases. To get the intuition, how different parameters affect this decision, two conflicting scenarios are discussed below.

1. The exponential nature of the thermal model allows the processor to perform more execution at high temperatures as the temperature rise in the active phase is slower and the temperature fall in the cooling phase is faster. The leakage current also increases at high temperature and results in additional energy consumption.

2. Conversely, when the processor cools down to low temperatures, its temperature rises faster in the active phase and falls slower in the cooling phase. The leakage current is also relatively smaller at low temperatures.

A trade-off between performance and the energy consumption is evident from these two scenarios. Moreover, a shorter cooling cycle also increase the number of sleep transitions, which is not desirable due to an overhead associated to each sleep state transition. On contrary, a long cooling phase decreases the energy consumption by reduced sleep transitions. The amount of execution that a processor can deliver should be related to the thermal constraint. The available utilisation defines such metric, which is formally presented as follow.

Definition 34 (Available Utilisation)**.** *The available utilisation of the system is the maximum amount of execution per unit time delivered by the processor while respecting the thermal constraint.*

Let $Tm_{\max} : Tm_{\max} \leq Tm_{cri}$ be the upper threshold temperature after which the scheduler initiates the cooling phase. The scheduler allows the processor to execute unless its temperature reaches $Tm_{\max}$. Similarly, the cooling phase is terminated when the temperature reaches a lower threshold temperature $Tm_o : Tm_o < Tm_{\max}$. The available utilisation U_{avail} of the processor with such repetitive cycles can be defined as given in Equation 4.45, where t_a is the time processor takes in active state to reach from Tm_o to $Tm_{\max}$ and t_c is the time it takes to cool down to Tm_o from $Tm_{\max}$.

$$U_{avail} \stackrel{\text{def}}{=} \frac{t_a}{t_a + t_c} \tag{4.45}$$

The execution is performed during t_a time interval, while t_c is the idle time. Using the empirical data given in the work of Yang et al. [YCTK10], Figure 4.22 plots the temperature profile of the processor versus time. The cooling phase and the execution phase are exponential functions and the rate of change in temperature is higher in the beginning of their respective phases. This illustrates the fact that one can execute more by setting $Tm_{\max}$ and Tm_o at high temperatures to get more performance. The available utilisation of the processor for different lengths of execution times in active phase (t_a) are presented in Figure 4.23. The value of $Tm_{\max}$ is fixed to $400K$. It is evident that the increase in duration of the active phase of the processor reduces the available utilisation (i.e., the amount of work done per unit time) of the processor. In a uniprocessor RT systems,

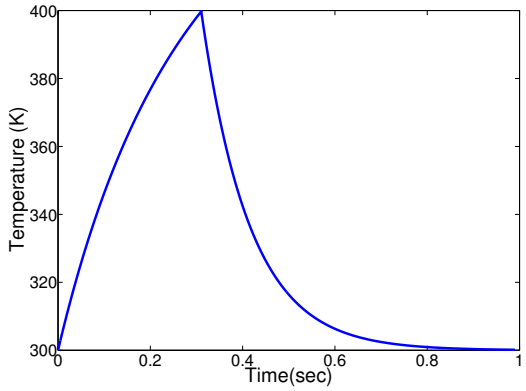


Figure 4.22: Temperature profile

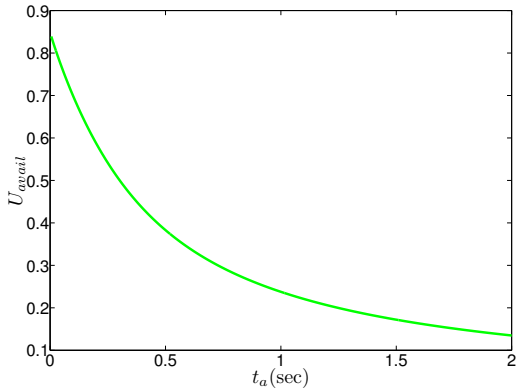


Figure 4.23: U_{avail} vs t_a

the worst-case execution requirements are known a-priori. Given the execution requirements in terms of U_{avail}, one can vary the values of t_a and t_c, to reduce energy consumption while respecting the thermal constraint. The transformation from execution time requirement to available utilisation is discussed in Section 4.5.3.

The available utilisation of the processor given in Equation 4.45 is defined as a function of time. It can be defined as a function of temperature as well. Assume, a processor transitions into a sleep state in the cooling phase then the value of $T_{dor}(\check{t},0) = T_{act}(\hat{t},t)$ and represented as $Tm_{\max}$ and similarly, $T_{dor}(\check{t},t) = T_{act}(\hat{t},0)$ and replaced with Tm_o. In this case, Equation 4.42 and Equation 4.43 can be used to replace the corresponding values of t_a and t_c respectively to define U_{avail} as a function of temperature as given in Equation 4.46. The unknown variables in Equation 4.46 are Tm_o and $Tm_{\max}$.

$$U_{avail} \stackrel{\text{def}}{=} \frac{\hat{\beta}\ln\left(\frac{(\theta_1+Tm_oa)(\theta_2+Tm_{\max}a)}{(\theta_2+Tm_oa)(\theta_1+Tm_{\max}a)}\right)}{\hat{\beta}\ln\left(\frac{(\theta_1+Tm_oa)(\theta_2+Tm_{\max}a)}{(\theta_2+Tm_oa)(\theta_1+Tm_{\max}a)}\right)-(\theta_1-\theta_2)\ln\left(\frac{\eta-Tm_o}{\eta-Tm_{\max}}\right)} \tag{4.46}$$

4.5.2.2 Energy Consumption of RT Systems under Thermal Constraint

The energy consumption of the processor with leakage-aware TCDPM can be minimised through two different factors.

1. Initiating the sleep state for longer intervals to reduce the total cost of sleep transitions and to maximise the idle period in low power state.

2. Running the system at low operating temperatures to avoid the higher leakage-power dissipation at high temperatures (i.e., setting $Tm_{\max}$ and Tm_o to low temperatures).

In the first case, duration of the sleep intervals is increased, the processor gets more time to cool down. This effect decreases the available utilisation of the processor as the temperature rises at faster rate at low temperatures in active mode and on contrary, the rate of cooling is slower at

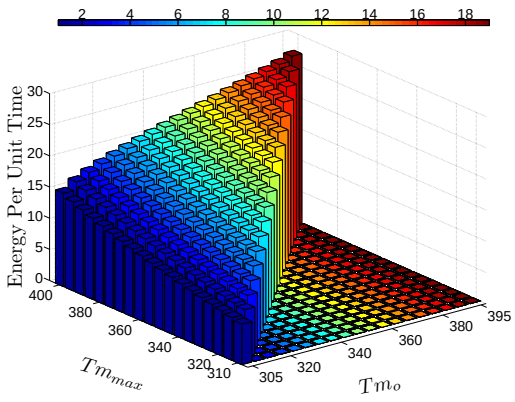


Figure 4.24: Energy vs operating temperature range

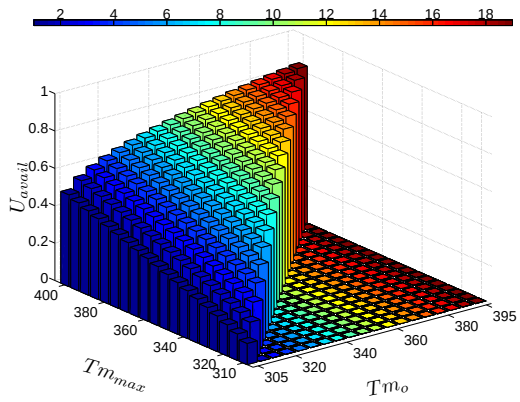


Figure 4.25: U_{avail} vs operating temperature range

low temperatures. In the second case, running a system at high temperature increases the leakage-power dissipation. However, if the operating temperature range, i.e., both $Tm_{\max}$ and Tm_o, is shifted to low temperatures, the available utilisation of the processor also decreases because of the same aforementioned reason. In both cases the decrease in available utilisation is due to a reduction in the duty cycle.

An optimal solution should consider both factors mentioned above to minimise the overall energy consumption. Nevertheless, intuition is clear that the energy consumption in TCDPM is reduced by running the processor at the lowest possible available utilisation (decreasing the duty cycle). One can propose different techniques to find the optimal set of $Tm_{\max}$ and Tm_o considering both factors for different values of U_{avail}. However, the objective of this research effort is not to find such values, rather to show that idealised DVFS algorithms are equivalent to TCDPM in a sense that both have the same objective to run the system at low available utilisation U_{avail} whenever it is possible.

The intuitions mentioned above are justified with the help of experimental results presented as follows. Figure 4.24 shows the energy consumption per unit of time (power) of the processor for different values of Tm_o and $Tm_{\max}$. It is evident that the energy consumption of the processor increases with an increase in the values of Tm_o and/or $Tm_{\max}$. Figure 4.25 presents available utilisation of the system against different values of Tm_o and $Tm_{\max}$. Available utilisation of the system increases with an increase in the operating temperature range. Combining the observations given in Figure 4.24 and Figure 4.25, it can be deduced that the high execution requirement (high performance) can only be achieved by operating the processor at high temperatures. Given $Tm_{\max}$ and Tm_o, the values of t_c and t_a can be determined by using Equation 4.42 and Equation 4.43. As a first approximation it is assumed that the value U_{avail} is computed by fixing $Tm_{\max}$ to Tm_{cri} and varying Tm_o.

4.5.3 Equivalence of Idealised DVFS and TCDPM

The available utilisation U_{avail} given in Equation 4.45 provides the execution per unit time for long time intervals (i.e., $\Delta t \gg t_c$), which is virtually equivalent to the normalised speed of the processor. The reduction in the amount of work per unit time (i.e., available utilisation or virtual speed of the processor) also decreases the energy consumption of the system. This occurs as the amount of work per unit time is decreased by reducing the duty cycle in TCDPM which can be achieved either by allowing the processor to stay longer in the sleep state or by decreasing the operating temperature range (i.e., $Tm_{\max}$ and Tm_0) of the system. This virtual reduction of speed also means prolonging the execution time of the tasks as the temperature rise is exponential and execution per unit of time does not scale linearly with a decrease in temperature.

The traditional idealised DVFS theory is also based on a convex function of the power dissipation. The decrease in speed/frequency of the processor though saves energy but also prolongs the execution time of the given workload by running the processor slower. In real DVFS, the execution time does not scale linearly with the processor speed $\frac{1}{f_v}$ (for example, memory access time does not scale with the processor frequency) [SPH07]. However, the above assumption is often made in the literature.

Under TCDPM, the execution of the workload is performed at full speed and it behaves almost at 50% speed when given a 50% duty cycle (available utilisation). Similarly, in idealised DVFS, it is assumed the execution scales by a factor of $\frac{1}{f_v}$. If the frequency is 50%, the execution time scales by a factor of 2 which is equivalent to 50% duty cycle in TCDPM at full speed. Moreover, another reason for similarity is that idealised DVFS has a continuously spectrum of available frequencies and similarly, TCDPM can represent the duty cycle in any ratio. If frequencies are normalised in idealised DVFS, there is a correlation between idealised DVFS frequencies and normalised speed (duty cycle) in TCDPM. In both cases the objective is to reduce the amount of work per unit time to reduce the overall energy consumption. The similarities between these two problems allow us to apply existing DVFS algorithms on TCDPM to reduce the energy consumption with some minor modifications in the schedulability analysis and/or speed modifications in TCDPM.

In DVFS, the amount of work per unit time is reduced by decreasing the physical frequency of the processor. The processor runs the instruction at slow but constant rate. The schedulability of the sporadic task model in DVFS is preserved if $\frac{f_v}{f_{V^m}} \geq U$, where f_v is the processor's frequency at any time t and f_{V^m} is its maximum frequency. On the other side, the suspension of the execution in the cooling phase of TCDPM may cause some of the tasks to miss their deadlines under EDF. Let us consider one task in isolation to show how it can miss its deadline and then propose a method to avoid it. Later in this section, this analysis is extended for multiple sporadic tasks.

Case 1) Single Task: Figure 4.26 represents TCDPM processing in an execution vs time graph commonly known as a service curve. The continuous line step function represents the ideal-case, where the task starts its execution in the beginning of the active phase. The straight line beneath it shows the gradient of execution i.e., U_{avail}. Assume, a worst-case scenario, i.e., the task arrives

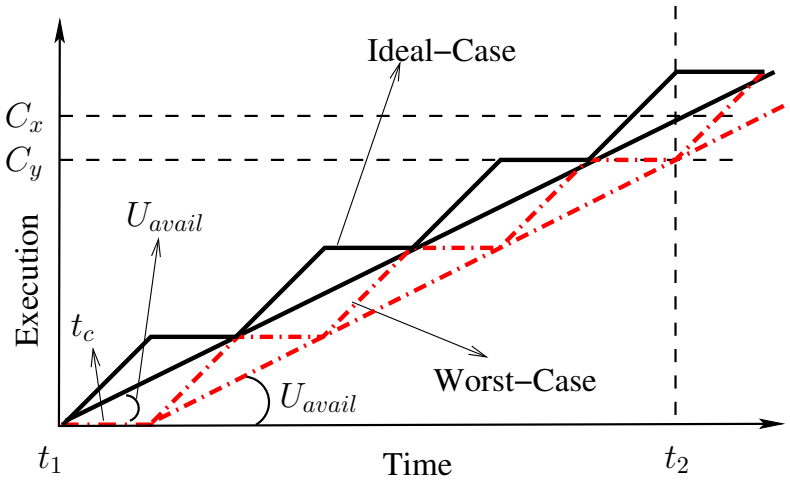


Figure 4.26: Service curve

in the beginning of the cooling phase and suffers an initial delay of t_c, it may miss its deadline (see dotted step function in Figure 4.26). This delay reduces the effective amount of work that a processor should deliver per unit time to meet all deadlines in the schedule. Assume t_1 is the initial time instant and t_2 is any time instant such that $t_2 > t_1$ && $t_2 > t_c$. The amount of work done in ideal-case in the interval $\Delta t = t_2 - t_1$ will be equal to $\Delta t\ U_{avail} = C_x$. While, in worst-case with an initial delay of t_c it will be equal to $U_{avail}\ \Delta t - U_{avail}\ t_c = C_y$. By substituting the value of C_x and rearranging, $C_x - C_y = U_{avail}\ t_c$. This is the maximum delay that a task can have in its T_i. As both U_{avail} and t_c are positive entities, the processor has executed in the worst-case $U_{avail}\ t_c$ time units less than in the ideal-case. To preserve the schedulability, the scheduler needs to satisfy two conditions given below.

- *Condition 1:* The effect of the additional delay of t_c should be accounted in the requested utilisation. The effect of this error is quantified by computing the requested utilisation U_{req} as given Equation 4.47. The length of the cooling phase used in Equation 4.47 corresponds to the time interval computed in the ideal-case (no blocking in the beginning of execution phase). The scaling of $U_{avail} \geq U_{req}$ ensures that the extra amount of work done per unit time will be greater than or equal to $\frac{t_c}{T_i}$. Afterwards, the requested utilisation is used to compute the lengths of new cooling t_c^{ud} and active phases t_a^{ud}.

$$U_{req} \stackrel{\text{def}}{=} \frac{C_i}{T_i} + \frac{t_c}{T_i} \tag{4.47}$$

- *Condition 2:* The schedulability of the single task is ensured if its minimum inter-arrival time satisfies the condition given in Equation 4.48, where $mod(a,b)$ represents the modulus operator and provides the remainder of $\frac{a}{b}$. Equation 4.48 computes the number of active phases required to execute the task and adds the corresponding cooling phase, and ensures

it is greater than the minimum inter-arrival time and relative deadline of the task to preserve the schedulability of the system.

$$T_i \quad > \quad \left\lfloor \frac{C_i}{t_a^{ud}} \right\rfloor \left(t_a^{ud} + t_c^{ud} \right) + \left(mod(C_i, t_a^{ud}) \right) + t_c^{ud} \tag{4.48}$$

Both Equation 4.47 and Equation 4.48 are sufficient conditions.

Case 2) Multiple Tasks Case: This analysis is extended to multiple sporadic tasks to ensure their schedulability. Similar to the single task case, the schedulers needs to satisfy two conditions.

- *Condition 1:* First of all, a slight modification is made in U_{req} as given in Equation 4.49. The additional factor corresponding to the blocking in the cooling phase $\frac{t_c}{T_i}$ is replaced with $\frac{t_c}{\min\limits_{\forall \tau_i \in \tau} (T_i)}$. For each period of the highest priority task the amount of extra work will be equal to $U_{avail}\ t_c$. Similar to a single task case, the value of t_c is obtained by considering the ideal-case and the original value of U_{avail} is raised to U_{req} to ensure the schedulability of all tasks in the given task-set. Again, the lengths of cooling t_c^{ud} and active t_a^{ud} phases are determined based on this new value of U_{req}.

$$U_{req} \quad = \quad \sum_{\forall \tau_i \in \tau} \frac{C_i}{T_i} + \frac{t_c}{\min\limits_{\forall \tau_i \in \tau} (T_i)} \tag{4.49}$$

- *Condition 2:* In the multiple tasks case, all the tasks should satisfy the condition given in Equation 4.50 to check that they are getting enough active phases in their minimum inter-arrival time to compete their execution to ensure the schedulability.

$$\forall \tau_i \in \tau, \qquad T_i \quad > \quad \left\lfloor \frac{C_i}{t_a^{ud}} \right\rfloor \left(t_a^{ud} + t_c^{ud} \right) + \left(mod(C_i, t_a^{ud}) \right) + t_c^{ud} \tag{4.50}$$

The quantisation error that occurs in TCDPM due to cooling and active phases is bounded to $\frac{t_c}{\min\limits_{\forall \tau_i \in \tau} (T_i)}$. This is a pessimistic but safe bound. Similar to single task, Equation 4.49 and Equation 4.50 are sufficient conditions.

Lets consider the other effects (that may affect the schedulability of tasks) such as if a task is executing with a worst-case scenario and other tasks are released during its execution. The arriving task may have higher or lower priority when compared to the currently executing task. If there is an arrival of a lower priority task(s) the normal execution is not interrupted at all as it has to wait for the currently running task to complete its execution. Now consider the effect of the higher priority task τ_i. The schedulability of the higher priority task τ_i is ensured by Equation 4.50. The

phasing of τ_i with respect to the phasing of the cooling is of no concern as the overall execution requirement is only increased by C_i. Similarly, it can be shown that by adding extra tasks, the schedulability of the system remains unaffected.

4.5.4 Case Study

This section shows that TCDPM problem can be solved with existing DVFS algorithms. For demonstration purpose, two DVFS algorithms for RT systems from the work of Pillai and Shin [PS01] are considered in this case study. It is assumed all the frequency set-points of the processor are normalised with the maximum frequency of the processor.

4.5.4.1 Static Allocation of Frequency

In the first algorithm of Pillai and Shin [PS01], it is assumed that all the tasks execute for their worst-case and they find statically the operating frequency of the processor. The operating frequency f_o of the processor is set to Uf_{V^m} and the normalised frequency of the system is equal to $\frac{f_o}{f_{V^m}} = U$. The execution time of all the tasks are scaled by a factor of $\frac{1}{f_o}$. As it have been mentioned in previous section that the available utilisation U_{avail} in TCDPM corresponds to the normalised speed of the processor in DVFS. The duty cycle of the system in TCDPM should be set greater than or equal to the normalised frequency of the system in DVFS to get an equivalent system. To do so, the requested utilisation U_{req} that is a summation of total utilisation of the given task-set and an additional error factor to compensate the potential additional delay of the cooling phase is computed for the given system. After computing the requested utilisation of the system, the value of available utilisation is set to $U_{avail} \geq U_{req}$. This new selected value of U_{avail} in turn is used to estimate the active and cooling phase durations. Afterwards, periods of all the tasks are checked for the condition given in Equation 4.50 to ensure the temporal correctness of the system. The duty cycle achieved with the estimated active and cooling phases is greater than or equal to the normalised frequency of the system in DVFS and ensures that the task-set gets enough time to execute the given workload without missing any deadlines.

4.5.4.2 Dynamic Allocation of Frequency

Pillai and Shin [PS01] have exploited the execution slack to further reduce the operating frequency. On the early completion of any task the unused execution time is reclaimed and the utilisation of the system is recomputed by considering the actual execution time of the current task. The operating frequency is set accordingly with this newly computed system utilisation. The individual utilisation of the task considering its actual execution time is used until its next arrival. On any task arrival, the system utilisation is computed again by replacing the previous individual utilisation of the currently arrived task with $\frac{C_i}{T_i}$. The operating frequency is changed accordingly. This algorithm does the frequency adjustment on the task arrival and on its completion.

Similar to Pillai and Shin's approach [PS01] , TCDPM should also make decisions about changing U_{avail} at the arrival and the completion of all tasks. For the temporal correctness, U_{avail} should be greater than or equal to U_{req} (i.e., $U_{avail} \geq U_{req}$). U_{req} is composed of two components. The first component computes the current utilisation of the system, while second factor considers the effect of potential blocking due to the cooling phase. A change in current utilisation of the system will vary the cooling phase, which in turn will affect the blocking time (i.e., second factor in U_{req}). To eliminate this issue, it is assumed that $t_c^{\max}$ is the maximum possible cooling time in the system. This value can be estimated by setting $Tm_{\max}$ and Tm_o to their feasible extremes (i.e., $Tm_{\max} = Tm_{cri}$ and $Tm_o = Tm_{amb}$). In theory the value of $t_c^{\max}$ can reach to infinity if Tm_o is set equal to Tm_{amb}. Therefore, for practical purposes Tm_o can be set to a value $Tm_{amb} + t_{th}$, where t_{th} is a small offset to keep $t_c^{\max}$ in a reasonable limit. If $\min_{\forall \tau_i \in \tau}(T_i) \gg t_c^{\max}$, then second component in U_{req} equation can be replaced with $\frac{t_c^{\max}}{\min_{\forall \tau_i \in \tau}(T_i)}$. Any task in a task-set cannot suffer from a blocking greater than $t_c^{\max}$. The first component of U_{req} equation (that estimates the current required utilisation) can be computed in a similar way as computed in Pillai and Shin's approach [PS01]. However, there is just one exception, if a task arrives in the cooling phase, then the processor needs to wait for the completion of the current cooling phase to make decision about the new U_{avail}.

Reducing Pessimism: The blocking factor of $\frac{t_c^{\max}}{\min_{\forall \tau_i \in \tau}(T_i)}$ in U_{req} equation is a pessimistic bound. The tasks rarely face such a large blocking time. Another less pessimistic approach is also presented to compute U_{req}. Assume, the previous cooling phase has a length of t_c^{old}. On every task completion or new task arrival in the **active phase**, the individual utilisation U_i of the task is updated and the total system utilisation is recomputed. Considering this new value of total system utilisation, the potential length of the next cooling phase is estimated and denoted as t_c^{new}. The value of U_{req} is set to $\sum_{\forall \tau_i \in \tau} U_i + \frac{t_c^{new}}{\min_{\forall \tau_i \in \tau}(T_i)}$. However, if there is a new task τ_i arrival in the **cooling phase** of the system, its processing is postponed by the end of this cooling phase. At the end of the cooling phase, the total system utilisation is computed by considering τ_i's worst-case execution time and the value of t_c^{new} is determined. If t_c^{new} is shorter than the current cooling phase time, than τ_i has suffered an extra delay. To compensate for this extra delay, its individual utilisation U_i is set to $\frac{C_i + \max(t - r_{i,k} - t_c^{new}, 0)}{T_i}$, where $r_{i,k}$ is the absolute release time of τ_i and t is the current time instant at the end of cooling phase. With this new value of U_i and t_c^{new}, the value of U_{req} is computed as $U_{req} = \sum_{\forall \tau_i \in \tau} U_i + \frac{t_c^{new}}{\min_{\forall \tau_i \in \tau}(T_i)}$. U_{avail} is then set to any feasible value greater than or equal to U_{req} and the corresponding values of t_c and t_a are computed. Note that the effective values for t_c and t_a are computed based on this new value of U_{avail} and the intermediate values of t_c^{new} used to estimate U_i is ignored.

One more concern that scheduler needs to deal with is the idle mode. If a processor has no workload to execute, it transitions into a sleep mode. It is equivalent to the early start of a cooling phase. However, the sleep state is terminated on the arrival of a new task. The delay caused due

to this sleep transition can be included in the individual utilisation of the arrived task and that is $U_i = \frac{C_i + tw_1 + ts_1}{T_i}$. Such additional overhead can be ignored, if the processor has an idle mode with zero transition delay to and from active mode. Similar to the examples given in this case study, any other DVFS algorithm can be similarly ported and applied in TCDPM setting.

4.5.5 Implementation Concerns

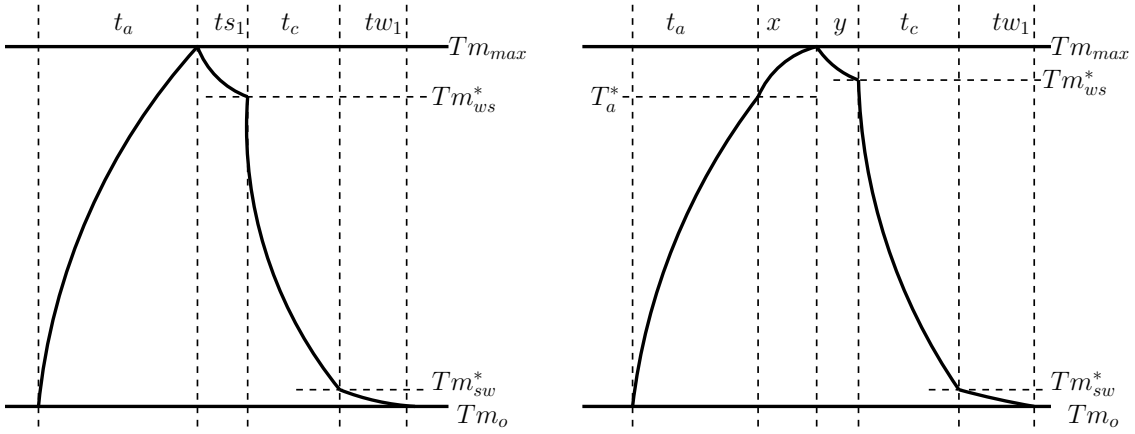
4.5.5.1 Computation of U_{avail}, Tm_o and $Tm_{\max}$

In order to reduce the online complexity of the algorithm, an offline table for U_{avail} is computed that contains the corresponding values of $Tm_{\max}$, Tm_o, t_a and t_c. Given the values of $Tm_{\max}$ and Tm_o, the values of t_c and t_a can be easily computed for the required table. The values of $Tm_{\max}$ and Tm_o against U_{avail} can be computed through various techniques such as exhaustive exploration, dynamic programming, approximation algorithm in which a value of $Tm_{\max}$ is fixed and Tm_o is varied to get different values of U_{avail}. The values of this table are platform dependent only and are estimated once for the given platform. This table reduces the online complexity of the algorithm to $O(log_2(x))$ to obtain $Tm_{\max}, Tm_o, t_a$ and t_c against U_{avail}, where x is the number of U_{avail} entries in the table. The length of this table defines the resolution of U_{avail}. In case of non-linear relation of U_{avail} and the energy consumption, the efficient distribution is to get high resolution of U_{avail} where the rate of change of energy consumption is high.

4.5.5.2 Transition Overheads of the Sleep State

Equation 4.45 assumes the sleep state has no overhead. However, in reality each sleep transition has a time and energy overhead. The energy overhead comes from the fact that it has to store the current status of the system (e.g. cache write-backs). These overheads may have an impact on the system temperature, which in turn also affect the available utilisation. Two different cases are considered as given below.

Transition Phase Decreases the Temperature: Consider a case when the temperature decreases as the processor transitions into a sleep state as shown in Figure 4.27-*i*. The active state of the processor is unaffected. The cooling phase may be affected as the complete circuitry is not off during the transition phase and the cooling time may be different from the cooling behaviour in the sleep state. Therefore, the cooling phase is divided into three intervals as shown in Figure 4.27-*i*. The curves given in Figure 4.27 are arbitrary and their main purpose is to illustrate the differences. In the start of the cooling phase the processor transitions into a sleep state for ts_1 time units and temperature at the end of this transition is Tm^*_{ws}. The sleep state lasts for t_c time units between Tm^*_{ws} and Tm^*_{sw}. The value of Tm^*_{sw} is set such that a system takes tw_1 time units to approach Tm_o in a transition out phase. The available utilisation in this case can be represented as given in Equation 4.51.



i) Temperature decrease *ii) Temperature increase*

Figure 4.27: Temperature decreases or increase in transition phase

$$U_{avail} = \frac{t_a}{t_a + t_c + ts_1 + tw_1} \tag{4.51}$$

Transition Phase Increases the Temperature: In a case, when the temperature of the system increases as a processor transitions into a sleep state, the execution phase is stopped such that it does not cross $Tm_{\max}$ (Figure 4.27-*ii*). The transition into a sleep state is divided into two parts. 1) Let x be the time unit processor takes to do the processing related to the transition of the sleep state (e.g., cache write-backs, IRQ to serve sleep request). 2) y is the transition time needed to initiate the sleep state after the processing phase and Tm^*_{ws} is the temperature after y time units. Assume T^*_a is the temperature such that if the sleep state is initiated it finishes its sleep related processing before the temperature reaches to $Tm_{\max}$. In this case, U_{avail} is computed as shown in Equation 4.52.

$$U_{avail} = \frac{t_a + x}{t_a + x + y + t_c + tw_1} \tag{4.52}$$

4.6 Evaluation of Thermal-Aware Energy Management Approach

This section analyses the behaviour of the algorithms presented in the case study (Section 4.5.4) across different dimensions. The results presented in the work of Pillai and Shin [PS01] of the chosen algorithms are compared with the same algorithms ported in TCDPM setting. This comparison shows that the trend of energy saving is consistent demonstrating the equivalence of the idealised DVFS and TCDPM. Note that this result section does not show the direct energy saving comparison of idealised DVFS and TCDPM rather demonstrates the fact that idealised DVFS

Parameters	Values
Task-set sizes $\|\tau\|$	$\{5, \underline{10}, 15, \ldots, 50\}$
Inter-arrival time T_i for RT tasks	$[30ms, 50ms]$
Sporadic delay limit $\Gamma \in$	$\{\underline{0}, 0.05, 0.1, \ldots, 1\}$
BCET limit C^b	$\{\underline{0.2}, 0.25.0.3, \ldots, 1\}$
System utilisation U	$\{0.35, 0.40, 0.45, \underline{0.5}, \ldots, 0.7\}$
Energy overhead Es_1 (mJoules)	$\{\underline{10}\}$
$\hat{\alpha}$ (K/Joules)	$\{26, 27, 28, \ldots, 35, \underline{35.62}\}$
Dynamic power P_{dyn} (Watts)	$\{0.5, 1, 1.5, \ldots, \underline{5}, \ldots, 9\}$

Table 4.4: Overview of simulator parameters used to evaluate thermal-aware energy management algorithms

can be applied in TCDPM setting to save energy. The amount of energy saved in both cases is obviously different and depends on a number of different hardware parameters.

The SPARTS simulator discussed in Section 3.2 is extended to incorporate the thermal-aware models and the approached presented in the case studies. It is used with the following parameters given in Table 4.4 with the default values underlined. In the context of this research, only hard real-time type tasks are considered. The values of $\hat{\beta}$, P_1, $ts_1 = tw_1$, $\mathscr{A}$, $\mathscr{B}$, Es_1, Tm_{amb} and $Tm_{\max}$ are adopted from Yang et al. [YCTK10] work and are fixed to $9.52/$ sec, 50μ Watt, 5 msec, 0.0002188 Watt$/K^2$, -8.5143 Watt, 10 mJoules, 300 K and 373 K respectively. Other parameters such as P_{dyn}, $\hat{\alpha}$ are considered as a variable in different set of experiments to vary the hardware platform behaviour.

All the results presented below are normalised to the highest value in the respective graph. The default value of the parameter is considered if not mentioned in the description of the experiment. The static frequency allocation algorithm, dynamic frequency allocation algorithm and dynamic frequency allocation algorithm with reduced pessimism are labelled as SFA, DFA and DFA-LP respectively. As the results section frequently refers to the simulations results of Pillai and Shin [PS01], therefore, it is important to mention that SFA and DFA in an idealised DVFS setting are termed as static EDF (staticEDF) and cycle-conservative EDF (ccEDF) respectively. It has been observed that the difference of overall energy consumption between DFA and DFA-LP is negligible with few exceptions. Therefore, only the comparison of SFA and DFA is presented, while DFA-LP is only mentioned where it makes considerable difference when compared to DFA.

Initially, the effect of change in the system utilisation is studied for three different approaches. Figure 4.28 presents the results for a task-set size of 10 with $\Gamma = 0$ and $C^b = 0.2$. The increase in system utilisation obviously increases the energy consumption. This trend is consistent with the results of Pillai and Shin [PS01], where, the energy consumption of staticEDF and ccEDF also increases with the system utilisation. The interesting fact that is evident from Figure 4.28 is the difference of energy consumption of SFA and DFA. SFA only exploits the spare capacity available in the system schedule called static slack and computes U_{avail} considering WCET of the tasks. On the other hand, DFA makes use of the static slack in the offline phase, and utilises the execution slack while recomputing U_{avail} online on early completion of all tasks. The execution slack helps

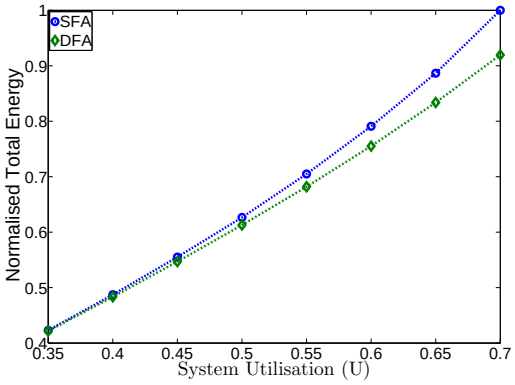


Figure 4.28: Variation in system utilisation

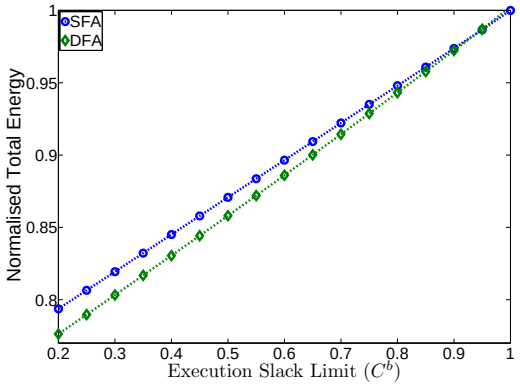


Figure 4.29: Variation in execution slack

to further reduce the energy consumption by decreasing the demand on U_{avail}. The same reason explains the difference of energy consumption at a utilisation of $U = 1$. DFA consumes approximately 9% less energy when compared to SFA. Furthermore, SFA and DFA behave identical if it is assumed that all the tasks execute for their WCET. The same is true for staticEDF and ccEDF.

The effect of variation in the execution slack available in the schedule with $\Gamma = 0$, $U = 0.5$ and task-set size of 10 is analysed in Figure 4.29. The value of C^b is varied from 0.2 to 1. The increase in the value of C^b means a decrease in the execution slack. At $C^b = 1$, the amount of execution slack becomes zero as all tasks execute for their WCET. Therefore, at such utilisation ($U = 1$) both algorithms (SFA and DFA) perform identical due to unavailability of any source of slack and the same observation holds for staticEDF and ccEDF. In general, the energy consumption increases with a decrease in the amount of execution slack in schedule. The similar trend is followed by the energy consumption of staticEDF and ccEDF in the idealised DVFS that verifies the equivalence. The DFA algorithm exploits execution slack by adapting U_{avail} to a low value according to the system requirement. Nevertheless, SFA only makes use of the execution slack by initiating early start of the cooling phase and keeping the processor in a sleep state till the arrival of a new task. However, it does not readjust U_{avail} to decrease the energy consumption. Likewise to DFA, ccEDF can reduce the processor's frequency with execution slack to save the energy. The energy consumption of staticEDF is not affected with a variation in the execution slack as Pillai and Shin [PS01] assumed in their experimental setup that energy consumption of the processor is zero in the idle phase with negligible transition overhead. The earlier start of cooling phase helps to save energy in SFA.

The effect of the task-set size is analysed in Figure 4.30. The DFA-LP algorithm is also included in this experiment for comparison. The task-set size is varied from 5 to 50. This experiments shows the difference of energy consumption does not depends on the task-set size as the difference is negligible. For instance, the maximum difference of energy consumption exists between a task-set size of 35 and 45, and is equal to $\approx 1\%$. The main factors are the system utilisation and the amount of dynamic slack in the schedule. The staticEDF and ccEDF also behave similar for different task-set sizes.

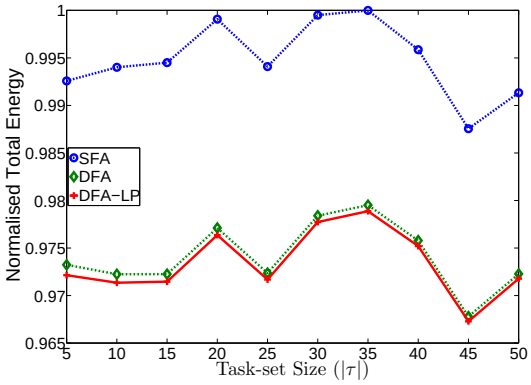


Figure 4.30: Variation in number of tasks

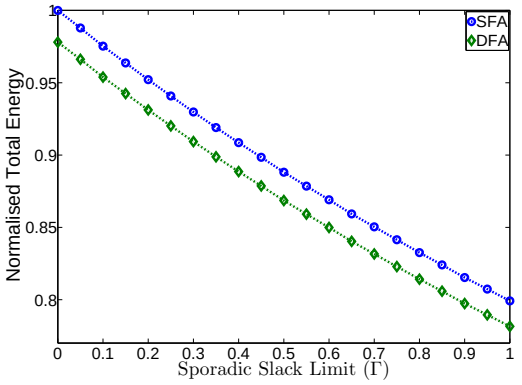


Figure 4.31: Variation in sporadic slack

Figure 4.31 presents the behaviour of the system towards sporadic slack in the schedule, which is not investigated in the work of Pillai and Shin [PS01] due to the assumed periodic task-model. The task-set size, C^b and U are fixed to default values. Γ is varied from 0 to 1 with a step size of 0.05. The increase in the value of Γ increases the sporadic slack. Normally, it is impossible to determine sporadic slack beforehand. Therefore, SFA and DFA make use of such slack in the cooling phase by extending it till next task arrives. The behaviour of all the algorithms is approximately linear with the variation of Γ. The energy consumption reduces to 20% from $\Gamma = 0$ (no sporadic slack) to $\Gamma = 1$. Hence, the proposed algorithms can exploit sporadic slack implicitly.

After analysing the task-level properties, the hardware platform specific parameters ($\hat{\alpha}$ and P_{dyn}) are considered. The hardware platform specific parameters $\hat{\alpha}$ presented here is not evaluated by Pillai and Shin [PS01] as it is specific to the temperature model and is not relevant in DVFS case. This parameter is discussed to demonstrate the behaviour of the TCDPM algorithms for different types of hardware platforms.

$\hat{\alpha}$ is the inverse of thermal capacitance C_{th} and has a unit of $K/Joules$. The higher value of $\hat{\alpha}$ implies that the hardware platform heats up quickly, but the cooling phase is independent of it. Figure 4.32 shows the variation in the energy consumption with different values of $\hat{\alpha}$. The values of P_{dyn} and Es_1 are fixed to 5 Watts and 0.01 Joules respectively. The processor heats up at a fast rate for a large value of $\hat{\alpha}$, therefore, the active phase shortens with an increase in the value of $\hat{\alpha}$. This decreases the leakage-power dissipation, as the processor stays for a shorter period of time at high temperatures. Oppose to this, a low of value of $\hat{\alpha}$ has long active period and it takes longer to heat up the processor. The processor though executes more but consumes more leakage energy. Therefore, Figure 4.32 shows an increasing value of $\hat{\alpha}$ decreases energy consumption. The difference in energy consumption of SFA and DFA increases at high values of $\hat{\alpha}$. This is motivated by the fact that when the processor heats up quickly it is important to exploit the available slack consciously. On one side, the processor's active phase is shorter and hence, its leakage-power dissipation reduces. On the other side, the decrease in active phase time means processor needs to operate at high U_{avail} values, i.e., short cooling phase. Hence, the effect of execution slack becomes important to extend the cooling phase by reducing the duty cycle i.e., U_{avail}.

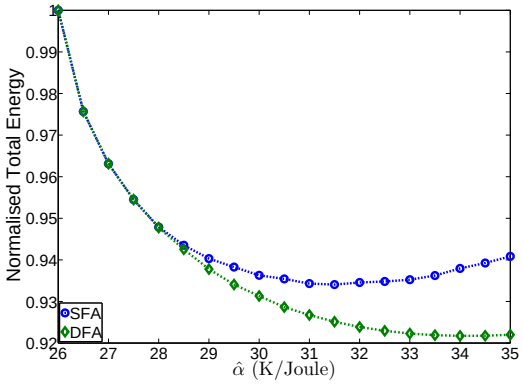


Figure 4.32: Variation in $\hat{\alpha}$

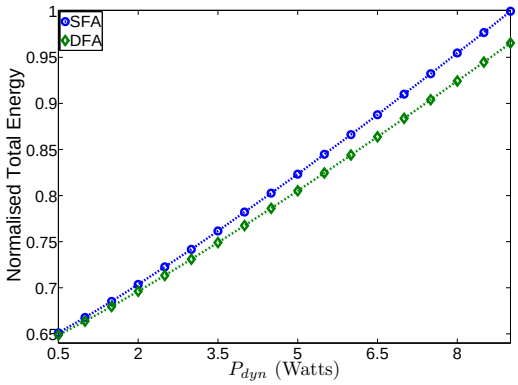


Figure 4.33: Variation in P_{dyn}

The thermal behaviour of the processor is modified by increasing the dynamic power dissipation. Both leakage and dynamic power dissipation contribute to the temperature increase. Varying the dynamic power dissipation not only changes the thermal behaviour but also the varies the ratio of dynamic to leakage-power dissipation. The dynamic power dissipation is varied from 0.5 Watts to 9 Watts. The leakage-power dissipation at ambient temperature is 11.178 Watts. Therefore, the ratio of P_{dyn} to P_{lkg} varies between 0.0447 to 0.8052. As can be seen in Figure 4.33, the increase of dynamic power increases the energy consumption. There is an approximately 35% rise in energy by varying P_{dyn} from 0.5 Watts to 9 Watts. Furthermore, the difference in energy consumption of SFA and DFA also increases with an increase in dynamic power. The increase in dynamic power increases the temperature and it heats up at a fast rate. Therefore, the active phase of the processor decreases and that consequently decreases the length of cooling phase. The leakage-power dissipation parameters are not altered, hence, it varies proportionally and does not affect the behaviour. The shortening of active and cooling phase enhances the need for effective management of U_{avail}, which is obviously better in DFA when compared to SFA. Therefore, DFA performs better at high dynamic power dissipation. A similar parameter called idle level (ratio of energy consumption in idle cycle to energy consumption in active cycle) is explored by Pillai and Shin [PS01]. If the power dissipation in idle mode is considered constant, then idle level essentially means a variation in dynamic power. The increase in idle level mean decrease in dynamic power and vice versa. Pillai and Shin [PS01] showed that a decrease in the idle level increases the energy consumption of staticEDF and ccEDF. This is consistent with the results presented in Figure 4.33 and shows that an increase in dynamic power (decrease in idle level) increases the energy consumption of SFA and DFA.

Each sleep transition has an energy overhead associated to it, which is modelled as Es_1 in the given system model. The frequent sleep transitions are undesirable and increase the energy consumption. The number of sleep transitions of the previous experiments is shown in Figure 4.34. In this experiment, the normalised sleep transitions of DFA-LP are also included in the comparison as well. An increase in dynamic power heats up the processor quickly and decreases the active phase, which in turn also increases the number of sleep transitions. This effect is evident from

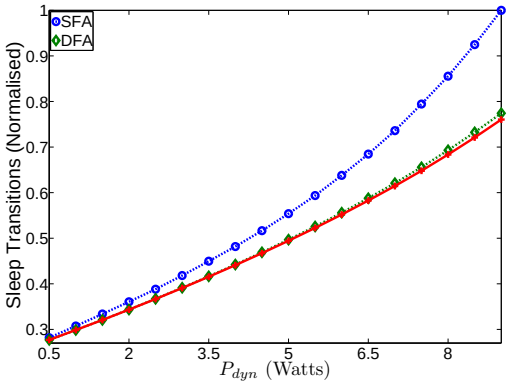


Figure 4.34: Number of sleep transitions

Figure 4.34 that shows an increase in the number of sleep transition with an rise in dynamic power dissipation. As mentioned previously, DFA and also DFA-LP manage U_{avail} effectively and extend their cooling phase, hence their number of sleep transition are fewer when compared to SFA. Especially at high dynamic power dissipation, this difference reaches to approximately 22%. Furthermore, the difference of DFA and DFA-LP is also visible in this experiment at high value of dynamic power dissipation that shows the minor gains of the proposed optimisation.

Chapter 5

Device Power Management

An ever increasing need for extra functionality in a single embedded system demands for extra I/O devices, which are usually connected externally and are expensive in terms of energy consumption. State-of-the-art avionics, automotive electronics, mobile phones and control systems are typical domains in which variety of different I/O devices are deployed to perform distinct functions. As mentioned previously, these I/O device are of different types including memories, LCD displays, touch screens, amplifiers, microphones, accelerometer, gyroscope, compass, camera, temperature sensor, barometer, routers etc. Though a processor is a large source of energy consumption but the current trend of increasing number of I/O devices have often include I/O device as a major contributor in energy consumption of embedded systems [CH10]. Therefore, I/O devices are of particular concern in embedded systems and provide extra opportunities to reduce energy consumption. Similar to a processor, the increased energy consumption of I/O device can be reduced through power saving state(s) by turning-off certain parts of the device.

In real-time systems, the device switching must be perform consciously to avoid any task missing its deadline incurring due to the time penalty associated to transition between two different states of a device. The request instant and access interval of the I/O devices within tasks execution time is usually not easy to determine beforehand considering the dynamic nature of modern applications. In order to guarantee temporal correctness of a RT system with I/O devices, the device transition delay to bring the device up from a sleep state needs to be taken into account in the schedulability analysis. Considering the variability of a device usage instant and access interval, traditional device-scheduling algorithms made a safe but pessimistic assumption to enable the active state of all the devices used by a task throughout its active time. This category of device scheduling is known as inter-task device scheduling. However, in most cases, I/O devices are used for a very short duration of time. For instance, consider any image processing application on embedded platforms (face/thumb recognition) that reads a image and afterwards, the majority of processing is performed to extract the required features from the sampled data. Therefore, inter-task device scheduling wastes substantial amount of energy. In contrast to this, intra-task device scheduling wakes device up when it is requested by a task. However, online intra-task device scheduling for HRT systems has not been explored in the past.

To ensure temporal correctness, the scheduler has to compensate for the transition delays of a device requested on demand, as the corresponding task's execution is suspended until its associated device reaches the active state. Technological advancements have decreased the overheads associated with a device by several orders of magnitude, for example, a solid state storage device has an extremely low overhead when compared to the conventional storages disks. The reduced overhead of sleep states and the pessimism involve in the inter-task device motivates to explore this new paradigm of intra-task device scheduling for the RT systems.

This chapter presents an intra-task device scheduling algorithm for RT systems that wakes up a device on demand and reduces its active time while ensuring system schedulability. This intra-task device scheduling algorithm exploits different sources of slack and extended for devices with multiple sleep states to further minimise the overall device energy consumption. The proposed algorithms have lower complexity when compared to the conservative inter-task device scheduling algorithms. The system model used relaxes some of the assumptions commonly made in the state-of-the-art that restrict their practical relevance. Apart from the aforementioned advantages, the proposed algorithms are shown to demonstrate substantial energy savings.

5.1 Preliminaries

This work assumes a sporadic constrained-deadline task-model with the specifications mentioned in Section 3.1.1. It also employs the RBED framework [BBLB03] and the parameters corresponding to the RBED framework are mentioned in Section 3.1.2. A unicore platform with a set of devices λ is assumed in this work. The parameters of a device are given in Section 3.1.3.2. A task τ_i uses a device λ_i exclusively, i.e., each device λ_i is associated to exactly one task and no sharing is allowed. It also means a task cannot access more than one devices. The example of such a system is device drivers in microkernel based system, where each device driver task is associated to one device. For simplicity sake, the number of devices in the system are consider equal to the number of tasks, i.e., $W = \ell$. However, this is not a restriction. For the ease of notation, in this chapter, an extra parameter is added into a task tuple. The task tuple is defined as $\tau_i \stackrel{\text{def}}{=} \langle C_i, D_i, T_i, \lambda_i \rangle$. This extra parameter is the device λ_i used by a task τ_i.

It is assumed a device is used once during the execution of a job, but the exact time instant of the device usage along with its duration within a job's execution time is not known a-priori. Even if the device usage instant is known within the task's execution time, there are also other factors that cause an inherent variability of the device usage instant. For instance, an execution of the task τ_i instance using a device λ_i can be delayed by higher priority jobs causing variable execution times delays. With such an inherent variability, it is complex to find the exact instant of the device usage time beforehand in the schedule. The optimal solution can only be determined with the complete information about the schedule, which cannot be determined in realistic scenarios due to the inherent variability in the sporadic task model. Therefore, this work focuses on different heuristics for the device scheduling problem and models a realistic behaviour of applications commonly deployed on RT systems with the given system-model.

In intra-task device scheduling, a device is only woken-up on demand to reduce its active time which consequently minimises its energy consumption. However, the transition time imposes an extra overhead and alters the schedule, as the task has to wait for a device to become active. An alternative solution of inserting extra wake-up calls ahead of the device usage into the application code is impractical. A slack management algorithm in the scheduler is needed to collate the idle intervals. The proposed intra-task device scheduling algorithms explicitly collect and utilise two main sources of slacks as given below.

Definition 35. *[Device Budget] The device budget D_b of the system is the maximum available spare time in the schedulability test that can be used to compensate for devices transition delays without causing any application to miss its deadline under worst-case assumptions.*

The device budget comes from static slack in the schedule. It is formally presented in Definition 35. A lower bound on the size of the device budget is determined by considering the temporal correctness of the schedule and the static sleep interval or minimum idle interval χ_{min} defined in Section 4.1.4 and quantified in Equation 4.26 is used as a device budget D_b (i.e., $D_b = \chi_{min}$).

The execution slack S_e is also explicitly exploited along with the device budget D_b. The slack management algorithm presented in Section 3.1.5 is also used to collate the execution slack. However, the method 2 (extending slack deadline on job arrival) in the slack preservation phase is opted in this work. The proposed intra-task device scheduling algorithms do not depend on this simplistic slack reclamation algorithm. Any existing slack management algorithm can be integrated with minimal effort as only the size and the deadline of the slack is important irrespective of the method collecting it. Moreover, the consumption of sporadic slack is implicit within proposed algorithm and will be explained in later sections.

5.2 A Single Sleep State per Device Model

Considering the complex nature of the problem, initially, it is assumed in this section that each device λ_i has only a single sleep state $\S_1^{\lambda_i}$. Therefore, λ_i can only transition into and out of $\S_1^{\lambda_i}$. The same problem with multiple sleep state devices has extra challenges which will be addressed in Section 5.4. This section presents an intra-task device scheduling algorithm called static slack container but before going into the details of this algorithm, some of the notations and symbols used throughout the coming sections are introduced below with their brief description.

- ϕ : The set of all sleep states of devices in the system.
- Φ: The set of intra-task device-scheduling compatible sleep states that have sufficient capacity to transition into and out of a sleep state before their next arrival, i.e., $\Phi \stackrel{\text{def}}{=} \{\S_n^{\lambda_j} \in \phi : D_j - C_j \geq tsw_n^{\lambda_j}, \forall j = \{1,\ldots,W\}, \forall n \in j\}$.
- λ_i^{NUT}: The next utilisation time of any device λ_i. This value is the next expected release time of the job $j_{i,k}$ using device λ_i. It is computed with reference to the previous release information $r_{i,k}$, i.e., $\lambda_i^{NUT} = r_{i,k} + T_i$.

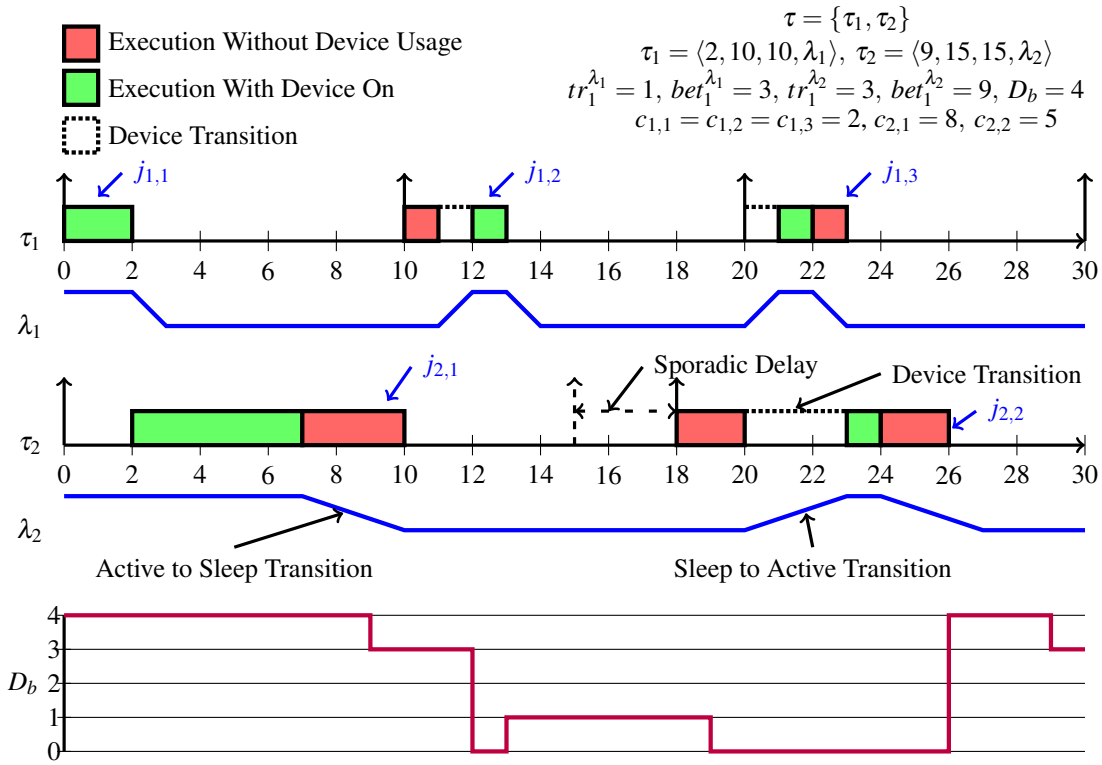


Figure 5.1: Example with two tasks ($\tau_1 = \langle 2, 10, 10, \lambda_1 \rangle,\ \tau_2 = \langle 9, 15, 15, \lambda_2 \rangle$)

- SSSR: The static slack service register contains references to inactive devices currently in a sleep state and that have acquired their wake-up budget from D_b. These devices are waiting for the requests from their corresponding jobs.

- Ξ_i^t: The amount of pending workload of higher priority than a job $j_{i,k}$ currently residing in the ready queue at time instant t. In other words, all released jobs currently residing in the ready queue at time t having deadline less than or equal to $d_{i,k}$.

- ESSR: The execution slack service register contains references to all inactive devices currently in a sleep state prolonging their wake-up time with execution slack S_e.

Example 5. *The running example depicted in Figure 5.1 is used throughout this section to visualise and explain the different concepts involved in the static slack container algorithm presented in Section 5.2.1. The task-set in this example is composed of two tasks. The parameters of these two tasks are given as follows:* $\tau = \{\tau_1 = \langle 2, 10, 10, \lambda_1 \rangle, \tau_2 = \langle 9, 15, 15, \lambda_2 \rangle\}$. *Both devices* λ_1 *and* λ_2 *have a single sleep state.* λ_1*'s sleep state* $\S_1^{\lambda_1}$ *has a transition delay* $tr_1^{\lambda_1} = 1$ *and a break-even-time* $bet_1^{\lambda_1} = 3$. *Similarly,* λ_2*'s sleep state* $\S_1^{\lambda_2}$ *has* $tr_1^{\lambda_2} = 3$ *and* $bet_1^{\lambda_2} = 9$. *Moreover, device budget* D_b *of the given task-set is computed to be* 4, *i.e.,* $D_b = 4$. *All jobs of* τ_1 *execute for* 2 *time units. The first job* $j_{2,1}$ *of* τ_2 *executes for* 8 *time units, and second job* $j_{2,2}$ *executes for* 5 *time units.*

5.2.1 Static Slack Container Algorithm (SSC)

The static slack container algorithm (SSC) uses the device budget D_b and the execution slack S_e to compensate for devices transition delays and maintain the schedulability with a global objective to minimise the devices energy consumption. The pseudo-code of the SSC algorithm is shown in Algorithm 6. This algorithm has different routines corresponding to different situations. These routines are explained as follows.

5.2.1.1 Offline Phase

Initially, the proposed algorithm identifies the set of intra-task device scheduling compatible devices using Definition 36. In the given example shown in Figure 5.1, all the devices are intra-task device-scheduling compatible. The non-compatible devices may be shut-down, only in a case, when their corresponding tasks execute for less than their C_i and $D_i - c_{i,k} \geq tsw_1^{\lambda_i}$. However, a scheduler needs to ensure the wake-up of non-compatible devices before their corresponding task starts its execution.

Definition 36 (Intra-task Device Scheduling Compatible Device)**.** *A device λ_i associated to a task τ_i will be compatible with intra-task device scheduling if its sleep state $\S_1^{\lambda_i}$ belongs to the set Φ, i.e., $\lambda_i \in \lambda : \S_1^{\lambda_i} \in \Phi$.*

5.2.1.2 Scheduling in Static Slack Container

On a system boot usually all the devices are in active mode. If a running $j_{i,k}$ requests the associate device λ_i and λ_i is in active mode, the job will continue its execution. In our example (Figure 5.1), $j_{1,1}$ and $j_{2,1}$ find their corresponding devices active and continue their execution. However, if λ_i is in a sleep mode, then $j_{i,k}$ is pre-empted and inserted into the device waiting queue. Once the interrupt service routine (ISR) signals that λ_i is in active mode, $j_{i,k}$ is enqueued again in the ready queue and scheduled according to its priority. For instance, $j_{1,2}, j_{1,3}$ and $j_{2,2}$ in Figure 5.1 wait for their corresponding devices to transition out of their sleep state and use them in the interval $[12,13], [21,22]$ and $[23,24]$ respectively. On every job $j_{i,k}$ release, the next earliest utilisation time λ_i^{NUT} of its device λ_i is updated as $\lambda_i^{NUT} = r_{i,k} + T_i$.

5.2.1.3 Device Shut-Down

Once $j_{i,k}$ has completed its use of λ_i, the driver tries to shut it down. The proposed algorithm takes an opportunistic approach in case a sleep state of $\lambda_i \in \Phi$ and performs the shut down when the difference of next utilisation time λ_i^{NUT} of λ_i and current time instant t is greater or equal to its total transition delay $tsw_1^{\lambda_i}$ (i.e. $\lambda_i^{NUT} - t \geq tsw_1^{\lambda_i}$). We are speculative to considered $tsw_1^{\lambda_i}$ instead of $bet_1^{\lambda_i}$ with an expectation that slack will be eventually available in near future. A timer is set to $\lambda_i^{NUT} - tr_1^{\lambda_i}$ to wake up the device before λ_i^{NUT}. Note that λ_i's wake-up procedure is only performed at $\lambda_i^{NUT} - tr_1^{\lambda_i}$ when D_b or S_e is not sufficient to postpone the device wakeup further. This device wake-up procedure is described in Section 5.2.1.4. Lines 4 to 20 in Algorithm 6

Algorithm 6 Static Slack Container Algorithm (SSC)

Offline Phase
Identify the intra-task scheduling compatible devices. $\lambda_i \in \lambda : \S_1^{\lambda_i} \in \Phi$
Calculate the device budget D_b for a given task-set τ
Device Shut-Down Procedure:
When $j_{i,k}$ has used λ_i, consider the following criteria to shut-down λ_i and set the corresponding entry in the sorted list of timer.
if ($\lambda_i \in \lambda : \S_1^{\lambda_i} \in \Phi$) **then**
    if ($\lambda_i^{NUT} - t \geq tsw_1^{\lambda_i}$) **then**
        Shut-down the device λ_i
        Timer = $\lambda_i^{NUT} - tr_1^{\lambda_i}$
    else
        Keep the device λ_i on
    end if
else
    if ($\lambda_i^{NUT} - t > bet_1^{\lambda_i}$) **then**
        Shut-down the device λ_i
        Timer = $\lambda_i^{NUT} - tr_1^{\lambda_i}$
    else
        Leave the device λ_i on (otherwise we cannot guarantee the schedulability)
    end if
end if
Device Wake-up Procedure:
When the initial timer to wake-up λ_i expires.
if ($\lambda_i \in \lambda : \S_1^{\lambda_i} \in \Phi$) **then**
    if ($tr_1^{\lambda_i} \leq D_b$) **then**
        $D_b = D_b - tr_1^{\lambda_i}$
        Keep the device λ_i off and register its entry in the SSSR
    else if ($S_e^{sz} > tr_1^{\lambda_i} \&\& S_e^{dl} \leq d_{i,k}$) **then**
        Where $d_{i,k}$ is the deadline of the job $j_{i,k}$ that will require λ_i in future.
        if $\Xi_i^t > 0$ **then**
            Register the device λ_i in ESSR
            Timer = $S_e^{sz} - tr_1^{\lambda_i} + \Xi_i^t$
        else
            Timer= $S_e^{sz} - tr_1^{\lambda_i}$
        end if
    else
        Wake-up the device λ_i
    end if
else if ($\lambda_i \in \lambda : \S_1^{\lambda_i} \notin \Phi$) **then**
    Wake-up the device λ_i
end if
Device Budget D_b Replenishment:
if Ready Queue Empty && Device Waiting Queue Empty **then**
    D_b = Initial value of $D_b - \sum_{\lambda_i \in \text{SSSR}} tr_1^{\lambda_i}$
end if

correspond to the device shut-down mechanism. In the given example (Figure 5.1), all jobs of τ_1 have enough time to shut-down the device. $j_{2,1}$ completes its device related execution at time instant 7 and has a difference of 8 time units from its next utilisation time λ_2^{NUT} of 15, which is less than its $bet_1^{\lambda_2}$. However, λ_2 initiates a sleep transition with an expectation that device usage will be delayed due to the sporadic delay of the task arrival and the total sleep duration will be more than $bet_1^{\lambda_2}$. Similarly, $j_{2,2}$ has $\lambda_i^{NUT} - t = 8 > tsw_1^{\lambda_i}$ and it initiates a sleep transition based on the same reasoning given for $j_{2,1}$.

On the other hand, the condition ($\lambda_i^{NUT} - t \geq tsw_1^{\lambda_i}$) may not be applied to devices not compatible with intra-task device scheduling, i.e., $\lambda_i \in \lambda : \S_1^{\lambda_i} \notin \Phi$, as the scheduler needs to ensure that λ_i should be active before its λ_i^{NUT}. Nevertheless, we can still shut-down these devices with a condition that the jobs related to these devices execute less than their C_i and $\lambda_i^{NUT} - t > bet_1^{\lambda_i}$. A timer for this type of device is also set to $\lambda_i^{NUT} - tr_1^{\lambda_i}$ to wake it up before its next earliest utilisation time λ_i^{NUT}.

5.2.1.4 Device Wake-up

This algorithm has the main objective to extend the sleep interval of the devices already in a sleep mode. Whenever, a timer associated to any $\lambda_i \in \lambda : \S_1^{\lambda_i} \in \Phi$ expires, the scheduler considers slack resources (i.e., device budget D_b or the execution slack S_e) to prolong the currently inactive device λ_i. However, this process to prolong the sleep interval of λ_i is not considered for $\lambda_i \in \lambda : \S_1^{\lambda_i} \notin \Phi$, for which a timer expiration triggers a process to activate the device without any further delay to activate it before λ_i^{NUT}. The pseudo-code of the device wake-up procedure is given in Algorithm 6 from line number 21 to 40. The device budget D_b is the major source of slack used to extend the sleep interval of the devices. Once a timer associated to any $\lambda_i \in \lambda : \S_1^{\lambda_i} \in \Phi$ fires, the scheduler firstly tries to utilise D_b. $D_b \geq tr_1^{\lambda_i}$ allows to further procrastinate the device λ_i activation process, while ensuring the schedulability of the system (Equation 4.26). Consequently, $tr_1^{\lambda_i}$ time is deducted from D_b and λ_i is registered in a special register called SSSR. SSSR holds all inactive devices that acquired their wake-up budget from D_b. The wake-up calls to the devices, acquired their transition budget from D_b, are only made when requested by their corresponding jobs. Furthermore, no timers are associated to these devices unless they are requested to transition out of their sleep states, as their wake-up transition time is already deducted from D_b. This process to procrastinate the transition of λ_i from its sleep state to an active mode and combining it with a wake-up call on demand allows to exploit the sporadic slack in the schedule as well as the delays due to higher priority workload and device usage in later stages of the task. For instance, consider the next job $j_{i,k}$ corresponding to λ_i arrives with some delay after its T_i. λ_i will only transition out of its sleep state when requested by $j_{i,k}$, hence, λ_i will stay in a sleep mode during this sporadic delay.

In our example presented in Figure 5.1, the timers associated to λ_1 servicing $j_{1,1}$, $j_{1,2}$ and $j_{1,3}$ expires at time instances $9, 19$ and 29 respectively. Similarly, timer associated to λ_2 which is being utilised by $j_{2,1}$ expires at time instant 12. The scheduler deducts D_b equal to 1 time unit at time instances $9, 19$ and 29 for the device λ_1 and extend its sleep state until it is requested again by the

subsequent job. Similarly, scheduler deduct 3 time units for λ_2 from D_b at time instant 12 (3 time units before its λ_2^{NUT}) and keep the device in a sleep mode unless requested by $j_{2,2}$. The next job $j_{2,2}$ associated to λ_2 arrives at time instant 18 with 3 units of delay from its original arrival time of 15. λ_2 is requested again at time instant 20 and hence it remains in a sleep state for a duration of 10 time units including this sporadic delay interval of 3 time units. Therefore, it can be seen that all devices which get their budget from D_b can exploit the sporadic slack, if it exists.

In case $D_b < tr_1^{\lambda_i}$, the scheduler relies on the S_e. λ_i associated to $j_{i,k}$ is eligible for S_e if and only if $d_{i,k}$ of $j_{i,k}$ that will utilise λ_i in the future is greater than or equal to the deadline of the execution slack S_e^{dl}. $d_{i,k}$ of $j_{i,k}$ not released yet can be conservatively predicted by considering its past release information and T_i. The duration of the pending high-priority workload Ξ_i^t compared to $j_{i,k}$ that currently resides in the ready queue at time instant t is also added while computing the total shut-down interval of the device. The next wake up time is set to $S_e^{sz} - tr_1^{\lambda_i} + \Xi_i^t$ and the corresponding device is registered in ESSR. A high priority workload from the future can also be included but it will increase the online complexity of the algorithm. Theorem 37 states the schedulability of the system remains unaffected when the execution slack is used to extend the sleep interval of the devices.

Theorem 37. *Assume the system is* EDF*-schedulable and let t denote the current time instant in the corresponding schedule. Let us consider the job $j_{i,k}$ with deadline $d_{i,k}$ released at time instant $t+\varepsilon$ ($\forall \varepsilon \geq tr_1^{\lambda_i}$). Assume $hp(j_{i,k},t)$ denote the set of all jobs in the ready queue at time t with a priority higher than or equal to $j_{i,k}$ and assume Ξ_i^t is the pending higher priority workload generated by $hp(j_{i,k},t)$.*

If the following two conditions hold:

1. *$d_{i,k}$ is greater than or equal to the execution slack deadline S_e^{dl},*
2. *The slack from $hp(j_{i,k},t)$ is not collected.*

Then the transition-out phase of device λ_i can be delayed for $S_e^{sz} + \Xi_i^t - tr_1^{\lambda_i}$ without jeopardising the schedulability of the system.

Proof. This theorem is proved by contradiction. Let $d_{i,k} \geq S_e^{dl}$ and the slack from $hp(j_{i,k},t)$ is not collected. Assume the transition-out phase of the device λ_i is delayed for $S_e^{sz} + \Xi_i^t - tr_1^{\lambda_i}$ and it causes the system to miss a deadline.Algorithm 6 sets the wake-up time of the device λ_i after its usage to $\lambda_i^{NUT} - tr_1^{\lambda_i}$. The extension in the sleep state of a device is performed at its wake-up time. Therefore, without loss of generality, $t = \lambda_i^{NUT} - tr_1^{\lambda_i}$. At time instant t, the job $j_{i,k}$ under consideration has an absolute deadline $d_{i,k} = t + tr_1^{\lambda_i} + D_i$. The job $j_{i,k}$ will miss its deadline if the inequality in Equation 5.1 holds.

$$t + C_i + \Xi_i^t + S_e^{sz} + tr_1^{\lambda_i} > d_{i,k} \tag{5.1}$$

$$t + C_i + \Xi_i^t + S_e^{sz} + tr_1^{\lambda_i} > t + tr_1^{\lambda_i} + D_i \tag{5.2}$$

$$C_i + \Xi_i^t + S_e^{sz} > D_i \tag{5.3}$$

Equation 5.3 contradicts the fact that system is EDF-schedulable as the response time of $j_{i,k}$ is greater than its absolute deadline. Indeed, this is not possible in our system and is explained as follows. Ξ_i^t is the interference from the high priority jobs residing in the ready queue at time t. S_e^{sz} is the slack from the previously finished jobs (executed less than their WCET) having deadline less than or equal to $d_{i,k}$. In the slack management algorithm (Algorithm 6) the deadline of the execution slack is extended, i) when the execution slack from the low priority job is added to the slack container or ii) when a low priority job executes in the presence of execution slack. In any case, if the deadline of the execution slack $S_e^{dl} \leq d_{i,k}$ then the execution slack comes from the leftover execution of completed jobs having priority higher than or equal to $j_{i,k}$. As a consequence of this, the interference from S_e^{sz} and Ξ_i^t in combination with WCET of a job $j_{i,k}$ must be less than D_i for the overall system to be considered schedulable with EDF. Hence, the transition-out phase of the device λ_i should be delayed for greater than $S_e^{sz} + \Xi_i^t - tr_1^{\lambda_i}$ to miss the deadline of $j_{i,k}$. Therefore, theorem holds. □

In a case $\{(D_b < tr_1^{\lambda_i}) \&\& (S_e^{dl} > d_{i,k} || S_e^{sz} < tr_1^{\lambda_i})\}$, the high priority workload currently in the ready queue and from future releases could also be considered. However, the computation of high priority workload increases the online overhead and is avoided in the SSC algorithm to limit the complexity.

5.2.1.5 Device Budget D_b Replenishment

The device budget is replenished in idle mode. The system is considered idle when the ready queue is empty and there is no job $j_{i,k}$ in the device waiting queue waiting for its λ_i to transition out of its sleep state. The replenishment of D_b is done in accordance with the following Theorem 38. The line numbers 41 to 44 in Algorithm 6 corresponds to the pseudo-code of D_b replenishment.

Theorem 38. *Assume, the system is* EDF*-schedulable. Let t be a time instant when the system is idle such that the ready queue along with the device waiting queue is empty. If the replenishment equal to $D_b - \sum_{\lambda_i \in \text{SSSR}} tr_1^{\lambda_i}$ occurs at time t then the schedulability of the system is preserved.*

Proof. This is a direct proof. The transition-out delays of the devices associated to the jobs registered in SSSR is modelled as a job $j_{f,k}$ that has an execution time of $\sum_{\lambda_i \in \text{SSSR}} tr_1^{\lambda_i}$ and a deadline equal to $\min_{\tau_i \in \tau} D_i$. In the idle state at time t, in order to maximise the workload, a critical instant is assumed, in which all the tasks release their jobs synchronously. The job $j_{f,k}$ is co-scheduled with τ at time t. The demand of the task-set with an addition of $j_{f,k}$ is denoted as $\text{DBF}(\tau + j_{f,k}, L)$ and given in Equation 5.4, where L is any absolute deadline in the schedule.

$$\text{DBF}(\tau + j_{f,k}, L) \stackrel{\text{def}}{=} \sum_{i=1}^{\ell} \left(\left\lfloor \frac{L - D_i}{T_i} \right\rfloor + 1 \right) C_i + \sum_{\lambda_i \in \text{SSSR}} tr_1^{\lambda_i} \tag{5.4}$$

$$= \text{DBF}(\tau, L) + \sum_{\lambda_i \in \text{SSSR}} tr_1^{\lambda_i} \tag{5.5}$$

Assume the device budget with a demand of $\mathrm{DBF}(\tau + j_{f,k}, L)$ is equal to D'_b. From Lemma 27 and Equation 4.26, the device budget can be expressed as given in Equation 5.6, where L^* is the first idle time in the schedule.

$$
\begin{aligned}
D'_b &= \min_{\forall L \leq L^*} (L - \mathrm{DBF}(\tau + j_{f,k}, L)) && (5.6)\\
&= \min_{\forall L \leq L^*} (L - \mathrm{DBF}(\tau, L) - \sum_{\lambda_i \in \mathrm{SSSR}} tr_1^{\lambda_i}) && (5.7)\\
&= \min_{\forall L \leq L^*} (L - \mathrm{DBF}(\tau, L)) - \sum_{\lambda_i \in \mathrm{SSSR}} tr_1^{\lambda_i} && (5.8)\\
&= D_b - \sum_{\lambda_i \in \mathrm{SSSR}} tr_1^{\lambda_i} && (5.9)
\end{aligned}
$$

Equation 5.9 shows the desired device budget $D'_b = D_b - \sum_{\lambda_i \in \mathrm{SSSR}} tr_1^{\lambda_i}$. The scheduler does not allocate for more than D_b time units to the devices to compensate for the transition-out delays, therefore, $\sum_{\lambda_i \in \mathrm{SSSR}} tr_1^{\lambda_i} \leq D_b$. This means the value of D'_b will always be positive and the system will preserve the schedulability with a replenishment of $D'_b = D_b - \sum_{\lambda_i \in \mathrm{SSSR}} tr_1^{\lambda_i}$. Hence, this theorem holds. □

The replenishment of D_b in an example presented in Figure 5.1 is performed according to the criterion defined in Theorem 38 at time instances 13 and 26. At the first time instant of 13, λ_2 is still in sleep mode and previously registered its entry in SSSR at time instant 12, therefore, D_b is only replenished with a budget equal to its initial value minus the device transition delay $tr_1^{\lambda_2}$ of λ_2 (i.e., $4-3=1$). However, at time instant 26 both the devices are in sleep states but not registered in SSSR, hence, D_b is replenished with a budget equal to its initial value of 4.

5.3 Device Budget Reclamation

The device budget D_b is a precious resource in our intra-task device scheduling algorithm; therefore, a device budget reclamation algorithm is proposed to collect the unused portion of it. Initially, the potential sources involved to reclaim D_b are defined and afterwards the device budget reclamation algorithm is discussed in details.

5.3.1 Terminologies and Basic Idea

Some of the abbreviations used throughout this section are given below.

- HPW : The workload of tasks' instances with higher or equal priority than τ_j's instance executed in the device λ_i's sleep state $\S_n^{\lambda_i}$ transition-out interval of $tr_n^{\lambda_i}$.
- LPW : The workload of tasks' instances with lower priority than τ_j's instance executed in the device λ_i's sleep state $\S_n^{\lambda_i}$ transition-out interval of $tr_n^{\lambda_i}$.

- IPW : Suppose LPW executed during τ_i's device λ_i transition-out phase then the intermediate priority workload (IPW) corresponds to those tasks' instances which can be released in future and have a priority between LPW and τ_i's instance.

- λ_i^{start}: Assume a device λ_i is in a transition out phase, i.e., transiting from a sleep state to an active state, then λ_i^{start} is the absolute start time of such transition.

- λ_i^{ready}: Assume a device λ_i has completed its transition out phase, i.e., transitioned out from a sleep state to an active state, then λ_i^{ready} is the absolute end time of such transition.

- Transitioning Device λ_t: At time instant t, a device λ_t currently in transition-out phase selected as a reference for reclamation purposes.

Device budget D_b is only reclaimed from devices which have corresponding entries in the SSSR; i.e., devices which have their allocations from a device budget to compensate for their transition delay. All devices discussed onwards in this section are assumed to have an entry in SSSR (i.e., $\lambda_i \in$ SSSR). Devices $\lambda_i \notin$ SSSR have not received D_b and hence are not eligible for reclamation. D_b by definition is the highest priority budget in the schedule. When $j_{i,k}$ is allocated a part of D_b to compensate for its device transition, analysis assumes this additional budget will be consumed by $j_{i,k}$ as a part of execution. This assumption is made for a case when there is no other job executing and/or waiting for its device transition during this interval. When there is another job executing or waiting for its device transition, the device budget may be reclaimed under certain conditions elaborated below. In the former case (another job executing or also termed as execution overlap), the reclaimed budget size depends on the priority of the workload executed in this interval. Similarly, if another job(s) is waiting for its device transition then such scenario is termed as device overlap and can be considered for device budget reclamation. For instance in Figure 5.1, within an interval of $[20,21]$ device transition time of both devices (λ_1 and λ_2) overlaps, similarly, in an interval of $[21,22]$, the execution of $j_{1,3}$ overlaps with the transition of λ_2. These two potential sources to reclaim D_b are discussed below in details. A reclaimed budget is not added back in the example given in Figure 5.1 for the ease of presentation.

5.3.2 Sources to Reclaim Device Budget

5.3.2.1 Device Overlap

In this scenario, multiple jobs are waiting for their device's active state. It is evident that a scheduler should only consider a budget consumption of a single device in the overlapping period as their wake-up transition occur in parallel. In our example (Figure 5.1) the device budget of 1 time unit can be reclaimed at time instant $[20,21]$, as device transitions of λ_1 and λ_2 overlap, and the scheduler should only consider a transition delay of one device.

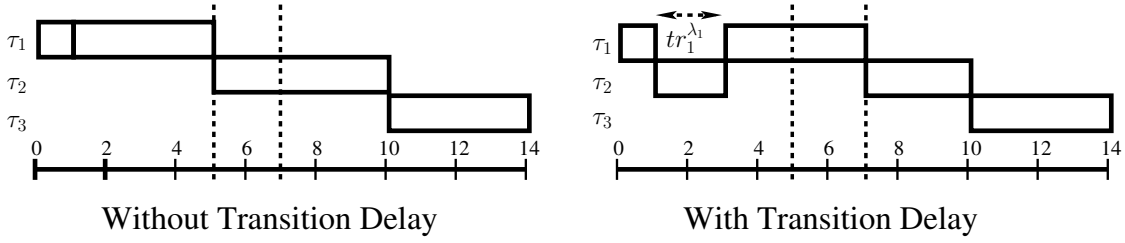


Figure 5.2: Low priority workload overlap

5.3.2.2 Execution Overlap

Assume a job $j_{i,k}$, currently waiting for the device λ_i in a transition phase from its sleep state to an active state. λ_i's transition interval is denoted as $[t_1, t_2]$. In this scenario, an overlap of $[t_1, t_2]$ with the execution of other jobs $\neq j_{i,k}$ is explored. The execution overlap is divided into two types based on its priority when compared to the priority of $j_{i,k}$, i.e., high or low priority workload. These two types are discussed separately.

- **High Priority Workload Overlap:** All jobs having deadline earlier than or equal to the deadline $d_{i,k}$ of $j_{i,k}$ are considered as high priority workload. If HPW executes during $[t_1, t_2]$, then the size of their overlap can be reclaimed as delayed execution of $j_{i,k}$ due to transition time of its device does not affect the high priority workload. Therefore, in our example given in Figure 5.1, a device budget of 1 time unit can be reclaimed in an interval of $[21, 22]$ for an overlap of λ_2 with an execution of $j_{1,3}$.

- **Low Priority Workload Overlap:** In a scenario, when LPW executes in $[t_1, t_2]$ then the scheduler needs to consider IPW that may execute during the leftover execution time of $j_{i,k}$. This IPW consists of jobs that will release in future and have deadlines between the earliest deadline of the LPW that executed in $[t_1, t_2]$ and the deadline of $j_{i,k}$. The IPW can be predicted by considering the previous release information.

The delayed execution of $j_{i,k}$ due to its device λ_i transition in this scenario can only affect the workload that we define as the IPW. LPW that was executed during $[t_1, t_2]$ will just switch their execution slots with $j_{i,k}$ execution by an amount they have executed in $[t_1, t_2]$. For instance consider an example shown in Figure 5.2. Assume, the priorities of $\tau_1 > \tau_2 > \tau_3$ corresponds to the priority of their instances. τ_2's and τ_3's instances have finished their execution at the same time in both cases (with or without transition delay). τ_2's instance has performed its execution during the transition phase of λ_1 and allowed τ_1's instance to swap its execution slot. Jobs having priority higher and lower than $j_{i,k}$ will not be affected in any case. The reclaimed budget is added back to D_b. Such reclamation is formally proven with Theorem 39.

Theorem 39. *If there's a low priority job $j_{p,q}$ executing during a wake-up transition of λ_i registered in* SSSR *and there is no intermediate priority (*IPW*) jobs then the wake-up transition time overlapped with $j_{p,q}$ execution can be reclaimed without causing any task to miss its deadline.*

Proof. This is a direct proof. Assume a job $j_{i,k}$ requires a device λ_i registered in SSSR at time instant t_o during its execution. The execution of $j_{i,k}$ is suspended during the transition-out phase of λ_i, i.e., in an interval $[t_o, t_o + tr_1^{\lambda_i}]$. Assume another job $j_{p,q}$ executes for X time units during the transition-out phase of λ_i and has low priority when compared to $j_{i,k}$ (i.e., $d_{i,k} < d_{p,q}$). From Section 5.3.1, IPW is the execution time of the jobs having priority greater than or equal to $j_{p,q}$ and less than $j_{i,k}$. Let, t be the time instant when $j_{p,q}$ starts its execution in the transition-out phase of λ_i. All the jobs are assumed to execute for their WCET to consider maximum workload in the schedule (sustainability property of EDF). The execution time of jobs having higher priority when compared to $j_{i,k}$ executed between $[t_o, t']$ is represented as HP. In a case $j_{i,k}$ requires its device λ_i then $t' \stackrel{def}{=} t_o + HP + tr_1^{\lambda_i} + c_{i,k} + c_{p,q} - X$ be the time instant when the execution of both $j_{i,k}$ and $j_{p,q}$ is completed.

To prove the theorem, we have to show two cases. 1) The portion of the jobs executed in an interval $[t_o, t']$ will remain the same with or without the transition-out delay of λ_i. 2) The schedule at any time instant $t'' > t'$ will remain unaffected.

Case 1) In the time interval $[t_o, t']$, the portion of the execution of the high priority jobs (HP) and $j_{i,k}$ is the same irrespective of the transition-out delay of λ_i. Now let us consider the portion of the execution of the job $j_{p,q}$. If the transition-out delay of λ_i occurs then the job $j_{p,q}$ executes its portion equal to X in the transition-out phase of λ_i, while the rest of it execution $c_{p,q} - X$ will execute after the completion of HP and $j_{i,k}$. However, in the absence of transition-out delay of λ_i, the job $j_{i,k}$ will finish its execution at most $tr_1^{\lambda_i}$ time units earlier than the previous case (with transition of λ_i). As there is no intermediate priority job released prior to the completion of job $j_{i,k}$ (using λ_i) by assumption, $tr_1^{\lambda_i} + c_{p,q} - X$ units are available to $j_{p,q}$ in an interval $[t_o, t']$ after the completion of $j_{i,k}$. Now consider two sub-cases. i) $X = tr_1^{\lambda_i}$: In this sub-case, the amount of execution available to $j_{p,q}$ in an interval $[t_o, t']$ will be equal to $c_{p,q}$. ii) $X < tr_1^{\lambda_i}$: In this sub-case, $tr_1^{\lambda_i} - X$ amount of execution will be left unused in an interval $[t_o, t']$. This extra time unit comes from the execution of $j_{i,k}$ in an interval $[t_o, t]$ as a portion of device budget allocated to λ_i is not used and hence available as a slack. This time interval is modelled as a fake job $j_{f,k}$ and scheduled after $j_{p,q}$. Irrespective of this fake job $j_{f,k}$ in the system, the job $j_{p,q}$ in both cases (with or without transition of λ_i) gets the same portion of execution in an interval $[t_o, t']$.

Case 2) The HP and $j_{i,k}$ gets the same portion of execution in an interval $[t_o, t']$ irrespective of device λ_i transition-out delay. Similarly, $j_{p,q}$ also gets same portion of execution in both cases. The only difference between the two schedules is the scheduling time of fake job $j_{f,k}$ generated from the difference of $t - t_o$. In the first case it is schedule in the start of $[t_o, t]$, while in the later case it can be scheduled at $[t' - (t - t_o), t']$, without affecting any deadline in the schedule. Hence, the schedule at $t'' > t'$ is not affected. The X unit of time can be reclaimed and used again thanks to Lemma 27. Hence, the above theorem holds. □

Algorithm 7 Device Budget Reclamation Algorithm

1: **if** (λ_i^{start} && No Transitioning Device λ_t Exists) **then**
2: Transitioning Device $\lambda_t = \lambda_i$
3: HPW $= 0$
4: LPW $= 0$
5: **else if** (λ_i^{start} && $\lambda_i^{ready} \leq \lambda_t^{ready}$) **then**
6: $D_b = D_b + tr_n^{\lambda_i}$
7: **else if** (λ_i^{start} && $\lambda_i^{ready} > \lambda_t^{ready}$) **then**
8: $D_b = D_b + \lambda_t^{ready} - \lambda_i^{start}$
9: $D_b = D_b +$ HPW
10: **if** (IPW Does not Exists) **then**
11: $D_b = D_b +$ LPW
12: **end if**
13: HPW $= 0$
14: LPW $= 0$
15: Transitioning Device $\lambda_t = \lambda_i$
16: **else if** (Transitioning Device is in Active State, λ_t^{ready}) **then**
17: $D_b = D_b +$ HPW
18: **if** (IPW Does not Exists) **then**
19: $D_b = D_b +$ LPW
20: **end if**
21: HPW $= 0$
22: LPW $= 0$
23: Transitioning Device $\lambda_t =$ NULL
24: **end if**

25: **Accounting** HPW **and** LPW**:**
26: **if** (τ_i's Instance Starts Execution && λ_t Exists) **then**
27: $\alpha = t$
28: **end if**
29: **if** ((τ_i's Instance Stopped && λ_t Exists) || (λ_t^{ready}) || (λ_t Changes)) **then**
30: **if** (τ_i's Instance $\in$ HPW) **then**
31: HPW$+= t - \alpha$
32: **else if** (τ_i's Instance $\in$ LPW) **then**
33: LPW$+= t - \alpha$
34: **end if**
35: $\alpha = t$
36: **end if**

5.3.3 Device Budget Reclamation Algorithm

The pseudo-code of the device budget reclamation algorithm is given in Algorithm 7. This algorithm works for both single and multiple sleep state devices. Therefore, it can be applied to all online intra-task device scheduling algorithms explained in this chapter. It is based on the four different principles given below.

- **Rule 1:** Assume λ_i started its transition from a sleep state and there is no other device in SSSR in transition phase then the device λ_i is marked as a transitioning device λ_t. There will be no device budget reclamation in this scenario. However, a scheduler will start monitoring and counting the execution occurring from this time instant. The lines 1 to 4 in Algorithm 7 correspond to this rule.

- **Rule 2:** A device λ_i started its transition in the presence of any transitioning device λ_t and has $\lambda_i^{ready} \leq \lambda_t^{ready}$, will be considered for device budget reclamation. In this scenario, a device budget of $tr_n^{\lambda_i}$ will be reclaimed from an overlap of λ_i with λ_t. The pseudo-code of this rule is presented in lines 5 and 6 of Algorithm 7. However, at this moment no overlap of λ_i with other tasks executions (HPW and/or LPW) is considered, as this will be covered in rule 3 and 4.

- **Rule 3:** In case $\lambda_i^{ready} > \lambda_t^{ready}$, then the device budget of $\lambda_t^{ready} - \lambda_i^{start}$ will be reclaimed. At the same moment all HPW executed during the interval $[\lambda_i^{start}, t]$ will be reclaimed as the device budget. Similarly, LPW executed in this interval will also be reclaimed as the device budget given there exists no IPW. Both HPW and LPW counters are initialised to zero to avoid their double reclamation. Moreover, λ_i will be marked as a new transitioning device (i.e., $\lambda_t = \lambda_i$). All the steps corresponding to this rule are given in lines 7 to 15.

- **Rule 4:** The transitioning device λ_t^{ready} time instant also triggers this routine to reclaim the device budget. This event occurs on the completion of the transition phase of λ_t. Similar to rule 3, a device budget is reclaimed from HPW and/or LPW (given the condition IPW does not exits). In case IPW exists, device budget from LPW is not reclaimed. Both counters are initialised to zero. See lines 16 to 24 for the pseudo-code of this rule.

The routine used to account HPW and LPW is given in lines 26 to 36 of Algorithm 7 and is explained as follows. The start time of an execution of a task τ_i's instance in the presence of a transitioning device is recorded in α. If any of the following three events occurs: 1) τ_i's instance stops its execution in the presence of λ_t or 2) transitioning device becomes active or 3) currently active transitioning device is replaced with another device, then the current execution time of τ_i's instance is added accordingly in HPW or LPW depending on its priority. Afterwards, the value of α is updated to the current time instant t. Lines 25 to 36 in Algorithm 7 represent this routine.

One of the advantage of our device budget reclamation algorithm is that a device budget due to device overlap is reclaimed in the beginning of the device transition. This allows to use this device budget for other devices to compensate for their transitions. Moreover, HPW and/or LPW overlap is recycled either at λ_t^{ready} time instant (rule 4) or when λ_t is replaced with the new device (rule 3). For instant, consider our running example given in Figure 5.1. The transition out phase of λ_1 and λ_2 overlap in an interval $[20, 21]$. The device budget due to a device overlap can be reclaimed at time instant 20. The high priority workload executing during a transition phase of λ_2 can be reclaimed at time instant $\lambda_2^{ready} = 23$.

5.4 Multiple Sleep States Per Device Model

5.4.1 Base Idea

This section generalises the device model and allows each device to have more than one sleep state. The multiple sleep states per device model has some additional challenges in distributing the available slack in the schedule when compared to the one allowing only a single sleep state. Compared to the SSC algorithm, the multiple sleep states per device model requires following additional steps to efficiently utilise available slack in the schedule. Firstly, the locally most efficient sleep state among those available needs to be identified pruning away the energy-wise less efficient sleep states. Secondly, the effect on the other devices and eventually on the global energy minimisation has to be considered, as this selected locally efficient sleep state might not be globally efficient. This is caused by the duration between the average activation times of the devices, since devices which on average have longer idle periods save more energy than those strictly periodically used. The objective is to quantify the effect of each sleep state on the overall device energy consumption and to find the right choice of a sleep state for each device to acquire the global goal of minimising the device energy consumption with the given slack. To achieve such objective, the problem of multiple sleep states per device is divided into a stepwise process. There are three major steps of this approach.

1. Find the effect of device's sleep states on the overall device energy consumption.

2. Categorise the device's sleep states based on the order of their effect on the global device energy minimisation.

3. Sort the devices as well with respect to their sleep states order.

The information given in aforementioned three steps is used to define a number of heuristics.

5.4.2 Energy-Density Function

The efficiency of the device's sleep state is measured by its energy saving ability. The energy saving offered by a particular sleep state of a device not only depends on the hardware platform but it is also affected by the task's properties using such a device. The average energy consumption of a device is considered as the focus is on the heuristics. In order to estimate the average energy saving offered by a device's sleep state, an average distance between the two subsequent jobs of the task using this device is needed. This distance depends on many factors such as T_i, actual execution time of task and device usage time. While it is feasible to identify the time window of the device usage after the start of the job execution, such estimation is not consider here. This is driven by the observation that the difference between earliest and latest usage is usually large due to substantial run time variation and high priority interference, hence the ensuing accounting overhead is not justified.

An energy-density function is developed that computes the average energy saving offered by a device λ_i's sleep state $\S_n^{\lambda_i}$ when used by the task τ_i. The energy-density function does not only consider the device properties but also includes the effect of task-level properties. To compute such value, a task is considered in isolation. Assume, an average distance between two consecutive jobs of a task denoted as $\bar{T}_i$. The energy consumption of the device λ_i is computed for an interval of $\bar{T}_i$, with an assumption that it transitions into a sleep state $\S_n^{\lambda_i}$ while not performing the device related processing. This value is subtracted from the energy consumption of the same device λ_i in its active mode for $\bar{T}_i$. The energy saving value is normalised to $\bar{T}_i$ of the task τ_i. The energy-density function corresponding to the sleep state $\S_n^{\lambda_i}$ of a device λ_i is denoted as $\lambda_i^{ED_n}$ and presented in Equation 5.10.

$$\lambda_i^{ED_n} = \frac{\bar{T}_i P_A^{\lambda_i} - \overbrace{\left(\bar{T}_i - \bar{C}_i - 2tr_n^{\lambda_i}\right) P_n^{\lambda_i}}^{SleepEnergy} - \overbrace{\left(Es_n^{\lambda_i} + \bar{C}_i P_A^{\lambda_i}\right)}^{ActiveEnergy}}{\bar{T}_i} \tag{5.10}$$

It has three components. The component $\bar{T}_i P_A^{\lambda_i}$ is the energy consumption of the device in the active state for $\bar{T}_i$. The energy consumption of the device during a sleep mode is represented with a second component of $\left(\bar{T}_i - \bar{C}_i - 2tr_n^{\lambda_i}\right) P_n^{\lambda_i}$. Finally, the energy consumed in the transition phases plus the active phase (device usage time) is included in the third component $\left(Es_n^{\lambda_i} + \bar{C}_i P_A^{\lambda_i}\right)$.

5.4.3 Devices and their Sleep State Categorisation

The energy-density function $\lambda_i^{ED_n}$ allows to find the energy saving offered by a device λ_i's sleep state $\S_n^{\lambda_i}$ per unit time. This function helps to prioritise the devices among each other. Moreover, multiple sleep state devices can also prioritise its sleep states with the help of this function. The process used to prioritise the devices and their sleep states is explained as follows.

Initially, the sleep states of the devices not compatible with the intra-task device scheduling are identified and eliminated from this process. A sleep state is not compatible with the intra-task device scheduling algorithm, if $\{\S_n^{\lambda_i} \in \phi : D_i - C_i < tsw_k^{\lambda_i}\}$ (see Section 5.2 for the definition of intra-task compatible sleep states set Φ). These sleep states are not considered for the allocation of any type of slack. The remaining intra-task device scheduling compatible sleep states of all devices are sorted with respect to their energy densities values in descending order. The devices are prioritised based on the order of their sleep-state energy-density values. The first occurrence of any device's sleep state defines its order among the list of devices.

For example, assume three devices, each having three sleep states and all are intra-task device scheduling compatible. Assume the descending order of the sleep states with respect to their energy densities values is $\{\S_1^{\lambda_2}, \S_0^{\lambda_2}, \S_2^{\lambda_3}, \S_2^{\lambda_2}, \S_2^{\lambda_1}, \S_1^{\lambda_1}, \S_1^{\lambda_3}, \S_0^{\lambda_1}, \S_0^{\lambda_3}\}$. The priority of the devices considering any type of slack would be $\lambda_2, \lambda_3, \lambda_1$.

Similarly, the order of the sleep states within a device from the most efficient to the least efficient is also estimated based on the energy-density values of these sleep states. In the example given above, the order of the sleep states within the device $\lambda_1 = \{\S_2^{\lambda_1}, \S_1^{\lambda_1}, \S_0^{\lambda_1}\}$, $\lambda_2 = \{\S_1^{\lambda_2}, \S_0^{\lambda_2}, \S_2^{\lambda_2}\}$

and $\lambda_3 = \{\S_2^{\lambda_3}, \S_1^{\lambda_3}, \S_0^{\lambda_3}\}$. Once devices and their sleep states are prioritised, a number of heuristics can be proposed exploiting this information. This chapter presents three different heuristics.

5.4.4 Offline Algorithm for Multiple Sleep State Devices (SSC^o)

The offline algorithm for multiple sleep state devices (SSC^o) is the simplest of all approaches and has negligible online device scheduling overhead. The pseudo-code of the algorithm is presented in Algorithm 8. The algorithm statically prioritises the devices among each other and finds the most efficient sleep state for each device. The unused system utilisation of $U_{leftover} = 1 - \sum_{i=1}^{\ell} \frac{C_i}{D_i}$ is distributed to compensate for the transition delays of the devices. One can also think of using the device budget D_b for offline distribution instead of $U_{leftover}$. In such distribution, if a device is considered as a candidate, the transition overhead of its all releases should be deducted from D_b, which is very pessimistic. Therefore, it is decided to use $U_{leftover}$.

The technique given in Section 5.4.2 and Section 5.4.3 is used to prioritise the devices. The energy-density for each sleep state is computed. The intra-task device scheduling compatible sleep state with the highest density value is selected for each device and represented as $\S_h^{\lambda_i}$. In the online phase, the device λ_i transitions into this selected sleep state. The corresponding devices are sorted with respect to the energy-density of the selected sleep state $\S_h^{\lambda_i}$. If none of the sleep states of λ_i are compatible with the intra-task device scheduling, then a sleep state $\S_{nc}^{\lambda_i}$ with the highest energy-density value is associated to such device λ_i to be used online for sleep transitions. However these devices are not considered for a share in $U_{leftover}$ distribution.

The devices request their share of $\frac{tsw_h^{\lambda_i}}{D_i}$ from the unused system utilisation of $U_{leftover}$ following the priority of the devices determined. If a device λ_i has $\frac{tsw_h^{\lambda_i}}{D_i} \leq U_{leftover}$, the value of $U_{leftover}$ is decremented by $\frac{tsw_h^{\lambda_i}}{D_i}$ before considering the next device in the order. However, if the device's $\frac{tsw_h^{\lambda_i}}{D_i} > U_{leftover}$, the SSC^o algorithm will skip this device and move on to the next device. This process is similar to the self suspending tasks model, where the self suspension part of the task is added into the task's execution time. The only difference is that we give priority to the devices which promise higher energy savings. The lines 7 to 12 of the pseudo-code in Algorithm 8 corresponds to $U_{leftover}$ distribution. The distribution of $U_{leftover}$ can also be modelled as a knapsack problem and solved through existing approaches.

The shut-down and wake-up procedure of the devices which have their share from $U_{leftover}$ is straightforward. All devices which have a share in $U_{leftover}$ are shut-down immediately, once they have been used by their respective tasks. These devices only transition out from their respective sleep states when requested by their corresponding tasks. The jobs requesting these devices are moved into the device waiting queue during the transition phase of their devices. No timer is needed for such devices. Devices which do not have a share in $U_{leftover}$ can also be turned-off after it has been used with a condition that $\lambda_i^{NUT} - t > bet_h^{\lambda_i}$. A timer is set to wake-up the

Algorithm 8 Offline Algorithm for Multiple Sleep State Devices (SSC^{o})

Offline Phase:
$\forall \lambda_i$ and its sleep states in $\vec{\S^{\lambda_i}}$, compute the energy-density function (Equation 5.10)
Remove sleep states not compatible with intra-task device scheduling,
i.e., $\S_n^{\lambda_i} \notin \Phi$, where, $\Phi = \{\S_n^{\lambda_i} \in \phi : D_i - C_i \geq tsw_n^{\lambda_i}\}$
For each device λ_i, find the intra-task device scheduling compatible sleep state with the maximum value of energy-density function and name it $\S_h^{\lambda_i}$
Sort λ_i in descending order with respect to their energy-density values of $\S_h^{\lambda_i}$
Compute $U_{leftover} = 1 - \sum_{i=1}^{\ell} \frac{C_i}{D_i}$
for $(\forall \lambda_i : \S_h^{\lambda_i} \in \Phi)$ **do**
    if $(U_{leftover} - \frac{tsw_h^{\lambda_i}}{D_i} \geq 0))$ **then**
        λ_i gets share from $U_{leftover}$ to compensate for its transitions delays of $\S_h^{\lambda_i}$
        $U_{leftover} -= \frac{tsw_h^{\lambda_i}}{D_i}$
    end if
end for
If none of sleep states of λ_i are intra-task device scheduling compatible, select the sleep state $\S_{nc}^{\lambda_i}$ for λ_i with the highest energy-density value
Device Shut-Down Procedure:
When $j_{i,k}$ has used λ_i, consider the following criteria to shut-down λ_i and set the corresponding entry in the sorted list of timer if required.
if $(\S_h^{\lambda_i})$ **then**
    if (λ_i has a share in $U_{leftover}$) **then**
        Initiate transition of λ_i into $\S_h^{\lambda_i}$
    else if $(\lambda_i^{NUT} - t \geq bet_h^{\lambda_i})$ **then**
        Initiate transition of λ_i into $\S_h^{\lambda_i}$
        Timer = $\lambda_i^{NUT} - tr_h^{\lambda_i}$
    else
        Keep λ_i on
    end if
else if $(\S_{nc}^{\lambda_i})$ **then**
    if $(\lambda_i^{NUT} - t > bet_{nc}^{\lambda_i})$ **then**
        Initiate transition of λ_i into $\S_{nc}^{\lambda_i}$
        Timer = $\lambda_i^{NUT} - tr_{nc}^{\lambda_i}$
    else
        Leave λ_i on
    end if
end if
Device Wake-up Procedure:
if ($\S_h^{\lambda_i}$&& has share in $U_{leftover}$) **then**
    Wake-up when requested
else if ($\S_{nc}^{\lambda_i}||(\S_h^{\lambda_i}$&& has no share in $U_{leftover}$) **then**
    Wake-up when timer associate to λ_i expires
end if

corresponding device before the next arrival of its associated task. If none of the sleep states of λ_i are compatible with the intra-task device scheduling then it is only considered for shut-down if the condition $\lambda_i^{NUT} - t > bet_{nc}^{\lambda_i}$ is met. A timer is set to wake-up this device before the arrival of the next job of its associated task.

The main advantage of SSC^{o} is its simplicity. It has an online complexity of $O(1)$. Most of the device related decision are taken offline. However, it does not exploit the execution slack S_e. The static slack and sporadic slack are also exploited partially. Therefore, it underutilises the opportunities available to reduce the device energy consumption.

5.4.5 Static Slack Container Algorithm with Multiple Sleep State Devices (SSC^{m})

The proposed static slack container algorithm with multiple sleep state devices (SSC^{m}) avails the opportunities ignored by the SSC^{o} algorithm and exploits all forms of slacks (static, execution and sporadic slack) to minimise device energy consumption. The problem of multiple sleep states per device can be transformed to a single sleep state per device problem using the technique given in Section 5.4.2 and Section 5.4.3. In this algorithm, the most effective sleep state for each device is identified offline and the static slack container algorithm is applied online to trigger the shut-down or wake-up procedure of the devices. Similar to the offline phase of the previous algorithm (SSC^{o}), it prioritises the sleep states for each device and selects the most efficient sleep state for each device. As this is an online algorithm and the order of the devices does not matter, therefore, $U_{leftover}$ is not distributed offline. In this algorithm, device budget D_b and the execution slack S_e is used to compensate for the transition delays of devices. Moreover, the sporadic slack is exploited implicitly to reduce the device power dissipation (see Algorithm 1). The pseudo-code of this is given in Algorithm 9. The lines 1 to 4 illustrates the sleep state selection process.

The basic principles of the scheduling phase, next utilisation time λ_i^{NUT} determination, device wake-up/shut-down procedure and device budget D_b replenishment mechanism remains similar to the SSC algorithm provided in Algorithm 6. The SSC^{m} algorithm reduces the online search space of the device scheduling algorithm and utilises all forms of slacks to minimise the device energy consumption. However, it partially exploits the extra opportunity offered by the multiple sleep states of devices. For example, assume a sleep state $\S_h^{\lambda_i}$ is selected for a device λ_i. At time instant t, a task finishes its device usage and considers its device for shut-down. If $tsw_h^{\lambda_i} > \lambda_i^{NUT} - t$, scheduler refrains its transition and keep it active. At the same time instant, may be another sleep state $\S_j^{\lambda_i}$ of this device can satisfy this condition $tsw_j^{\lambda_i} \leq \lambda_i^{NUT} - t$ and allows the device to transition into a less efficient sleep state. The loss of opportunity results as a consequences of trading energy consumption for algorithmic complexity.

5.4.6 Aggressive Static Slack Container Algorithm for Multiple Sleep State Devices (SSC^{a})

The aggressive static slack container algorithm for multiple sleep state devices (SSC^{a}) exploits more successfully the opportunities to minimise the energy consumption of the devices. It does

Algorithm 9 Static Slack Container Algorithm for Multiple Sleep State Devices (SSC^{m})

Offline Phase:
$\forall \lambda_i$ and its sleep states in $\vec{\S^{\lambda_i}}$, compute the energy-density function (Equation 5.10)
Remove sleep states not compatible with intra-task device scheduling,
i.e., $\S_n^{\lambda_i} \notin \Phi$, where, $\Phi = \{\S_n^{\lambda_i} \in \phi : D_i - C_i \geq tsw_n^{\lambda_i}\}$
For each device λ_i, find the intra-task device scheduling compatible sleep state with the maximum value of energy-density function and name it $\S_h^{\lambda_i}$
Device Shut-Down Procedure:
When $j_{i,k}$ has used λ_i, consider the following criteria to shut-down λ_i and set the corresponding entry in the sorted list of timer if required.
if $(\S_h^{\lambda_i})$ **then**
    if $(\lambda_i^{NUT} - t \geq tsw_h^{\lambda_i})$ **then**
        Shut-down λ_i into $\S_h^{\lambda_i}$
        Timer = $\lambda_i^{NUT} - tr_h^{\lambda_i}$
    else
        Keep λ_i on
    end if
else if $(\S_{nc}^{\lambda_i})$ **then**
    if $(\lambda_i^{NUT} - t > bet_{nc}^{\lambda_i})$ **then**
        Shut-down λ_i into $\S_{nc}^{\lambda_i}$
        Timer = $\lambda_i^{NUT} - tr_{nc}^{\lambda_i}$
    else
        Leave λ_i on
    end if
end if
Device Wake-up Procedure:
When the initial timer to wake-up λ_i expires.
if $(\S_h^{\lambda_i})$ **then**
    if $(tr_h^{\lambda_i} \leq D_b)$ **then**
        $D_b = D_b - tr_h^{\lambda_i}$
        Keep λ_i off and register its entry in the SSSR
    else if $(S_e^{sz} > tr_h^{\lambda_i} \&\& S_e^{dl} \leq d_{i,k})$ **then**
        Where $d_{i,k}$ is the deadline of the job $j_{i,k}$ that will require λ_i in future.
        if $\Xi_i^t > 0$ **then**
            Register the device λ_i in ESSR
            Timer = $S_e^{sz} - tr_h^{\lambda_i} + \Xi_i^t$
        else
            Timer= $S_e^{sz} - tr_h^{\lambda_i}$
        end if
    else
        Wake-up the device λ_i
    end if
else if $(\S_{nc}^{\lambda_i})$ **then**
    Wake-up the device λ_i
end if

Algorithm 10 Aggressive Static Slack Container Algorithm for Multiple Sleep State Devices (SSC[a])

Offline Phase:
$\forall \lambda_i$ and its sleep states in $\vec{\S^{\lambda_i}}$, compute the energy-density function (Equation 5.10)
Sort the sleep states of each device in descending order with respect to their energy-density values
Mark sleep states not compatible with intra-task device scheduling,
i.e., $\S_n^{\lambda_i} \notin \Phi$, where, $\Phi = \{\S_n^{\lambda_i} \in \phi : D_i - C_i \geq tsw_n^{\lambda_i}\}$

Device Shut-Down Procedure:
When $j_{i,k}$ has used λ_i, consider the following criteria to shut-down λ_i and set the corresponding entry in the sorted list of timer if required.
if (Any Sleep State of $\lambda_i \in \Phi$) **then**
    $Flag = 1$
    for $(k = 0; k < SizeOf(\vec{\S^{\lambda_i}}); k++)$ **do**
        if $(\S_k^{\lambda_i} \in \Phi$ && $\lambda_i^{NUT} - t \geq tsw_k^{\lambda_i})$ **then**
            Shut-down λ_i into $\S_k^{\lambda_i}$
            Timer = $\lambda_i^{NUT} - tr_k^{\lambda_i}$
            $Flag = 0$
            Break
        end if
    end for
    if $(Flag)$ **then**
        Keep λ_i on
    end if
else
    $Flag = 1$
    for $(j = 0; j < SizeOf(\vec{\S^{\lambda_i}}); j++)$ **do**
        if $(\lambda_i^{NUT} - t > bet_j^{\lambda_i})$ **then**
            Shut-down λ_i into $\S_j^{\lambda_i}$
            Timer = $\lambda_i^{NUT} - tr_j^{\lambda_i}$
            $Flag = 0$
            Break
        end if
    end for
    if $(Flag)$ **then**
        Leave λ_i on
    end if
end if

not statically selects the most efficient sleep state but rather checks online the feasibility of each sleep state. Therefore, it overcomes the issue aforementioned in Algorithm 9 arising from the static allocation of the most efficient sleep state and is more aggressive to initiate device shut-down.

The pseudo-code of this algorithm is presented in Algorithm 10. In the offline phase, the energy-density of each sleep state is determined to prioritise the sleep states for each device. In-

stead of statically choosing any sleep state for a device to use online, this algorithm sorts the sleep states for each device in a descending order of their energy-density values. In the online phase the sleep states are considered in this predetermined order (i.e., descending order of their energy-density values) for shut-down. The scheduling, next utilisation time λ_i^{NUT}, device wake-up procedure and device budget D_b replenishment process are same when compared to SSC$^{\mathrm{m}}$.

The device shut-down phase differs from the previous algorithms. In the shut-down phase, a scheduler checks a device can safely transition into its most efficient sleep state. In case such a sleep state can not be safely initialised, the algorithm selects the next sleep state in the order to check its feasibility and so on. The device is kept in an active mode if and only if none of the sleep state satisfies the condition of $\lambda_i^{NUT} - t \geq tsw_n^{\lambda_i}$. A similar procedure is applied on the devices with no sleep states compatible with the intra-task device scheduling. The condition $\lambda_i^{NUT} - t > bet_n^{\lambda_i}$ is checked for each sleep state in the pre-determined order. The pseudo-code of the shut-down procedure in SSC$^{\mathrm{a}}$ is illustrated in Algorithm 10 from line number 5 to 33.

5.5 Evaluation of Device Power Management Algorithms

This section initially compares the overhead analysis of the proposed algorithms with the state-of-the-art and then show simulation results for a variety of different parameters.

5.5.1 Complexity Comparison

Recall W is the total number devices in the system. The complexity of the near optimal algorithm MDO [SC05] is $O(W\mathrm{H}^2)$, where H is the hyper-period. SYS-EDF [CG05] has a complexity of $O(V^m \times 2^W)$, where V^m is the number of frequency set-points. The complexity of EEDS [CG06] algorithm is $O(W\ell)$. Their algorithm performs the device transition decision on every job release, job completion and when the timer to reactivate the device expires. The state of all the devices is re-evaluated on each of the instants mentioned above. DFR-RMS [DA08a] has the same complexity of EEDS, i.e., $O(W\ell)$. The complexity of COLORS [CHT$^+$09] is $O(\nu\ell\varepsilon(\ln\ell\varepsilon)) + O(\nu\ell^2\varepsilon)$, where ε denotes the ratio of the largest period to the smallest period and ν presents the sum of the number of peripheral intervals in all tasks.

The SSC algorithm proposes the efficient device energy saving algorithm with low complexity. The overall complexity of our algorithm is $O(\ell)$. In our algorithm, a device state decision is made when a job requests the device, a job completes its execution or when the timer to activate the device expires. Unlike the state-of-the-art, only a device related to this job will be serviced, the status of the other devices is not re-evaluated. Within the algorithm, the only routine that has to compute the high priority work load has the complexity of $O(\ell)$. This routine is only used when the timer associated to a device expires and D_b is insufficient. Otherwise, all the other routines have the constant complexity of $O(1)$. The device budget reclamation algorithm has the same complexity of $O(\ell)$.

Let us consider the complexity of the multiple sleep state devices algorithms. Assume each device λ_i has Y_i number of sleep states. The offline complexity of the SSC^{o}, SSC^{m} and SSC^{a} algorithms is no more than $O(\sum_{i=1}^{W} Y_i)$. The online complexity of SSC^{o} is $O(1)$ (i.e., just one comparison is needed). The online complexity of all parts of SSC^{m} is same as the SSC algorithm. Similarly, the online complexity of the different parts of SSC^{a} is similar to the SSC algorithm except the device shut-down procedure. The complexity of the device shut-down procedure of this algorithm is $O(\max\{Y_1, Y_2, \cdots, Y_\ell\})$.

All device power management algorithms have a timer management system. The timer management mechanism store all the timers in a sorted list with respect to their expiration time. It has a complexity of $O(\ln \ell)$.

5.5.2 Experimental Setup

The SPARTS simulator is extended to incorporate the I/O devices and account their temporal behaviour and power dissipation. The proposed device power management algorithms are implemented in SPARTS. The state-of-the-art algorithm EEDS [CG06] is also implemented in SPARTS for the comparative analysis. SPARTS is used with the parameters specified in Table 5.1. The default parameters are underlined. While conducting experiments, it has been observed that borrowing of the budget in BE tasks from their future releases has negligible impact on the device power management. Hence, within the presented experiments, we concentrate on those which do not use BE borrowing.

Parameters	Values
Task-set sizes $\lvert\tau\rvert$	$\{5, \underline{10}, 25, 50\}$
Share of RT/BE tasks $\xi = \{\xi_1, \xi_2\}$	$\{\langle \underline{40\%, 60\%} \rangle, \langle 60\%, 40\% \rangle\}$
Inter-arrival time T_i for RT tasks	$[50ms, 200ms]$
Inter-arrival time T_i for BE tasks	$[200ms, 500ms]$
Sporadic delay limit $\Gamma \in$	$\{\underline{0}, 0.2, 0.4, 0.6, 0.8, 1\}$
BCET limit C^b	$\{0.25, 0.5, 0.75, \underline{1}\}$

Table 5.1: Simulator parameters used to evaluate device power management algorithms

Each task is allocated a device. The device usage time of each job within its execution is controlled with two variables that define the lower and upper bounds. These bounds are defined in a percentage of job's actual execution time. Suppose the percentage share of the device usage time, represented as Ω, in any job's actual execution time is randomly selected between two limits. Then the device usage time of a λ_i in any job $j_{i,k}$ is estimated as $\Omega\, c_{i,k}$. The overall system utilisation is varied from 0.1 to 0.95 with an increment of 0.05. Each task-set is simulated for 100 seconds. The power model for different devices used in our algorithms is based on their data sheets values shown in Table 5.2. All the parameters given in Table 5.2 are in milliwatts (for power) and milliseconds (for time). The transition time of the device's sleep state has been assumed when not given in its data sheet. Devices are selected for tasks in a round-robin way starting from the top of Table 5.2.

No.	Device Name	P^a	P^s	P^{tr}	t^{tr}
1.	Texas Instrument *CC*2430 [Tex07]	80.7	.0009	40	.525
2.	TJA1043 Transceiver [NXP13]	325	.01	162.5	.05
3.	Mica2Mote [Cro]	29	.145	72.5	5
4.	Lin Transceiver *NCV*7321 [ON 13]	19.2	.12	9.6	.15
5.	SST Flash *SST*39*LF*020 [CG06]	125	1	50	1
6.	SimpleTech Flash Card [CG06]	225	20	100	2
7.	MicroSSD(8GB)[mic11]	412.5	2.31	≈ 0	≈ 0
8.	Realtech Ethernet Chip [CG06]	190	85	125	10
9.	IBM MicroDrive [CG06]	1300	100	500	12

Table 5.2: Parameters of different devices

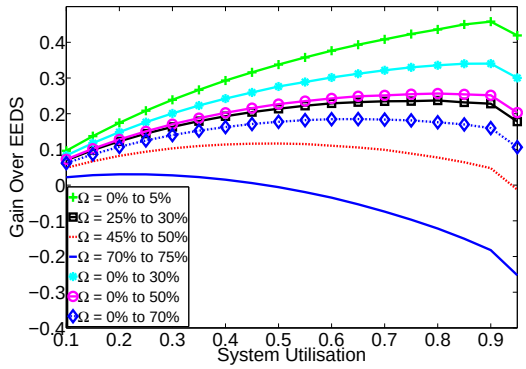


Figure 5.3: Variation in Ω

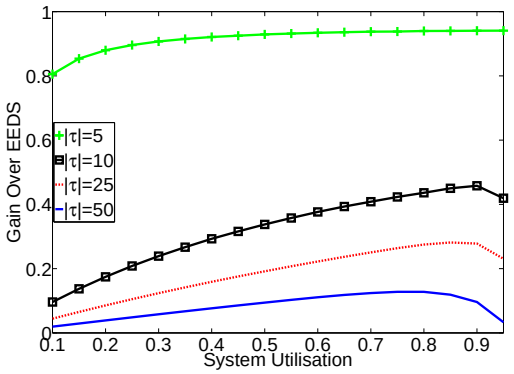


Figure 5.4: Variation in $|\tau|$ against U

5.5.3 Simulation Results of a Single Sleep State Devices Model

This section presents the comparison of the single sleep state devices algorithm SSC with EEDS by varying a set of different parameters. For each scenario, the reduction of the energy consumption of our algorithm over the EEDS algorithm is computed. The effect of variation in device usage time Ω on the gain of SSC over EEDS is illustrated in Figure 5.3. If the percentage of the device usage time is within a range of 0% to 70% of $c_{i,k}$, SSC outperforms EEDS. However, if all the jobs use their corresponding devices for a high percentage of their $c_{i,k}$, the performance of SSC declines eventually. For example, consider that the jobs use their corresponding devices for more than 70% of $c_{i,k}$, EEDS performance tends to rise after $U \geq 0.5$. Similarly, if the device usage time is in an interval of 45% to 50% of $c_{i,k}$ then EEDS saves slightly more energy after $U = 0.95$. This occurs because intra-task device scheduling algorithm is designed with a consideration that devices are used for very short intervals of time, allowing them to utilise sleep states within a wake-up on demand setting. EEDS is favourable for the systems with very high usage of device times (e.g. 70% to 75%). There is a dip in the gain at very high utilisations. This effect occurs due to an increase in the interference of high priority tasks at high utilisations. The SSC algorithm maintains the active state of the device until the task finishes its peripheral computation. The increase in interference of high priority tasks extends the active state of the devices and raises

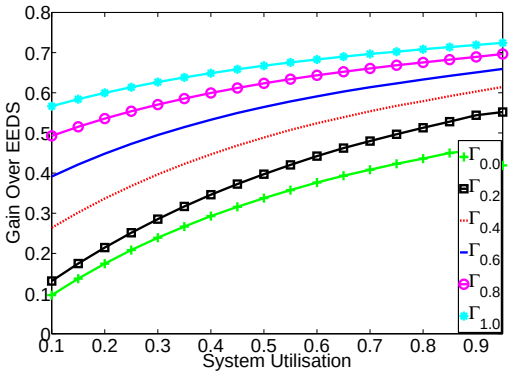


Figure 5.5: Variation in Γ

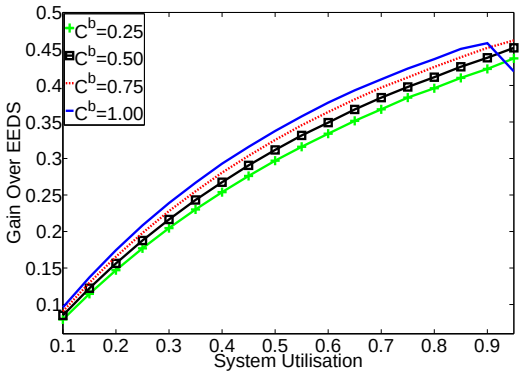


Figure 5.6: Variation in C^b ($|\tau| = 10$)

the energy consumption of the SSC algorithm. For all following experiments, the device usage time Ω for each job is selected randomly between 0% to 5% of the corresponding job's $c_{i,k}$ which reflects a more realistic application. The sole purpose of this selection is to present the effect of variation in difference parameters. All results presented in the subsequent figures scale with the gains presented in Figure 5.3 for different values of Ω.

The energy gain of SSC over EEDS is explored for different task-set sizes against the different system utilisations as shown in Figure 5.4. Figure 5.4 shows that the gain of SSC increases with an increase in the system utilisation. EEDS cannot extend sleep intervals of the devices at high utilisation. Moreover, with an increase in the task-set size, D_b has to service extra devices and thus the gain of SSC decreases. The gain of $|\tau| = 5$ is disproportionally high when compared to other task-set sizes, as favourable devices with less overheads are used in this experiment to illustrate that the effectiveness of SSC tends to rise with a decrease in device transition overheads.

The sporadic slack is also varied for different task-set sizes. Only the results of task-set size of 10 is depicted in Figure 5.5. An increase in Γ injects more sporadic slack in the schedule, and hence, extra sporadic slack allows for larger gains in energy consumption by SSC. SSC makes an efficient use of the sporadic slack as a device is only woken up on demand and kept in sleep mode if the task arrives later than its T_i. Contrary to this, EEDS has the requirement to keep the device on during C_i; therefore, devices are woken up assuming a worst-case scenario of task arrival after every T_i. As, it is impossible to predict the sporadic slack in the schedule, no mechanism to make use of the sporadic slack can be integrated into EEDS. Moreover, it is also observed that if we increase the task-set size the difference between the gain at low values of Γ increases when compared to the high values of Γ.

The third experiment highlights, the effect of variations in C^b (execution slack) on the energy gain of SSC over EEDS for different task-set sizes. The variation in C^b for task-set sizes of only 10 and 50 are shown in Figure 5.6 and Figure 5.7 respectively. SSC performs well with an increase in system utilisation. However, the gain decreases with an increase in execution slack for an obvious reason that if tasks finish their execution earlier than C_i, EEDS has a chance to turn their corresponding devices off immediately afterwards. There is one oddity in the form of a crossover

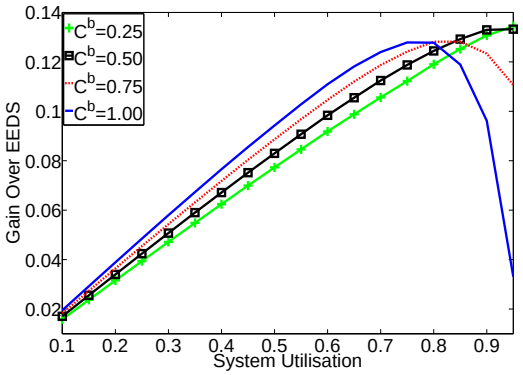


Figure 5.7: Variation in C^b ($|\tau| = 50$)

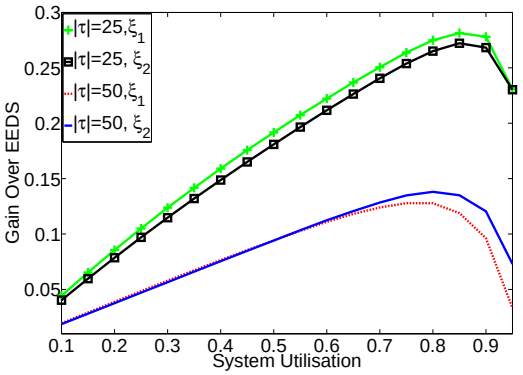


Figure 5.8: Variation in ξ

of high values C^b on the low values of C^b towards very high utilisations. This behaviour is more prominent for large task-set sizes as shown in Figure 5.7 for a task-set size of 50. This crossover point tends to move towards lower utilisations with an increase in the number of tasks. At very high utilisations, the interference caused by the high priority workload increases. The probability of the interference caused by the high priority workload also increases with an increase in C^b. The SSC algorithm is sensitive to this interference as a device's active state is maintained until a task finishes its peripheral computation. The high priority workload can pre-empt this task and can increase its device's active state length.

Figure 5.8 demonstrates the effect of variation in ξ for task-set sizes of 25 and 50. It has been observed that the gain in energy consumption of ξ_1 is higher than ξ_2 for small task-set sizes ($|\tau| \leq 25$) at all utilisations. In ξ_1, the percentage of BE tasks in task-set size is greater than ξ_2. Recall that BE tasks usually run for long intervals. Therefore, EEDS keeps the devices on for longer durations with ξ_1 when compared to ξ_2 and consequently consumes more energy. However, as the task-set size increases to 50, ξ_1's gain decreases over ξ_2 after a utilisation of 0.65. This effect is motivated by the interference of the high priority workload. As the number of long running BE tasks increases, the interference caused by their high priority workload also increases. The effect that SSC has the ability to turn their devices off early for long running BE tasks when compared to EEDS is dominated by the interference caused by their high priority workload at high utilisations for large task-set sizes and eventually causes a crossover of ξ_1 over ξ_2 in Figure 5.8.

To evaluate the scalability of EEDS and SSC with an increase in the number of devices, their simulation time and number of sleep decisions taken are compared for different task-set sizes. EDF is considered in this comparison because this algorithm is sleep agnostic and provides a baseline for the fair comparison. Actual simulated time is 100 seconds. Figure 5.9 shows the simulation times of the EEDS, EDF and SSC algorithm for different task-set sizes ranging from 10 to 200. It is evident that for large task-set sizes the simulation time of EEDS is much higher than the SSC and EDF algorithms. Similarly, Figure 5.10 compares the number of sleep decisions taken by EEDS and SSC for different task-set sizes. The number of sleep decisions of EEDS are also very high when compared to the SSC algorithm and these presented curves have rising trend.

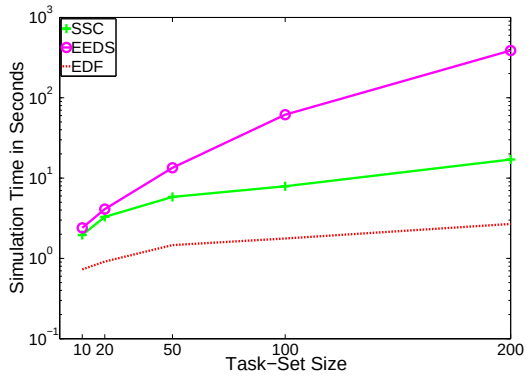


Figure 5.9: Simulation time comparison

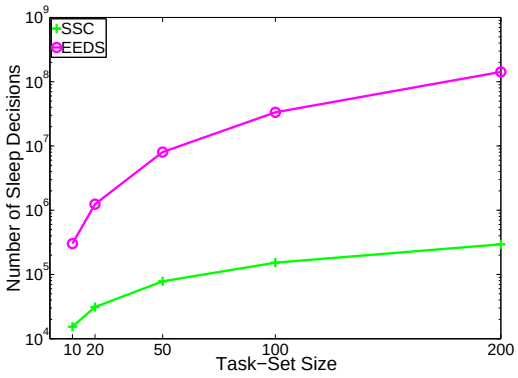


Figure 5.10: Sleep decisions comparison

5.5.4 Simulation Results of the Multiple Sleep State Devices Model

In this section, the performance of the algorithms corresponding to the multiple sleep state devices is analysed. Each device is assumed to have 3 sleep states numbered from 1 to 3. The devices given in Table 5.2 have only a single sleep state each, therefore, the other two sleep states are generated for all devices. The sleep state of a device given in Table 5.2 is considered to be its second sleep state. The parameters for the first and third sleep states are generated as follows. The power dissipation of $P_1^{\lambda_i}$ is considered to be 50% to 80% of active power dissipation of the device, i.e., $P_1^{\lambda_i} = [0.5, 0.8]P_A^{\lambda_i}$. The third sleep state is chosen to has a power dissipation of 20% to 50% of second sleep state, i.e., $P_3^{\lambda_i} = [0.2, 0.5]P_2^{\lambda_i}$. The transition delay of the first sleep state is considered 10% of the second sleep state ($tr_1^{\lambda_i} = 0.1tr_2^{\lambda_i}$) and similarly, for the third sleep state $tr_3^{\lambda_i} = [2.5, 6]tr_2^{\lambda_i}$. The random numbers are generated in these intervals for the different devices to generate a non-linear set of sleep states.

The performance of the three different heuristics (SSC$^{\text{o}}$, SSC$^{\text{m}}$ and SSC$^{\text{a}}$) is evaluated though extensive simulations. By applying the energy-density function, one can transform the multiple sleep state problem to single sleep state problem and consider EEDS algorithm for comparison. However, such comparison does not give any extra insights as both SSC and EEDS will have same sleep states, and will yield same results as discussed in the single sleep state model section. Therefore, three different algorithm are compared against each other and the EEDS algorithm is not included in the comparison.

The energy-density function has a major impact on the overall device energy consumption in our algorithms. The SSC$^{\text{o}}$ algorithm is used to evaluate the performance of the energy-density function. Each device selects the most efficient sleep state in this algorithm statically. Therefore, it is possible to generate all the possible combinations of assignment of different sleep states in different devices. The number of combinations of assignments of different sleep states in different devices with different task-set sizes varies by a factor of 3^{ℓ} (assuming each task uses a single device). Therefore, small task-set sizes of 5 to 10 are used for this experiment. The results of all these combinations for different task-set sizes are presented with the help of a boxplot given in Figure 5.11. Each candle presents the data corresponds to a task-set size given on the X-axis.

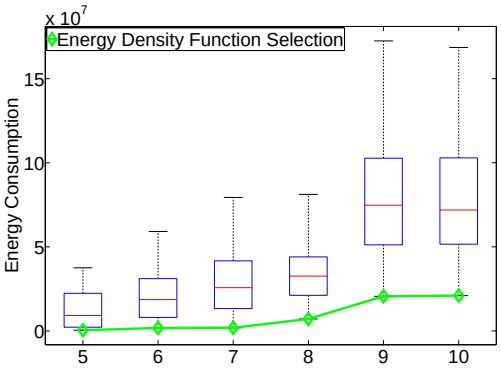


Figure 5.11: Efficiency of $\lambda_i^{ED_n}$

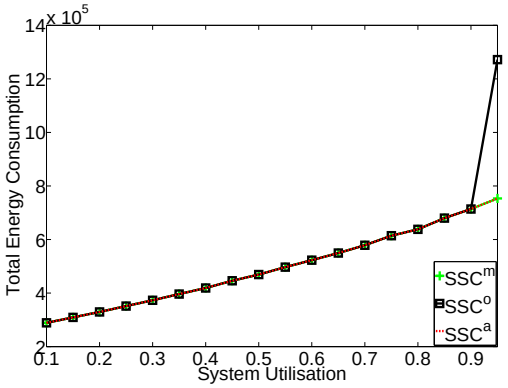


Figure 5.12: Variation in τ ($|\tau| = 5$)

The central mark is the median of energy consumption of all combinations, while the bottom and the top edges of the box are 25% and 75% percentile respectively. The top most and the bottom most values presents the maximum and the minimum energy consumption respectively for the correspond task-set size. The energy consumption with the sleep states selected for each device using the energy-density function is also shown in Figure 5.11. It is evident in most of the cases the energy consumption with our selection through energy-density function is equal the minimum energy consumption. The utilisation of the system is set to $U = 0.5$ for this experiment. We varied the utilisation and noticed that the results do not substantially change compared to what is shown in Figure 5.11.

In the rest of this section, the results of different heuristics (SSC^o, SSC^m and SSC^a) presented for multiple sleep state devices are compared with each other. In the first experiment, the task-set size is varied from 5 to 50. Figure 5.12 and Figure 5.13 only illustrate and compare the total energy consumption of different heuristics for the task-set sizes of 5 and 50 respectively. All algorithms performs similar for a small task-set size of 5. The transition overhead of the devices is small and the cumulative system resources are enough to compensate for their transitions. However, as the utilisation passes 0.9, the performance of the SSC^o algorithm degrades as $U_{leftover}$ is insufficient to accommodate transition delays of all the devices. On the other side (Figure 5.13), a very large task-set size of 50 distinguish the following three main features. 1) SSC^a always performs well and consumes less energy when compared to other two approaches due to its aggressive nature to utilise every opportunity to initiate a sleep state. 2) The gap between SSC^m and SSC^a widens with the increase in task-set size. 3) SSC^o consumes slightly less energy when compared to SSC^m for small utilisations values. The cross over point of SSC^o over SSC^m algorithm slides towards the low utilisations as the task-set size increases. This is motivated by an increase in the number of devices. $U_{leftover}$ cannot accommodate more devices at high utilisations. Moreover, devices are turned-on for larger intervals at higher utilisations due to the longer busy intervals. However, SSC^m performance enhances with an increase in the system utilisation. Its ability to collate execution slack and efficiently use device budget helps to increase the sleep duration of the devices at high utilisations. Moreover, longer busy intervals at high utilisations provide extra opportunity to

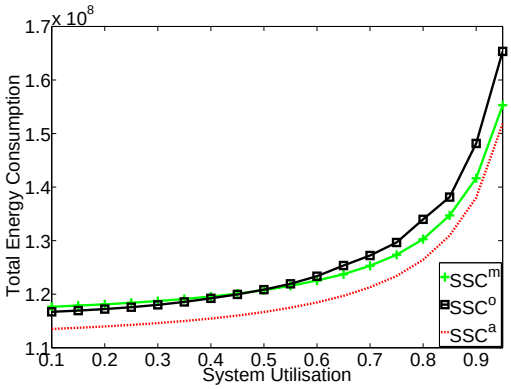


Figure 5.13: Variation in τ ($|\tau| = 50$)

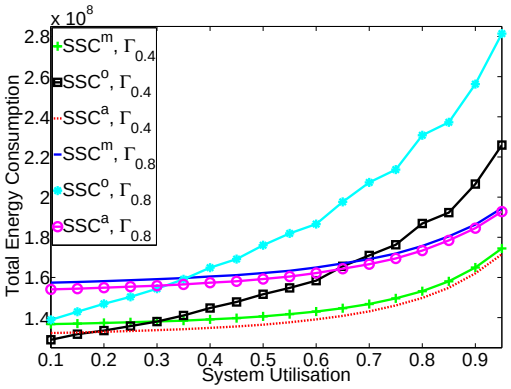


Figure 5.14: Variation in Γ ($|\tau| = 50$)

the device budget reclamation algorithm due to an overlap of devices/execution. This reclaimed budget can be allocated to the devices previously not being allocated static slack.

The total energy consumption of all heuristics for different value of Γ and task-set sizes are observed in this analysis. At low utilisations SSC^{o} performs better when compared to the SSC^{m} and SSC^{a} algorithms. However, at high utilisation, its performance degrades at a very high rate. This observation is evident in Figure 5.14 that illustrates the total device energy consumption of only 0.4 and 0.8 sporadic delay limit for a task-set size of 50. The crossover point of SSC^{o} over SSC^{m}/SSC^{a} shift toward right with an increase in the sporadic delay limit. Though SSC^{o}'s performance enhances at high Γ for low utilisations but at the same time at high utilisations with the same Γ its difference also increases from SSC^{m}/SSC^{a}. SSC^{o} statically allocates the static slack to the most efficient devices. Hence, these devices remain in low power states during the extra sporadic delay. The SSC^{m} and SSC^{a} algorithms are more opportunistic approaches and the static slack in the form of a device budget is distributed among devices in a first come first severed basis. Therefore, their performance is low when compared to SSC^{o} at low utilisations as the static slack is not targeted to the more amenable devices. However, the SSC^{o}'s ability to serve more device decreases at high utilisations and has adverse effect on the overall energy consumption. The SSC^{m} and SSC^{a} algorithms can use and reclaim device budget efficiently to prolong the sleep states of more devices, and hence save more energy at high utilisations.

In the same context, the difference between SSC^{m} and SSC^{a} algorithms virtually stays the same, as both algorithms work on the same principles except that SSC^{a} exploits every opportunity (available slack) to shut-down the devices even in the shallower sleep states. Another interesting observation is an effect of Γ over the energy consumption of the same algorithm. It has been observed that for large task-set sizes the energy consumption of the algorithm increases with an increase in the value of Γ. For instance, consider energy consumption of SSC^{m} given in Figure 5.14 which increase with the value of Γ. Conversely, the energy consumption decreases with an increase in the value of Γ for small task-set sizes. This behaviour is evident from Figure 5.15 that presents the energy consumption of SSC^{m} for two different values Γ for a task-set size of 5. The rational behind such behaviour is the availability of the sufficient slack in the schedule to

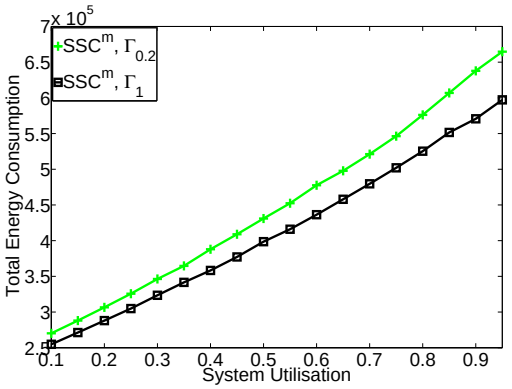


Figure 5.15: Variation in Γ ($|\tau| = 5$)

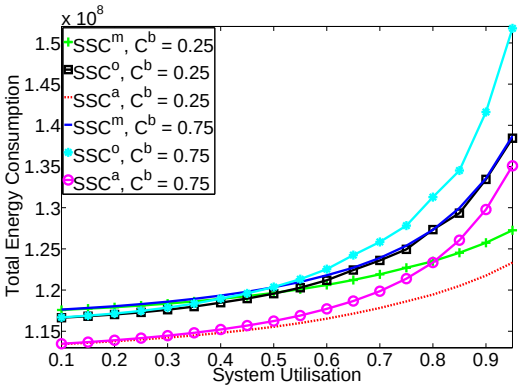


Figure 5.16: Variation in C^b ($|\tau| = 50$)

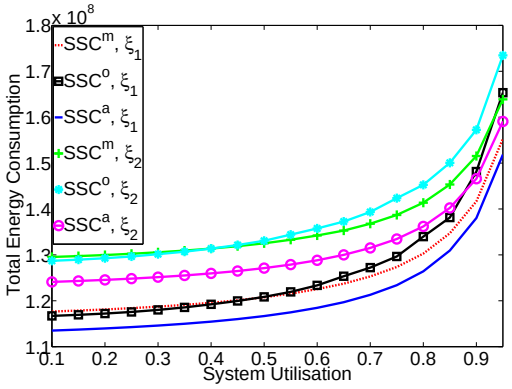


Figure 5.17: Variation in ξ ($|\tau| = 50$)

accommodate the transition delays. If the slack available to the algorithm is sufficient to accommodate all the transitions delays, the energy consumption decreases with an increase in the value of Γ. However, if it is insufficient, some of the devices have to stay awake for longer duration due to their sporadic delays and hence the energy consumption increases with increasing value of Γ.

The effect of execution slack variation is also analysed for different task-set sizes. Figure 5.16 only presents the results for the two different values of $C^b = 0.25, 0.75$ with a task-set size of 50. A small value of C^b means more execution slack. One of the observations is that the energy consumption of all the algorithms decreases with an increase in the execution slack at high utilisations. However, execution slack variation poses no effect at low utilisations. A high execution slack at high utilisations decreases the busy intervals as the interference caused by the high priority workload is low and the tasks finish their executions earlier. Therefore, as the execution slack increases devices can be sent to sleep states earlier compared to a case when tasks execute for longer intervals due the interference cause by the high priority workload. A second effect is the widening of the gap between the SSC^o and SSC^m/SSC^a algorithms with a decrease in execution slack at high utilisations. Moreover, the gap between SSC^m and SSC^a decreases at very high utilisations with a decease in the execution slack. The extra execution slack allows SSC^a to shut-down its devices into a inefficient sleep states if the slack is not sufficient for the most efficient sleep state.

The distribution of the task-set size between RT and BE task has a high impact on the energy consumption of the devices. Figure 5.17 presents the results for the two different distributions. For all algorithms, ξ_1 consumes less energy when compared to ξ_2. This observation holds for all task-set sizes. ξ_1 has more long-period BE type tasks, which allows to shut-down their devices for longer intervals. Therefore, ξ_1 consumes less energy when compared to ξ_2. The difference between ξ_1 and ξ_2 for all algorithms decreases at high utilisations. This behaviour is caused by an increase in the interference of high priority workload to long-period BE tasks.

Chapter 6

Global Scheduler and Power Management

In the last decade, the semiconductor industry has experienced a paradigm shift from single processor design to multicore architecture era. As mentioned previously, a multicore architecture combines two or more processing units (cores) into a single package (single or multiple dies) and can execute programs simultaneously. This is driven by the fact that the increase in clock speed to enhance the performance of the processor has hit its limits as the performance per watt became costly at high frequencies and the energy dissipated at high frequencies requires special packaging techniques to reduce the generated temperature. The complexity involved in scaling the instructions level parallelism with increasing clock speed also favours a multicore design. The increasing difference between memory and processor speed is another factor that caused this paradigm shift. The simple cache design of single processor has helped to solved this issue up to some extent but it cannot be sustained in the long term. The multicore design has allowed to maintain Moore's law along with relatively high performance per watt ratio. It has the potential to execute multiple instructions in parallel which has to be mostly driven by the efficient programming tools. A high performance per area ratio allows to pack extra functionality on a single chip. However, not all of this comes for free. The multicore architecture also have some challenges and issues. Among others, power and temperature management are two main concerns. Most of the multicore platforms use level-1 distributed caches and level-2 shared caches that give rise to coherence challenges. Moreover, applications should be written in a way to exploit the full potential offered by these architecture.

Homogeneous multicore platforms (or identical multicore platforms) have been widely deployed in cutting-edge application domains such as the mobile-phone, avionics and automotive industry. One of the major issues is the energy management of such platforms to prolong the battery life of the embedded devices. The processing units in homogeneous multicores have most characteristics of a single processor. These cores are equipped with sleep states to shut-down the certain parts of the core on demand with some additional overhead associated to each sleep transition. The effect of leakage current in multicore is exaggerated due to the thermal issues involved

in such platforms. Conscious exploitation of such sleep states is required which is a non-trivial exercise especially in the context of HRT systems. Over the last two decades, the RT community has been actively developing sophisticated algorithms for HRT systems to guarantee the timing constraints of all tasks while scheduling the tasks on the cores. The task-to-core scheduling can usually be performed by using a scheduler which belongs to one of the following aforementioned three classes, (1) partitioned schedulers; (2) global schedulers, and finally (3) semi-partitioned schedulers. The partitioned schedulers are the most common choice as the uniprocessor scheduling techniques can be directly applied after the task-partitioning. However, global schedulers are now emerging as an alternative choice since applications are becoming more and more complex and this type of schedulers offers more flexibility in terms of scheduling solutions. Furthermore, the global scheduling algorithms remove the need of partitioning the tasks among the processors. This chapter focuses on global schedulers.

The state-of-the-art have tackled the problem of reducing the total energy consumption upon homogeneous multicore platforms with limited or intermittent power supply, but mainly by using partitioned-schedulers. However, the results tackling the same problem with global scheduling techniques are very limited and only focus on DVFS capabilities of the multicore platform to reduce the energy consumption. Moreover, to the best of our knowledge, the related work on static power dissipation minimisation in the context of global-schedulers does not exist. This work fills this gap. The proposed algorithms (*i*) exploits the spare capacity in the schedule of a set of tasks on each core to either initiate a sleep state on this core or prolong the sleep state of the cores already in sleep state; and (*ii*) has low complexity with up to 70% of the energy consumption saved in idle intervals – when a core is not performing any execution – over the baseline global-EDF schedule [DL78].

6.1 Preliminaries

In this section, the extensions in the system model along with the key concepts (expected release time, usable execution slack, usable idle slack) needed to understand our proposed global power management (GPM) algorithm presented in Section 6.2 are discussed in details.

6.1.1 Extensions in the System Model

A task-set composed of ℓ sporadic constrained-deadline tasks is assumed in this chapter. The standard parameters of the tasks and their jobs defined in Section 3.1.1 are adopted. The budget $a_{i,k}$ of a job is initialised with C_i at the release of $j_{i,k}$. A symmetric multicore platform (SMP) $\pi \stackrel{\text{def}}{=} \{\pi_1^1, \pi_2^1, \ldots, \pi_M^1\}$ composed of M homogeneous cores is considered. Homogeneous means that all the cores have the same computational capabilities and are of same processor type. A core has the same properties as mentioned in Section 3.1.3.1. As all the processors are of same type, therefore, to simplify the notations, the superscript of the processor is dropped in this chapter, i.e., π_m^1 is represented as π_m. Tasks are scheduled with a *fully-preemptive global earliest deadline first* scheduler (GEDF) in which: (*i*) a constant priority is assigned to each job upon its release (the

earlier the absolute deadline $d_{i,k}$ of a job, the higher its priority); and (*ii*) a running job can be interrupted at any time-instant and have its execution resumed later at no cost or penalty on the same core as, or a different core from, the one on which it was executing prior to the interruption. Note that different jobs of the same task will have different priorities.

6.1.2 Expected Release Time

Since $D_i \leq T_i$ for task τ_i (with $i \in [1,\ell]$), there is at most one active job from task τ_i at any time instant t. Let $\gamma \stackrel{\text{def}}{=} \{\gamma_1, \gamma_2, \ldots, \gamma_\ell\}$ denote the set which maintains the information about the "next earliest release time" of the next job to be released by each task. Such an information can easily be computed from the last release time of each task in τ. For each task τ_i, if $j_{i,k}$ is the last released job, then it holds true that $\gamma_i = r_{i,k} + T_i$ as T_i is the minimum inter-arrival time between two consecutive jobs of τ_i. Note that one element of γ is updated on every job release. In the rest of the chapter, it is assumed that γ is sorted in a non-decreasing order with respect to their earliest expected release times. Without any loss of generality, the first element of γ is denoted as $\gamma^{(1)}$, the i^{th} element as $\gamma^{(i)}$, and the last element as $\gamma^{(\ell)}$. Note that the order of the elements in γ after sorting may not be same as the indices of the tasks, i.e., $\gamma^{(i)}$ may not be equal to γ_i.

6.1.3 Usable Execution Slack

In this chapter, the execution slack generated by a job $j_{i,k}$ is denoted as $S_e(j_{i,k})$. Formally speaking, if t denotes the time at which $j_{i,k}$ finishes its execution, then $S_e(j_{i,k})$ is the remaining execution budget of $j_{i,k}$ at time t, i.e., $S_e(j_{i,k}) = a_{i,k}$. Assume, a job $j_{i,k}$ completes its execution at time t and produces an execution slack $S_e(j_{i,k})$. Only a part of this slack can be used to reduce the power dissipation of the platform at time t with our proposed algorithm in this chapter. The size of $S_e(j_{i,k})$ is split in two parts: (*i*) the usable execution slack $S_e^u(j_{i,k})$, which may be used directly at time t, and (*ii*) the non-usable execution slack $S_e^{nu}(j_{i,k})$ which cannot be used at time t directly. Note that $S_e^{sz}(j_{i,k}) \stackrel{\text{def}}{=} S_e^u(j_{i,k}) + S_e^{nu}(j_{i,k})$ and the deadline of the usable as well as non-usable slack is same, i.e., $S_e^{dl}(j_{i,k})$.

Let t' denote the earliest time instant from time t at which job $j_{i,k}$ would have been pre-empted by any other higher priority job if it had executed for its WCET. Then, it holds true that the usable execution slack (i.e., the slack directly usable at time t) $S_e^u(j_{i,k})$ is smaller than or equal to $t' - t$. Informally speaking, the size of $S_e^u(j_{i,k})$ can be computed by observing the "next earliest arrival" of a job with a higher priority than $j_{i,k}$. To find this earliest time, the scheduler uses γ as presented in Section 6.1.2 and searches for the first element $\gamma_q \in \gamma$ such that $d_{i,k} > \gamma_q + D_q$, i.e., the next job released by task τ_q has a higher priority than $j_{i,k}$ assuming that it is released as soon as it is legally permitted to do so. To correctly understand this, recall that: (*i*) γ is sorted in a non-decreasing order of expected release times; and (*ii*) jobs are scheduled according to GEDF, i.e., the earlier the deadline of a job, the higher its priority. Consequently, the usable execution slack

$S_e^u(j_{i,k})$ is provided by Equation 6.1. From Equation 6.1, $S_e^u(j_{i,k})$ is always a non-negative number ($S_e^u(j_{i,k}) \geq 0$) in the one hand, and in the other hand, $S_e^u(j_{i,k})$ is bounded by $\min\{S_e^{sz}(j_{i,k}), \gamma_q - t\}$.

$$S_e^u(j_{i,k}) \stackrel{\text{def}}{=} \max\left\{0, \min\{S_e^{sz}(j_{i,k}), \gamma_q - t\}\right\} \tag{6.1}$$

The usable execution slack $S_e^u(j_{i,k})$ generated by the completion of job $j_{i,k}$ on a core π_m at time instant t is stored in a container associated to π_m and denoted as $\pi_{m,S_e^u}(t)$. Note that $\pi_{m,S_e^u}(\cdot)$ is a monotonically decreasing function between two replenishments. Consequently, if t_r is the first instant after t where $\pi_{m,S_e^u}(\cdot)$ is replenished due to the generation of usable execution slack on π_m, then for any time instant t' such that $t \leq t' < t_r$ it holds that: $\pi_{m,S_e^u}(t') \stackrel{\text{def}}{=} \max\{0,\ \pi_{m,S_e^u}(t) - (t' - t)\}$.

6.1.4 Usable Idle Slack

Idle times in the schedule can be caused by both static and dynamic slack. However, recall this work considers only a portion of the dynamic slack, namely the execution slack. The execution slack which is not directly usable to send a core into a sleep state results in idle times, since the actual workload executing on the platform is less than initially expected, i.e., the jobs executed for less than their WCET. Similarly, the static slack is a consequence of a non fully loaded platform, i.e., the sum of the utilisations of the tasks is less than the platform capacity, and hence results in idle times.

Instead of staying idle, a core could enter a sleep state and hence save some energy. Before entering a sleep state, the scheduler must however know for how long the core can stay asleep without jeopardising the correctness of the schedule. This time is lower-bounded by the time for which the core would have stayed idle (see Lemma 41 in Section 6.3). However, because of sporadic task delays, the scheduler cannot know the exact length of the idle time beforehand. It can nevertheless lower-bound this time by assuming the earliest expected release time in the system using the set γ. Indeed, a core may stop being idle as soon as a new job is ready to be executed.

The size of the usable idle slack available at time t on an idle core π_m is denoted as $S_i^u(t, \pi_m)$. Knowing that γ is sorted in a non-decreasing order of earliest release time, the usable idle slack of the first idle core at time instant t is equal to $\gamma^{(1)} - t$, while the usable idle slack of the n^{th} idle core is $\gamma^{(n)} - t$.

6.2 Proposed Energy Saving Algorithm

The intuitive idea of the algorithm proposed in this paper is to exploit the available slack in the schedule to initiate a sleep state on one or multiple cores for energy minimisation purposes. However, the sporadic slack is difficult to predict beforehand. Therefore, this algorithm focuses only on the static slack and the execution slack in the online phase of the proposed algorithm.

6.2.1 Exploiting the Usable Execution Slack

The usable execution slack $S_e^u(j_{i,k})$ produced at time instant t by a job $j_{i,k}$ on a core π_m can be used in three different ways to reduce the power dissipation of the platform.

6.2.1.1 Initiating a Sleep State on a Given Core

The usable execution slack $S_e^u(j_{i,k})$ can be assigned to π_m, i.e., $\pi_{m,S_e^u}(t) = S_e^u(j_{i,k})$, and then used to initiate a sleep state on this core. If its size is greater than the break-even time bet_n of any sleep state, π_m can transition into a sleep state. The sleep state is selected such that it minimises the energy consumption on the total slack length. The chosen sleep state for π_m is therefore the sleep state $\S_n$ such that $\S_n$ minimises the function $Cons(\pi_{m,S_e^u}(t),n)$ given in Equation 6.2. Please note that all the processors have the same set of sleep states. Core π_m will then wake up at time $t_w = t + \pi_{m,S_e^u}(t)$.

$$Cons(x,n) \stackrel{\text{def}}{=} Es_n + (x - 2 \times tr_n)P_n \tag{6.2}$$

6.2.1.2 Donation to Other Jobs

The usable execution slack $\pi_{m,S_e^u}(t)$ not sufficient to initiate a sleep state on π_m at time t can be passed on to other jobs with low priorities executing on π_m in the time interval $[t, t + \pi_{m,S_e^u}(t))$. This action is performed with the expectation that the execution slack on π_m will aggregate with the slack of the jobs executing in this time interval. This accumulation allows us to add up portions of execution slack distributed in the schedule which would have otherwise been too small to enable a core to enter a sleep state. To this end achievement, the budget $a_{p,q}$ of job $j_{p,q}$ executing in this interval $[t, t + \pi_{m,S_e^u}(t))$ is not reduced. Consequently, job $j_{p,q}$ will eventually finish before having consumed its execution budget $a_{p,q}$ and generate more execution slack than initially expected.

To keep the correctness of the schedule (see Lemma 42 in Section 6.3), the active jobs must be assigned first to cores without any execution slack. Then, if there is no core without execution slack that can execute job $j_{p,q}$, it is assigned to a core π_m with $\pi_{m,S_e^u}(t) > 0$. In that case, its execution budget $a_{p,q}$ is not decreased when executing, unless $\pi_{m,S_e^u}(t)$ becomes equal to 0 or $j_{p,q}$ migrates to another core π_s without execution slack (i.e., $\pi_{s,S_e^u}(t) = 0$). The rational behind such a condition is explained as follows. Job assigned to a core with usable execution slack in the presence of an idle core does not decrement its budget. However, in the original schedule — where all the jobs execute for their WCET — the budget of this job would have decremented as scheduler would have assigned it to an idle core.

Example 6. *Consider a system with three tasks* $\tau_1 \stackrel{\text{def}}{=} < 4, 12, 12 >$, $\tau_2 \stackrel{\text{def}}{=} < 10, 20, 20 >$ *and* $\tau_3 \stackrel{\text{def}}{=} < 6, 24, 24 >$, *and two cores* π_m *and* π_s. *Under* GEDF *at* $t = 0$, τ_1 *is assigned the highest priority and* τ_3 *is assigned the lowest priority. Initially,* τ_1 *and* τ_2 *are assigned to* π_m *and* π_s, *respectively, as shown in Figure 6.1. However, task* τ_1 *completes its execution earlier than its WCET and generates a usable execution slack of* 2 *time units at* $t = 2$ *(Figure 6.2). Assume that a usable execution slack*

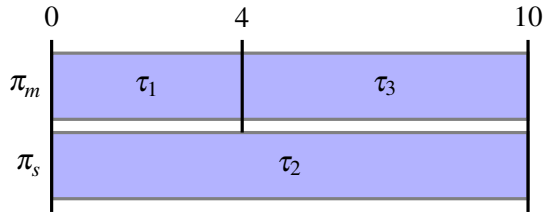


Figure 6.1: Initial schedule when all tasks execute for their WCET

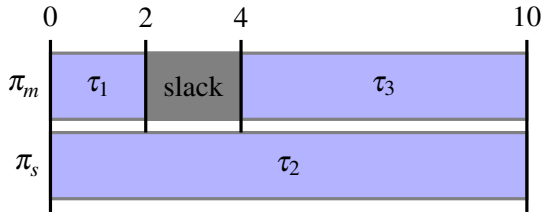


Figure 6.2: Task τ_1 generates a slack at time instant 2

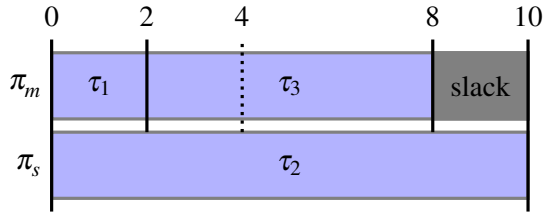


Figure 6.3: Task τ_3 starts its execution earlier at time instant 2

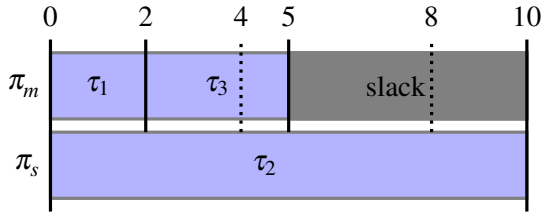


Figure 6.4: Task τ_3 generates a slack at time instant 5

of 2 time units cannot be used to initiate a sleep state. Therefore, τ_3 starts its execution earlier on π_m ($t = 2$) as illustrated in Figure 6.3. The execution budget $a_{3,k}$ of task τ_3 is initialised to its WCET, i.e, $a_{3,k} = 6$ time units at $t = 2$. As τ_3 is executing within a usable execution slack of τ_1 from $t = 2$ to $t = 4$, its execution budget $a_{3,k}$ is not decremented. Therefore, it will have an execution budget of $a_{3,k} = 6$ time units at $t = 4$. If τ_3 should execute for its WCET, then it would complete its execution at $t = 8$ and still have $a_{3,k} = 2$, thereby generating an execution slack of 2 time units at $t = 8$. On the other hand, if τ_3 completes its execution earlier than its WCET, lets say at $t = 5$ as shown in Figure 6.4, then, the execution slack generated by τ_3 will be equal to 5 time units as τ_3 will execute from its budget in an interval $[4,5]$. Note that the execution slack generated by τ_3 at $t = 5$ is the summation of the slack generated by τ_1 and τ_3, i.e., $(2+3=5)$. This example shows how a distributed execution slack can be accumulated.

There is also a possibility that a job assigned to core with usable execution slack and executing inside a usable execution slack interval generates additional slack, i.e., a job generates execution slack on π_m at time t while $\pi_{m,S_e^u}(t) > 0$. In order to avoid any alteration of the original schedule and reduce the complexity of the algorithm, such newly generated slack is not taken into account and ignored.

6.2.1.3 Donation to Other Cores

A portion of the usable execution slack can be considered for donation to other cores already in sleep states. This donation is performed such that it minimises the overall energy consumption. The usable slack can be donated only to cores that are already in a sleep state and not in their wake-up transition phase. Assume that a core π_s is in a sleep state $\S_n$ at time t and is supposed to wake-up at time instant $t_w > t$. Also assume that a job $j_{i,k}$ just induced a usable execution slack $S_e^u(j_{i,k})$ on core π_m at time t. The non-overlapping time interval between $S_e^u(j_{i,k})$ on core π_m and

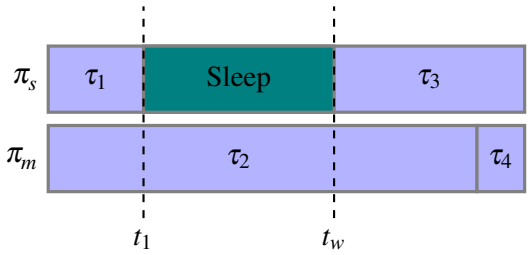


Figure 6.5: Schedule if τ_2 executes for its WCET

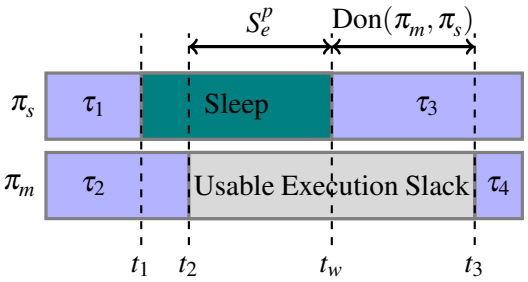


Figure 6.6: Schedule when τ_2 completes early at time t_2

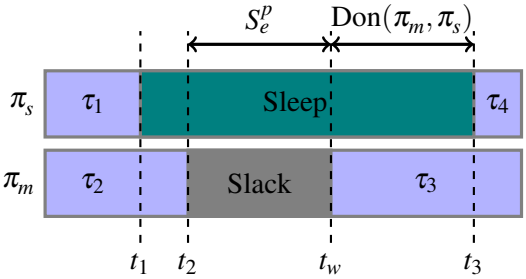


Figure 6.7: Schedule after a slack donation from π_m to π_s

the sleep interval on core π_s can be donated to π_s to prolong its sleep state. The donation size is denoted as $\mathrm{Don}(\pi_m,\pi_s)$ and computed with Equation 6.3.

$$\mathrm{Don}(\pi_m,\pi_s) = \begin{cases} t + S_e^u(j_{i,k}) - t_w & \text{if } t + S_e^u(j_{i,k}) > t_w \\ 0 & \text{if } t + S_e^u(j_{i,k}) \leq t_w \end{cases} \tag{6.3}$$

For practical reasons, a slack donation from π_m to π_s is prohibited if π_s is in the transition phase from "sleep" to "active" state at time t. Indeed, the transition from one state to another can never be aborted and π_s should already be considered as no longer being in sleep state.

In case the donation has been successful, the wake-up time of π_s can be extended to $t_w + \mathrm{Don}(\pi_m,\pi_s)$. The remaining portion of the usable execution slack on π_m, which is parallel with the sleep interval of π_s, is denoted as S_e^p and its length is $S_e^p = S_e^u(j_{i,k}) - \mathrm{Don}(\pi_m,\pi_s)$. It can either be: (1) assigned to the execution slack of π_m (i.e., $\pi_{m,S_e^u}(t) = S_e^p$) or used to initiate a sleep state on π_m, as explained in Section 6.2.1.1, or (2) given to jobs executing in the time interval $[t, t + \pi_{m,S_e^u}(t))$ as discussed in Section 6.2.1.2. To better understand the slack donation process, consider the example given below.

Example 7. *Assume a system with four tasks $(\tau_1,\tau_2,\tau_3,\tau_4)$ and two cores. The priorities at time instant t are such that the lower the index of a task, the higher its priority. At time instant t_1, a core π_s is in a sleep state for an interval $[t_1,t_w]$ and π_m is executing task τ_2 as shown in Figure 6.5. Task τ_2 completes its execution earlier than its WCET and generates a slack at t_2 (see Figure 6.6). Assume that the scheduler donates the slack to the other core already in sleep state, i.e., π_s. The size of donation in this scenario will be* $\mathrm{Don}(\pi_m,\pi_s) = t_2 + \pi_{m,S_e^u}(t_2) - t_w = t_3 - t_w$. *The portion of*

the parallel slack is $S_e^p = t_w - t_2$. As demonstrated in Figure 6.7, this technique allows us to extend the sleep interval of the cores already in a sleep state.

The criterion to decide whether to donate the slack produced on core π_m to another core π_s (to extend its sleep duration), or use it on π_m (to initiate a sleep state) is based on a comparison of the total energy saving in both cases. Assume, a slack $S_e^u(j_{i,k})$ is produced on core π_m. The scheduler tries to donate a portion of this slack to core π_s currently in sleep state $\S_q$. It computes and compares the energy consumption with and without donation. The energy consumption in case of donation is equal to: (*i*) the energy consumption on core π_m in the most energy efficient sleep state for a duration equal to the parallel portion of the slack S_e^p, augmented with (*ii*) the energy consumption of the donated portion $\mathrm{Don}(\pi_m, \pi_s)$ on π_s in its sleep state $\S_q$. Such an energy consumption is expressed as $E_D(\pi_m, \pi_s)$ and computed with Equation 6.4, by using function $\alpha(x)$ (Equation 6.6) which selects the sleep state that minimises the energy consumption during an interval x, and returns the energy consumption in that sleep state. Similarly, the energy consumption without donation is $E_W(\pi_m)$. It is equal to the energy consumption on core π_m for duration $S_e^u(j_{i,k})$ time units in the most energy efficient sleep state and is computed by using Equation 6.5. The comparison of $E_W(\pi_m)$ and $E_D(\pi_m, \pi_s)$ allows to decide whether to donate a portion of the usable execution slack of π_m to π_s or not.

$$E_D(\pi_m, \pi_s) = P_q \times \mathrm{Don}(\pi_m, \pi_s) + \alpha(S_e^p) \tag{6.4}$$

$$E_W(\pi_m) = \alpha(S_e^u(j_{i,k})) \tag{6.5}$$

$$\alpha(x) = \begin{cases} \min\limits_{n \in \mathbb{N}} \{Cons(x,n) \mid x > bet_n\} & \text{if } \exists n \mid x > bet_n \\ x \times P_{idle} & \text{Otherwise} \end{cases} \tag{6.6}$$

6.2.2 Exploiting the Usable Idle Slack

The usable idle slack $S_i^u(t, \pi_m)$ available on a core π_m at time t is only used to initiate a sleep state and is never donated to other cores or jobs. The size of the usable idle slack depends on the order in which the cores have become idle in the system (see Section 6.1.4). A sleep state that minimises Equation 6.2 is selected for the idle core. Core π_m transitions to an idle state if the size of the usable idle slack is not sufficient to initiate a sleep state, i.e., there is no $n \in \mathbb{N}$ such that $S_i^u(t, \pi_m) > bet_n$.

6.2.3 Algorithmic Summary

The pseudo-code of the proposed global power management algorithm named GPM, is presented in Algorithm 11. A job $j_{i,k}$ on its completion on π_m may generate execution slack $S_e(j_{i,k})$. The scheduler initially compute the usable execution slack $S_e^u(j_{i,k})$ through Equation 6.1. If there exists already slack on a core π_m (i.e., $\pi_{m,S_e^u}(t) > 0$), the newly generated usable execution slack $S_e^u(j_{i,k})$

Algorithm 11 Global Power Management Algorithm (GPM)

Whenever a job $j_{i,k}$ Generates an Execution Slack on Core π_m at time instant t:
Compute the usable execution slack $S_e^u(j_{i,k})$ (Equation 6.1)
if ($S_e^u(j_{i,k}) > 0$ && $\pi_{m,S_e^u}(t) \neq 0$) **then**
    Discard $S_e^u(j_{i,k})$ /* Slack generated inside a slack */
else
    $\pi_{m,S_e^u}(t) = S_e^u(j_{i,k})$
end if
PerformSlackDonationProcess(π_m, $\pi_{m,S_e^u}(t)$)
InitiateSleepOnCores()

PerformSlackDonationProcess(π_m, $\pi_{m,S_e^u}(t)$)
for all $s = \{1,\ldots,M\} \setminus m$ **do**
    if (π_s in Sleep State && π_s not in Wakeup Phase) **then**
        Compute $\mathrm{Don}(\pi_m, \pi_s)$ (Equation 6.3)
        if ($\mathrm{Don}(\pi_m, \pi_s) > 0$ && $E_D(\pi_m, \pi_s) < E_W(\pi_m)$) **then**
            $\pi_{s,S_e^u}(t) = \pi_{m,S_e^u}(t) + \mathrm{Don}(\pi_m, \pi_s)$
            $\pi_{m,S_e^u}(t) = \pi_{m,S_e^u}(t) - \mathrm{Don}(\pi_m, \pi_s)$
        end if
    end if
end for

InitiateSleepOnCores()
$next = 1$
for all $j = \{1,\ldots,M\}$ **do**
    if (π_j is in Active State && $\pi_{j,S_e^u}(t)$ is Sufficient to Initiate a Sleep State) **then**
        Send π_j into a sleep state
    else if (π_j is in Sleep State && $\pi_{j,S_e^u}(t)$ Updated) **then**
        Extend the sleep state to $t + \pi_{j,S_e^u}(t)$
    else if (π_j is in Idle Mode && $\max\{\gamma^{(next)} - t, \pi_{j,S_e^u}(t)\}$ is Sufficient to Initiate a Sleep State) **then**
        Sleep for $\max\{\gamma^{(next)} - t, \pi_{j,S_e^u}(t)\}$
        $next = next + 1$
    else
        Keep π_j in idle mode
    end if
end for

Scheduling:
Assign jobs from the ready queue to the active and idle cores in priority order. First assign to cores without slack and then to cores with usable execution slack

Execution Inside a Usable Execution Slack:
A job $j_{i,k}$ executing on a core π_m with usable execution slack $\pi_{m,S_e^u}(t)$ does not decrease its execution time from its budget $a_{i,k}$

is discarded as the execution slack is generated inside another execution slack. Otherwise, the slack $\pi_{m,S_e^u}(t)$ of core π_m is updated to $S_e^u(j_{i,k})$.

A slack donation process is performed on core π_m to any other core π_s already in a sleep state, if it satisfies the following conditions: (*i*) π_s is not in wake-up transition phase, (*ii*) there is slack to donate, i.e., $\mathrm{Don}(\pi_m,\pi_s) > 0$ (computed with Equation 6.3) and (*iii*) it is energy efficient, i.e., $E_D(\pi_m,\pi_s) < E_W(\pi_m)$ (see Equation 6.4 and Equation 6.5). Note that the first condition is needed to satisfy the physical property of the sleep state that requires to complete its transition once initiated. The size of the donated usable execution slack is $\mathrm{Don}(\pi_m,\pi_s)$ and is equal to the non-overlapping time interval between the slack of π_m and the sleep state time of π_s. This donation process continues as long as π_m has slack to donate.

Afterwards, the scheduler tries to initiate a sleep state on active cores. A core π_j, in active state with sufficient usable execution slack, transitions into a sleep state (i.e., there exists $n \in \mathbb{N}$ such that $\pi_{j,S_e^u}(t) > bet_n$). Similarly, any update in the usable execution slack of a core already in sleep extends its sleep interval. An idle core can be considered to initiate a sleep state, if the maximum of usable idle slack (computed as explained in Section 6.1.4) and the usable execution slack is sufficient to initiate a sleep state (i.e., $\exists n \in \mathbb{N} \mid \max\{\gamma^{(next)} - t, \pi_{j,S_e^u}(t)\} > bet_n$). In all other cases, the core stays in idle mode.

Ready jobs at any time t are scheduled in a GEDF manner on the active and idle cores by favouring cores without slack. If there is no idle core without usable execution slack then a job is allocated to an idle core with usable execution slack. A job $j_{i,k}$ executing on a core with usable execution slack (i.e., $\pi_{m,S_e^u}(t) > 0$) does not decrease its budget $a_{i,k}$.

6.3 Proof of Correctness

In order to proof the correctness of Algorithm 11, the correctness of its elements — i) usable execution slack consumption mechanism, ii) usable idle slack consumption mechanism, iii) budget handing in the presence of usable execution slack, and iv) usable execution slack donation process — should be proven. Recall that for a task-set τ and a homogeneous platform π, GEDF is *"sustainable"* with respect to the execution requirements of the tasks [HL94]. That is, if the instance of execution τ^{worst} of τ, in which all the jobs execute for their WCETs, is schedulable, then any other instance of execution of τ, in which the jobs execute for less than their WCET, is also schedulable.

Lemma 40. *Let a task-set τ be scheduled by using* GEDF *on a homogeneous platform π. Let $\pi_m \in \pi$ be a core with a usable execution slack $\pi_{m,S_e^u}(t)$ at time t. Let us assume that this slack was caused by the completion of job $j_{i,k}$. Then, π_m can sleep from t to $t + \pi_{m,S_e^u}(t)$ without jeopardising the correctness of the schedule.*

Proof. Let $\mathscr{S}^{ref}$ be the schedule of τ that would have been produced if job $j_{i,k}$ would have executed for its WCET and $\mathscr{S}$ be the actual schedule of τ. By definition of the usable execution slack (see Section 6.1.3), $j_{i,k}$ cannot be pre-empted before $t + \pi_{m,S_e^u}(t)$ in $\mathscr{S}^{\mathrm{ref}}$. Hence, no other

job, but $j_{i,k}$ can execute on π_m within $[t, t+\pi_{m,S_e^u}(t))$ (see Figure 6.5 for an example). Since $j_{i,k}$ has already completed its execution at time t in $\mathscr{S}$ and GEDF is sustainable with respect to the execution requirements, then core π_m can either be kept idle or it can enter a sleep state during interval $[t, t+\pi_{m,S_e^u}(t))$ without modifying the scheduling decisions on any of the other cores and in the future in comparison to $\mathscr{S}^{\text{ref}}$ (see Figure 6.6). □

Lemma 41. *Core π_m with a usable idle slack $S_i^u(t,\pi_m)$ at time t can sleep from time t to $t+S_i^u(t,\pi_m)$ without jeopardising the correctness of the schedule.*

Proof. By definition of the usable idle slack (see Section 6.1.4), no job will ever be executed on π_m before time instant $t+S_i^u(t,\pi_m)$. Hence, π_m can sleep within $[t, t+S_i^u(t,\pi_m))$ without modifying the schedule. □

Lemma 42. *Let π_m be a core with a usable execution slack $\pi_{m,S_e^u}(t)$ at time t. Let us assume that this execution slack was caused by the completion of job $j_{i,k}$. Let $j_{p,q} \neq j_{i,k}$ be any other active job and let us assume that there is no core without execution slack that can execute $j_{p,q}$. Then, core π_m can execute $j_{p,q}$ in interval $[t, t+\pi_{m,S_e^u}(t))$ without reducing its execution budget $a_{p,q}$ and without jeopardising the correctness of the schedule.*

Proof. Let us assume τ^{worst} is schedulable on π by using GEDF. Let $\mathscr{S}^{\text{ref}}$ be the schedule produced by τ^{worst} and $\mathscr{S}$ be the actual schedule of τ. By definition of the usable execution slack (see Section 6.1.3), job $j_{i,k}$ cannot be pre-empted before $t+\pi_{m,S_e^u}(t)$. Moreover, because at any time $t_p \in [t, t+\pi_{m,S_e^u}(t))$, job $j_{p,q}$ cannot be executed by any other core with no execution slack, but on π_m in $\mathscr{S}$, then $j_{p,q}$ cannot be scheduled at time t_p in $\mathscr{S}^{\text{ref}}$ (see Figure 6.1). Therefore, by not reducing the remaining execution budget $a_{p,q}$ of $j_{p,q}$ when it executes on π_m in $[t, t+\pi_{m,S_e^u}(t))$, its remaining execution budget in $\mathscr{S}$ remains identical to the one in $\mathscr{S}^{\text{ref}}$. However, the actual remaining execution time of $j_{p,q}$ diminishes in $\mathscr{S}$ in comparison to $\mathscr{S}^{\text{ref}}$ (it executes in the actual schedule but not in $\mathscr{S}^{\text{ref}}$). By the sustainability property of GEDF, this does not impact the correctness of the schedule. □

Lemma 43. *Let π_m be a core with a usable execution slack $\pi_{m,S_e^u}(t)$ at time t, and π_s a core in a sleep state which is not transitioning from sleep to an active state. If the usable execution slack $\pi_{m,S_e^u}(t)$ of π_m is reduced by $\text{Don}(\pi_m,\pi_s)$, then the sleep state length of π_s can be extended by $\text{Don}(\pi_m,\pi_s)$ without jeopardising the correctness of the schedule.*

Proof. Lemma 40 proves that core π_m can go into a sleep state from time instant t to time instant $t+\pi_{m,S_e^u}(t)$ without jeopardising the correctness of the schedule. Furthermore, Equation 6.3 shows that π_s will wake up at time $(t+\pi_{m,S_e^u}(t)-\text{Don}(\pi_m,\pi_s))$ (see Figure 6.6). Therefore, in the time interval $\left[t+\pi_{m,S_e^u}(t)-\text{Don}(\pi_m,\pi_s),\ t+\pi_{m,S_e^u}(t)\right)$, among cores π_m and π_s, one would be active whilst the other would be sleeping. As π is homogeneous (i.e., all cores are identical and interchangeable), and because GEDF does not differentiate between cores but rather only schedules the highest priority jobs on the available cores, it does not matter which of π_m or π_s is sleeping in $\left[t+\pi_{m,S_e^u}(t)-\text{Don}(\pi_m,\pi_s), t+\pi_{m,S_e^u}(t)\right)$. Consequently, the sleep state of π_s can be prolonged

until $t + \pi_{m,S_e^u}(t)$ if π_m wakes up at or before $(t + \pi_{m,S_e^u}(t) - \text{Don}(\pi_m, \pi_s))$ (see Figure 6.7). The lemma follows. □

Theorem 44. *Algorithm 11 does not impact the correctness of the schedule produced by* GEDF.

Proof. This theorem is a direct consequence of Lemma 40 to Lemma 43. □

6.4 Evaluation of Global Power Management Algorithm

In this section, initially the experimental setup is presented that is used to evaluate the performance of the proposed algorithm. Afterwards, the proposed algorithm is compared against GEDF and an "over optimal algorithm", which assumes no transition overhead for deepest sleep state and schedules tasks with GEDF.

6.4.1 Experimental Setup

To demonstrate the effectiveness of the proposed approach, the SPARTS simulator has been extended to support global schedulers. The proposed global power management algorithm along with GEDF are implemented for the experiments. SPARTS is used with the parameters given in Table 6.1. The underline values are the default parameters assumed if not specified in the description of the individual experiment. An implicit deadline task model is assumed throughout the evaluation section, i.e., $D_i = T_i$. The utilisation factor ζ is a helper variable used to vary the overall system utilisation. The utilisation factor ζ is varied from 0.1 to 0.8 with an increment of 0.05. The overall system utilisation U is computed by multiplying the utilisation factor with the number of cores in the platform, i.e., $U = M \times \zeta$.

The vast variety of symmetric multicore platforms are available in the market with diverse hardware characteristics. A multicore platform is modelled such that it is composed of Freescale PowerQUICC III integrated Communications Processors *MPC*8536 [Sem]. The specifications of this processor are given in Table 4.2. The number of cores of *MPC*8536 on symmetric multicore platform is varied from 2 to 8 as given in Table 6.1.

Parameters	Values
Task-set sizes $\|\tau\|$	$\{15, \underline{20}, 30, 50\}$
Share of RT/BE tasks ξ	$\{\langle 50\%, 50\% \rangle\}$
Inter-arrival time T_i for RT tasks	$[30ms, 50ms]$
Inter-arrival time T_i for BE tasks	$[50ms, 200ms]$
Sporadic delay limit $\Gamma \in$	$\{0, \underline{0.10}, 0.20, 0.30, 0.40\}$
BCET limit C^b	$\{0.25, \underline{0.50}, 0.75, 1\}$
Utilisation factor ζ	$\langle 0.1 : 0.05 : 0.8 \rangle$
Number of cores M	$\langle 2, \underline{4}, 8 \rangle$

Table 6.1: Overview of simulator parameters used to evaluate global power management algorithm

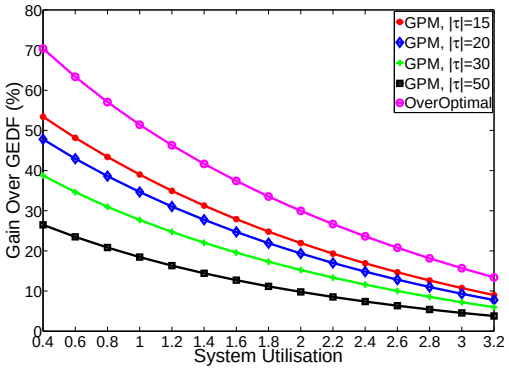


Figure 6.8: Variation in $|\tau|$

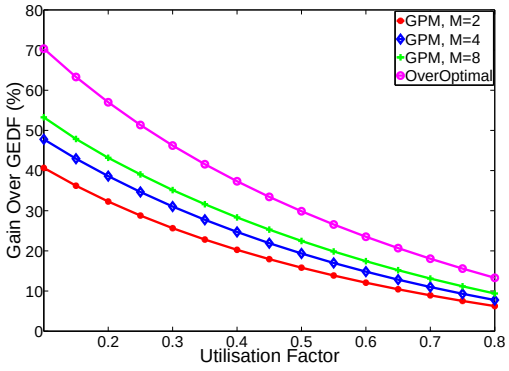


Figure 6.9: Variation in number of cores

6.4.2 Simulation Results of the GPM Algorithm

It is assumed that the processor transition into an idle state (typical mode) when the ready queue is empty in GEDF. GEDF is used as a baseline in the presented results. Also, a conservative lower-bound on energy consumption is determined using an over-optimal algorithm. To derive this bound, the tasks are scheduled with the GEDF scheduling policy and cores transition into the deepest sleep state instantly (without any transition time and energy overhead) as soon as they become idle. Such an over-optimal algorithm gives us a safe lower-bound which is clearly not reachable in practice. It is represented as OverOptimal in the graphs. All the task-sets not schedulable with the GEDF scheduling algorithm are discarded from the results. The results presented in this section show the gain of total energy consumption over the GEDF algorithm. The total energy consumption is the summation of active, idle and sleep state energy consumption.

Figure 6.8 demonstrates the effect of variation in the number of tasks on GPM assuming four cores in the platform. The gain of GPM over GEDF decreases with an increase in task-set size. The increase in task-set size increases the energy consumption of GPM due to the following reason. The usable execution slack is bounded by the next higher priority task arrival that can pre-empt it. Similarly, the idle slack is bounded by the next task arrival in the system. Therefore, the chance of getting pre-empted by higher priority jobs or next task arrival in the system increases with the size of the task-set. This explains the reason for an increase in energy consumption with different task-set sizes but an identical utilisation. The performance of GPM also degrades gradually, with an increase in overall system utilisation. At high utilisation, GPM cannot find efficient sleep states to reduce the energy consumption. Moreover, the slack in the schedule also decreases with an increase in the system utilisation. Hence, its energy consumption increases at high utilisations. In the best-case, the gain over GEDF varies between 3.78% and 53.4% in this experiment. The gain of OverOptimal against GEDF at different utilisation stays same for different task-set sizes as the size of idle interval remains same. Therefore, only a single line is presented in the graph. The gain of OverOptimal bound also decreases with an increase in the system utilisation due to the same aforementioned reason of decrease in the available slack. In the best case, the difference of gain between OverOptimal and GPM ranges between 4.4% to 16.96%.

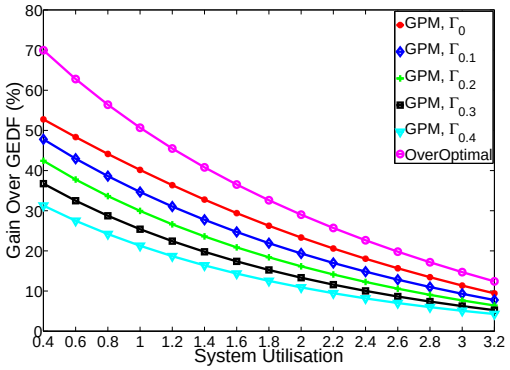


Figure 6.10: Variation in Γ

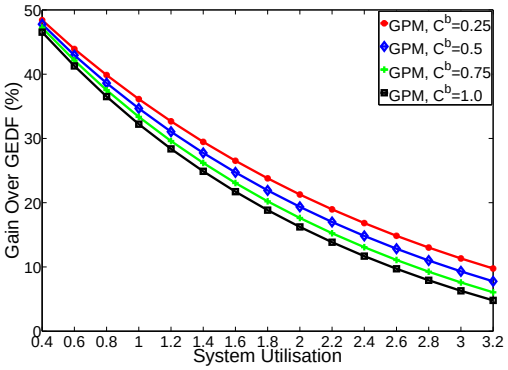


Figure 6.11: Variation in C^b (GPM)

The number of cores on the given multicore platform also plays an important role in the overall energy consumption of the platform. Their effect is presented in Figure 6.9. Please note that the multicore platforms compared in Figure 6.9 are virtually equivalent. As the number of core increases, the utilisation factor defined as $\frac{U}{M}$ is kept constant, thereby allowing a fair comparison. The performance of GPM increases with the number of cores. The multicore platform with large number of cores provides more room for energy minimisation and slack donation. Moreover, the chances of tasks getting pre-empted also decreases with an increase in the number of cores with same task-set sizes. Hence, GPM works well with an increase in the number of cores, which demonstrates its scalability feature. Similar to the task-set size variation, the number of cores also does not affect the gain of OverOptimal against GEDF at different utilisation and hence, only a single line is shown for OverOptimal. In the best case, the difference of gain between OverOptimal and GPM is bounded by a limit of $[3.9\%, 17.1\%]$.

The effect of sporadic slack on the given algorithm is shown in Figure 6.10. The value of the sporadic slack limit is varied from 0% to 40%. GPM cannot exploit the sporadic slack in the schedule. The energy saving shown in this graphs comes from the idle intervals generated through execution and static slack in the schedule. Both, GEDF and GPM stay in idle state for the sporadic slack in the system. Hence, the increase in sporadic slack decreases the relative gain of GPM when compared to the GEDF algorithm. As the idle interval increases in the schedule OverOptimal bound varies for different values of sporadic slack. It is evident that the gain of OverOptimal at 40% sporadic slack is maximum. However, the difference between the gains of OverOptimal against GEDF at different utilisation for different size of sporadic slack stays within an range of 1.4% to 3.4% approximately. The sake of clarity, the curve corresponding to OverOptimal is only plotted for a sporadic slack size of 40%. In this experiment, the gain of GPM reaches upto 52.75% for the fully periodic task-set. In the best case, OverOptimal and GPM has a difference that starts from 17.21% at low utilisation to 2.98% at high utilisation.

The best-case execution time limit C^b allows to control the amount of execution slack in the system. A low value means high execution slack. The gain of GPM over GEDF on different values of execution slack is illustrated in Figure 6.11. The increase of execution slack obviously decreases the energy as it helps to prolong the sleep intervals. It can be seen that the execution

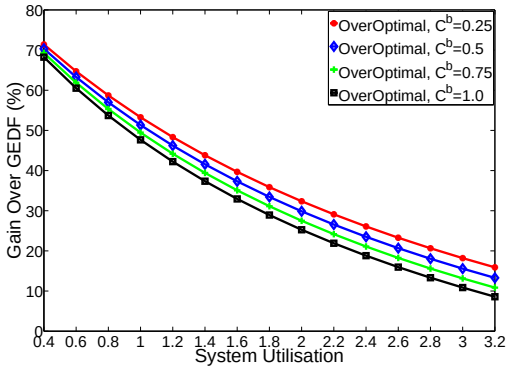


Figure 6.12: Variation in C^b (OverOptimal)

slack has low impact at low utilisations (all curves are close from each other). Indeed, the static slack plays a more dominating factor at low utilisations when compared to execution slack. On high utilisations, the effect of execution slack is evident (the curves move away from each other) and GPM gain up to $\approx 5\%$ of energy efficiency with an average increase of 75% in execution slack. In best case the gain reaches upto to approximately 48.4% and drops to 4.8% in the worst-case.

The gain of OverOptimal is also plotted for different values of execution slack as presented Figure 6.12. It is evident that the gain decrease with the decrease in the execution slack. Similar to GPM the gain difference between the different curves is small at low utilisations but increases at high utilisation. This occurs as the execution slack is minute compared to the static slack at low utilisations. The execution slack varies the gain in OverOptimal within a range of $[3.15\%, 7.32\%]$. In the best case, the maximum difference between GPM and OverOptimal for an execution slack of size 25% is approximately 22.98%, and the minimum difference is 6.13%. Hence, the gain of the proposed algorithm is not far from the OverOptimal curve.

Chapter 7

Partitioned Multicore Power Management

To efficiently deploy complex algorithms an increasing number of heterogeneous platforms have emerged. A heterogeneous multicore platform is composed of multiple heterogeneous cores or core types geared to perform specific tasks well and cheap. Although heterogeneous multicore platforms look an attractive choice for modern RT embedded systems, the integration of such platform in the RT domain is far from trivial. The complexity of the design leads to a pessimistic analysis to provide the timing guarantees and this problem is acerbated if unrelated heterogeneous multicore platforms are considered. The major issue in unrelated heterogeneous multicore platform comes from the fact that there is no relation between given task-set and the available unrelated cores. Extracting the optimum performance out of such a platform is an open problem. Apart from the aforementioned inherent challenges, heterogeneous multicores are emerging in RT community and researchers in this domain are working on different orthogonal issues. Partitioned scheduling algorithms are commonly used on heterogeneous multicore platforms. One of the issues is to efficiently map the given task-set such that it minimises the energy consumption of the platform. The task-to-core mapping in a heterogeneous multicore platform is a NP-hard problem as it appears to be special case of bin-packing.

Similar to other multicore platforms, energy efficiency is equally important in heterogeneous multicore platforms. Even when the application is technically feasible upon the targeted platform in the sense that the platform can provide a sufficient computing capacity for the execution of the application, it has become unreasonable to expect to implement such a system without addressing the issue of minimising its energy consumption. To this end, chip manufacturers are putting considerable efforts that aligns neatly with the desired wish-list of most embedded systems. The energy consumption of a heterogeneous platform can be reduced in two different ways. Firstly, a task can be allocated to a core where it consumes minimum dynamic power. Secondly, tasks on a core can be grouped such that scheduler allows efficient sleep states in the online phase which in turn reduces the static power dissipation. However, a global minima can only be achieved through a hybrid approach that considers both factors altogether.

The inherent complexity of the problem forced us to divide the algorithm of task allocation into two phases. In the first phase of allocation, dynamic power dissipation of the given platform is minimised. The second phase of allocation trades-off the increase in dynamic power dissipation with reduced leakage-power dissipation of the platform. Traditional task assignment algorithms aim to reduce the active power dissipation of the system by assigning the tasks to the core where it consumes the least dynamic power dissipation, while ignoring the effect of such allocation on the static power dissipation. The management of the static power dissipation of the processor is an orthogonal issue as it depends on the properties of the tasks such as their respective minimum inter-arrival times and worst-case execution times. For instance, assume the task assignment that generates a large fraction over time in combination with a short period task. The core may not be able to exploit such idle intervals in the schedule through deeper sleep states due to a combination of the larger transition overhead of those and a short period task. The proposed two phase strategy is novel in a sense it considers both dynamic and static power dissipation while considering a generic power model. Initially, the allocation algorithms are proposed for a heterogeneous multicore platform without DVFS capability, and later on this assumption is relaxed to a general platform with DVFS and sleep state capabilities exploited altogether to minimise energy consumption.

7.1 Extensions in the System Model

The system model (that includes hardware platforms, task model and power model) used in this work is explained as follows.

7.1.1 Hardware Platform

This chapter considers a partitioned unrelated heterogeneous multicore platform $\pi \stackrel{\text{def}}{=} \{\pi^1, \pi^2, \ldots, \pi^M\}$ composed of M different types of processors/cores. Each core type $\pi^m = \{P_A^m, P_I^m, \vec{\S}^m, \vec{f}^m\}$ is characterised by unique power dissipation and execution capabilities, and consists of an active state, an idle state, a number of frequency set points and a set of sleep states. These parameters and the specification of a processor type π^m are defined in Section 3.1.3. A single processing unit of each processor type π^m (with $m = 1, 2, \ldots, M$) is assumed in this work for the separation of concerns and ease of notation. However, it is not limited to this restriction. The total number of processors in the platform will be equal to M. A core type π^m running at a frequency f_v^m is represented as $\pi^{m,v}$. The terms core, processor or processor type are used interchangeably throughout this chapter.

7.1.2 Task Model

A task-set τ is composed of ℓ independent tasks. The parameters of the individual task defined in Section 3.1.1 are modified as follows. Each task is characterised by a quadruple $\tau_i \stackrel{\text{def}}{=} \langle \vec{\vec{C}}_i, D_i, T_i, \vec{\vec{E}}_i \rangle$, where $\vec{\vec{C}}_i \stackrel{\text{def}}{=} (\vec{C}_i^1, \vec{C}_i^2, \ldots, \vec{C}_i^M)$ is the vector of execution profiles of τ_i on the core types and $\vec{\vec{E}}_i \stackrel{\text{def}}{=}$

$(\bar{E}_i^1, \bar{E}_i^2, \ldots, \bar{E}_i^M)$ is the vector of energy profiles of τ_i associated to $\vec{C}_i$. For the core type π^m, the execution profile $\vec{C}_i^m \stackrel{\text{def}}{=} {}^t(C_{i,1}^m, C_{i,2}^m, \ldots, C_{i,V^m}^m)$ is the vector of worst-case execution times of τ_i where $C_{i,v}^m$ (with $v = 1, 2, \ldots, V^m$) is the worst-case execution time of τ_i at frequency f_v^m. Similarly, the energy profile $\bar{E}_i^m \stackrel{\text{def}}{=} {}^t(\bar{E}_{i,1}^m, \bar{E}_{i,2}^m, \ldots, \bar{E}_{i,V^m}^m)$ is the vector of average energy consumptions of τ_i such that $\bar{E}_{i,v}^m$ (with $v = 1, 2, \ldots, V^m$) is the average energy consumption of τ_i at frequency f_v^m. For the sake of brevity, every task τ_i (with $i = 1, 2, \ldots, \ell$) is assumed to have an implicit deadline, i.e., $D_i = T_i$. A subset of a task-set τ allocated to a core π^m is denoted as τ^m. The individual utilisation of task τ_i on π^m at frequency f_v^m is defined as $U_{i,v}^m \stackrel{\text{def}}{=} \frac{C_{i,v}^m}{T_i}$. The utilisation of a core type $\pi^{m,v}$ running at frequency f_v^m is denoted as $U^{m,v}$. It is the summation of individual utilisation of the tasks allocated to it, i.e., $U^{m,v} \stackrel{\text{def}}{=} \sum_{\forall \tau_i \in \tau^m} \frac{U_{i,v}^m}{T_i}$. Jobs of the same task are allowed to vary their execution between τ_i's BCET and WCET. The ERTH algorithm discussed in Section 4.2.1 is used on each processor to manage the energy consumption.

7.1.3 Power Model

The power model used in state-of-the-art algorithms assumes the energy consumption of an application on a processor is only a function of its execution time. However, in reality, the energy consumption on a certain processor type depends also on the set of instructions it has to execute to perform the desired functionality. Different instructions use different parts of CPU, and hence may result in a different energy consumption. Therefore, two application with identical execution time may consume different energy depending on the characteristics of the instructions used, and the number of cache misses involved. Secondly, the static power dissipation of the platform cannot be regarded as a constant factor. If the energy saving mechanism is based on sleep states then the static power dissipation depends on the energy characteristics of the used sleep states. This wok employs a more refined power model where energy consumption of a platform is not constant per unit time, rather depends on the behaviour of the application, the sleep-states characteristics of the processor and the use of sleep states by the scheduling algorithm.

The average energy consumption of tasks on different platforms for different frequency set-points can be determined offline using any existing approach (e.g., an energy measurement technique based on performance monitoring counters [SLSPH09]). A naïve power model can also be considered by either assuming that the energy consumption of the core is constant per time unit in the active mode or by setting the worst-case energy consumption as an optimisation target. The preference of a task to a processor type is determined with respect to its active energy consumption.

Definition 45 (Favourite Core)**.** *The favourite core for a task is the one where its active energy consumption is minimal.*

Definition 46 (Least Preferred Core)**.** *The least preferred core of a task is the one where its active energy consumption is maximal.*

7.2 Allocation Heuristics (Non-DVFS)

In order to tackle active and static power dissipation, a two phase algorithm is proposed to perform the task assignment for the given M-type heterogeneous platform. The first phase of the algorithm optimises the assignment such that it reduces the active energy consumption of the platform. The second phase trades tasks active energy consumption to enhance the ability of the processors to use more efficient sleep states in order to reduce static power dissipation of the system. Initially, the system model is relaxed and a platform without DVFS capability is assumed, i.e., each core type is assumed to have a single active power state (no DVFS), idle state and set of different sleep states. Later in the discussion, the system model is extended to the more general platforms with DVFS capabilities. For ease of notation in the system model discussed above, the indexes corresponding to the frequency are dropped in this section. For example, an individual utilisation is denoted as U_i^m and the same holds for other symbols. Considering the non-DVFS platform, the problem discussed in this section considers a M-type Heterogeneous platform with several sleep states per core assuming their energy/time overhead in a setting of partitioned scheduling and map a given task-set onto this platform such that the overall energy consumption (active + sleep) of the system is minimised.

7.2.1 First Phase of Allocation

The two different assignment algorithms are proposed to reduce the dynamic power dissipation of the platform.

7.2.1.1 Least Loss Energy Density Algorithm (LLED)

The least lost energy density algorithm (LLED) attempts to allocate tasks to their favourite core to optimise the individual task's energy consumption. However, not all tasks may be allocated to their respective favourite core type due to the limited capacity on each core. In such scenario, where more than one tasks are competing for their favourite core type, the tasks among each other on the same core type should be ranked. To derive such ranking, a metric called energy density of a task is defined as follows.

Definition 47 (Energy Density)**.** *The average energy consumption of a task τ_i per unit time on a core type π^m is called the energy density of a task ED_i^m, as shown in Equation 7.1.*

$$ED_i^m \stackrel{def}{=} \frac{\bar{E}_i^m}{T_i} \tag{7.1}$$

As there is only single active state, therefore, only the energy consumption at the maximum frequency is considered in the definition of energy density. The energy density value of a task does not provide any global perspective on how the power dissipation of the system changes when a certain task is not allocated to its preferred core type. The global perspective can be achieved through another metric termed as density difference (*DD*).

Definition 48 (Density Difference)**.** *The density difference of a task τ_i on a core π^m is the quantity of extra energy consumption per unit time that the system will consume when compare to its energy consumption per unit time on π^m, if the task τ_i is allocated to the next higher energy consuming core instead of current preferred core π^m. Mathematically, it is defined in Equation 7.2*

$$DD_i^m \stackrel{def}{=} min\{ED_i^k : k \neq m \wedge ED_i^k \geq ED_i^m\} - ED_i^m \tag{7.2}$$

The density difference is determined by subtracting the energy density of the task on the current core type from the next higher energy density value of the same task on another core. To get the ranking of the tasks on the given core, all tasks are sorted on this core in descending order with respect to their density difference values. The tasks from the top of the list i.e., tasks with higher density difference values are allocated first. The intuition behind such a mechanism is to reduce the losses by allocating the tasks first which suffer a larger energy penalty when moved to another core. The process can be started from any core type. A task allocated to a core is not considered for an allocation on any other core where it consumes more energy when compared to its currently allocated core. The same procedure is repeated for all cores. In the worst-case, the process iterates over each core ℓ times.

The pseudo-code of least loss energy density algorithm (LLED) is given in Algorithm 12. Initially, the energy density ED_i^m of every task is computed on all core types (line 2). Using energy density values, the density difference DD values of all tasks are estimated on each core and stored in a matrix called MT (line 3-6, 10). Note that MT_w^q value in a matrix MT corresponds to the density difference DD_w^q of τ_w on a core type π^q. To obtain the density difference DD_w^y of the task τ_w on its least preferred core type π^y ($\pi^y : ED_w^y = \max_{x=1,\cdots,M} ED_w^x$), its energy density ED_w^y on the least preferred core type π^y is subtracted from 0 (line 8) to obtain a negative value, i.e., $0 - ED_w^y$. Afterwards, the LLED algorithm iterates through the processors in any order (for example, the processors indices can be used to order them). Starting from the first core type π^q, all tasks on π^q have their entries in MT^q sorted in descending order with respect to their MT_w^q (i.e., DD) values. The proposed algorithm (LLED) iterates by picking a task from the top of the sorted list and attempts to allocate it to π^q. For instance, τ_x is the current task on top of the sorted list with respect to density difference values on core π^q. The algorithm attempts to allocate τ_x to π^q. If π^q can accommodate τ_x (line 17-18), it does not consider τ_x on other cores for which this inequality $\bar{E}_x^m \geq \bar{E}_x^q$ holds and removes its entries of DD values in MT matrix (line 19). In other words, τ_x is not considered for allocation on other core types where it consumes more or equal energy compared to this core type π^q. If the task τ_x was previously allocated to these higher energy consuming core types, it is deallocated on such cores (line 20). Once the allocation for τ_x is completed on π^q, LLED attempts to allocate the next task in the sorted list. If any of the task in the order cannot be allocated to π^q, the algorithm moves to the next core type instead of checking the next tasks in the order. This action is performed to avoid allocation of any unfavourable task to the current core type, which may have a chance of allocation in the next iteration. The same procedure is repeated for the next core type and so on. On completion of the first iteration, the

Algorithm 12 First Phase: Least Loss Energy Density (LLED)

1: $U^m = 0$ for each core π^m
2: Compute ED_i^m for each τ_i on each core
3: **for** $q = 1$ to M **do** [/*For all processor types*/]
4: **for** $w = 1$ to ℓ **do** [/*For all tasks*/]
5: **if** $ED_w^q \neq \max_{x=1,\ldots,M} ED_w^x$ **then**
6: $ED_w^r = \min_{x=\{1,\ldots,M\}\setminus q \;\&\&\; ED_w^x \geq ED_w^q} ED_w^x$
7: **else**
8: $ED_w^r = 0$
9: **end if**
10: $MT_w^q = ED_w^r - ED_w^q$
11: **end for**
12: **end for**
13: **for all** Tasks ℓ **do**
14: **for** $q = 1$ to M **do** [/*For all processors types*/]
15: Sort all tasks having entry in MT^q, with respect to MT_w^q values in descending order
16: **for all** $\tau_w \in \tau$ on Core Type π^q in Descending Order of MT_w^q Values **do**
17: **if** $U^q + U_w^q \leq 1$ **then**
18: Assign τ_w to π^q
19: $\forall x \in [1,\ldots,M]\setminus q$, Remove $MT_w^x \; iff(\bar{E}_w^x \geq \bar{E}_w^q)$
20: $\forall x \in [1,\ldots,M]\setminus q,\; U^x -= U_w^x \; iff(\bar{E}_w^x \geq \bar{E}_w^q \;\&\&\; \tau_w$ is assigned)
21: **else**
22: Break
23: **end if**
24: **end for**
25: **end for**
26: **end for**

algorithm starts again from the first processor type. These iterations are repeated until all the tasks are allocated to exactly one core type. In the worst-case, the LLED algorithm has to check each task in each core type for ℓ times. Lines $13-26$ in Algorithm 12 corresponds to these steps. Therefore, complexity of the LLED algorithm is $O(\ell^2 \times M)$. The working of the LLED algorithm is demonstrated with the help of an example given below.

Example 8. *Consider a set of* 4 *tasks and* 3 *core types. The tasks specifications are given in Figure 7.1(a). Entries under each core type specifies* $C_i^m, \bar{E}_i^m, ED_i^m$ *for* τ_i*. The density difference DD values are computed for all tasks and presented in Figure 7.1(b). As an example, the DD value of* τ_1 *in* π^1 *is computed by an expression* $ED_1^2 - ED_1^1$*.* LLED *can start from the first core type* π^1 *and sorts the tasks in descending order of DD values as presented in the first column of Figure 7.1(c).* τ_4 *can be allocated to* π^1*, therefore, its entry that consumes more energy compared to this core type is deleted in* π^3 *type.* τ_2 *cannot be allocated, therefore* LLED *moves to* π^2 *and sorts the task-set according. In core type* π^2*,* τ_1 *and* τ_4 *can be allocated.* τ_1*'s entry in* π^3 *and* τ_4*'s entries on* $\pi^1 \& \pi^3$ *is deleted due to higher energy consumption. Similarly, after appropriate sorting of tasks with respect to their DD values on* π^3*,* τ_2 *and* τ_3 *can be allocated to* π^3*. Therefore,*

	π^1	π^2	π^3	T_i
τ_1	4.5/16.5/1.65	3/17.2/1.72	7/52.5/5.25	10
τ_2	8/37.65/2.51	10/65.1/4.34	8/57/3.80	15
τ_3	18/84/2.80	12/78.9/2.63	10/75.9/2.53	30
τ_4	60/259.2/2.16	35/210/1.75	80/649.2/5.41	120

(a) $C_i^m/\bar{E}_i^m/ED_i^m$ values

	π^1	π^2	π^3
τ_1	0.07	3.53	−5.25
τ_2	1.29	−4.34	0.54
τ_3	−2.8	0.17	0.10
τ_4	3.25	0.41	−5.41

(b) Density difference (DD) in MT

π^1	π^2	π^3
~~τ_4~~	τ_1	τ_2
τ_2	τ_4	τ_3
τ_1	~~τ_3~~	~~τ_1~~
~~τ_3~~	~~τ_2~~	~~τ_4~~

(c) 1^{st} iteration

π^1	π^2	π^3
~~τ_4~~	~~τ_1~~	~~τ_2~~
τ_2	τ_4	τ_3
τ_1	~~τ_3~~	~~τ_1~~
~~τ_3~~	~~τ_2~~	~~τ_4~~

(d) 2^{nd} iteration

Figure 7.1: First phase mapping of least loss energy density algorithm

τ_2's entry in π^2 and τ_3's entry in π^2, π^1 are deleted. This completes first iteration and status of the tasks after first iteration are shown in Figure 7.1(c). Similarly, LLED *performs the second iteration. On π^1, the τ_4's entry is deleted, so it is not considered for allocation and the algorithm attempts to allocated next task in the order (τ_2). The rest of the process is similar to the first iteration. The end result of 2^{nd} iteration is shown in Figure 7.1(d).* LLED *does not need any further iterations as all the tasks are assigned. The worst-case number of iterations is equal to the task-set size.*

Algorithm 13 Alternative First Phase: Maximum Minimum (MM)

1: $U^m = 0$ for each core π^m
2: Compute ED_i^m for each τ_i on each core
3: $\forall \tau_i$: Find $ED_i^{max} = \max_{x=1,\ldots,M} ED_w^x$
4: $\forall \tau_i$: Find $ED_i^{min} = \min_{x=1,\ldots,M} ED_w^x$
5: Sort task-set with respect to $\{ED_i^{max} - ED_i^{min}\}$ in descending order
6: **for all** Tasks $i = 1$ to ℓ **do**
7: Sort cores with respect to the energy consumption of τ_i in ascending order
8: **for all** Processors $j = 1$ to M **do**
9: **if** $U^j + U_i^j \leq 1$ **then**
10: Assign τ_i to π^j
11: $U^j += U_i^j$
12: Break
13: **end if**
14: **end for**
15: **end for**

7.2.1.2 MaxMin Algorithm (MM)

Another simple heuristic MaxMin labelled as MM can be used to assign tasks in M-type heterogeneous platform to reduce the active power dissipation. The pseudo-code of the MM algorithm is given in Algorithm 13. Assume, ED_i^{min} is the energy density of task τ_i on its most favourite core type, while ED_i^{max} corresponds to its energy density on the least preferred core type. This heuristic for each task computes the difference of ED_i^{max} and ED_i^{min}, i.e., $ED_i^{max} - ED_i^{min}$. All tasks are globally sorted in descending order with respect to this difference (line 5). The MM algorithm picks a task from the top of the list and assigns to its favourite core type. If the favourite core cannot accommodate this task, an allocation attempt is made for next core type in its ascending order of energy consumption (line 8-14). If the task is assigned to a core type, the utilisation of the corresponding core type is incremented accordingly. The MM algorithm has a low complexity of $O(\ell \times M)$. The example given below assigns the the task-set given in Figure 7.1(a).

τ_i	ED_i^{max}	ED_i^{min}	$ED_i^{max} - ED_i^{min}$	Order	Allocated Core
τ_1	5.25	1.65	3.6	2	π^1
τ_2	4.34	2.51	1.83	3	π^1
τ_3	2.80	2.53	0.27	4	π^3
τ_4	5.41	1.75	3.66	1	π^2

Table 7.1: Tasks allocation through the MM algorithm

Example 9. *The* MM *algorithm determines* ED_i^{min}, ED_i^{max} *and* $ED_1^{max} - ED_1^{min}$ *for each task given in Figure 7.1. The computed values are presented in Table 7.1. The tasks are sorted with respect to the descending order of* $ED_i^{max} - ED_i^{min}$. *The sorted order in our example is* $\tau_4 > \tau_1 > \tau_2 > \tau_3$. *Afterwards, the tasks are assigned in this sorted order to their favourite core. The* MM *algorithm firstly allocates* τ_4 *to* π^2, τ_1 *and* τ_2 *to* π^1 *and finally assigns* τ_3 *to* π^3. *In the given example, the algorithm managed to allocate tasks to their most favourite cores.*

7.2.2 Second Phase of Optimisation

While, the first phase of allocations is derived with an objective to optimise an individual task's active energy consumption, it does not consider its effect on the mechanism to reduce the static power dissipation. For instance, a core may have less active energy consumption but a small group of tasks allocated to it may prevent it from using a deeper and more efficient sleep state in the idle intervals of the schedule to reduce the static power dissipation of the system. In this second phase of optimisation, our proposed algorithm analyses the properties of the allocated tasks to a core in this broader context and considers its effect on the core's ability to use more efficient sleep states by trading off higher active energy consumption of a task for energy savings in sleep states.

As mentioned previously, ERTH is used on individual cores as a power saving algorithm. In the context of multicore platform, the static sleep interval of the processor type π^m is denoted as χ_{min}^m. It can be determined using DBF assuming a synchronous release of all tasks allocated to

Algorithm 14 Second Phase of Task Mapping (SP)

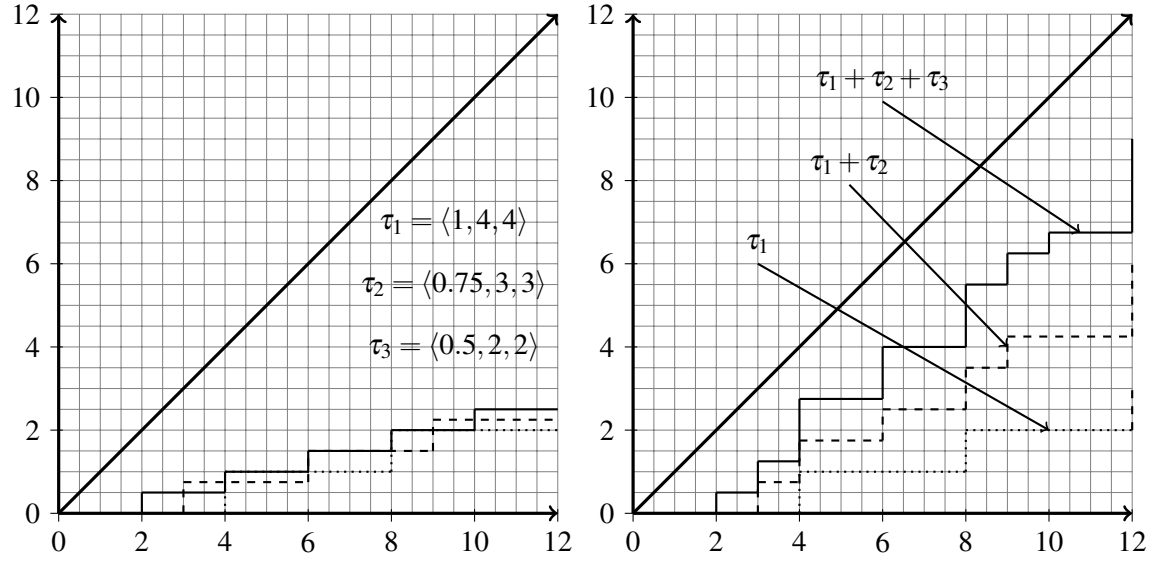
```
1: repeat
2:     Previous Assignment = Current Assignment
3:     Energy Old = Energy New
4:     Sort tasks in descending order on each core (allocated in the first phase) w.r.t T_i − C_i^m
5:     On each core π^m compute χ_i^m for each task allocated to it
6:     Group tasks on each core by achievable sleep states
7:     Order core by gains when removing group
8:     Feasible = TRUE
9:     for all Processor Types M do
10:        for all Tasks in a Top Group do
11:            Compute the local cost of migration in terms of energy consumption of this task
       for all other cores
12:            Sort other cores by decreasing order of cost
13:            for all Cores Except the Core of the Currently Assigned Task do
14:                if Feasible on Core then
15:                    Assign to a core
16:                    Success = True
17:                    Break
18:                end if
19:            end for
20:            if !Success then
21:                Feasible = FALSE
22:                Break
23:            end if
24:        end for
25:        if Feasible && Energy New < Energy Old then
26:            Break
27:        else
28:            Undo all assignments
29:        end if
30:    end for
31: until Previous Assignment == Current Assignment
```

a core π^m, i.e., $\chi^m_{min} \stackrel{\text{def}}{=} \min_{\forall L \leq L^*} (L - \text{DBF}(L))$. For further details about static sleep interval, please revisit Equation 4.26 and Lemma 28. The ERTH algorithm initiates a sleep transition online when the processor is idle or has sufficient execution slack. The duration of χ^m_{min} defines which sleep state can be used online on core π^m.

As has been discussed, the properties of tasks involved in the computation of χ^m_{min} have a high impact on its value. For example, tasks with shorter difference between their T_i and C^m_i give a small value of χ^m_{min} and restrict usage of those sleep states with $bet^m_n > T_i - C^m_i$. The intuition behind the second phase is to collate tasks on a core with similar properties such that it can use a more efficient sleep state. As a heterogeneous platform is assumed in this work, each core has sleep states with different characteristics. One or more tasks restricting a more efficient sleep state on one core may not effect the sleep state on another core and hence can be considered for

migration. However, the algorithm must ensure that such a migration reduces the overall average energy consumption of a system. The proposed heuristic (Algorithm 14) performs such a trade-off.

A first step necessary for Algorithm 14 is to take as an input the tasks assignment of the first phase and sort tasks on each core with respect to their difference between T_i and C_i^m in descending order (line 4 in Algorithm 14). Consider one of the core type π^m and assume ℓ^m are the number of tasks allocated to it in the first phase (through LLED or MM). The second phase initially computes the maximum time interval of the sleep duration also known as a static sleep interval with just one task picked from the top the sorted list (with respect to $T_i - C_i^m$) of tasks allocated to π^m. This value is denoted as χ_1^m and computed through DBF. As there is just one task, therefore, $\chi_1^m = T_i - C_i^m$. Now the next task is superimposed on the current DBF and new static sleep interval $\chi_2^m = \min_{\forall L \leq L^*} (L - \text{DBF}(L))$ is computed. Similarly, a third task is superimposed and correspondingly χ_3^m is computed. This process is repeated for all sorted tasks allocated to π^m to obtain a set of static sleep interval values called $\chi^m = \{\chi_1^m, \chi_2^m, \chi_3^m, \ldots, \chi_{\ell^m}^m\}$. This step corresponds to the line 5 of Algorithm 14. As the tasks are superimposed in the descending order of $T_i - C_i^m$, therefore, one of the property of χ^m is that $\chi_1^m \geq \chi_2^m \geq \chi_3^m \geq \ldots \geq \chi_{\ell^m}^m$. Moreover, $\chi_{min}^m \stackrel{\text{def}}{=} \chi_{\ell^m}^m$. To illustrate the computation of χ^m set, consider the following example.



(a) Individual Demand (b) Superimposed

Figure 7.2: Demand bound function to demonstrate the computation of static sleep interval set in the second phase of optimisation with tasks $\tau_1 = \langle 1,4,4 \rangle, \tau_2 = \langle 0.75,3,3 \rangle$ and $\tau_3 = \langle 0.5,2,2 \rangle$

Example 10. *Assume, three tasks* $\tau_i = \langle C_i^m, D_i, T_i \rangle \Rightarrow \tau_1 = \langle 1,4,4 \rangle, \tau_2 = \langle 0.75,3,3 \rangle, \tau_3 = \langle 0.5,2,2 \rangle$ *sorted in the descending order of* $T_i - C_i^m$ *and allocated to* π^m. *Individual demands of these tasks are shown in Figure 7.2(a). Firstly,* χ_1^m *with* τ_1 *is computed, i.e.,* 3 *time units. Then* τ_2 *is superimposed on* τ_1 *and* χ_2^m *is computed, which is equal to* 2.25 *time units. Finally,* τ_3 *is superimposed on the demand of* $\tau_1 + \tau_2$ *and* χ_3^m *is estimated to be* 1.25 *time units. These steps are demonstrated in Figure 7.2(b). This example has* $\chi^m = \{3, 2.25, 1.25\}$.

The number of elements in χ^m is equal to ℓ^m (task-set size on core π^m). Each element in χ^m indicates the static sleep interval with the corresponding number of tasks. A core π^m can use this sleep interval to initiate a sleep state, if the tasks used to compute such interval are allocated to it. The most efficient sleep state among the available set of sleep states in π^m is determined for all elements of χ^m using the following expression $\{\forall x \in \chi^m$, find $\S_n^m : \S_n^m$ minimises $Cons(x,n,m)\}$, where $Cons(x,n,m)$ is defined in Equation 7.3. χ^m holds the property that $\chi_1^m \geq \chi_2^m \geq \chi_3^m \geq \ldots \geq \chi_{\ell^m}^m$, therefore, χ_2^m cannot support a better sleep state when compared to χ_1^m and so on.

$$Cons(x,n,m) \stackrel{\text{def}}{=} Es_n^m + (x - 2 \times tr_n^m)P_n^m \tag{7.3}$$

After computing the sleep states for each χ^m element, the tasks that allow the same sleep state are grouped together. A set G_n^m is defined to hold the tasks for a sleep state $\S_n^m$. Starting from χ_1^m, τ_1 is added to a set of its computed sleep state. Similarly, τ_2 is added to the set corresponding to a sleep state determined for χ_2^m and so on. This step is demonstrated with an example.

Example 11. *Suppose, there are five elements in* $\chi^m = \{\chi_1^m, \chi_2^m, \chi_3^m, \chi_4^m, \chi_5^m\}$. *Assume, sleep states corresponding to these* χ^m *elements are determined to be* $\{\S_3^m, \S_3^m, \S_2^m, \S_1^m, \S_1^m\}$ *(shallower the sleep state lower the index). Then the tasks are added to the sets corresponding to the sleep states as follow:* $G_3^m = \{\tau_1, \tau_2\}$, $G_2^m = \{\tau_3\}$ *and* $G_1^m = \{\tau_4, \tau_5\}$. *These sets are referred as groups of tasks corresponding to different sleep states.*

These groups of tasks are ordered from the least efficient to the most efficient sleep state. Thus, if the top most group of tasks is removed, a core can achieve the next better sleep state. This complete process is repeated for all cores and finally there are different groups of tasks on each core corresponding to their different sleep states. This step is given in line 6 of Algorithm 14.

All cores compete to gain the next more efficient sleep state to save energy by relocating to other core the tasks in the top most group that enforces the less efficient sleep state. However, the algorithm first considers the core that results in the most system energy gain. To identify such core, the algorithm removes all the tasks associated to the first group (that cause less efficient sleep state) on each core. Let G_{top}^m correspond to the tasks in the top least efficient sleep state group on π^m. The energy saving by removing such a group from this core is equal to $\Delta\bar{E}^m$ as given in Equation 7.4. Where χ_{old}^m and χ_{new}^m are the static sleep intervals before and after removing G_{top}^m respectively on π^m. After computing χ_{old}^m and χ_{new}^m, their corresponding sleep states are determined. Suppose, $\S_{n1}^m$ and $\S_{n2}^m$ are the sleep states selected for χ_{old}^m and χ_{new}^m respectively. Moreover, $\bar{E}_{old}^m = Cons(\chi_{old}^m, n1, m)$ and $\bar{E}_{new}^m = Cons(\chi_{new}^m, n2, m)$ represent the energy consumption of a single sleep transition with and without G_{top}^m respectively, where $Cons(x,n,m)$ is defined

in Equation 7.3. All Cores are sorted in descending order with respect to $\Delta\bar{E}^m$ (line 7 of Algorithm 14). This determined order is used to attempt a reallocation of the top most group of cores.

$$\Delta\bar{E}^m = \sum_{\forall\tau_i\in\tau^m}\frac{\bar{E}_i^m}{T_i} + \frac{\left(1-\sum_{\forall\tau_i\in\tau^m}U_i^m\right)\bar{E}_{old}^m}{\chi_{old}^m} - \left(\sum_{\forall\tau_i\in\tau^m\setminus G_{top}^m}\frac{\bar{E}_i^m}{T_i} + \frac{\left(1-\sum_{\forall\tau_i\in\tau^m\setminus G_{top}^m}U_i^m\right)\bar{E}_{new}^m}{\chi_{new}^m}\right) \tag{7.4}$$

Assume $\pi^1,\pi^2,\ldots,\pi^m$ represent the cores in descending order of $\Delta\bar{E}^m$. Initially, π^1 is selected. G_{top}^1 represents the set of tasks in the top least efficient sleep state group of π^1. The local cost of migration $LC_{\tau_j}^o$ of each task $\tau_j \in G_{top}^1$ is computed on every other core type π^o excluding π^1. The expression to determine the local cost of migration $LC_{\tau_j}^o$ is presented in Equation 7.5. This value is the τ_j's energy density on π^o plus the energy consumption per unit of time in idle period with τ_j on π^o minus the energy consumption per unit of time in idle period without τ_j on π^o. Where, χ_{new}^o and χ_{old}^o are the static sleep intervals with and without including τ_j on π^o respectively. The sleep states corresponding to χ_{new}^o and χ_{old}^o are determined to be $\S_{n2}^o$ and $\S_{n1}^o$ respectively. $\bar{E}_{new}^o = Cons(\chi_{new}^o, n2, o)$ and $\bar{E}_{old}^o = Cons(\chi_{old}^o, n1, o)$ are the energy consumption of a single sleep transition with or without τ_j respectively. The algorithm sorts all core types in ascending order of $LC_{\tau_j}^o$ to move τ_j.

$$LC_{\tau_j}^o = \frac{\bar{E}_j^o}{T_j} + \frac{\left(1-\sum_{\forall\tau_i\in\tau^o+\tau_j}U_i^o\right)\bar{E}_{new}^o}{\chi_{new}^o} - \frac{\left(1-\sum_{\forall\tau_i\in\tau^o}U_i^o\right)\bar{E}_{old}^o}{\chi_{old}^o} \tag{7.5}$$

The algorithm attempts to assign it to a core type with the least migration cost provided it is schedulable on that core. This process is repeated $\forall\tau_j \in G_{top}^1$. Line 10 to 24 in Algorithm 14 corresponds to this step. In case any of the tasks $\tau_j \in G_{top}^1$ is not schedulable, all assignments are undone and the algorithm moves to the next core type. On the other side, if the assignments of G_{top}^1 are successful, the algorithm computes the new expected total energy (TE) consumption of the platform through Equation 7.6. Where χ_{min}^m is the static sleep interval computed for the set of tasks allocated to π^m, $\S_n^m$ is the corresponding sleep state that minimises the function $Cons(\chi_{min}^m, n, m)$ and $\bar{E}^{\S_n^m}$ is the energy consumption of a sleep state $\S_n^m$ when initiated for χ_{min}^m. This new value is compared with the previous expected total energy consumption. If the new expected TE consumption is less than the previous expected TE consumption, the proposed algorithm reiterates over all steps unless the energy consumption of the previous iteration is greater than this new iteration.

$$\text{TE} = \sum_{\forall\pi^m}\left\{\left(\sum_{\forall\tau_i\in\tau^m}\frac{\bar{E}_i^m}{T_i}\right) + \frac{\left(1-\sum_{\forall\tau_i\in\tau^m}U_i^m\right)\bar{E}^{\S_n^m}}{\chi_{min}^m}\right\} \tag{7.6}$$

The maximum number of groups (of tasks) in each processor type is equal to its number of sleep states and the algorithm migrates the complete group to another core. The complexity of each iteration is $O(\ell \times M)$. Theoretically, the complexity of the entire algorithm is combinatorial, as a migrant task from one core type can be reassigned to it in another iteration, but in practice it converges very quickly. The algorithm avoids already computed assignments with a constraint that new assignment should reduce the energy consumption. The actual computation time and the number of migrations are discussed in Section 7.4.1.2.

7.3 Allocation Heuristics (With DVFS)

The restriction of a single active state in the previous allocation heuristics is relaxed and the general system model presented in Section 7.1 is adopted in this section. More precisely, this work considers a M-type heterogeneous platform with multiple frequencies and sleep states per core along with their transition overheads (time/energy) in a setting of partitioned scheduling. The objective is to map a given set of ℓ independent sporadic tasks onto this platform such that the overall energy consumption (active + leakage) is minimised, without violating any timing constraint.

The problem under consideration is NP-hard in a strong sense even when there is only one core type (with multiple processing units), since the bin-packing problem appears to be its special case. Hence, no polynomial time algorithm can derive an optimal solution. Instead of seeking optimal solutions by checking all possible task-to-core mappings, a two phase allocation process that can efficiently derive approximated solutions in reasonable time is proposed in this section. Similar to the allocation heuristics presented in Section 7.2, the respective first phases of the allocation heuristics minimise the active energy consumption, while the second phase reduces the static energy consumption by allowing hardware platform to use efficient sleep states. These two phases are discussed as follows.

7.3.1 First Phase of Allocation

This phase optimises the task-to-core mapping such that the active energy consumption of the system is reduced. More precisely, the behaviour of each task at different frequencies on each core type is considered during its allocation process and each task is assigned to a specific core by following either the LLED algorithm or the MM algorithm as detailed in Section 7.2.1. Note that LLED or MM do not consider the multiple frequency set-points. This section generalises the work to encapsulate a more generic power model.

As already mentioned, an optimal solution can be achieved through an exhaustive search. In an exhaustive search, all task-to-core mappings are checked at all frequency set-points of different core types. This has the drawback of being computationally intensive and different tasks allocated to the same core type may be constrained to run at different frequency set-points. Consequently, the specific core would need to switch to the frequency associated to each task, which in turn may lead to an unreasonable rise in the frequency switching overhead. This issue is avoided by assuming that all tasks allocated to a core run at the same frequency, i.e., the frequency is changed

Algorithm 15 First Phase of Allocation

Require: τ_i and π^m
1: EnergyOld = ∞
2: Generate a set Λ
3: **for all** $\Lambda_i \in \Lambda$ **do**
4: Set the frequency of the core types according to the set-points mentioned in Λ_i
5: CurrentAssignment = Perform the allocations with the LLED or MM algorithm
6: EnergyNew = Compute expected energy consumption of CurrentAssignment
7: **if** (EnergyNew<EnergyOld && CurrentAssignment Feasible) **then**
8: SelectedAssignment = CurrentAssignment
9: EnergyOld = EnergyNew
10: Index = i
11: **end if**
12: **end for**
13: **return** Selected task-to-core mapping and its corresponding Λ_i

on core type level rather than task level. With this assumption, the reduction in active energy can be achieved by

1. reducing the frequency of the core types; or
2. allocating the tasks to their favourite core types.

The first phase considers both factors to reduce the active energy consumption. The allocation of the tasks to their favourite core types is performed with the help of LLED or MM algorithm (Section 7.2.1) with an assumption of static DVFS, i.e., each core type is associated to a single frequency. The first factor is tackled by considering the combinations of all frequencies on all core types while performing the allocation with LLED/MM. One returning the minimum expected energy consumption is selected for the second phase. It is reasonable to do so as the number of frequency set-points of a core type is usually limited. As V^m represents the number of frequencies in the processor type π^m, then the overall complexity of this procedure is $O\left(\left(\max_{j=0}^{M} V^j\right)^M\right)$.

The pseudo-code of the first phase is presented in Algorithm 15. Initially, the proposed algorithm generates the set $\Lambda \stackrel{\text{def}}{=} \{\Lambda_1, \Lambda_2, \ldots, \Lambda_b\}$ (with $b \in \mathbb{N}^+$) that represents all possible combinations of frequencies set-points on all core types (line 2). Each element $\Lambda_k \stackrel{\text{def}}{=} \langle f^1_{x^1}, f^2_{x^2}, \ldots, f^M_{x^M}\rangle$ (with $x^i \in \{1,2,\ldots,V^i\}$) of a set Λ is a tuple of frequency set-points representing a concrete frequency set-point for each of the core type in the platform. The algorithm chooses each element Λ_k from a set Λ one by one. The frequency set-points in a tuple Λ_k are assigned to their corresponding cores (line 4).

Example 12. *Assume a platform with three different processor types, i.e.,* $\pi \stackrel{\text{def}}{=} \{\pi^1, \pi^2, \pi^3\}$. *Each processor type has two frequency set-points. In this case, the set* $\Lambda = \{\langle f^1_1, f^2_1, f^3_1\rangle, \langle f^1_1, f^2_1, f^3_2\rangle, \langle f^1_1, f^2_2, f^3_1\rangle, \langle f^1_1, f^2_2, f^3_2\rangle, \langle f^1_2, f^2_1, f^3_1\rangle, \langle f^1_2, f^2_1, f^3_2\rangle, \langle f^1_2, f^2_2, f^3_1\rangle, \langle f^1_2, f^2_2, f^3_2\rangle\}$. *The total number of combinations are equal to* $2^3 = 8$. *Each element in* Λ *shows a concrete frequency for each processor type.*

Afterwards, the task-to-core allocation is performed by using either the LLED or MM algorithm. The LLED and MM algorithm use the metrics energy density and density difference while deciding on the task-to-core allocation. The metric energy density and density difference (introduced in Section 7.2.1.1) in the context of system model used in this section are refined as follows.

Definition 49 (Energy density $ED^m_{i,v}$)**.** *The energy density of a task τ_i on $\pi^{m,v}$ (allocated a frequency f^m_v) is defined as $ED^m_{i,v} \stackrel{\text{def}}{=} \frac{\bar{E}^m_{i,v}}{T_i}$, which corresponds to its average power dissipation (i.e., average energy consumption per unit time).*

Definition 50 (Density difference $DD^m_{i,v}$)**.** *The density difference of τ_i on $\pi^{m,v}$ is defined as $DD^m_{i,v} \stackrel{\text{def}}{=} \min\{ED^h_{i,r} : h \neq m \wedge ED^h_{i,r} \geq ED^m_{i,v}\} - ED^m_{i,v}$ (with $r \in \{1,2,\ldots,V^h\}$). This corresponds to the difference between the next higher energy density of τ_i on any other core type and the energy density on the current core type.*

These two definitions allow to compute the energy density and density difference metrics needed for LLED and MM heuristics to performs task-to-core allocation. Once any of these heuristics perform the allocation, the resulting assignment is stored in a variable "CurrentAssignment" (line 5). The expected energy consumption EEC of the current assignment is computed by using Equation 7.8 and this value is stored in a variable "EnergyNew" (line 6).

$$EEC^m_v = \sum_{\forall \tau_i \in \tau^m} \frac{\bar{E}^m_{i,v}}{T_i} + \left(\left(1 - \sum_{\forall \tau_i \in \tau^m} U^m_{i,v} \right) Sf_e \right) \tag{7.7}$$

$$EEC = \sum_{\forall \pi^{m,v} \in \pi} EEC^m_v \tag{7.8}$$

The expected energy consumption EEC^m_v computed for an individual core type by Equation 7.7 consists of two parts, the active and sleep energy. The factor $\sum_{\forall \tau_i \in \tau^m} \frac{\bar{E}^m_{i,v}}{T_i}$ contributes to the active part, while the remainder of the equation corresponds to the energy consumption in the idle mode. The value Sf_e is the amount of energy consumption per time unit in the idle mode. It can be defined as $Sf_e \stackrel{\text{def}}{=} \frac{Cons(x,n,m)}{x}$, where x is the length of the minimum idle interval (or static sleep interval) in the schedule of π^m. The value of x can be computed through DBFP, PROC or LC-EDF (revisit Section 4.1 about the details of DBFP, PROC and LC-EDF). The minimum idle interval in DBFP, PROC and LC-EDF is equal to χ_{min}, Z_{min} and Q_{min} respectively. Hence, the value of x can be easily computed by considering any of these methods. The selection of the method to compute the minimum idle interval is entirely a design choice. The length of x is then used to select the most efficient sleep state $\S^m_n$. The most efficient sleep state is the one that minimises the function $Cons(x,n,m)$. The designer also has the option to assume that $Sf_e = P^m_I$ to reduce the complexity, and thus, a trade-off exists between accuracy and complexity.

The expected energy consumption of a core is computed if any task is assigned to it by LLED/MM. In case, a core is not assigned any task in this allocation phase, the algorithm assumes that core transitions into the deepest sleep state $\S^m_n$. Algorithm 15 compares this newly

developed assignment with the previous assignments. The new assignment is selected as a potential candidate for the second phase if it satisfies the following two conditions (line 7).

1. Its expected energy consumption EEC is less than the previously selected assignment.
2. It is a feasible assignment, i.e., all temporal constraints are met.

Finally, the index of Λ_k is stored to keep track of the frequency set-points on different core types. After going through all the element of Λ, the task-to-core assignment that has the least expected energy consumption is selected for the second phase of optimisation to reduce the static energy consumption of the platform. Moreover, the frequencies of the cores corresponding to such assignment is also passed on as an input to the second phase of optimisation.

7.3.2 Second Phase of Optimisation

The first phase of the task-to-core mapping tries to allocate each task to its favourite core type in order to reduce the individual active energy consumption of each task. However, this process ignores its effect on the leakage energy consumption. As has been discussed, a core type might have less active energy consumption but a small group of tasks allocated to this core might prevent it from using a better sleep state to decrease the leakage energy consumption. Therefore, the objective of the second phase is to alter the task-to-core assignment performed in the first phase such that the overall expected energy consumption is reduced. It considers the effect of the task-to-core assignment on the leakage energy through the use of better sleep states. The output of the first phase is considered as an input to this phase along with the frequency set-point associated to each core type. The sleep state that can be used in the idle mode on each core type is determined at the end of the first phase, and can be improved through two different methods given below.

1. By reducing the number of tasks on the given core type to gather more space in the schedule to accommodate better sleep states with longer transition overhead and break-even time.
2. By increasing the frequency of the core to speed up the execution of the tasks and thus, allow the system to stay longer in the sleep state using less energy hungry sleep states.

As mentioned previously, the ERTH algorithm initiates the sleep state for a predetermined time interval while ensuring the system schedulability. A task with a short deadline restricts the use of efficient sleep states. The first method (shuffling the tasks around) to achieve a less energy hungry sleep state is useful especially in such scenario. Few tasks preventing from the use of better sleep states can be moved to the other core types or collated to one particular core type in order to improve the energy savings. However, instead of moving the task to other cores, one can also increase the frequency of the core to speed up the execution of such tasks so as to achieved better sleep states. This approach is particularly helpful for scenarios where the cost of moving tasks to another core type is higher compared to that of increasing the frequency, e.g., when most of the tasks are CPU intensive and/or tasks are not feasible on other core types.

Algorithm 16 Second Phase of Optimisation (SP)

Require: Initial task-to-core mapping performed in the first phase of allocation and a set of frequencies Λ_i

```
1: Initially all the cores are allotted their corresponding frequency in Λ_i
2: repeat
3:     CHANGE = FALSE
4:     for q = 1 to M do
5:         repeat
6:             FEASIBLE = FALSE
7:             if π^q is not Empty then
8:                 Partition tasks into Groups on each core type
9:                 Find G^q_top to achieve next better sleep state
10:                TotalEnergyOld = EEC (see Equation 7.8)
11:                for all τ_i ∈ G^q_top do
12:                    Compute the moving cost of τ_i on all core types
13:                    Sort cores with respect to the moving cost of τ_i
14:                    Allocate τ_i in this order
15:                end for
16:                if All Tasks in G^q_top are Assigned then
17:                    OldEnergy = ∞
18:                    for all v to V^q do
19:                        NewEnergy = EEC^q_v (see Equation 7.7)
20:                        if NewEnergy < OldEnergy then
21:                            OldEnergy = NewEnergy
22:                        end if
23:                    end for
24:                    TotalEnergyNew = EEC
25:                    if TotalEnergyNew < TotalEnergyOld then
26:                        CHANGE = SUCCESSFUL
27:                        FEASIBLE = TRUE
28:                    else
29:                        Undo the assignments of all tasks in G^q_top
30:                        Restore the frequency of π^q
31:                    end if
32:                end if
33:            end if
34:        until FEASIBLE == TRUE
35:    end for
36: until CHANGE == TRUE
37: return Task-to-core assignment
```

The proposed heuristic is a hybrid strategy that exploits both approaches to achieve better energy savings. This hybrid strategy is summarised as follows. The set of tasks on each core is sorted in a non-increasing order of the difference between deadlines and WCETs. Then, the tasks are removed from the top of the list to gain the next better sleep state. The expected energy consumption of the remaining tasks on the given core type is checked on different frequency set points. The

gain achieved after removing the tasks and selecting the better frequency is compared against the cost of moving those tasks to another core type. If it reduces the overall energy consumption, this process is performed again on the same core until no more improvements can be achieved. The number of times it is repeated is bounded by the number of tasks initially assigned to each core type. The same procedure is applied to the other cores as well and is repeated until the system reaches a configuration of task assignment where no further energy savings can be achieved.

The pseudo-code of the second phase of optimisation is presented in Algorithm 16. Initially, all cores are assigned a frequency set-points determined in the first phase of allocation. In this algorithm, the cores can be sorted in any order. This work assumes that the cores are sorted with respect to their core type index, i.e., from π^1 to π^m. Assume, a core type $\pi^{m,v}$ is selected for the optimisation. All the tasks τ^m assigned to this core type are sorted in descending order with respect to $T_i - C^m_{i,v}$. The proposed algorithm computes the static sleep interval $\chi^{m,v}_1$ of the first task in the sorted list in an isolation on a core allocated a frequency f^m_v. The second task is the sorted list is superimposed on it to get the static sleep interval with two task i.e., $\chi^{m,v}_2$. Similarly, the third task from the list is superimposed to compute $\chi^{m,v}_3$ and so on. This process is repeated for all sorted tasks allocated to π^m to obtain a set of static sleep intervals called $\chi^{m,v} \stackrel{\text{def}}{=} \{\chi^{m,v}_1, \chi^{m,v}_2, \chi^{m,v}_3, \ldots, \chi^{m,v}_{\ell^m}\}$. This step is similar to the one presented in the second phase of optimisation without DVFS system model (see Section 7.2.2). It also holds the non-increasing property of $\chi^{m,v}_1 \geq \chi^{m,v}_2 \geq \chi^{m,v}_3 \geq \cdots \geq \chi^{m,v}_{\ell^m}$ and $\chi^{m,v}_{min} \stackrel{\text{def}}{=} \chi^{m,v}_{\ell^m}$, where $\chi^{m,v}_{min}$ represents the static sleep interval of τ^m allocated to a core type $\pi^{m,v}$. Similar to the techniques used in Section 7.2.2, this algorithm computes the sleep state corresponding to each element of $\chi^{m,v}$. The tasks that allows similar sleep states are grouped together to obtain a set $G^{m,v} \stackrel{\text{def}}{=} \{G^{m,v}_n, G^{m,v}_{n-1}, \ldots, G^{m,v}_1\}$ (line 8). These steps are demonstrated with the help of an example.

Example 13. *Assume a task-set $\tau^m = \{\tau_1, \tau_2, \tau_3, \tau_4\}$ is allocated to a core type $\pi^{m,v}$. Let $\chi^{m,v} = \{\chi^{m,v}_1, \chi^{m,v}_2, \chi^{m,v}_3, \chi^{m,v}_4\}$ be the set of computed static sleep intervals determined by superimposing tasks one by one. The sleep state corresponding to the different elements of $\chi^{m,v}$ are $\{\S^m_3, \S^m_2, \S^m_2, \S^m_1\}$ respectively, where the lower the index of a sleep state represents the shallower sleep state. In this example, a task τ_1 alone on this core allows a sleep state $\S^m_3$. The superimposition of τ_2 and τ_3 on τ_1 leads to the next shallower sleep state $\S^m_2$. Finally, the superimposition of τ_4 forces the core to use $\S^m_1$ to respect all the temporal constraints. The task-set allocated to this core τ^m is partitioned into three groups, $G^{m,v}_3 = \{\tau_1\}$, $G^{m,v}_2 = \{\tau_2, \tau_3\}$ and $G^{m,v}_1 = \{\tau_4\}$. Thus, the removal of tasks in $G^{m,v}_1$ allows the scheduler to use the next better sleep state, i.e., $\S^m_2$.*

The group of tasks that allows to achieve a next better sleep state is denoted $G^{m,v}_{top}$. The set of tasks $G^{m,v}_{top}$ on the core under consideration is determined (line 9). The local cost of moving each task in $G^{m,v}_{top}$ to another core type is computed on all core types (line 12). For each task, the cores are sorted in a non-decreasing order of its local cost (line 13). The allocation of a task to other core types is tried out by following the ascending order of local cost of moving and this process is performed for all the tasks in $G^{m,v}_{top}$ (line 11-15). The local cost of a task on any other core type can be computed by subtracting the energy consumption of each core type with and without this

task as shown in Equation 7.5. Please note that the symbols in Equation 7.5 are replaced by their respective symbols with frequency indexes.

As previously stated that moving tasks in $G_{top}^{m,v}$ to other core types may allow $\pi^{m,v}$ to use less energy hungry sleep states. Similarly, the sleep states can be further improved by increasing the frequency of the core. Moreover, the expected energy consumption can also be reduced by decreasing the frequency of the core. The reason for such a behaviour is that there may be a case where reducing the frequency does not affect the feasibility of the new sleep state. In order to cover these scenarios, the total expected energy consumption EEC_v^m of the core π^m is determined $\forall v \in \{1,2,\ldots,V^m\}$ by using Equation 7.7 (line 18-23). Note that the alteration of the frequency forces the system to recompute the static sleep interval $\chi_{min}^{m,v}$. The frequency f_v^m of π^m that has the minimum expected energy consumption is selected as a new potential frequency for π^m, i.e., $\min\limits_{\forall v \in \{1,2,\ldots,V^m\}} EEC_v^m$.

After performing the above mentioned procedure, the energy gain of the system is computed by subtracting the overall energy consumption of the new allocation from the last one. The overall expected energy consumption EEC can be computed through Equation 7.8 (line 24). If the expected energy consumption is greater than that of the previous task-to-core allocation, the task relocation is undone and the frequency of the core type under consideration is set back to its previous value (line 28-31). Otherwise, if the frequency has been altered in the previous phase, the tasks on the core under consideration are grouped again. Then, $G_{top}^{m,v}$ is determined and the same procedure explained above is applied. This procedure is repeated until the approach can not improve on the energy savings. Once the algorithm cannot get any further gain from the current core, the next core type is considered and the same process applies. The order in which the cores are addressed can be cyclic or some gain factor can be computed to sort them. This work assumes a cyclic procedure with respect to processor type indices and the cycling stops if there is no change when compared to the previous iteration. One can limit the number of times the algorithm should iterate over, but on average, the experiments show that this approach converges very fast as explained in Section 7.4.2.2.

7.4 Evaluation of the Partitioned Multicore Allocation Heuristics

The proposed task-to-core allocation heuristics are implemented in SPARTS to evaluate their effectiveness against different system parameters. SPARTS selects one of the core type and reference it as a default core type π^D. The task-set is initially generated for π^D. The WCET $C_{i,1}^D$ of τ_i is deemed to be $U_{i,1}^D \times T_i$, where $U_{i,1}^D$ is the utilisation of τ_i on π^D at maximum frequency f_1^D. The average capacity of the available heterogeneous platform is computed with the help of the average speed-up factor, which in turn is computed by reference to the default core type. The average speed-up factor of a core π^m is denoted as κ^m and defined as follows in Equation 7.9.

$$\kappa^m \stackrel{\text{def}}{=} \frac{\text{Clock cycle length of } \pi^m}{\text{Clock cycle length of } \pi^D} \tag{7.9}$$

The average capacity of the heterogeneous platform is represented as U^{avg}. It is defined in Equation 7.10. Please note that each core type has a single processing unit.

$$U^{avg} \stackrel{\text{def}}{=} \frac{1}{\kappa^1} + \frac{1}{\kappa^2} + \cdots + \frac{1}{\kappa^m} \tag{7.10}$$

The effective utilisation of the platform is again controlled with help of another variable ζ and is scaled as $U^{eff} = \zeta \times U^{avg}$. The range of ζ is $(0,1]$. In order to generate the WCET of the task τ_i on different core types, its individual utilisation on each π^m is computed as a random number within a range $U^m_{i,1} = [(1-\beta),(1+\beta)] \times \kappa^m \times U^D_{i,1}$, where β is a characteristic factor that models the task differing scaling behaviour on different core types. These utilisations are in turn used to compute WCET of the tasks on different core types. Usually, the task does not scale linearly with the average speed-up factor, but rather depends on the underlying hardware as well as specific instructions executed in a job.

Parameters	Values
Task-set sizes $\lvert\ell\rvert$	$\{100, \underline{200}, 500\}$
Share of RT/BE tasks ξ	$\langle 30\%, 70\% \rangle$
Inter-arrival time T_i for RT tasks	$[30ms, 50ms]$
Inter-arrival time T_i for BE tasks	$[50ms, 200ms]$
Sporadic delay limit Γ	$\{\underline{0.1}\}$
BCET limit C^b	$\{\underline{0.1}\}$
Characteristic factor β	$\{10\%, \underline{20\%}, 40\%\}$
Helper variable ζ	$\{0.5 : 0.05 : 0.9\}$

Table 7.2: Overview of simulator parameters used to evalute non-DVFS heuristics

π^m	P^m_A	P^m_I	κ^m	P^m_1	P^m_2	P^m_3	P^m_4	tr^m_1	tr^m_2	tr^m_3	tr^m_4
$\pi^1(\pi^D)$	1.0	0.39	1.0	0.31	0.21	0.12	0.05	2	50	250	500
π^2	2.2	0.86	0.5	0.67	0.47	0.27	0.11	3	66	333	667
π^3	6.0	2.33	0.2	1.83	1.29	0.74	0.30	4	83	416	833
π^4	13.0	5.05	0.1	3.98	2.79	1.61	0.64	5	100	500	1000

Table 7.3: Heterogeneous multicore platform and its parameters

7.4.1 Simulation Results (Non-DVFS)

The performance of the heuristics proposed for non-DVFS platform is evaluated with the parameters of the SPARTS defined in Table 7.2. The underlined values are the default values if not specified in the description of an individual experiment. The hardware parameters of heterogeneous platform used in the experiments are shown in Table 7.3. As in the other evaluations of the thesis, the power model for the default core in the experiments is modelled after the FreeScale PowerQUICC III Integrated Communication Processor *MPC*8536 [Sem]. The FreeScalePowerQUICC III core specifications are given in Table 4.2. The values of the other core types are derived from this core type to generate a heterogeneous platform. Each core type is assumed

to have four sleep states, with $\{\S_x^m : x \in \{1,2,3,4\}\}$ representing different sleep states such as Doze, Nap, Sleep and Deep Sleep respectively. Their transition overheads are assumed and afterwards, the break-even-time are computed accordingly. All power dissipation parameters presented in Table 7.2 have a unit of watt, while the transition overhead time is given in μ seconds. There is also only a single unit of each core type. The average system capacity is computed to be $U^{avg} = \frac{1}{1} + \frac{1}{0.5} + \frac{1}{0.2} + \frac{1}{0.1} = 18$. As the helper variable ζ is changed within an interval of $[0.5, 0.9]$, therefore, the effective utilisation of the system U^{eff} is within a range of $[0.5, 0.9] \times 18 = [9, 16.2]$. The energy consumption of a task is, however, not a mere function of its execution time. As such the values of $\bar{E}_i^m$ are computed using the average execution time $\bar{C}_i^m$ and a random value similar to the utilisation conversion $\bar{E}_i^m = [1-\beta, 1+\beta] \times P_A^m \times \bar{C}_i^m$. The average execution time $\bar{C}_i^m$ can be estimated through profiling.

The state-of-the-art algorithms cannot be adapted to the generic power model used in this work and thus a direct comparison is at least extremely difficult. Therefore, the well know bin packing algorithms worst-fit decreasing (WFD) and first-fit (FF) are selected as a base line against which the proposed heuristics are compared. It has been shown by Aydin and Yang [AY03] that WFD performs better when compared to other conventional bin packing algorithms for homogeneous platforms. In the experiments, it has been observed that WFD performs worst when applied to mapping tasks to heterogeneous platforms. It was unable to schedule majority of tasks-set at higher utilisations making it hard to compare against proposed algorithms. Therefore, initially, the proposed approached is compared against the FF algorithm. Later on, the experiments comparing WFD against FF are also presented. Moreover, the FF algorithm allocates the tasks sorted with respect to their D_i or T_i following the order from the slowest core type to the fastest core type. The results under labels LLED-SP and MM-SP represent the second phase applied on the allocation of LLED and MM respectively. Two different scenarios are created. First scenario models the system with very efficient sleep states having low transition overhead (time and energy), while the second scenario models the system, with substantially less efficient sleep states. All results are normalised to the corresponding values of the FF algorithm.

7.4.1.1 Exploring a Small Break-even-time (SBET)

In this scenario, as the overhead of the sleep state is low, therefore, different cores can still achieve the most efficient sleep state even at high utilisation. This scenario does not leave much room for the second phase to save any additional energy when compared to LLED. Nevertheless, MM-SP saves energy in some cases when compared to MM but it is fairly minimal. Therefore, this scenario compares the energy consumption of MM-SP and LLED-SP, instead of comparing the first and the second phases.

Firstly, the performance of LLED-SP and MM-SP is analysed for different number of core types. Figure 7.3 shows the normalised energy consumption of the system for only 4 core types. The figure for 2 cores looks similar to Figure 7.3 but does not provide as high energy gains over FF due to the limited scope for optimisation. With 4 core types, initially, the difference of LLED-SP

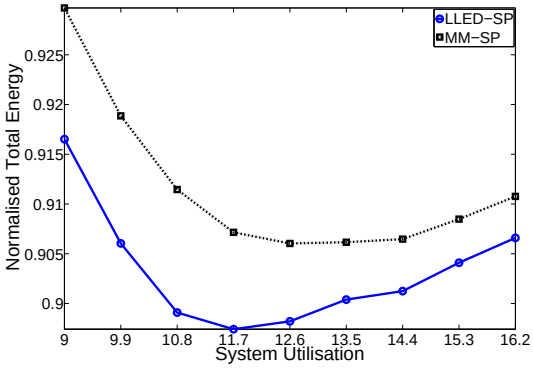


Figure 7.3: (SBET) 4 core types

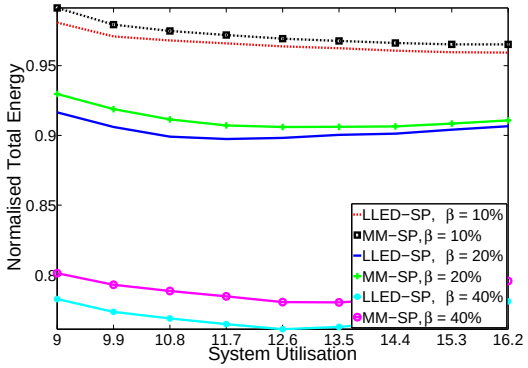


Figure 7.4: (SBET) Variation in β

and MM-SP increases but then starts to shrink towards higher utilisations. The reason for this behaviour is quite obvious in that LLED-SP and MM-SP has more chance at low utilisation to allocate task to their favourite core. However, towards, high utilisations, this flexibility decreases along with their difference. In the best-case, LLED-SP consumes 10% less energy when compared to FF, while MM-SP saves energy slightly under 10%.

The effect of variation in the characteristic factor β on the normalised total energy consumption is demonstrated in Figure 7.4. β controls the variation dynamic power consumption of the task and models its variation from the average dynamic power dissipation of the core. Figure 7.4 demonstrates that the energy consumption of both approaches decrease with an increase in the range of β. The developed power model on average favours the slow core. However, this factor (β) can change this behaviour. With $\beta = 10\%$, small portion of tasks are more favourable to the fast cores. Hence, the FF algorithm that fills the slowest core first does a few task allocation to their unfavourable cores. Consequently, the gains of LLED-SP and MM-SP are less at $\beta = 10\%$. However, as the β range increases, the tasks probability to favour a fast core becomes higher. Therefore, LLED-SP and MM-SP give better allocations for higher values of β. Similar to the previous observation, the difference of MM-SP and LLED-SP is higher at low utilisation and decreases with an increase in the system utilisation.

Figure 7.5 demonstrates the effect of task-set size variation on the given allocations mechanism. In general a large task-set size increases the probability of the tasks to be allocated to their unfavourable core with FF. Therefore, the relative energy consumption of the LLED-SP and MM-SP algorithms decreases with an increase in the task-set size. However, this saving reduces with an increase in the effective utilisation. At low utilisations, the difference of energy consumption between different task-set sizes is small for both LLED-SP and MM-SP. This different increase with an increase in effective system utilisation and deteriorates again at high utilisations. The algorithm can easily allocate tasks to their favourite cores at low utilisation, and this becomes challenging with an increase in system utilisation. At very high utilisation, the scope of energy saving also decreases in the second phase. For small task-set size of 100, FF also performs well at low utilisation. However, this effect deteriorates with increasing effective system utilisation.

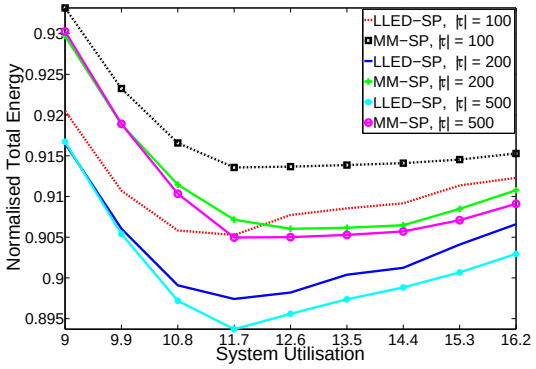


Figure 7.5: (SBET) Variation in $|\tau|$

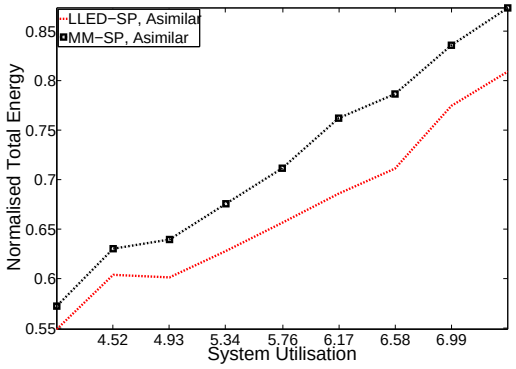


Figure 7.6: (SBET) Asimilar platform

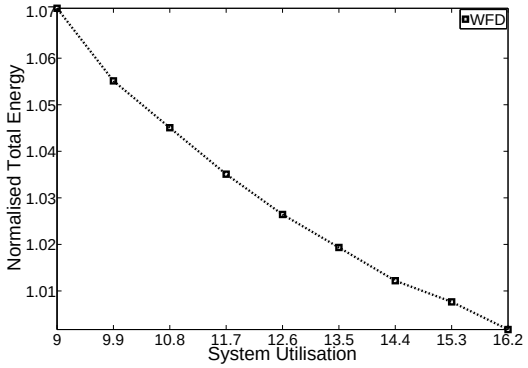


Figure 7.7: (SBET) 4 core types (WFD)

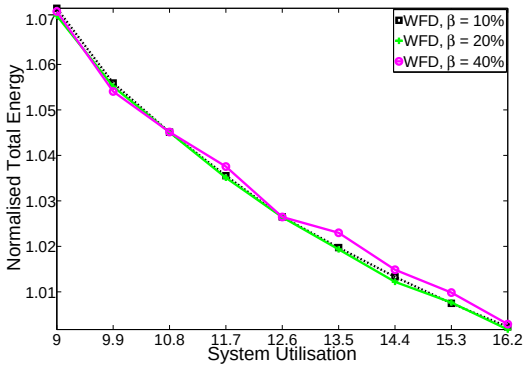


Figure 7.8: (SBET) Variation in β (WFD)

Processors types given in Table 7.3 have a similar ratio of $\frac{P_A^x}{P_A^y} \approx \frac{\kappa^y}{\kappa^x}$. SPARTS generates a case where this ratio is not the same and tasks always favour the same core i.e., $\frac{P_A^x}{P_A^y} \neq \frac{\kappa^y}{\kappa^x}$. This case allows to evaluate a system, where all tasks are competing for the best core types. For this experiment, the heterogeneous platform given in Table 7.3 is modified to asimilar heterogeneous platform by changing the κ^m values of different cores from $1, 0.5, 0.2, 0.1$ to $1, 0.6, 0.45, 0.3$ respectively. The average capacity of the asimilar platform is $U^{avg} = \frac{1}{1} + \frac{1}{0.6} + \frac{1}{0.45} + \frac{1}{0.3} = 8.22$. The effective utilisation U^{eff} is varied within a range of $[0.5, 0.9] \times 8.22 = [4.11, 7.4]$. Figure 7.6 presents the results for the asimilar platform. The relative energy consumption of LLED-SP and MM-SP is low at low utilisation and gradually increases towards high utilisation. All algorithms, attempt to allocate tasks in order from the slowest core to the fastest core. LLED-SP can rank tasks in an efficient way and saves more energy. Similarly, MM-SP also performs better when compared to FF as it also does some ranking of the tasks but FF does not prioritise the tasks to account for global energy benefits.

Comparison With Worst Fit Decreasing (WFD): The WFD algorithm is also implemented in SPARTS and compared against the FF algorithm. All values are normalised to the corresponding results of FF. Figure 7.7 demonstrates the energy consumption of WFD for four different cores types. It is evident that WFD performs worse when compared to the FF algorithm. Its per-

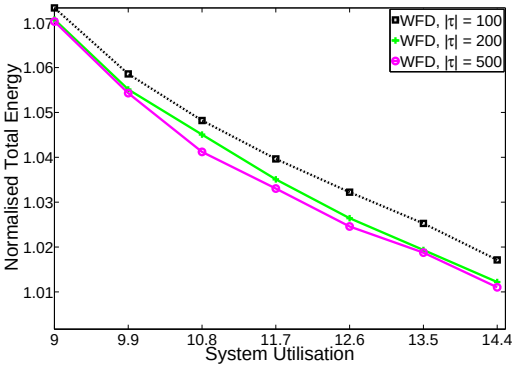


Figure 7.9: (SBET) Variation in $|\tau|$ (WFD)

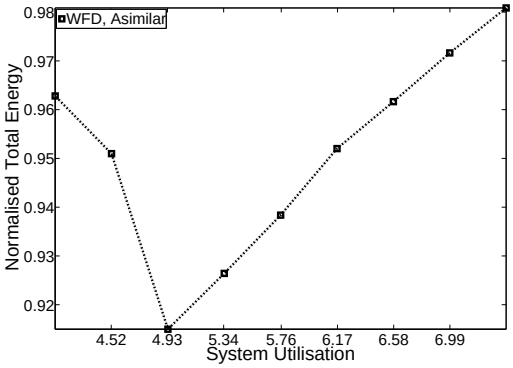


Figure 7.10: (SBET) Asimilar platform (WFD)

formance slightly increases with an increase in the system utilisation and the difference of energy consumption with FF decreases. WFD follows the similar trend for different value of β as shown in Figure 7.8. One interesting observation is that change of β has very similar effect on both WFD and FF decreasing algorithms.

The effect of different task-set sizes on the *WDF* algorithm is also compared as shown in Figure 7.9. The performance of the WFD algorithm increases with an increase in the task-set size. Moreover, the utilisation of the system is only varied upto 14.4 in this experiment because the WFD algorithm was not able to schedule most of the task-sets on higher utilisations. For the asimilar platform, Figure 7.10 shows WFD performs better than FF. However, its performance is substantially worse when compared to our algorithms.

7.4.1.2 Exploring a Large Break-even-time (LBET)

In this scenario, the heterogeneous platform is modelled such that the core types have large overheads of sleep transitions (time/energy). Such model is generated by scaling the transition delays of all the sleeps states by a factor of 12 and their bet_n^m determined accordingly. An interesting result shows that it is not necessary that tasks assigned to their favourite core will always reduce the overall system energy consumption. In this scenario, the overall energy consumption depends mostly on the characteristics of the core and it depends less on those of the tasks. This fact will be evident in the following experiments, in which LLED, MM, LLED-SP and MM-SP algorithms are compared against each other. The base line is still the corresponding energy consumption of FF. Furthermore, the range of ζ is increased to $[0.4, 0.9]$ with a step size of 0.05 for this scenario.

Figure 7.11 shows the normalised total energy consumption of system for 4 core types. At low utilisation, though LLED and MM had a chance to allocate tasks to their favourite core but globally it is not energy efficient. The reasons is that these algorithms are not accounting the effect of their allocation on the core sleep states. The FF algorithm which is also sleep state agnostic allocation mechanism surprisingly performs well compared to LLED and MM. It allocates the core from the slowest one and allows fast core to have empty space to use their efficient sleep state. However, our LLED-SP and MM-SP algorithm compares well to FF at low utilisations and

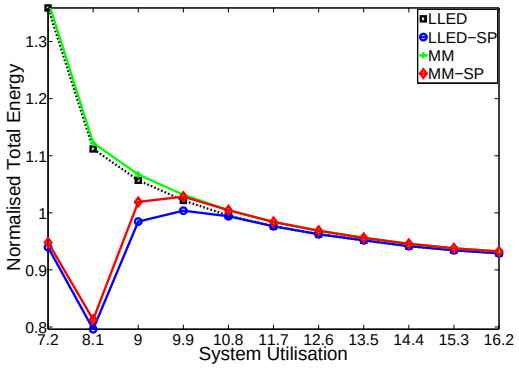


Figure 7.11: (LBET) 4 core types

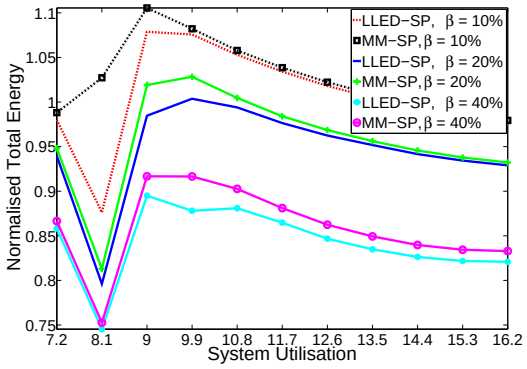


Figure 7.12: (LBET) Variation in β

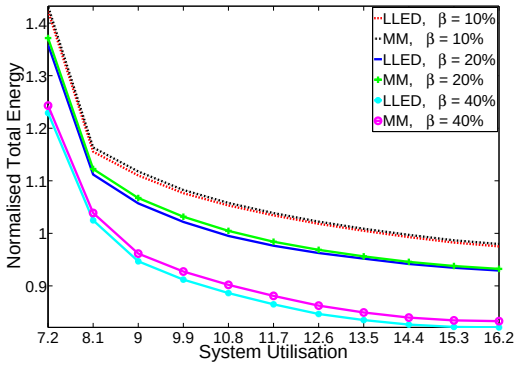


Figure 7.13: (LBET) Variation in β

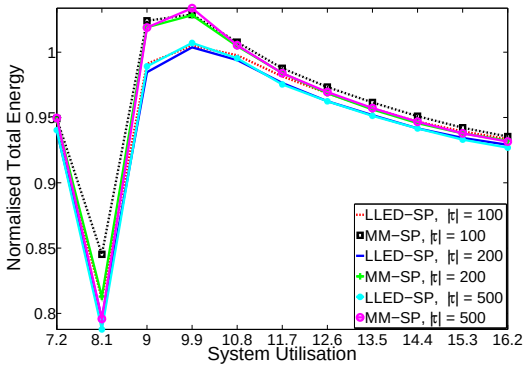


Figure 7.14: (LBET) Variation in $|\tau|$

compensates for the wrong allocations done by LLED and MM respectively. It is interesting to see that for low utilisations LLED-SP and MM-SP achieves substantial gain. For high utilisations, LLED and MM energy consumption reduces when compared to FF. Hence, a combination of initial first phase allocation (LLED or MM) with the second phase is a good choice for most system utilisations, except for some corner cases (e.g., at utilisation of 9.9 in Figure 7.11). In the detailed analysis of utilisations between 7.2 and 9, it has been observed that FF loses the efficient sleep states earlier than LLED-SP or MM-SP. Hence, the energy consumption of LLED-SP and MM-SP is dropped at $U^{eff} = 8.1$ when compared to FF. It is also evident from Figure 7.11 that the performance of LLED always dominates MM, and similarly, the performance of LLED-SP over MM-SP.

The variation in the characteristics factor β is demonstrated in Figure 7.12 and Figure 7.13. Similar to the results in SBET (Figure 7.4), the performance of LLED-SP and MM-SP given in Figure 7.12 increases with an increase in the value of β and the similar trend is followed by LLED and MM in Figure 7.13. Figure 7.12 also shows that LLED-SP always dominates MM-SP and the same is true in Figure 7.13 for LLED and MM.

The effect of variation in the task-set size is presented in Figure 7.14 and Figure 7.15. Unlike to Figure 7.5, in this scenario the task-set size does not make any difference on the performance of all the algorithms. To evaluate the platform, where all the tasks prefer similar core type, the

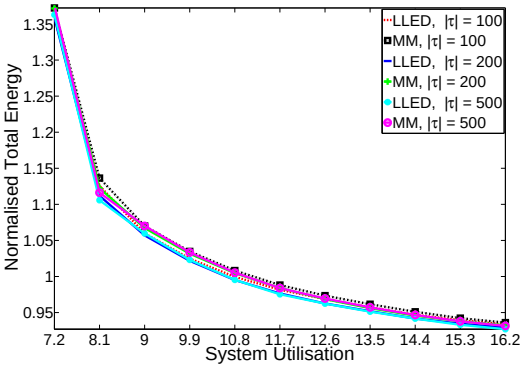


Figure 7.15: (LBET) Variation in $|\tau|$

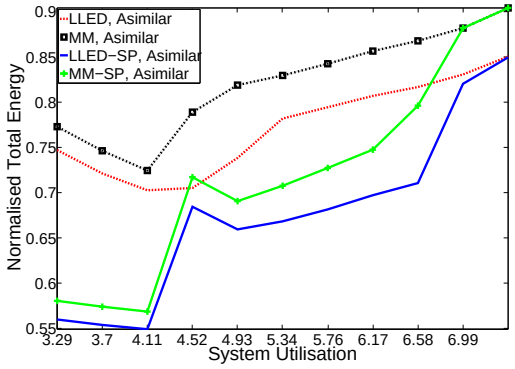


Figure 7.16: (LBET) Asimilar platform

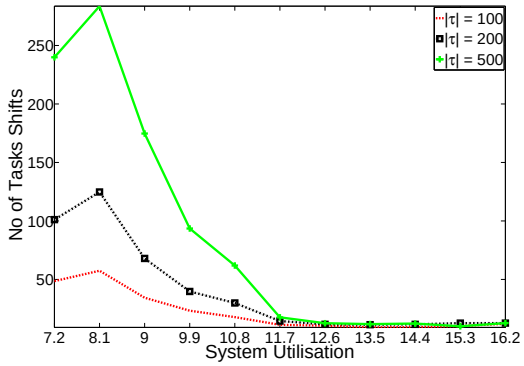


Figure 7.17: (LBET) Decisions

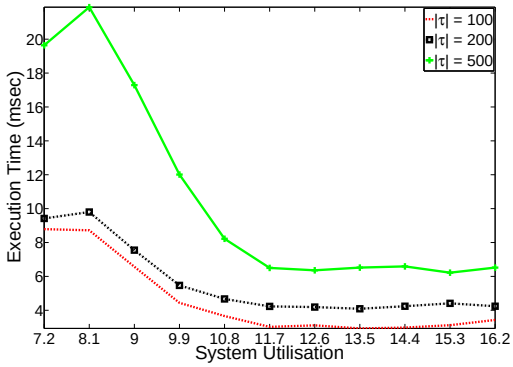


Figure 7.18: (LBET) Time calculation

same setup of Figure 7.6 is adopted. The results of this experiment are shown in Figure 7.16. All the algorithms follow the same race to allocate tasks to the slowest core. Furthermore, LLED performance dominated over MM, and towards high utilisations, it even consumes less energy compared to MM-SP. Overall, LLED-SP performs better for all utilisations.

Figure 7.17 and Figure 7.18 present the execution time and the number of tasks migrations between different core types of the second phase of allocation respectively. The results in Figure 7.18 are generated with a server having 8 Intel Xenon 1.60GHz processors and a memory size of 8GB. The allocation process of the second phase is very fast even for a large task-set size of 500. Figure 7.17 shows that the number of migrations (also execution time) decrease with an increase in effective utilisation as the tasks have less freedom to manoeuvre due to high utilisations. Less loaded system ($U^{eff} = 7.2$) allows cores to use their more efficient sleep state anyway. Therefore, $U^{eff} = 7.2$ has fewer number of migrations (executions time) when compared to $U^{eff} = 8.1$.

Comparison With Worst Fit Decreasing (WFD)**:** Similar to the previous scenario, WFD is also compared against FF in this scenario. All the values of the WFD algorithm are normalised to the corresponding values of FF for consistency. For four core types, normalised energy consumption of the WFD algorithm is shown in Figure 7.19. The shape of the curve is similar to the previous scenario (Figure 7.7). The difference of energy consumption between FF and WFD is higher in this scenario when compared to first scenario. The variation in β and task-set sizes also

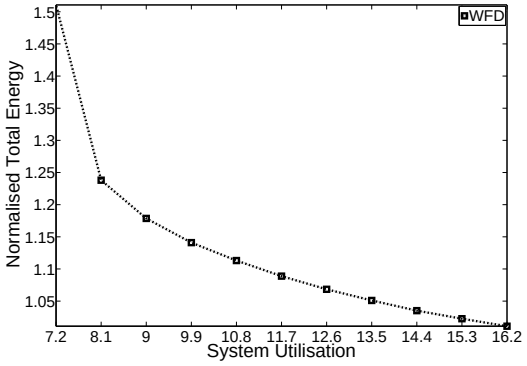


Figure 7.19: (LBET) 4 core types (WFD)

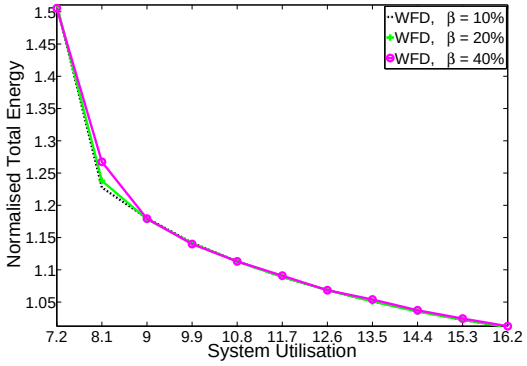


Figure 7.20: (LBET) Variation in β (WFD)

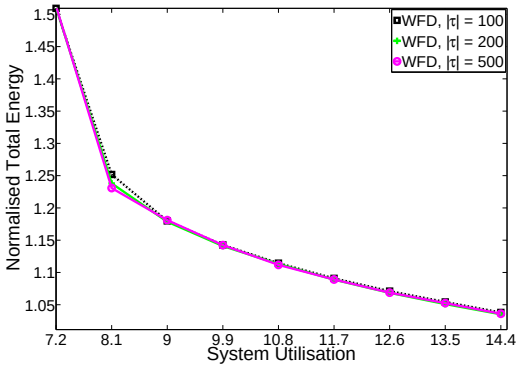


Figure 7.21: (LBET) Variation in $|\tau|$ (WFD)

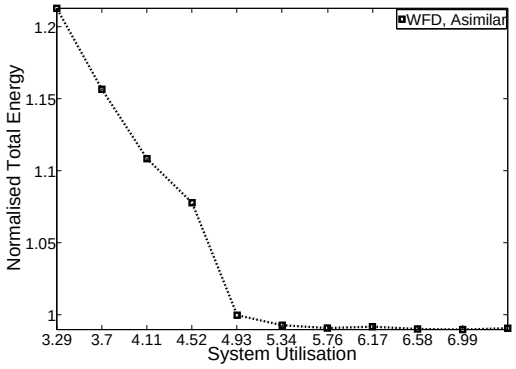


Figure 7.22: (LBET) Asimilar platform (WFD)

have the similar effects when compared to previous scenario. The normalised energy consumption for different values of β and different sizes are shown in Figure 7.20 and Figure 7.21 respectively. In this scenario, the energy consumption of WFD is higher for the asimilar platform as shown in Figure 7.22.

7.4.2 Simulation Results (With DVFS)

To evaluate the performance of the heuristics proposed for DVFS enable heterogeneous multicore platform, SPARTS is set up with the parameters presented in Table 7.4. The underlined values are the default values if not specified in the description of an individual experiment. In order to determine the execution time of the tasks at different frequency set points, one can perform the measurements to generate the exact values. However, in this work, the execution time of tasks given at maximum frequency is modelled to other frequencies based on the nature of their instructions. Some instructions operate on memory where a change in frequency does not affect the execution time much. CPU intensive instructions scale well with the frequency scaling. Some of the instructions along with their surrounding instructions scale initially but gradually their scaling towards high frequencies fades away. The details of the model used to scale the execution time at different frequencies is explained as follows. First, the number of instructions of the tasks are computed on π^m from its $C^m_{i,1}$. To do so, it is assumed that the execution time is composed of three

Parameters	Values
Task-set sizes $\|\ell\|$	$\{\underline{40}, 60, 80\}$
Share of RT/BE tasks ξ	$\langle 50\%, 50\% \rangle$
Inter-arrival time T_i for RT tasks	$[30ms, 50ms]$
Inter-arrival time T_i for BE tasks	$[50ms, 200ms]$
Sporadic delay limit Γ	$\{\underline{0.1}\}$
BCET limit C^b	$\{\underline{0.5}\}$
Characteristic Factor β	$\{10\%, \underline{20\%}, 40\%\}$
Helper variable ζ	$\{0.4 : 0.05 : 0.9\}$

Table 7.4: Overview of simulator parameters used to evaluate the allocation heuristics proposed for heterogeneous platform with DVFS capabilities

parts: a CPU bound time share (x), memory bound time share (y) and latency hiding time share (z). The sum of these parts yields $x+y+z = 100\% \times C^m_{i,1}$. The CPU bound time share x is randomly selected from the interval $[20\%, 80\%] \times C^m_{i,1}$, the latency hiding time share z is randomly chosen from interval $[0\%, 20\%] \times C^m_{i,1}$ and the memory bound time share $y = (100\% - x - z) \times C^m_{i,1}$. It is assumed that each memory access require 70 cycles and that a CPU bound instruction is executed in one cycle at maximum speed. Finally, the number of CPU intensive instructions is equal to x.

The memory bound and latency hiding instructions are equal to $\frac{y}{70\text{cycles}}$ and $\frac{z}{70\text{cycles}}$, respectively. While computing the execution time at different frequencies, it is assumed that a memory bound instruction takes the same amount of time on all frequencies whereas the CPU bound instructions scale as $\frac{1}{f} \times x$. To determine the execution time of latency hiding instructions on different frequency set points, three intervals $(0, f_{LH})$, $[f_{LH}, f_{Sat}]$ and $(f_{Sat}, 1]$ are defined as shown in Figure 7.23, where f_{LH} is latency hiding low limit and f_{Sat} is the latency hiding saturation frequency. Note that latency hiding techniques takes full benefit at low frequency set points and hence behave like CPU intensive instructions. The interval defined for such behaviour is $(0, f_{LH})$. With an increase in the frequency, the behaviour of the latency hiding degrades and the execution time of an instruction starts to transition from a CPU intensive instruction towards a memory intensive one. The interval $[f_{LH}, f_{Sat}]$ will correspond to such a behaviour. The transition from a CPU intensive instruction towards a memory intensive instruction is approximated linearly as shown in Figure 7.23. Suppose that t_{LH} and t_{Sat} are the execution times of an instruction at frequencies f_{LH} and f_{Sat}, respectively. By assuming an inverse relationship between execution time with frequency, the execution time at the saturation frequency f_{Sat} is $t_z = \frac{t_{Sat} + t_{LH}}{2}$. Hence, the execution time of instructions in a range $[f_{LH}, f_{Sat}]$ is computed with the help of the equation of a line: $t = t_{LH} + \frac{t_z - t_{LH}}{f_{Sat} - f_{LH}}(f - f_{LH})$, where f is the input frequency and t is the execution time of the instruction. It is assumed in interval $(f_{Sat}, 1]$ that the execution time of an instruction is constant and equal to t_{Sat}. Note that in this experimental setup, the values of f_{LH} and f_{Sat} are fixed to 40% and 80% of the maximum frequency, respectively.

The heterogeneous multicore platform in our experiment setup is composed of four core types with the parameters given in Table 7.3 and Table 7.5. Table 7.3 specifies the parameter of the sleep

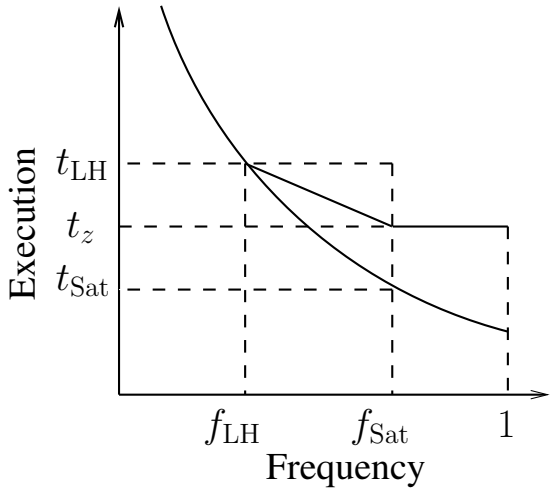


Figure 7.23: Latency hiding instruction scaling

states, while Table 7.5 defines the parameters corresponding to different frequency set-points. Similar to the non-DVFS case, the speed-up factors are chosen to be $1, 0.5, 0.2, 0.1$ for π^D, π^2, π^3 and π^4, respectively. The frequencies of π^D and other core types (π^2, π^3, π^4) are randomly chosen from interval $[(1-\alpha),(1+\alpha)] \times f_v^D$ (with $v = 1,2,\ldots,6$), where α is the variability factor and is set to 25%. The average power dissipation of a core π^m at frequency f_v^m is represented as $P_{f_v^m}^m$. The average power dissipation at different frequencies are randomly chosen from interval $[(1-\alpha),(1+\alpha)] \times P_{f_v^D}^D$ (with $v = 1,2,\ldots,6$). Each core is assumed to have four different sleep states $\{\S_x^m : x \in \{1,2,3,4\}\}$ representing Doze, Nap, Sleep and Deep Sleep, respectively. Different core types select the power dissipation of the sleep states from an interval $[(1-\alpha),(1+\alpha)] \times P_n^D$ (with $n = 1,2,3,4$), while power dissipation of sleep states of the default core π^D is modelled from the sleep states parameters of PowerQUICC III Table 4.2. The average capacity with the given parameters is equal to $U^{avg} = \frac{1}{1} + \frac{1}{0.5} + \frac{1}{0.2} + \frac{1}{0.1} = 18$. With the variation of ζ in $[0.4, 0.9]$, the effective utilisation U^{eff} is within $[0.4, 0.9] \times 18 = [7.2, 16.2]$. However, the results at $U^{eff} = 16.2$ are not presented in this section as the FF algorithm fails to schedule task-sets correctly in most of the cases.

Similar to the non-DVFS case, the FF algorithm is considered as a baseline to illustrate the performance of the proposed heuristics since the state-of-art cannot be extended to the realistic power models used in this chapter. In the experimental setup, FF is granted an ideal configuration in which all the cores are aligned from the slowest to the fastest core (i.e., lowest to highest power

π^m	κ^m	$P_{f_1^m}^m / P_{\mathrm{A}}^m$	$P_{f_2^m}^m$	$P_{f_3^m}^m$	$P_{f_4^m}^m$	$P_{f_5^m}^m$	$P_{f_6^m}^m$
$\pi^1(\pi^D)$	1.0	1	0.88	0.85	0.79	0.76	0.73
π^2	0.5	2.2	1.99	1.95	1.81	1.79	1.51
π^3	0.2	6	5.48	5.46	4.74	4.26	3.58
π^4	0.1	13	11.66	11.27	10.93	10.61	9.58

Table 7.5: Frequency specification of the heterogeneous multicore platform

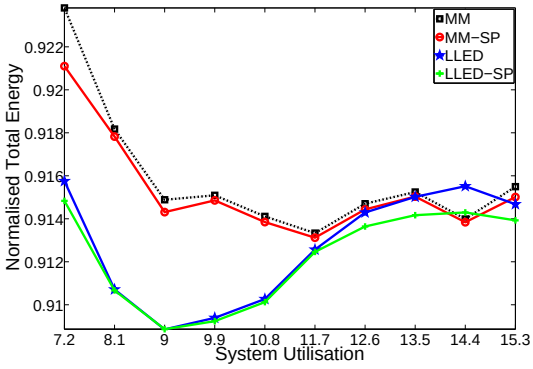


Figure 7.24: (SBET) 4 core types

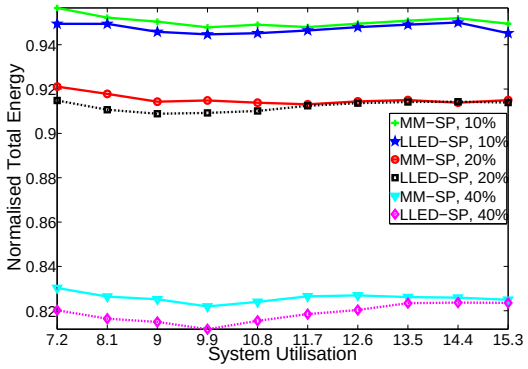


Figure 7.25: (SBET) Variation in β

dissipation). On average the slower cores are favourite. This allows FF to fill the most efficient core first. The task-set for FF is sorted in a non-decreasing order of their T_i. The energy consumption is tested for all elements of Λ and the one with the least expected energy consumption is chosen for the comparison. All the values in the graphs are normalised to the corresponding values of the FF algorithm. The results under labels LLED-SP and MM-SP represent the second phase of optimisation (in DVFS case) applied on the allocation performed with LLED and MM in the first phase respectively. Again two different scenarios are simulated in this result section.

7.4.2.1 Exploring a Small Break-even-time (SBET)

This scenario assumes a low transition overhead of the sleep states (see Table 7.3). The efficient sleep states allow the system to attain better sleep states even at high utilisations. Initially, the normalised total energy consumption of task-to-core allocations performed by LLED, LLED-SP, MM and MM-SP is presented in Figure 7.24 for 4 core types. The energy saving are less at low utilisations as all the tasks can be easily allocated to their favourite core types and the best sleep states can be used as well. With an increase in the system utilisation, the energy savings increases as the system scope to optimise the allocation increases. However at higher utilisation it starts to decrease again as the system does not have enough capacity to perform the optimisation. The difference between LLED-SP and MM-SP increases in the beginning and shrinks towards the higher utilisations as the potential to optimise the allocation decreases. The same experiment is performed for 2 core types and it has been observed that the energy savings increases with the number of core types. The difference between first phase LLED/MM and second phase LLED-SP/MM-SP of the two corresponding algorithms is small in this scenario. The potential for the second phase to reduce energy consumption is limited due to the existence of efficient sleep states that have small transition delay and break-even times. However, the difference between LLED and LLED-SP increases at high utilisations as all the core types cannot use better sleep states any more and hence, the sleep states optimisation gains its importance. Therefore, in this scenario, only LLED-SP and MM-SP are compared against each other.

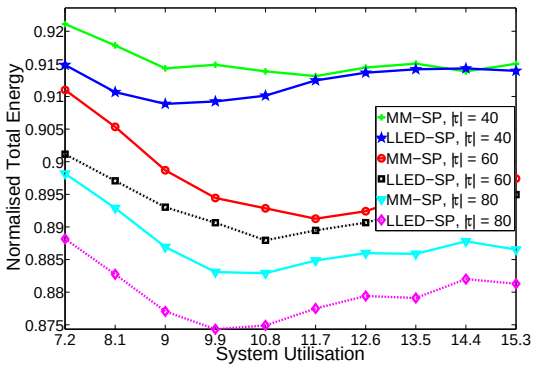


Figure 7.26: (SBET) Variation in $|\tau|$

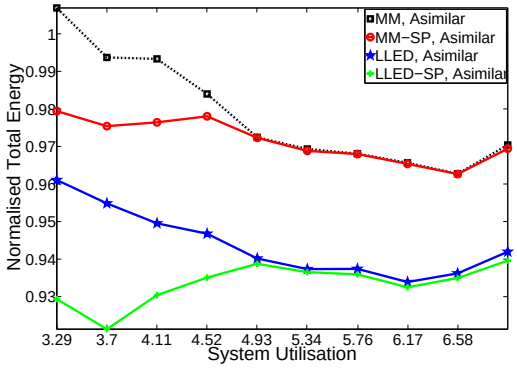


Figure 7.27: (SBET) Asimilar platform

The effect of variation in the characteristic factor β is observed in Figure 7.25. β controls the behaviour of the tasks on different core types. The power model on average favours the slow core to provide ideal condition to FF. The increase in the value of β changes this behaviour. Therefore, the normalised energy consumption of the proposed algorithms decreases with an increase in the value of β. In the best case, the proposed algorithms save energy over FF up to $18\%, 9\%, 5\%$ for β values of $40\%, 20\%, 10\%$, respectively. Such saving is considered worthwhile in the embedded systems industry. These trends highlight that the proposed algorithms will perform even better in unrelated heterogeneous platforms as the increase of β increases the difference between LLED-SP and MM-SP. Similar to the previous case, the difference between LLED-SP and MM-SP is higher at low utilisations and decreases at higher utilisations due to the aforementioned reason.

The performance of LLED-SP and MM-SP is evaluated for different task-set sizes as shown in Figure 7.26. Large task-set sizes increase the probability of FF to allocate tasks to their unfavourable core types. Therefore, the energy saving increases. Moreover, the difference between LLED-SP and MM-SP increases with large task-set sizes. This shows the lower quality of MM-SP tasks sorting while doing the allocation, compared to LLED-SP.

Similar to the non-DVFS case, asimilar hardware platform is also generated here such that certain cores will become favourite irrespective of their speed or power relation with other cores. Tasks will always compete for those favourite cores to save the energy. Now the assumption that the slower core types on average are favourite is not valid any more. To generate such platforms, the speed-up factor κ^m of the core types given in Table 7.5 is varied from $1, 0.5, 0.2, 0.1$ to $1, 0.6, 0.45, 0.3$, respectively. The available capacity U^{avg} of the platform is $U^{avg} = 8.22$. The effective utilisation U^{eff} will be varied within $[0.5, 0.85] \times 8.22 = [4.11, 6.98]$. The results of such a platform is presented in Figure 7.27. As the order of the core types is no longer favourable to FF, the energy savings of the proposed algorithms increases with respect to an increase in the system utilisations. The difference between the first and second phases is also prominent here at low utilisations.

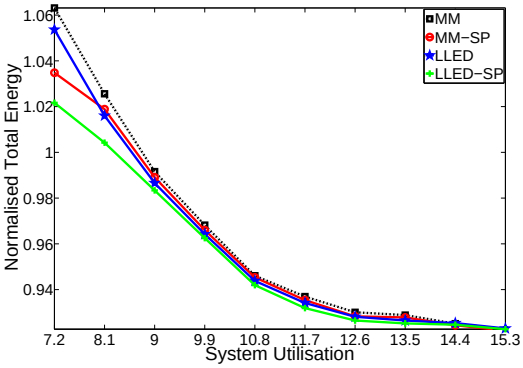


Figure 7.28: (LBET) 4 core types

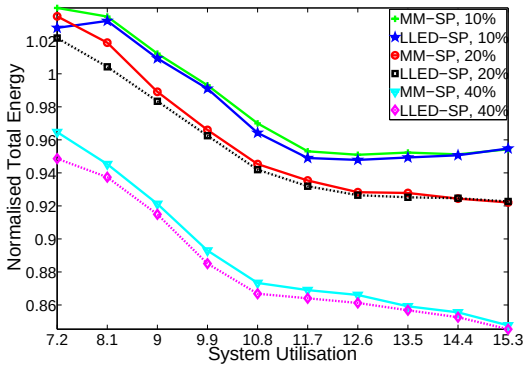


Figure 7.29: (LBET) Variation in β

7.4.2.2 Exploring a Large Break-even-time (LBET)

The performance of the proposed algorithms is evaluated on a heterogeneous platform that does not have efficient sleep states. This platform is modelled by scaling the transition delay of the sleep states of the given platform (see Table 7.5) with a factor of 12. It has been noticed that the task-to-favourite core type allocation is not always efficient and it depends on the core type characteristics as well.

The normalised energy consumption of the platform on the different core types is presented in Figure 7.28 for 4 core types. At low utilisations LLED allocates the tasks to their favourite core types in order to minimise the dynamic power dissipation. However, such an allocation is not globally optimal as the energy consumption in the idle interval depends on the core sleep states. The FF algorithm performs well at low utilisations. Indeed, it packs tasks on slower core types and thus allows faster core types to use efficient sleep states. However, as the system utilisation increases this effect diminishes as the system needs to accommodate the whole task-set. The performance of our algorithms improves with an increase in the system utilisation. The second phase compensates for the wrong task-to-core allocation done in the first phase up to some extent at low utilisations. LLED-SP improves the allocation of LLED in the best case up to 5%, while MM-SP improves over MM up to 2.5%. The second phase finds a good balance between the active energy of a core and the sleep states. Moreover, the results of LLED-SP dominates over MM-SP, while LLED performs better when compared to MM. The same experiment is repeated for 2 core types and the results are similar.

Figure 7.29 shows the normalised energy consumption of LLED-SP and MM-SP for different values of β. Similar to the previous case, the energy saving over FF increases with an increase in β. LLED-SP always dominates MM-SP in this scenario as well. The effect of different task-set sizes on the proposed algorithms is presented in Figure 7.30. The resulted behaviour is consistent with the previous scenario, that is, larger task-set sizes increase the gains of our algorithm.

Figure 7.31 presents the energy consumption of different algorithms on asimilar platform. It adapts the same setting as described in previous scenario. Similar to SBET scenario, the difference between the first and second phase is larger at low utilisations and this difference diminished

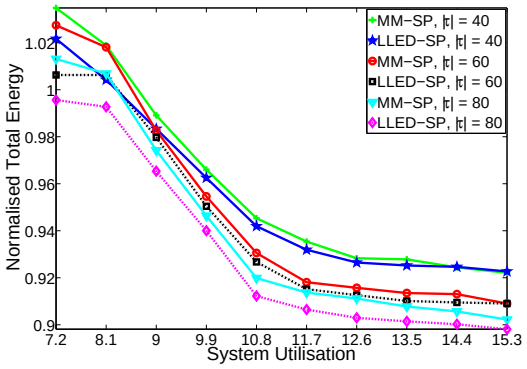


Figure 7.30: (LBET) Variation in $|\tau|$

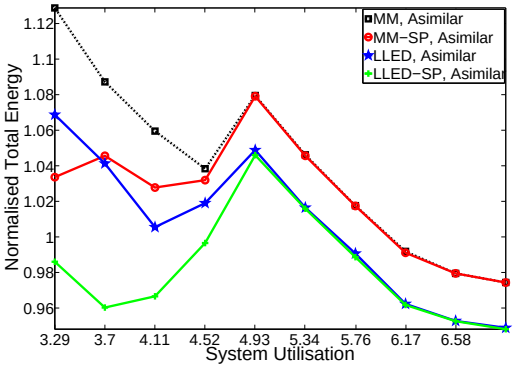


Figure 7.31: (LBET) Asimilar platform

at high utilisations. The wrong task-to-core allocations performed by LLED and MM at low utilisations are improved by applying the second phase. The effect of not packing the cores at low utilisations and allowing faster cores to use efficient sleeps is also evident in Figure 7.31. In general, LLED-SP has dominated with respect to energy saving in majority of the situations.

Finally, the execution time of LLED-SP to perform the task mapping for different task-set sizes at different utilisations are computed on a server with 8 Intel Xenon 1.60GHz processors with a memory size of 8GB. For task-set sizes of 100, 75 and 50, both phases take on average around 40, 20 and 8 seconds respectively. The execution time decreases with the increase in utilisation as the optimisation scope decreases.

Chapter 8

Conclusions, Perspective and Future Directions

Real-time systems have become a major part of the competitive embedded computing landscape. These systems interact with the environment and are deployed in various domains to increase automation, precision and efficiency. Apart from RT constraints, most of these systems are nomadic in nature and hence are battery powered or have limited/intermittent power supply. Energy efficiency is a prime design metric in such systems. In order to increase the energy efficiency, it is important to follow the current trends in power dissipation of modern embedded platforms. Following Moore's law, hardware vendors have doubled the number of transistors on a same die every two years and consequently, increased the computational capabilities of modern hardware platforms. However, the process of CMOS miniaturisation has raised several issues. One of them is an increase in leakage power dissipation. The leakage current in modern hardware platforms has increased to such an extent that the leakage power dissipation has started to dominate the dynamic power dissipation. This dissertation focused on system-level techniques that exploits different sleep states of the platform to reduce the leakage current of processors (both in unicore and multicore setting) and I/O devices.

8.1 Summary of the Work

8.1.1 Unicore Power Management

Unicore platforms are still an attractive choice in low-end RT systems due to their simple design process and high predictability when compared to multicore platforms. This thesis proposed several leakage-aware energy management techniques for single processor platforms. Procrastination is one of the famous leakage-aware scheduling algorithms in the literature to reduce the energy consumption. This method delays the execution of newly arrived jobs to prolong a sleep state of the processor. The proposed DBFP approach based on procrastination scheduling computes the optimal procrastination interval of the given task-set. It has been shown theoretically and experimentally that DBFP dominates over the best state-of-the-art procrastination algorithm SRA.

The simulations results with our experimental setup showed that the average sleep interval can be increased up to 75%, while reducible energy consumption (REC) can be raised up to 55%. The online complexity of DBFP is the same when compare to that of SRA. Applicability to the constrained deadline task model is an additional benefit of the proposed approach. However, procrastination scheduling needs an external hardware to implement its power saving algorithm. Such an external hardware obviously has its own energy cost and partly negates what these algorithms are aiming to achieve. To avoid the limitation of an external hardware in procrastination scheduling several alternative leakage-aware RTH scheduling algorithms (ERTH, IRTH and LWRTH) are proposed. The online complexity of these algorithms is low when compared to the related work while their energy savings are comparable to SRA. This work includes the discussion of the effect of sleep states on the number of pre-emptions. Our analysis showed that the side effect of using sleep states results in a reduced average number of pre-emptions. This in turn helps to achieve the goal of minimal energy consumption and improves on the required reservations.

The power density of modern processors has also increased with technology scaling. It demands efficient thermal management techniques to operate the system within given temperature limits. Furthermore, the leakage current increases exponentially with a rise in a system's temperature. TCDPM is a kind of DTM technique that exploits sleep states to cool down the system. In this thesis, it is shown that idealised DVFS and TCDPM are very similar in their treatment by RT algorithms, and with some minor modifications in the schedulability analysis and online mechanisms, the work done for DVFS algorithms can be applied to TCDPM to save energy. This strategy allowed to relax the assumptions commonly made in the literature (such as frame based RT system, single task, neglecting energy and temperature independent leakage power dissipation) of TCDPM and to apply it to a more generic RT task model using dynamic priorities. The proof of concept is shown with the help of a case study on a DVFS algorithm taken from the literature.

8.1.2 Device Power Management

The demand of extra functionality has increased the number of I/O devices. These I/O devices are usually connected externally and their energy consumption has become a considerable portion of the overall energy consumption. There are several techniques in the literature based on inter-task device scheduling. This thesis explored a new paradigm of intra-task device scheduling. The intra-task device scheduling strategy makes device wake-up calls on demand rather than keeping it unnecessary active throughout the execution time of a task using such a device. Initially, it considers a device model with a single sleep state and later extends it to multiple sleep-state devices. In order to compensate for the transition delays, it exploits execution and static slack explicitly, while the sporadic slack is utilised implicitly. The proposed approaches exploit with a low complexity the benefits of multiple sleep-state devices to save extra energy. Moreover, the intra-task device-scheduling strategy scales nicely with an increase in the number of devices (even with multiple sleep-state devices) due to its low complexity. It is not only complexity-wise efficient but the energy saving of the proposed algorithms outperforms the state-of-the-art when the average device usage time is less than 50% of tasks execution times. The extensive simulations

results demonstrate the effects of variations in different parameters. The presented strategy opens new opportunities for the efficient device-energy minimisation algorithms for future generations of embedded systems with a large number of I/O devices.

8.1.3 Multicore Power Management with Global Scheduling

The semiconductor industry has a paradigm shift from uniprocessor to multicore platforms as rising performance needs can no longer be sustained by increasing clock frequency. Among different types of multicore platforms, homogeneous multicore platforms are commonly deployed in RT applications. A number of scheduling algorithms are designed in literature to schedule given task-set on such platforms. Global scheduling has emerged as a potential candidate that provides flexibility in terms of scheduling solutions and eliminates the need to partition the given task-set among available cores. This thesis presented a leakage-aware energy management algorithm, called GPM, for a system using the GEDF scheduler on a homogeneous multicore platform. The proposed algorithm exploits the usable execution and idle slack to either transition a core into a sleep state, or prolong the sleep interval of the cores currently in sleep state. This is the first effort in the direction of leakage-aware energy management whilst tasks are scheduled by using GEDF. The effectiveness of the proposed algorithm is demonstrated through exhaustive evaluation results on a simulator modelled after a Freescale PowerQUICC III based multicore platform. Those simulations showed that the energy saving offered by the GPM algorithm can compete with the conservative lower-bound on the energy consumption, i.e., the OverOptimal algorithm.

8.1.4 Partitioned Multicore Power Management

Heterogeneous multicore platforms are becoming popular in industry to deploy complex applications. It has the flexibility to perform specific tasks well and cheap. Normally, the given task-set is partitioned among different types of cores on heterogeneous platform. Tasks assigned in the allocation phase on each core are scheduled through any uniprocessor scheduling algorithm. The problem of task-to-core mapping is solved through energy-aware allocation algorithms with an objective to reduce the average-case energy consumption of the system, while satisfying RT constraints. The proposed allocation heuristics consider both dynamic and leakage energy consumption into account while performing the tasks assignment and are based on a more realistic power model than used in the state of the art. Due to the complex nature of the problem, the proposed heuristics are divided into two phases. In the first phase, tasks are assigned to their favourite core with an objective to reduce their individual dynamic power dissipation. In the second phase, the increased dynamic power dissipation of tasks are traded with an improved sleep states on cores to reduce the overall energy consumption of the multicore platform. Initially, a system model considers a platform without DVFS capabilities and later extends it to a more general system model that can be employed on heterogeneous platforms with DVFS capabilities. One interesting observation of this work is that a task assignment to its favourite core type is not always beneficial. Hence, the leakage power dissipation has a major role in allocation to reduce the overall energy consumption.

8.2 Limitations and Future Directions

8.2.1 Dependent Task Model

Many applications in the RT domain are modelled as dependent task models with precedence and communication constraints. The research presented in this thesis is limited to the independent task model. One interesting direction to follow is an extension of the proposed leakage-aware scheduling algorithms to a communicating task-model. Optimal leakage-aware procrastination scheduling for a task-set with precedence constraint may be the first step to solve this issue, which can be later extended to avoid the external hardware to implement such an algorithm. The procrastination interval of tasks with precedence constraints will be highly affected by their preceding tasks and this cascading effect will propagate from a source to an output node. I believe a dependent task model has a high potential to save energy as most of the tasks are waiting for their inputs due to precedence constraints.

This problem is much more interesting in a setting of a multicore platform. Task-to-core mapping of a task-set with precedence constraints with an aim to reduce the energy consumption is an important and interesting problem to tackle. The task allocation process has to consider three main factors.

- Energy consumption of a task on an individual core.
- Its effect on the overall energy consumption of a core combined with other allocated tasks to such core.
- Communication overhead generated by such an allocation due to its precedence constraint.

In homogeneous multicore platforms, the first factor can be ignored as a task will consume same amount of energy on any core and the allocation process has to only deal with other two factors. The same problem is acerbated in heterogeneous multicore platforms that has to consider all factors. I believe a trade-off between the communication overhead and the effect of the allocation of a task on its energy consumption will result in reduction of the overall energy consumption.

8.2.2 Device Power Management

The proposed work on intra-task device scheduling is very limited and there exists a large potential to extend the current findings. There are several assumptions made in this work: a) a task has exclusive access to a device, b) a device is used once in the execution time of a job and c) a task can access only one device. A task may need more than one device and has the requirement to access them more than once. Similarly, it is also very common that tasks share devices among each other. Relaxing these assumptions may not be trivial but an interesting problem to solve. The system designer has the objective to reduce the overall energy consumption of the embedded system considering CPU as well as devices. A power management algorithm that considers the

energy consumption of both factors is outstanding. The proposed uniprocessor power saving algorithms exploit the available slack in the system to initiate and prolong the sleep states. Similarly, intra-task device scheduling algorithms utilise the available slack in the system to compensate the transition delays of on demand device wake-up calls. One of the interesting problems is to consciously distribute such slack among devices and CPU such that it minimises the overall energy consumption of the system. One of the very trivial and conservative method is to allocate different slack types (execution slack or static slack) to CPU or devices. Nevertheless, a global slack distributor can also be proposed to allocate the slack appropriately on demand to both CPU and devices. Another interesting direction is to consider the energy consumption of devices in a multicore context, as there exists a very limited work in this direction.

8.2.3 Multicore Power Management

The power model of the homogeneous multicore platform used in this thesis assumes that a core can transition into any of its sleep state independent of the state of other cores. In some platforms, this assumption is not valid and cores cannot transition into any arbitrary sleep states. A core can only enter a sleep state $\S_{n+1}$, if other cores are in a sleep state $\S_n$. Furthermore, once all cores transition into their deeper sleep state, other components of the platform not specific to any core such as shared caches, buses, frequency generator, power regulators etc, can be turned off as well. These additional components also have a set of sleep states to select depending on the following idle interval. The proposed power management algorithm for the homogeneous multicore platform can be extended to this more realistic power model. The major challenge in this direction is to align slack periods on cores to achieve deeper sleep states.

As discussed in Chapter 1 the big.LITTLE processing [ARMa] — a type of uniform multicore platform — has a high potential to save energy. The proposed allocation heuristic in Chapter 7 can be extended to an online power management algorithm that migrates jobs to a LITTLE processor based on the available slack in the system and/or remaining execution requirements. One possible way is to provide an online test that ensures that all tasks on a LITTLE processor can be co-scheduled with the remaining workload of a big processor. Obviously, the slack available on the LITTLE processor can be exploited for this purpose. This analyses can be further extended to incorporate the effect of caches that mainly depends on the architecture of the cache hierarchy. The migration of data intensive jobs should be avoided to minimise their data transfer overhead.

8.2.4 Massive Multicore Power Management

Massive-multicore or many-core hardware platforms are emerging in the industry due to their immense computational capabilities and flexibility to scale. These hardware platforms are composed of tiles, main memory and interconnect network. A tile contains a core, private cache and network switch. All tiles are interconnected with their neighbours through their network switch. The network-on-chip communication framework [BDM02] provides the connection among cores and the main memory. One of main reasons of this paradigm shift in the bus design is the contention

and the access time on shared bus architecture. Moreover, in such platforms, it is very expensive to provide voltage regulator for each core, and therefore, they are divided into voltage islands. All cores inside the voltage island operate on the same frequency and can transition into a low power sleep state to save energy.

One interesting and complex problem is to propose an online algorithm that either migrates the workload dynamically to transition a voltage island into a low power state or reduces its frequency to save energy, while satisfying RT constraints. To perform a migration, the scheduler needs to first account for the available slack in other voltage islands. Given the available slack is sufficient enough to migrate all tasks from one specific voltage island to other voltage islands, the scheduler needs to ensure temporal correctness of the system after including the cost of migration. The cost of migration includes several factors. Firstly, data related to migrant tasks in their local caches should be migrated to their destination cores which in turn generates traffic on the network. Secondly, memory request time of such migrating tasks also changes, and hence, its schedulability on the new core should be tested as well. Thirdly, the effect of this migration on the traffic of memory requests on other cores should be considered as well. A migration in such an online algorithm is only performed when it saves energy consumption after considering the overhead of all these factors. Otherwise, it can decrease the operating frequency of individual voltage islands to exploit the available slack.

Energy-aware allocation of tasks on such platforms not only depends on the energy consumption of individual cores, but also on the communication overhead between cores and the main memory. The nature of a task plays an important role in the allocation process on massive multicore platforms. A CPU intensive task placed away from the memory does not affect the communication overhead, however, the placement of a memory intensive task will influence the communication overhead heavily. Therefore, an allocation process should consider the number of requests made by a task while performing the allocation. Memory intensive tasks should be placed close to the main memory. Moreover, the assignment heuristic must consider the fact that placing CPU intensive tasks close to each other may create hotspots that should be avoided in such platform.

The state-of-the-art dealing with power management on massive multicore platforms with RT constraints is very limited. The existing solutions such as the one presented by Sayuti et al. [MSIGO13] rely on meta-heuristics to perform allocations but I believe constructive heuristics can be developed to obtain solutions fast. The key to constructive heuristic is to develop a density function that computes the effect of allocation on network links.

8.3 End Note

The research performed in this thesis has shown that energy consumption of embedded systems can be reduced with low complexity through the use of efficient sleep states for different kinds of hardware platforms (e.g., unicore, homogeneous multicore and heterogeneous multicore). Temporal isolation demanded in current RT systems between applications of different characteristics (HRT, SRT or BE) is also integrated in the proposed solutions.

Appendix A

Evaluation of CPU Power Management Algorithms

In this appendix, initially, the complexity comparison of all the algorithms is presented and then the extensive simulation results compare the proposed algorithms against the state-of-the-art on different parameters.

A.1 Overhead Analysis

The complexity of the proposed algorithms is compared with LC-EDF, PROC and SRA, as they are with their use of dynamic priorities closest to this work.

A.1.1 Complexity of LC-EDF

As it has been discussed in Section 4.1.1, all these algorithms (LC-EDF, PROC and SRA) initiate a sleep state in the idle mode. The LC-EDF algorithm has a smaller number of sleep states when compared to EDF as it combines several small idle intervals to initiate a sleep state for a long period of time. While in the sleep state, on each higher priority (shorter deadline) task arrival, the LC-EDF algorithm recomputes the new procrastination interval for that task, unless the schedule does not allow further procrastination. The online overhead of the LC-EDF algorithm depends on two main factors, 1) Number of times a sleep state is initiated, 2) The overhead of each sleep transition. The first factor depends on the total number of idle intervals in the schedule as LC-EDF initiates a sleep in idle mode. However, the overhead of each sleep transition depends on the task-set size. The complexity of each sleep transition in LC-EDF is $O(\ell^2)$.

A.1.2 Complexity of PROC and DBFP

The PROC method has an offline complexity of $O(\ell^2)$. The DBFP approach has an offline complexity of $O(\ell \times x)$, where $x = \sum_{\forall \tau_i \in \tau} \frac{\mathrm{H}}{T_i}$ is the number of jobs in the hyper-period H. The online complexity of the DBFP approach and the PROC method is the same and equals to $O(\ell)$. The

SRA algorithm [JG05] reclaims the execution slack from the schedule and uses it to further procrastinate the sleep interval. In a nutshell, on every release of a task during the sleep interval, the scheduler computes the available execution slack and compares it with the offline computed procrastination interval of that task. The maximum of these two values is considered while deciding on the reinitialisation of the timer. DBFP or PROC can be used in the offline phase of the SRA algorithm to compute the procrastination interval. The complexity to determine the available execution slack for a task is $O(\ell)$. As both PROC and DBFP has an online complexity of $O(\ell)$, therefore, combined with slack reclamation, the online complexity of the SRA algorithm is same as LC-EDF i.e., $O(\ell^2)$.

A.1.3 Complexity of ERTH

The alternative race-to-halt algorithms do not require any external hardware. The χ_{min} is used in all alternative race-to-halt algorithms and the offline complexity of its computation is same as presented for DBFP. The online complexity of ERTH can be divided into three different categories based on its three different principles.

- Firstly, if the sleep transition is initiated through principle 1, it requires just one comparison against the offline computed static sleep interval χ_{min}, i.e., $O(1)$.
- Secondly, a sleep states initiated with principle 2 require the computation of φ in order to obtain the maximum feasible sleep interval. The major overhead lies in the computation of ρ that could be obtained either offline or online. Offline Method: The interval for computing ρ offline is no more than the longest T_i in the task-set. Therefore, the maximum available gap can be computed offline for each deadline and sorted in an increasing order by time. The runtime overhead is to search the sorted array of maximum available gaps for each given interval, which can be done in $O(ln(p))$, where p is the number of intervals. Online Method: The online complexity to compute ρ depends on the number of jobs in an interval. The former method is used to compute ρ.
- Thirdly, in idle mode (principle 3), sleep state is initiated for χ_{min} interval without any check. Thus, sleep states initiated in idle mode do not have any online overhead.

Apart from its low complexity, the second advantage of ERTH is the existence of the fixed sleep-interval at the sleep-state initialisation instant. Once the processor initiates the sleep transition, no matter how many tasks arrive during the sleep mode, it will wake up after a defined limit (when the timer expires). The presented schedulability tests ensure that all jobs will meet their deadlines. This mechanism simplifies the system implementation and eliminates a need for external hardware to run the algorithm. Which in turn further reduce the complexity of the design, as external hardware requires extra communication overhead and increases integration issues.

A.1.4 Complexity of IRTH

The online overhead of IRTH is similarly divided into three categories.

- If the sleep state is initiated by a RT task (principle 1), its overhead is same as in ERTH principle 1, i.e., $O(1)$.

- In idle mode (principle 3), its complexity increases, as the algorithm has to search for the earliest possible future release in an array of γ. There are two ways to manage it. Firstly, a sorted array of γ can be stored and its first value can be used when the processor initiates a sleep transition. Thus the complexity of maintaining the array on each job arrival is $O(ln(\ell))$. However, when the processor initiates a sleep the overhead is low i.e., $O(1)$. Secondly, γ can be stored with respect to the task-ID and on each sleep invocation the algorithm traverses γ to find the minimum value. In this case complexity to update an array of γ on each job invocation is $O(1)$, however, each sleep transition has a complexity of $O(\ell)$. It is observed that the number of sleep transitions are fewer when compared to the number of jobs invocations. Therefore, the second approach is used. Thus the complexity of each sleep transition in IRTH through principle 3 is $O(\ell)$.

- The principle 2 of IRTH exploits the future release information (γ). Therefore, it is difficult to find the sleep interval offline, and hence, estimated online on each sleep invocation. To compute the complexity of a sleep transition in principle 2, it is assumed $\Theta = \frac{T_{max}}{T_{min}}$, where T_{max} is the maximum and T_{min} is the minimum inter-arrival time in the task-set. Then the complexity of each sleep transition in principle 2 is $O(\Theta \times \ell)$, as in worst-case the scheduler has to check the all possible job releases within T_{max}.

A.1.5 Complexity of LWRTH

The online complexity LWRTH is low when compared to IRTH. LWRTH only initiates a sleep state transition in idle mode. It relies on future release information array to maximise the energy efficiency. Similar to IRTH, tasks are stored with respect to their ID's and on each sleep invocation the algorithm traverses γ. Therefore, each sleep transition happening in LWRTH has a complexity of $O(\ell)$. This algorithm does not need any slack management algorithm, and moreover, its online complexity to initiate a sleep transition is also low when compared to ERTH and IRTH.

Finally, a system designer needs to perform a careful evaluation, while selecting among the available algorithms. IRTH clearly has the highest complexity when compared to ERTH and LWRTH but provides the best energy efficiency among them. The complexity comparison of ERTH and LWRTH is difficult. On one side, ERTH does not require to maintain a list of future release information, while LWRTH requires information which needs to be updated on every task's release. On the other hand, LWRTH has lower sleep transition overhead when compared to ERTH and does not exploit the execution slack generated from the slack management algorithm.

Parameters	Values
Task-set sizes $\|\tau\| \in$	$\{10, 20, \ldots, \underline{50}, \ldots, 100\}$
$T_{min} \in$	$\{\underline{30}, 40, \ldots, 100\}$
PUB $\in$	$\{1.1, 1.2, \ldots, \underline{1.5}, \ldots, 5\}$
BCET limit $C^b \in$	$\{0.2, 0.25, \ldots, \underline{1}\}$
Sporadic delay limit $\Gamma \in$	$\{\underline{0}, 0.05, \ldots, 1\}$

Table A.1: Overview of simulator parameters used to evaluate demand bound function based procrastination

No.	Power Mode	tr_n (μs)	bet_n (μs)	P_n ($Watts$)	Es_n ($\mu Joules$)
1.	Doze	5	225	3.7	42
2.	Nap	100	450	2.6	950
3.	Sleep	200	800	2.2	1980
4.	Deep Sleep	500	1400	0.6	5750
5.	Typical	0	0	4.7	0
6.	Maximum	-	-	12.1	-

Table A.2: Different sleep states parameters

A.2 Simulation Results of the DBFP Algorithm

A.2.1 Experimental Setup

The discrete event simulator SPARTS (simulator for power aware and real-Time systems) [NAP11a, NAP11b] discussed in Section 3.2 is used to evaluate the effectiveness of the DBFP approach. SPARTS is used with the parameters mentioned in Table A.1, where underlined values are the default values if not mentioned otherwise in the description of the experiment. The parameters C^b and Γ are used to generate wide variety of different tasks and their subsequent varying jobs. The periods of both BE and RT tasks are chosen from an interval, $T_{min}[1, \text{PUB}]$, where T_{min} is the lower bound and PUB (Period Upper Bound) is the variable used to define the upper bound on the interval. Each task-set with different parameters mentioned in Table A.1 is simulated for 100 times with different seed values to the random number generator and averaged. The simulation time of each task-set is 100 seconds.

The SRA algorithm [JG05] is an energy saving approach that takes procrastination intervals of the tasks determined through Jejurikar's method as an input. For a fair comparison, the same algorithm is used by just replacing the input phase with DBFP determined procrastination intervals. For simplicity sake, it is assumed that all the slack in the schedule (spare capacity) is reserved for the shut-down of the processor. Both variations of SRA are implemented in SPARTS and their sleep state is selected offline based on their respective minimum idle interval. It has already been shown in the state-of-the-art that SRA performs better than LC-EDF, hence, this section only considers SRA for the comparison.

The power model used for simulations is based on the Freescale PowerQUICC III Integrated Communications Processor MPC8536 [Sem]. The power dissipation values are taken from its data

sheet for different modes (Maximum, Typical, Doze, Nap, Sleep, Deep Sleep). The core frequency of 1500 MHz and core voltage of 1.1 V is used for all the experiments. The transition overheads are not mentioned in their data sheet, therefore, assumed values are used for four different sleep states. The transition overhead of the typical mode that corresponds to the idle state in our system model is considered negligible. The power values given in Table A.2 sum up core power and platform power dissipation. More details are available in the reference manual [Sem].

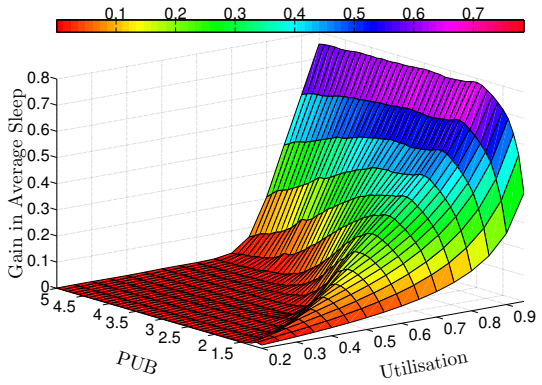


Figure A.1: Variation in T_{max} (sleep interval)

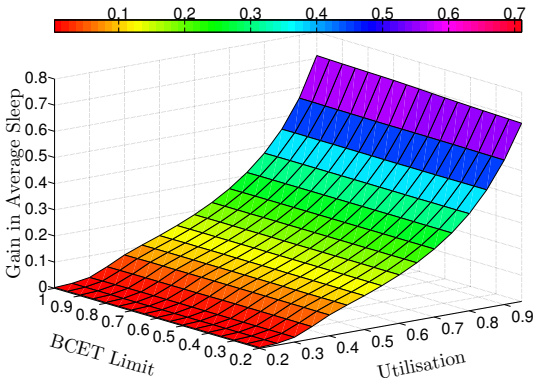


Figure A.2: Variation in C^b (sleep interval)

A.2.2 Analysing Average Sleep Interval

Figure A.1 presents the gain of DBFP over SRA with respect to average sleep interval for different values of U and PUB. The average sleep interval is computed by accumulating the idle time in the scheduling and dividing it by the number of sleep intervals. The gain of DBFP increases with an increase in system utilisation. Furthermore, the gain also increases by widening the interval to select T_i of the tasks. At low utilisation DBFP and SRA have enough slack to initiate longer sleep intervals. However, with an increase in system utilisation, the slack in the system decreases, and the procrastination intervals lengths have a high impact on the sleep intervals. Another reason for a high gain at high utilisation is the difference of minimum idle interval between SRA and DBFP. It has been shown in Lemma 24 that $\chi_{min} \geq Z_{min}$. Therefore, SRA starts to lose efficient sleep states at high utilisation, causing its frequent switching. In the best case, increase in the average sleep interval is approximately 75%.

The gain in average sleep interval is also computed by varying the utilisation against the BCET Limit C^b as shown in Figure A.2. Mostly, the gain occurs due to an increase in system utilisation, while the variation in C^b has a minute effect at a very high utilisation of 0.95. As both algorithms use the same mechanism to manage the slack, the difference is negligible. The change in sporadic delay limit Γ has been investigated in the experiments against different values of U. The effect of Γ is negligible as well. The variation in task-set size is demonstrated in Figure A.3 against different values of U. In the best case (i.e., $|\tau| = 100$), the gain reaches 75%. It is evident that the increase in task-set size increases the gain in average sleep intervals. This can be explained as follows. The procrastination interval of a high priority task is always bounded by the low priority

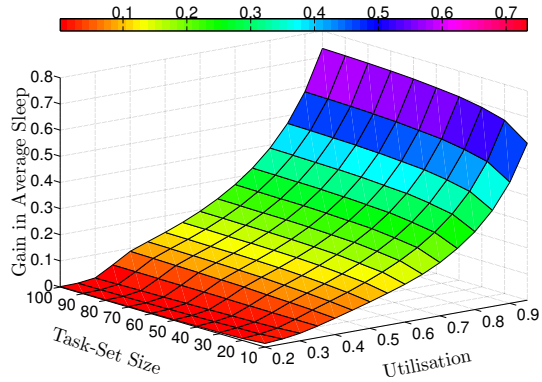


Figure A.3: Variation in $|\tau|$ (sleep interval)

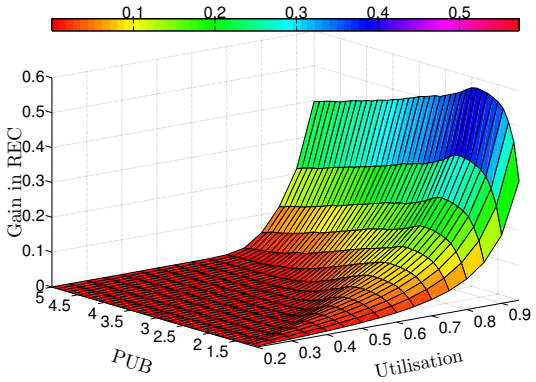


Figure A.4: Variation in T_{max} (REC)

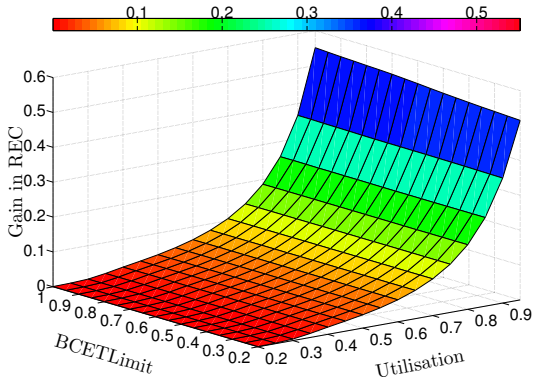


Figure A.5: Variation in C^b (REC)

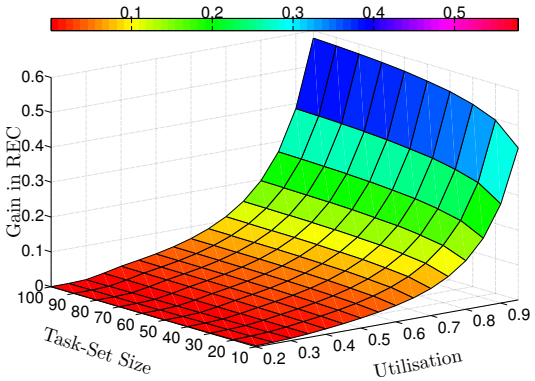


Figure A.6: Variation in $|\tau|$ (REC)

tasks in the given task-set. The difference between the procrastination intervals of different tasks between DBFP and SRA has a cascading effect. For instance, a low priority task τ_i having a procrastination interval Z_i smaller than that of a high priority task will have its Z_i scaled down due to Equation 4.4. If $Z_i < \chi_i$, then not only the difference exists at level τ_i but also $\forall \tau_k : k < i$. A large task-set has high probability to get this cascading effect.

A.2.3 Analysing Reducible Energy Consumption

The active energy consumption of the processor is the same in SRA and DBFP as only a single active state is assumed in this work. The difference comes in the energy consumption in idle intervals and termed as reducible energy consumption (REC). The gain of DBFP over SRA with respect to REC is compared for different parameters against system utilisation as demonstrated in Figure A.4, Figure A.5 and Figure A.6. In the best case, the gain in REC is approximately 55%. All the graphs have similar trends as explained in the description of their corresponding results with average sleep intervals.

Parameters	Values
Task-set sizes $\|\tau\|$	$\{10, 50, 200\}$
Share of RT/BE tasks $\xi = \{\xi_1, \xi_2\}$	$\{\langle 40\%, 60\% \rangle, \langle 60\%, 40\% \rangle\}$
Inter-arrival time T_i for RT tasks	$[30ms, 50ms]$
Inter-arrival time T_i for BE tasks	$[50ms, 1sec]$
Sporadic delay limit $\Gamma \in$	$\{0.1, 0.2\}$
BCET limit C^b	0.2
Sleep threshold Ψ_x in	$\{\underline{1}, 2, 5, 10, 20\}$

Table A.3: Overview of simulator parameters used to evaluate alternative race-to-halt algorithms

A.3 Simulation Results of ERTH, IRTH and LWRTH Algorithms

A.3.1 Experimental Setup

The proposed alternative race-to-halt algorithms (ERTH, IRTH, LWRTH) are implemented in SPARTS and compared against the state-of-the-art (SRA and LC-EDF). The LC-EDF algorithm is included in this comparison as it has some interesting properties. The SPARTS simulator is used with the parameters specified in Table A.3. Though not a fundamental requirement of the proposed algorithms, implicit deadlines $D_i = T_i$ are assumed for evaluation purposes. It is obvious that $D_i > T_i$ leads to greater saving opportunities, but does not provide greater insights. Overall system utilisation is varied from 0.2 to 1 with an increment of 0.05. In total, 1020 different task-sets configurations ($C^b, \Gamma, U_i, \cdots$ etc) are generated. The same power model (based on the Freescale PowerQUICC III Integrated Communications Processor MPC8536 [Sem]) specified in Table A.2 is used in these simulations.

A vast variety of CPUs are available in the market. They have diverse hardware architectures and consequently different power characteristics. In order to observe the effect of different types of hardware platforms on the proposed alternative race-to-halt algorithms, different power parameters of the processor are generated. In the system model, active and idle time of the CPU remain constant for a specific task-set. As the total energy consumption is normed, the factor among the power model parameters that affects the energy gain of an algorithm is the overhead of the sleep transitions. However, the overhead of the sleep transition is modelled by the break-even-time of the sleep state. Therefore, the power model parameters are altered to generate a distinct BET such that it is a multiple of the original BET by a factor of x. The different break-even-times are represented with Ψ_x called sleep threshold (Table A.3). The sleep threshold with a value of $x = 1$ denotes the BET of the original power model.

The overhead of all the algorithms including procrastination algorithms (LC-EDF and SRA) is considered negligible. This is obviously a favourable treatment for LC-EDF and SRA, as the time/energy overhead of the external specialised hardware is substantial. The SPARTS simulator takes into account the effect of the sleep state transition delays and its energy/time overhead is included in the power model. Any change in the parameters from those described above in Table A.3 is explicitly mentioned in the individual experiment description. Each point in the figure

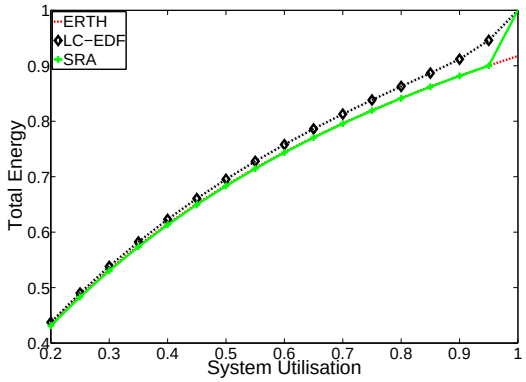


Figure A.7: Normalised total energy consumption (ξ_1 and $|\tau| = 200$)

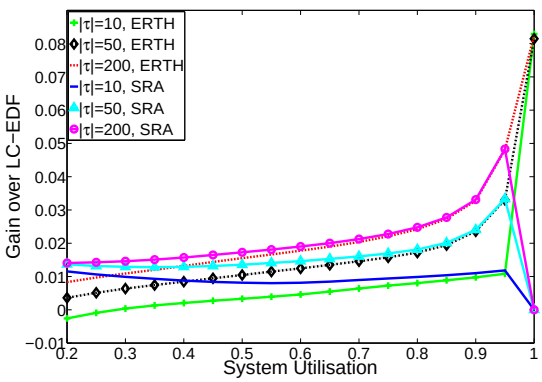


Figure A.8: Gain of ERTH and SRA over LC-EDF for different task-set sizes

present results averaged over 100 runs with different respective seed values as well as all different free parameters. As a baseline, ERTH is simulated without the use of sleep states (NS), i.e., processor uses typical power when it is not executing any task otherwise consume P_A during normal execution of tasks. All the results are normalised to the corresponding results of NS. Energy consumption of LC-EDF or SRA without using sleep state is identical to NS as the overall idle and active execution time remains same in both cases.

The RBED framework is used for the integration of applications with different criticality levels. The RBED framework allows an overrunning job to borrow from its future invocations [LB05]. Therefore, two different scenarios are considered. In scenario 1, it is assumed that $A_i = C_i$ for both task classes (RT and BE task). Moreover, $\Gamma_{0.1}$ is assumed for all experiments in scenario 1, as the difference is marginal when compared to $\Gamma_{0.2}$. Nevertheless, the effect of variation in Γ is explained later in scenario 2. In scenario 2, it is assumed, BE tasks often overrun beyond their allocated periodic budget A_i. The mean of the BE tasks actual-execution-time distribution is set to 85% of A_i in this scenario. As noted, to ensure timely completion of RT tasks, $A_i = C_i$. The borrowing mechanism [LB05] is also integrated in scenario 2 so that BE tasks can use their future budgets, if required.

A.3.2 Scenario 1 ($A_i = C_i$, $\forall$ task types)

A.3.2.1 Analysing Total Energy Consumption

The minimum sleep threshold value Ψ is set to 1 for the next six experiments. The total energy consumption of ERTH is compared against LC-EDF and SRA for a task-set size of 200 and a task distribution of ξ_1 in Figure A.7. All values are normalised to the corresponding values of NS. As it is evident, SRA performs comparable to ERTH except at high utilisations. Moreover, ERTH outperforms LC-EDF for all, but particularly for higher utilisations. With an increase in system utilisation, the maximum feasible idle interval (procrastination interval) computed by the LC-EDF algorithm shrinks. The given system model assumes multiple sleep states, while LC-EDF cannot

use more energy efficient sleep states with corresponding higher overhead bet_n since it will risk system schedulability.

The SRA algorithm saves more energy when compare to LC-EDF. Firstly, the procrastination interval computed for each task in SRA is greater than or equal to the procrastination interval determined by the LC-EDF algorithm. This increase in procrastination interval over LC-EDF enables SRA to select a more efficient sleep state offline while ensuring system schedulability. Secondly, it also benefits from the execution slack reclaimed online. On the other hand, the efficient slack management algorithm described in Section 3.1.5 also enables ERTH to accumulate the slack S_e and still use more efficient sleep states at high utilisation. Consequently, at high utilisations (especially at $U = 1$), the savings of ERTH are still larger when compared to SRA. This is motivated by the following observations. As already mentioned in the experimental setup, the resulting utilisation is less than the target utilisation by a very small factor of ε due to numerical rounding of the parameters used to generate a task-set. The secondary effect is the diversity in periods of task-set that rarely aligns and as a result the hyper-period of the given task-set is very long. Therefore, at high utilisations, the use of the demand bound function yields an actually usable χ_{min} due to the disparity of periods and deadlines. If one uses the utilisation based approach SRA, analytically this leads invariably to small intervals, due to the loss of accuracy when abstracting the workload through its worst-case utilisation. An example of this is reflected in the proof to Lemma 24.

At $U = 1$, ERTH creates idle intervals to save energy by exploiting the execution slack in the system. For a distribution of ξ_2, the processor consumes approximately 1% more energy when compared to ξ_1. In ERTH, it is due to the lesser usage of principle 2, as the system has fewer BE tasks in ξ_2. The LC-EDF and SRA algorithms depend on the period of the tasks. Extra tasks with long periods resulting in greater opportunities to save energy, therefore, ξ_2 consumes slightly more energy when compared to ξ_1.

A.3.2.2 Effect of Task-set Size on LC-EDF

An interesting observation may be noticed in the total energy consumption of LC-EDF: fine-grained large task-sets consume more energy when compared to the coarse-grained small task-sets at the same utilisation. Hence, LC-EDF is susceptible to the changes in the task-set size. The figure is not shown here but its main features are described in detail. The variation in the total energy consumption is up to 4% at high utilisations. The major reason of this variation in total energy consumption lies in the algorithm itself, which computes the procrastination interval. The procrastination interval is recomputed on every arrival of a job with deadline shorter than any of the currently delayed jobs. An increase in the task-set size means a higher probability of recomputing the procrastination interval. Each re-computation includes a nominal shortening of the procrastination interval and increasing the virtual utilisation in the process. Therefore, the energy consumption marginally increases with an increase in task-set size. The effect of task-set variation is also analysed for ERTH, IRTH and LWRTH. Oppose to LC-EDF, the task-set variation does not affect the total energy consumption of the processor with either of them. Consequently, ERTH and its variants are more robust, when it comes to task-set size variations. The task-set

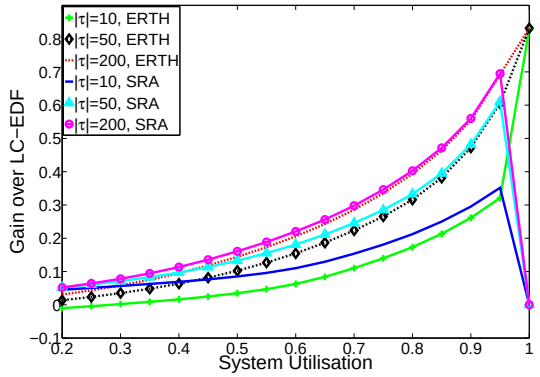


Figure A.9: Gain of ERTH and SRA over LC-EDF in idle interval (ξ_1)

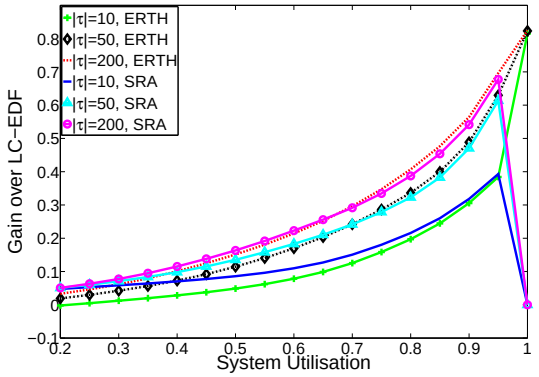


Figure A.10: Gain of ERTH and SRA over LC-EDF in idle interval (ξ_2)

size variation has negligible effect on the energy consumption of SRA, as it pre-computes the procrastination interval offline and exploits execution slack.

A.3.2.3 Analysing Overall Gain

The overall-gain of ERTH and SRA over LC-EDF for three different task-set sizes with a distribution of ξ_1 is depicted in Figure A.8. The formula used to compute the overall-again is $\frac{E_{\text{LC-EDF}} - E_x}{E_{\text{LC-EDF}}}$, where $E_{\text{LC-EDF}}$ is the total energy consumption of LC-EDF and E_x corresponds to the total energy consumption of SRA or ERTH. It is evident ERTH saves more energy compared to LC-EDF for larger task-set sizes. This happens due to dependency of LC-EDF on the task-set size as described above. However, SRA saves approximately 1% more energy when compared to ERTH at low utilisations. Its performance degrades towards high utilisations and the difference between SRA and ERTH slowly vanishes. The ERTH algorithm manages to save this energy without the support of extra hardware. If the energy consumption of the external hardware is more than 1% of the saving, then ERTH is still a better approach in terms of energy saving due to lower complexity when compared to SRA. The difference between two distributions ξ_1 and ξ_2 is small. Nevertheless, for all three different task-set sizes, the gain of ξ_2 dominates ξ_1 due to an increase in RT and decrease in BE tasks. One oddity exits at $U \leq 0.2$ for $|\tau| = 10$ as LC-EDF performs slightly better when compared to ERTH, but that difference is negligible. At such a low utilisation, the processor is consuming very little energy in any case. Note: In this case the overhead of the external hardware would be more pronounced.

A.3.2.4 Analysing Gain in Idle Interval

The amount of execution performed by ERTH, SRA and LC-EDF is the same. Hence, the only difference arises from the difference of energy consumption in the idle intervals of the schedule. Apart from the overall-gain that is shown in the Figure A.8, the gain of ERTH and SRA over LC-EDF is analysed only in the idle intervals. Simulation results are presented in Figure A.9 and Figure A.10 for two different distributions of ξ_1 and ξ_2 respectively. At high utilisations, the

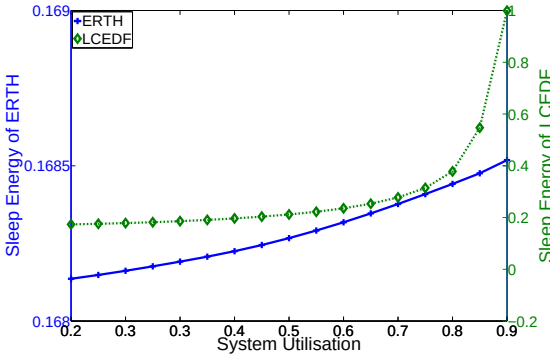


Figure A.11: Normalised sleep energy consumption (ξ_1 and $|\tau| = 200$)

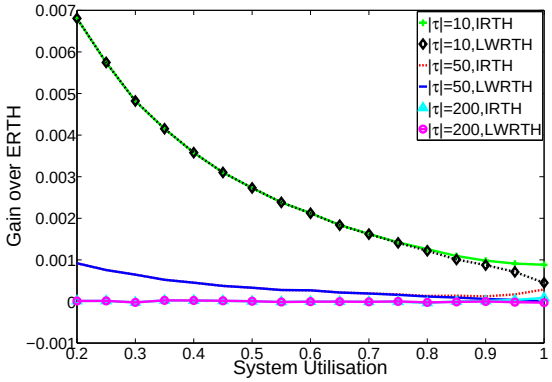


Figure A.12: Overall-gain of IRTH and LWRTH over ERTH (ξ_1)

gain shown in Figure A.9 is about a factor of 10 higher than can be seen in Figure A.8. Another important observation is that the gain of the SRA algorithm is smaller with ξ_2 when compared to ξ_1. At high utilisation, even the energy consumption of ERTH is reduced when compared to SRA. Hence, ERTH is favourable at high utilisations for a task-set containing small number of BE tasks.

The processor consumes typical power (idle power) only in one scenario, i.e., when the available interval is not feasible to initiate a sleep state due to either break-even-time limitation or when the scheduler cannot guarantee the real-time constraint. Otherwise the used energy depends on the selected sleep state. Figure A.11 compares ERTH with LC-EDF in terms of the normalised energy consumption in the idle interval. The idle energy consumption of ERTH and LC-EDF is normalised to the corresponding idle energy consumption of NS algorithm. The result indicates that LC-EDF performance degrades with an increase in system utilisation. LC-EDF selects the single most efficient sleep state among the set of available sleep states based on its maximally-feasible idle interval. As the system utilisation increases, the length of maximally-feasible idle interval shrinks. Consequently, LC-EDF cannot select the more efficient sleep states due to their higher transition delay bet_n which in turn leads to increased energy consumption. At $U = 1$, LC-EDF behaves similar to a system that is not using sleep states. Opposed to this ERTH can collate the available slack in the system, shows a smooth behaviour for all utilisations. The SRA algorithm behaves similar to ERTH from $U = 0.2$ to $U = 0.9$ and afterwards it follows LC-EDF.

A.3.2.5 Effect of Improved Slack Management Algorithm

The disadvantage of the proposed slack management algorithm is the poor slack distribution. The improved slack management approach used in the SRA algorithm is also integrated with the ERTH algorithm for the fair comparison. The gain of the ERTH algorithm with improved slack management over ERTH with simplistic slack management approach proposed in this work in the current experimental set-up is negligible. The reason behind such a behaviour is the fact that better slack distribution plays an important role for DVFS based algorithms, where the slack distribution among different tasks is important. However, when it comes to race-to-halt algorithms, slack

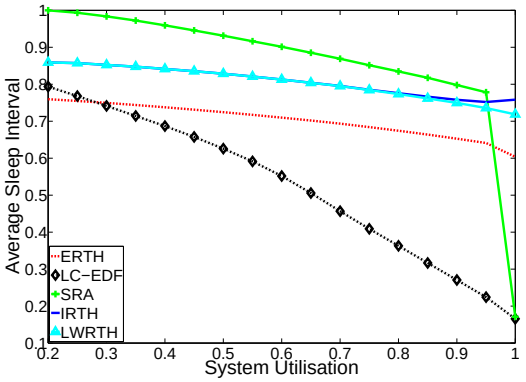


Figure A.13: Normalised average sleep interval ($|\tau| = 10$ and ξ_1)

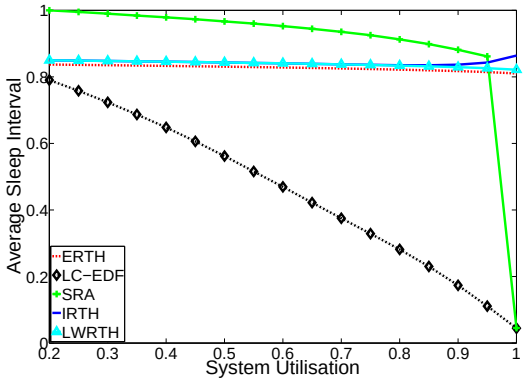


Figure A.14: Normalised average sleep interval ($|\tau| = 50$ and ξ_1)

accumulation is important than better slack distribution.

A.3.2.6 Overall-gain of IRTH and LWRTH

IRTH and LWRTH target the pessimism introduced in ERTH at the cost of extra overhead. In order to quantify their effectiveness, the overall-gain of IRTH and LWRTH over ERTH is analysed. The corresponding results are illustrated in Figure A.12 with a ξ_1 and $\Gamma_{0.1}$. Four important observations are evident from the results. Firstly, the gain decreases with an increase in task-set size. IRTH and LWRTH reduce the pessimism by utilising their past information to predict the future. The goal of these algorithms is to extend the sleep duration. Intuitively, one can argue with an increase in task-set size, future release information predicted is less helpful to extend the sleep interval. For a task-set size of 200, both IRTH and LWRTH behave very similar to ERTH. Secondly, with an increase in system utilisation, especially for a task-set sizes of 10 and 50, the overall gain decreases. The increase of system utilisation decreases the idle interval in the schedule and pushes the releases closer to each other. Hence, future release information becomes less important. Third observation is the difference between IRTH and LWRTH. IRTH exploits the execution slack explicitly in the system thus behaves superior to LWRTH at higher utilisation. Finally, the gains are moderate over ERTH but worthwhile in mobile systems.

A.3.2.7 Analysing Average Sleep Interval

The average sleep-interval of all the algorithms is determined by dividing the total sleep duration over the number of sleep transitions. Figure A.13 and Figure A.14 present the average sleep-interval against utilisation with a distribution of ξ_1 for task-set sizes of 10 and 50 respectively. Similarly, Figure A.15 and Figure A.16 demonstrate the results with ξ_2 for task-set sizes of 10 and 50 respectively. All the values in these results are normalised to the maximum average sleep-interval of the SRA algorithm in the corresponding task-set size. The results presented in these graphs are consistent with the previously explained results in Figure A.8, Figure A.9, Figure A.10 and Figure A.12. The reasons described for different behaviours of all algorithms in terms of

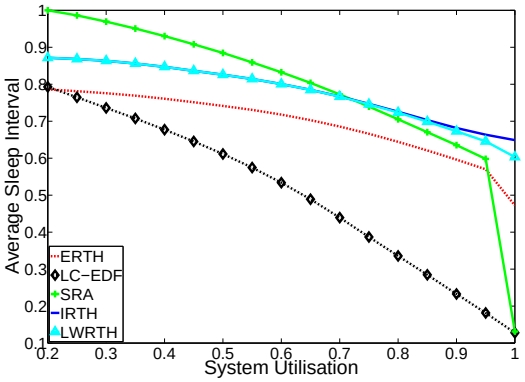


Figure A.15: Normalised average sleep interval ($|\tau| = 10$ and ξ_2)

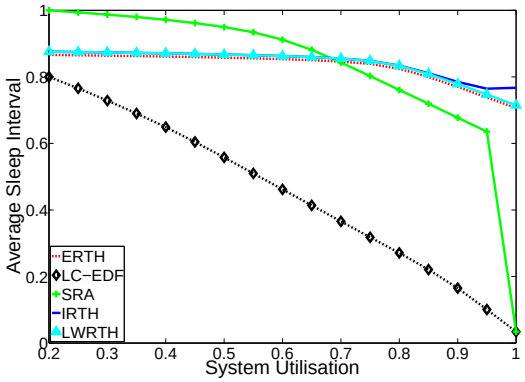


Figure A.16: Normalised average sleep interval ($|\tau| = 50$ and ξ_2)

energy consumption also applies on the average sleep-intervals. The SRA algorithm has the highest average sleep-interval compared to other algorithms for a distribution of ξ_1 except at $U = 1$. With the same setting SRA has the maximum gain in energy consumption over LC-EDF as shown in Figure A.8. LWRTH and IRTH behave the same at low utilisations but get diverted at high utilisations for both distributions (ξ_1 and ξ_2) (see also Figure A.12).

The average sleep interval of ERTH is around 10% lower when compared to IRTH and LWRTH for small task-set sizes for both distributions. However, for a large task-set size ($|\tau| = 50$) ERTH, LWRTH and IRTH behave the same for ξ_1 and ξ_2 except at very high utilisation, where IRTH has larger average sleep-intervals due to less pessimism in analysis. The SRA algorithm has lower average sleep interval when compared to IRTH and LWRTH with a distribution of ξ_2 for both task-set sizes ($|\tau| = 10$ and $|\tau| = 50$) after $U \geq 0.7$. ERTH behaves better against SRA for large task-set size after $U \geq 0.7$ for ξ_2 (with same setting gain in energy consumption of ERTH is also higher when compare to SRA in ξ_2, see Figure A.10). These results demonstrate that SRA does not behave better than the proposed algorithms with large number of RT tasks (i.e., for a distribution of ξ_2) at high utilisations. This conclusion is consistent to the observation of the energy consumption of these algorithms.

A.3.2.8 Analysing Sleep Threshold

To analyse the effect of different types of hardware platforms, the effect of a high sleep threshold Ψ that indicates the scaled value of bet_n obtained by altering the power model parameters is studied for ERTH, IRTH, LWRTH, SRA and LC-EDF for two different distributions of ξ_1 and ξ_2. This work analysed the different scaling factors of the sleep threshold given in Table A.3. The scaling of Ψ corresponds to a scaling of all bet_n. This is achieved by modifying the power model parameters such as sleep transitions overheads etc. Figure A.17 presents the energy consumption of ERTH for different values of Ψ with $|\tau| = 50$ and ξ_1. Naturally, an increase in bet_n is also reflected in higher overall energy consumption as depicted in Figure A.17. IRTH and LWRTH have the similar results for the different values of Ψ.

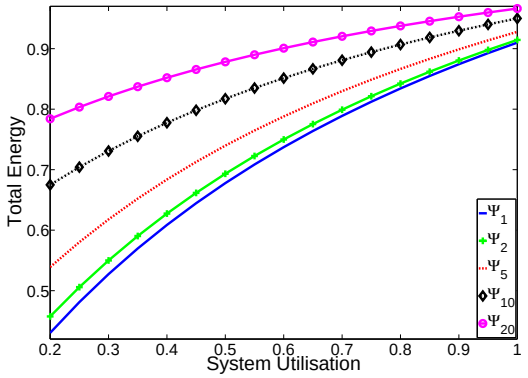


Figure A.17: Effect of sleep threshold change on total energy consumption of ERTH ($|\tau| = 50$ and ξ_1)

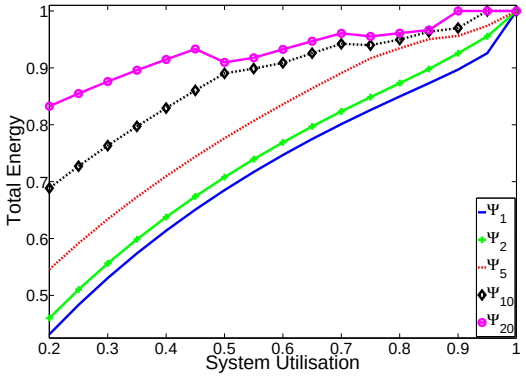


Figure A.18: Effect of sleep threshold change on total energy consumption of LC-EDF ($|\tau| = 50$ and ξ_1)

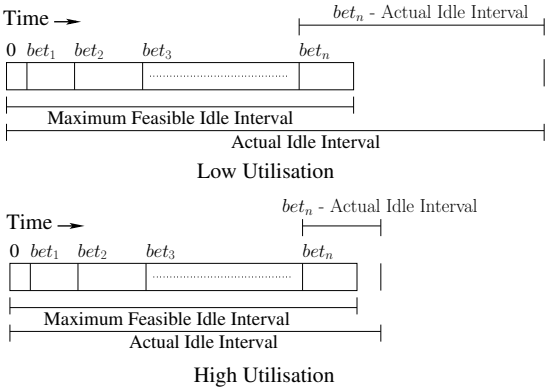


Figure A.19: Energy drop on same threshold of the LC-EDF algorithm

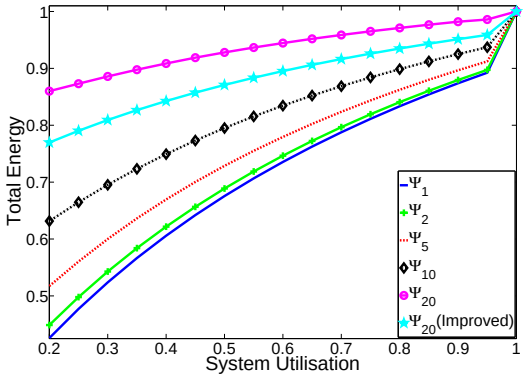


Figure A.20: Effect of sleep threshold change on total energy consumption of SRA ($|\tau| = 50$ and ξ_1)

LC-EDF suffers from a high dependence on different available sleep states. The effect of different sleep threshold values on LC-EDF is shown in Figure A.18 with $|\tau| = 50$ and ξ_1. LC-EDF uses a single sleep state for each utilisation and Ψ pair. Similar to ERTH the energy consumption in LC-EDF also increase with an increase in the value of bet_n. Abrupt variations on the same line of any sleep threshold refer to a switch to a different sleep state. An interesting observation in this graph is the drop of energy consumption for Ψ_{20} at $U = 0.5$ when compared to the energy consumption at $U = 0.45$, which is explained as follows with the help of Figure A.19. LC-EDF can compute a bound on the maximally-feasible idle interval in the schedule and states all the idle-intervals will be longer than this bound. An opportunistic approach of LC-EDF selects the most efficient sleep state considering a bound on the maximally-feasible idle interval. Usually the actual-idle-intervals are longer than this bound. However, with an increase in system utilisation, the difference of actual-idle-intervals and bet_n (as shown in Figure A.19) becomes smaller. Other sleep states with low transition overhead have greater margin to save energy when compared to the more efficient sleep state with higher transition delay. Hence, at $U = 0.5$ when system switches to

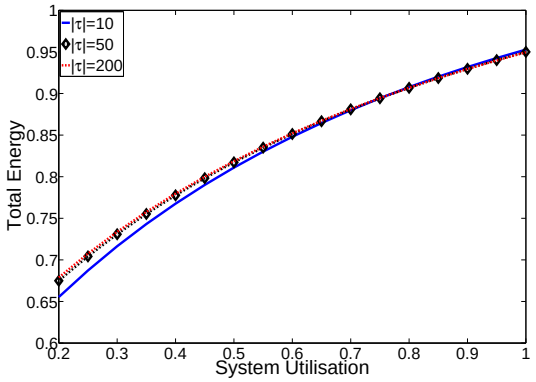


Figure A.21: Effect of sleep threshold Ψ_{10} on ERTH (ξ_1)

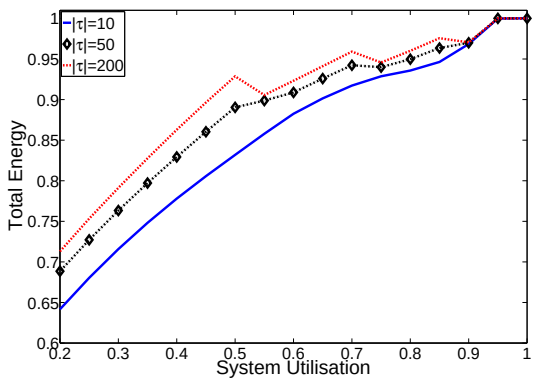


Figure A.22: Effect of sleep threshold Ψ_{10} on LC-EDF (ξ_1)

another less efficient sleep state it saves more compared to the one selected at $U = 0.45$ for Ψ_{20}.

The effect of different sleep thresholds on SRA for a distribution of ξ_1 and a task-set size of 50 is presented in Figure A.20. It behaves similar to ERTH for all thresholds except Ψ_{20}. The energy consumption of the SRA algorithm scales up for Ψ_{20} when compared to ERTH. To ensure the schedulability, the SRA algorithm selects its most efficient sleep state offline considering the minimum idle interval Z_{min}. However, the sleep state selected offline might not be a good choice as explained earlier for LC-EDF algorithm with the help of Figure A.19. One extension to the SRA algorithm is provided to enhance its performance by selecting the appropriate sleep state online based on the predicted idle interval. When the processor is in an idle mode, the next sleep duration is determined to greater than or equal to an interval $Z_{sleep} = max(Z_1, R_1^F)$, where Z_1 is the maximum procrastination interval allowed on the arrival of highest priority task and R_1^F is the execution slack available to the highest priority task. The sleep state is selected for the sleep interval Z_{sleep} online. The simulations results for Ψ_{20} are shown in Figure A.20 under a legend $\Psi_{20}(Improved)$. It clearly shows an effectiveness of the proposed modification. All the experiments presented in this section are repeated for the all settings with this modification. The results remains same for all cases except for the very high threshold of Ψ_{20}. Therefore, this modification is useful for the hardware platforms having sleep states with high sleep transition overheads. Though it increases the online overhead of sleep state selection but cannot perform worse in terms of energy saving when compared to original SRA algorithm.

The effect of Ψ is also analysed for different task-set sizes. The energy consumption of ERTH is presented in Figure A.21 for different task-set sizes with Ψ_{10} and ξ_1. The high sleep threshold managed to created a minute difference between the energy consumption of $|\tau| = 10$ when compared to other task-set sizes at low utilisations. At such a low utilisation tasks in a small task-set size are widely spread out and provide an extra opportunity to the scheduler to use most efficient sleep states in conjugation with principle 2. This experiment with the same values is repeated for LC-EDF as shown in Figure A.22. The spread along the vertical axis among the different task-set sizes is higher when compared to ERTH, which affirms the strong dependency of LC-EDF on the task-set size as explained in the beginning of Section A.3.2. The reason for the bumpy effect on

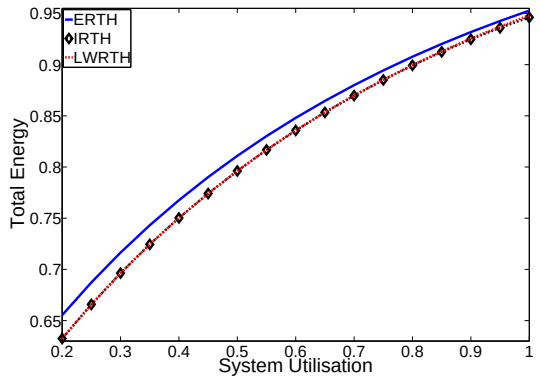


Figure A.23: Total energy consumption of ERTH, IRTH and LWRTH at Ψ_{10} with $|\tau| = 10$ and ξ_1

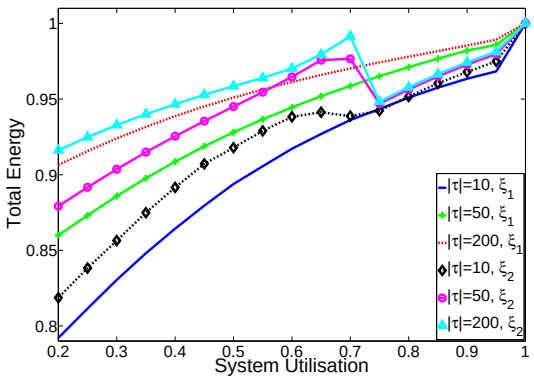


Figure A.24: Effect of two different distributions (ξ_1, ξ_2) on high sleep threshold (Ψ_{20}) with the SRA algorithm

the line of $|\tau| = 200$ is the same as already been explained earlier with Figure A.18. The same experiment have also been performed for IRTH and LWRTH. The graphs have the same shape as ERTH with a slight larger gap between $|\tau| = 10$ and other task-set sizes. It demonstrates that high sleep threshold slightly favours small task-set size at low utilisation.

The comparison of the energy consumption of ERTH, IRTH and LWRTH is illustrated in Figure A.23 with $|\tau| = 10$, Ψ_{10} and ξ_1. The increase in bet_n enhance the potential of IRTH and LWRTH to save more energy compared to ERTH. The future release information is useful at high sleep threshold values and helps to pick more efficient sleep states. The curve of LWRTH tends to rise at $U = 1$ when compared to IRTH but the difference in energy consumption is very small and negligible. Figure A.24 shows the effect of high threshold Ψ_{20} on different task-set sizes and two different distributions with SRA. First observation is the difference of energy consumption between different task-set sizes. Secondly, the dropped down of the energy consumption for a distribution of ξ_2 is due to the change of sleep states. The reason for such behaviour is already explained in conjunction with LC-EDF's similar behaviour with the help of Figure A.19. The performance of SRA algorithm suffers with a decrease in the number of BE tasks as the long period tasks allows longer procrastination interval.

Similarly, the effect of Ψ is also analysed for a distribution ξ_2 with all task-set sizes on all algorithms (ERTH, IRTH, LWRTH, SRA and LC-EDF). The only difference it makes is the increase in the total energy consumption. This is motivated in the reduced share of BE tasks. It reduces the opportunity in ERTH, IRTH and LWRTH to use the principle 2, and thus results in a extra energy consumption. SRA and LC-EDF algorithms (as mentioned earlier) depend on the periods of the tasks. Therefore, fewer tasks with longer periods decrease the opportunity to save on energy, hence ξ_2 results in more energy consumed when compared to the ξ_1.

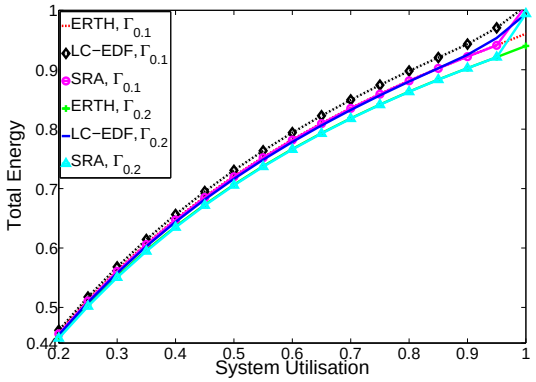


Figure A.25: Normalised total energy consumption with $|\tau| = 200$ and ξ_1

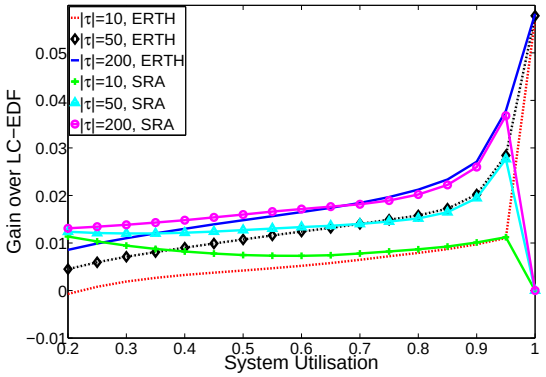


Figure A.26: Overall-gain of ERTH and SRA over LC-EDF (ξ_2 and $\Gamma_{0.1}$)

A.3.3 Scenario 2 $(RT \Rightarrow (A_i = C_i), BE \Rightarrow (A_i \leq C_i))$

In scenario 2, BE tasks are allowed to occasionally require more than their allocated budget A_i. All algorithms i.e., ERTH, IRTH, LWRTH, SRA and LC-EDF have been extended and allowed to borrow from the budget of future job releases of the same task. While it was of little consequence in scenario 1, it has to be noted that in scenario 2, ERTH and IRTH do not allocate execution slack to BE tasks. BE jobs usually overrun their budget and borrow from their future jobs, and hence, they are likely to consume the slack. However, the execution slack is only retained for energy management purposes. Thus, if the next job to execute is of BE type, the execution slack is maintained in the slack container and its deadline is updated as follows: $S_e^{dl} = \max\{S_e^{dl}, d_{i,k}\}$, where $d_{i,k}$ is the absolute deadline of the BE job under consideration.

A.3.3.1 Analysing Total Energy Consumption

The total energy consumption of ERTH, SRA and LC-EDF in this scenario is analysed for two different sporadic delay limits ($\Gamma_{0.1}, \Gamma_{0.2}$) and two different distributions (ξ_1,ξ_2) with $|\tau| = 200$. Figure A.25 demonstrates the effect of a variation in the sporadic delay limit. The distribution for this experiment is fixed to ξ_1. For the sake of clear representation, all the values of Figure A.25 are normalised to the corresponding results of NS with a distribution of $\Gamma_{0.1}$. $\Gamma_{0.1}$ and $\Gamma_{0.2}$ define an interval of 10% and 20% of T_i respectively for the sporadic delay to maneuver for a task T_i. The expansion of this interval means extra sporadic slack in the system when compared to the nominal utilisation. The sporadic slack is dealt implicitly in the proposed algorithms. Therefore, energy consumption is less with $\Gamma_{0.2}$ when compared to $\Gamma_{0.1}$ as shown in Figure A.25. Similarly, SRA and LC-EDF also have more room to initiate a sleep state as well. However, at higher utilisation; extra sporadic slack does not help to save energy in LC-EDF or SRA. These algorithms (LC-EDF and SRA) calculate their maximally-feasible idle interval based on the worst-case scenario i.e., each job of a task will be released as soon as possible with a difference of minimum inter-arrival. Therefore, at high utilisation, the feasible-sleep interval is usually short and they cannot utilise the more efficient sleep states effectively. As a general rule ERTH performs superior to LC-EDF,

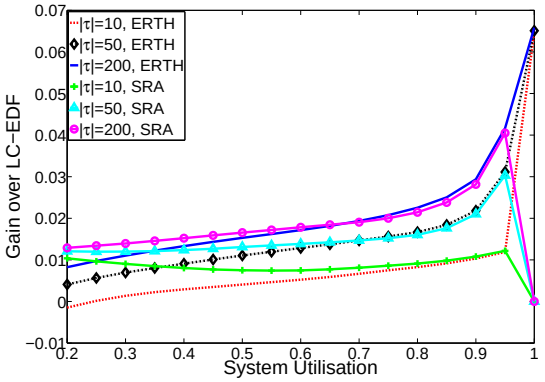


Figure A.27: Overall-gain of ERTH and SRA over LC-EDF (ξ_2 and $\Gamma_{0.2}$)

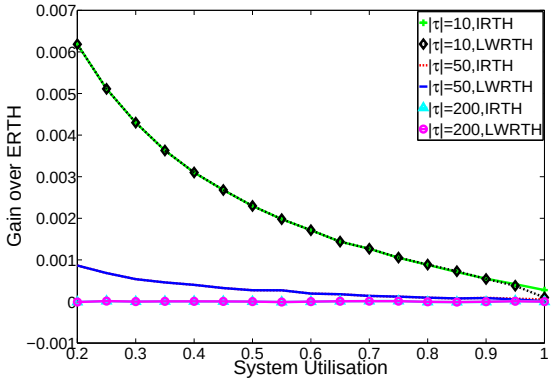


Figure A.28: Overall-gain of IRTH and LWRTH over ERTH (ξ_1 and $\Gamma_{0.1}$)

especially at higher utilisations the difference is prominent. However, it performs comparable to SRA but consumes less energy at higher utilisations. The same experiment is repeated with ξ_2, where the energy consumption of all algorithms decreases, the reason of which is explained in conjunction with the next experiment.

The energy consumption of two distributions (ξ_1,ξ_2) is studied with a fixed task-set size of $|\tau| = 200$ and $\Gamma_{0.1}$. The resulting figure (not shown here) has a similar shape when compared to Figure A.25 but the difference between ξ_1 and ξ_2 is slightly more pronounced. The energy consumption of SRA, LC-EDF and ERTH is reduced for ξ_2 when compared to ξ_1. The percentage of the BE tasks in ξ_2 is reduced to 40% and results in less borrowing. Therefore, the processor consumes less energy with ξ_2 when compared to ξ_1. Nevertheless, ERTH outperforms LC-EDF and comparable to SRA in both distributions (ξ_1, ξ_2), even with the borrowing mechanism integrated. The energy consumption of all algorithms decreases, when the same experiment is done with $\Gamma_{0.2}$ due to extra sporadic slack in the system. Moreover, it is also observed that the energy consumption of IRTH and LWRTH is similar to ERTH for the above mentioned two experiments. The borrowing effect dominates the total energy consumption and provides less room to manoeuvre for energy saving purposes.

A.3.3.2 Analysing Overall Gain

The overall energy consumption gain of ERTH and SRA over LC-EDF is analysed for scenario 2 for three task-set sizes ($|\tau| \in \{10, 50, 200\}$) with ξ_2 in Figure A.26 and Figure A.27 considering two different sporadic delay limits $\Gamma_{0.1}$ and $\Gamma_{0.2}$ respectively. Although sporadic slack is managed implicitly in all algorithms, ERTH outperforms LC-EDF, especially at high utilisation for large task-set sizes. The SRA algorithm performs better when compared to LC-EDF in all cases. For large task-set sizes, ERTH performs superior when compared to SRA at high utilisations and SRA performs better at low utilisations. The slight increase in gain of ERTH with $\Gamma_{0.2}$ over $\Gamma_{0.1}$ indicates an efficient implicit use of sporadic slack in ERTH. Nevertheless, LC-EDF performs slightly better when compared to ERTH only at $U = 0.2$ for a task-set size of 10 as LC-EDF can create

large gaps in this case. Similarly, with a distribution of ξ_1 (not shown here), the results indicate a slight decrease in overall gain. Major difference lies at $U = 1$, where it varies approximately about 1% of overall gain. However, for smaller utilisations the difference is less pronounced. This is a function of the reduced number of BE tasks in ξ_2 and the consequently smaller amount of borrowing in the system.

Figure A.26 shows the overall-gain of ERTH and SRA over LC-EDF with ξ_2 and $\Gamma_{0.1}$ and comparing that to Figure A.8 one can notice the reduced gains returned when borrowing. Generally, the gain of scenario 2 compared to scenario 1 is less at higher utilisations, but approximately the same at lower utilisations. The gain rises exponentially in Figure A.8, Figure A.26 after $U = 0.8$ for large task-set sizes. The overall energy gain of IRTH and LWRTH over ERTH is depicted in Figure A.28 for ξ_1 and $\Gamma_{0.1}$. Compared to Figure A.12, the overall gain has reduced in scenario 2. Moreover, IRTH and LWRTH behave identical when borrowing is enabled. Main reason is the extra execution requested by the BE task through borrowing i.e., an increase in effective utilisation.

A.3.3.3 Other Miscellaneous Factors

The normalised sleep state energy consumption of scenario 2 is similar to scenario 1. Moreover, the higher sleep threshold effect in scenario 2 is also identical to scenario 1 for IRTH, LWRTH, LC-EDF, SRA and ERTH with just one difference, i.e., energy consumption of the system increases in scenario 2. It occurs due to an increase in execution-time requirement of the BE tasks that occasionally overrun and borrow from their respective future releases. To summarise, for different combinations of ξ and Γ, an increase in gain occurs in the following ascending order $(\xi_2,\Gamma_{0.2})$, $(\xi_2,\Gamma_{0.1})$, $(\xi_1,\Gamma_{0.2})$ and $(\xi_1,\Gamma_{0.1})$. This is caused by an increase in sporadic delay limit, sporadic slack also increases, therefore the processor saves more energy. Similarly, if borrowing is enabled, the processor consumes extra energy. Thus with the highest sporadic delay limit and minimum borrowing $(\xi_2,\Gamma_{0.2})$ the energy consumption is least in scenario 2, whilst with least sporadic delay limit and most borrowing $(\xi_1,\Gamma_{0.1})$ energy consumption is maximised.

A.4 Pre-emptions Related Results

A side effect of the use of the sleep states is a change in the number of pre-emptions. In order to find the sleep state relation with the number of pre-emptions, the pre-emptions for all algorithms (ERTH, IRTH, LWRTH, SRA and LC-EDF) are counted for different parameters. The DBFP is not included in this evaluation as it is easier to get the trend based on the results of LC-EDF and SRA. The experimental setup defined for alternative race-to-halt algorithms and the parameters defined in Table 4.3 remain the same except some alterations in best-case execution-time limit C^b and sporadic delay limit Γ. The best-case execution-time limit C^b is varied from 0.25 to 1 with an increment of 0.25 (i.e., $C^b \in \{0.25, 0.5, 0.75, 1\}$). Similarly, the sporadic delay limit Γ is varied from 0 to 0.6 with an increment of 0.2 (i.e., $\Gamma \in \{0, 0.2, 0.4, 0.6\}$). For the representation purposes, only the two corner values for $\Gamma = (0, 0.6)$ and $C^b = (0.25, 1)$ are plotted, as the results for the other two values lies in between these two curves and scales linearly. All the values in the

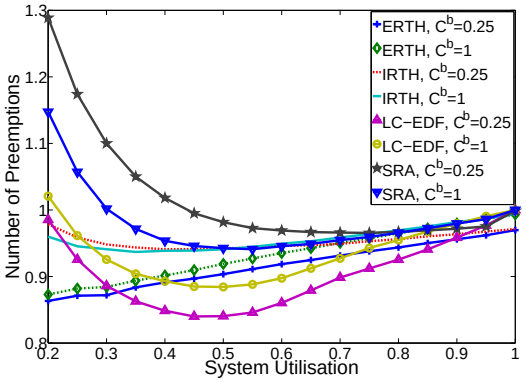


Figure A.29: Variation in C^b for $|\tau| = 10$ $(\Gamma_{0.2}, \xi_1)$

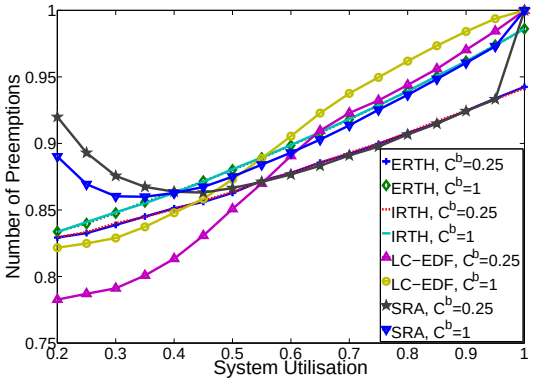


Figure A.30: Variation in C^b for $|\tau| = 50$ $(\Gamma_{0.2}, \xi_1)$

following experiments are normalised to the number of pre-emptions with earliest deadline first algorithm (EDF). The results shows that the pre-emption count for LWRTH is virtually identical to IRTH, therefore, for presentation purposes only results of IRTH are shown hereafter.

A.4.1 Scenario 1

In this scenario, it is assumed all the tasks have budget equal to their worst-case execution time, i.e., $A_i = C_i$.

A.4.1.1 Effect of C^b

The effect of best-case execution-time limit variation is shown in Figure A.29 with $|\tau| = 10$, $\Gamma_{0.2}$ and ξ_1. The results are plotted only for $C^b \in \{0.25, 1\}$, while the other two values of $C^b \in \{0.5, 0.75\}$ lie in between these two curves of the corresponding algorithm and scale linearly. First observation for small task-set size is the positive impact of $C^b = 0.25$ over $C^b = 1$ that holds for all utilisations with LC-EDF and ERTH, and only at high utilisations with SRA and IRTH (at low utilisations the opposite behaviour of C^b for SRA and IRTH will be explained later in the discussion). The reason is quite clear, an increase in the value of C^b potentially decreases the range of execution slack that a task can provide online. Thus $C^b = 1$ means no execution slack in the system. Overall all scheduling algorithms showed a positive impact of sleep states on the number of pre-emptions, except for one case in LC-EDF at $U = 0.2$ and for SRA at $U \leq 0.45$. By injecting more execution slack, the processor initiates more often a sleep state and hence lowers the number of pre-emptions.

As previously mentioned in Section 4.3, a sleep state that delays the execution can increase the pre-emptions by pushing it closer to higher priority tasks. While at the same time, the delayed execution caused by a sleep state can combine the job releases to reduce the pre-emption count. With a small task-set size, jobs releases are anyway dispersed at low utilisation. However, as the utilisation increases execution increases and jobs execution run into each other and cause a rise in the number of pre-emptions. SRA and LC-EDF initiate a sleep state in idle mode, starts estimating

the delay interval on the next job release and extend it as much as possible. This behaviour causes widely spread low priority jobs at low utilisation to come closer to high priority jobs and hence increase the pre-emption count. Moreover, at low utilisation, in EDF the number of pre-emptions are small and the use of sleep states cannot help much to reduce them. However, as the utilisation increases pre-emption count drops quickly for LC-EDF up to approximately a utilisation of 0.5 and for SRA up to $U = 0.65$. These algorithms (SRA and LC-EDF) can collate enough tasks releases to compensate the effect of extra pre-emption due to the delay of execution. Nevertheless, at a further increase in utilisation (beyond $U > 0.5$ for LC-EDF and $U > 0.65$ for SRA), the possibility to use sleep intervals also reduces for both LC-EDF and SRA, and consequently their ability to reduce the pre-emptions. Therefore, at $U = 1$, both algorithms cannot afford to initiate a sleep state and hence, the pre-emption count is the same as EDF.

The ERTH algorithm distributes the sleep states uniformly in the schedule and never pushes the sleep interval beyond the static sleep interval χ_{min}, even if there is a possibility to prolong the sleep interval. Not extending the sleep state to its limit pays off at low utilisation, as it never delays execution to increase the pre-emption count. However, ERTH ability to introduce sleep in the busy interval helps to save pre-emptions even at higher utilisations. Therefore, ERTH has linear curve from low to high utilisation for both C^b. The difference decreases towards high utilisation due to a decrease in the execution slack and hence fewer sleep states in the system.

IRTH and SRA show an oddity at low utilisations, as $C^b = 1$ has fewer pre-emptions compared to $C^b = 0.25$. In IRTH algorithm, sleep states are increased by utilising predicted future release information. Future release information is very useful especially to prolong the sleep interval for a small task-set size at low utilisations. It can be easily motivated by the curve of Figure A.12 that IRTH saves more energy at low utilisations for a task-set size of 10 due to extensively long sleep intervals. Similarly, SRA sleep intervals are even greater than or equal to all the algorithms. As a side effect of long sleep intervals, they assemble a large amount of work for later execution. This delayed execution later on encounters high priority tasks and causes additional pre-emptions. However, if the encountered high priority tasks execute for their C_i, the chances are higher that it might accumulate other tasks having priority higher than the backlog and less than the encountered high priority tasks. These intermediate priority tasks will not cause pre-emptions to a backlog. This effect causes the flip of $C^b = 0.25$ over $C^b = 1$ for low utilisations.

Large task-set sizes give a smoother curve, as shown in Figure A.30 for a task-set size of $|\tau| = 50$ with a distribution of ξ_1 and sporadic delay limit of $\Gamma_{0.2}$. All the algorithms have fewer number of pre-emptions when compared to EDF and also savings are larger when compared to a task-set size of 10. Oppose to small task-set size, the number of pre-emptions of SRA and LC-EDF are smaller than EDF at small utilisations in this case, as the probability of jobs being widely spread out is lower for large task-set size. IRTH also behaves identical to ERTH, as future release information is less effective for large task-sets. Similar to previous case, the curves for other two values of $C^b \in \{0.5, 0.75\}$ lies in between $C^b = 1$ and $C^b = 0.25$, and this observation holds for all algorithms.

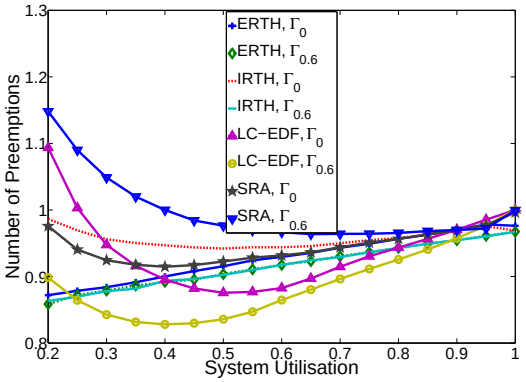


Figure A.31: Variation in Γ for $|\tau| = 10$ (ξ_1)

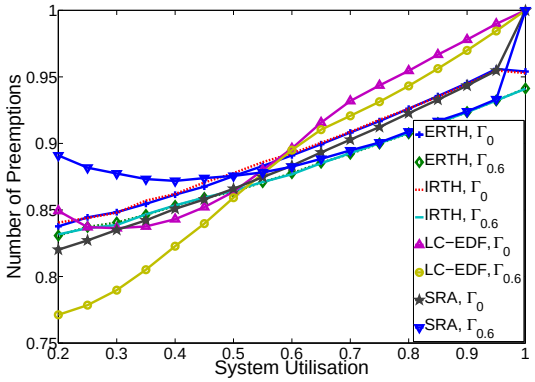


Figure A.32: Variation in Γ for $|\tau| = 50$ (ξ_1)

A.4.1.2 Effect of Γ

The effect of variation in sporadic delay limit Γ is illustrated in Figure A.31 for a task-set $|\tau| = 10$ and a distribution of ξ_1. All algorithms consume sporadic slack implicitly. An increase in the sporadic delay limit causes an increase in sporadic slack and that can increase the number and/or prolong the sleep transitions. Similar to the execution slack, sporadic slack also helps to decrease the number of pre-emptions for all algorithms except SRA. ERTH and LC-EDF behave similar to the Figure A.29, with a slight variation in the beginning and towards the end of utilisations. In IRTH, both sporadic delay limits ($\Gamma_0, \Gamma_{0.6}$) have access to same amount of future release information. Therefore, they also follow the same trend of saving on the number of pre-emptions with extra sporadic slack. However, the SRA algorithm that has the longest sleep intervals of all algorithms increases the number of pre-emptions when extra sporadic slack is available. This is motivated by the fact that widely spread out jobs in the EDF schedule are unlikely to preempt each other but SRA brings these jobs close to such a degree that they result in an increased number of pre-emptions at low utilisation. The other two sporadic delay limit ($\Gamma_{0.2}, \Gamma_{0.4}$ are bounded by $\Gamma_0, \Gamma_{0.6}$. A large task-set $|\tau| = 50$, brings IRTH curves close to ERTH (Figure A.32) because future release information becomes less important. Moreover, the pre-emption avoiding effect of LC-EDF at low and medium utilisations is reduced for larger task-set sizes, when comparing to smaller task-sets, due to the higher probability of tasks cutting idle intervals short, as illustrated in Figure A.32. Moreover, the SRA algorithm causes more pre-emptions with an increase in sporadic slack for low utilisations. However, for large utilisations it saves more pre-emptions with additional sporadic slack similar to other algorithms. Globally, whenever there is a possibility to increase the length of sleep state (either through execution slack or sporadic slack) at low utilisations, SRA increase the number of pre-emptions.

A.4.1.3 Effect of ξ

The effect of variation in the distribution ξ is demonstrated in Figure A.33 for $|\tau| = 10, \Gamma_{0.2}$ and $C^b = 0.5$. In general for all the algorithms, distribution ξ_2 saves more pre-emptions compared to ξ_1. BE tasks are more vulnerable to pre-emptions as they have longer periods along with their

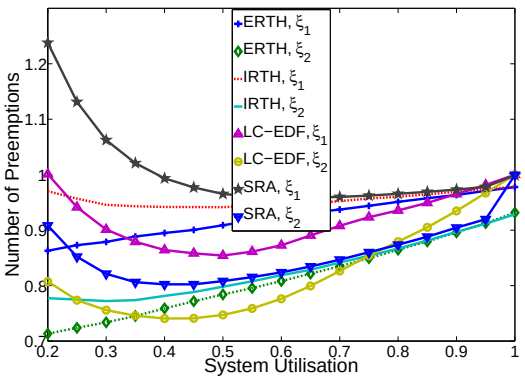


Figure A.33: Variation in ξ for $|\tau| = 10$ ($\Gamma_{0.2}, C^b = 0.5$)

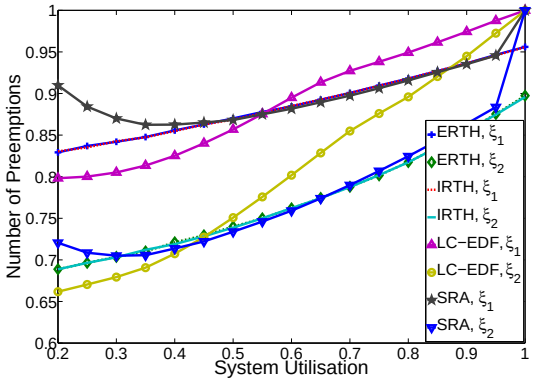


Figure A.34: Variation in ξ for $|\tau| = 50$ ($\Gamma_{0.2}, C^b = 0.5$)

execution. Therefore, ξ_1 having more BE tasks results in more pre-emptions, when compared to ξ_2. The same observation holds for the large task-set size as shown in Figure A.34.

A.4.2 Scenario 2

In this scenario, BE jobs occasionally require more than their respective budget and borrow from their future job releases.

A.4.2.1 Effect of C^b

Figure A.35 depicts the effect of variation in the best-case execution time limit C^b for a $|\tau| = 10$, $\Gamma_{0.2}$ and ξ_1. One of the interesting observation that holds for all algorithms in general is that now $C^b = 1$ offers fewer pre-emptions when compared to $C^b = 0.25$. Because of the borrowing, BE tasks add a great deal of backlog in addition to a backlog assembled due to sleep transitions. Therefore, it increases the probability to encounter higher priority tasks. Similar to the case explained for IRTH ($U \leq 0.5$) in Figure A.29, if the encountered higher priority tasks execute for their C_i, chances are higher that they will collect some of the tasks having priority in between backlog and the higher priority executing jobs. Thus $C^b = 1$ offers fewer pre-emptions compared to $C^b = 0.25$. Similar behaviour is observed for a large task-set size of 50 as shown in Figure A.36. Only exception is at $U \geq 0.9$ for particularly ERTH and IRTH. At such a high utilisation, sleep states save more pre-emptions when compared to an increment in pre-emptions due to its backlog.

A.4.2.2 Effect of Γ

A further experiment explores a variation in the sporadic delay limit Γ for all task-set sizes. The results show an increase at low utilisations when compared to a system without borrowing. Moreover, SRA with borrowing in the system saves more pre-emptions with an increase in the amount of sporadic slack. Thus, the number of pre-emptions is higher for Γ_0 when compared to $\Gamma_{0.6}$.

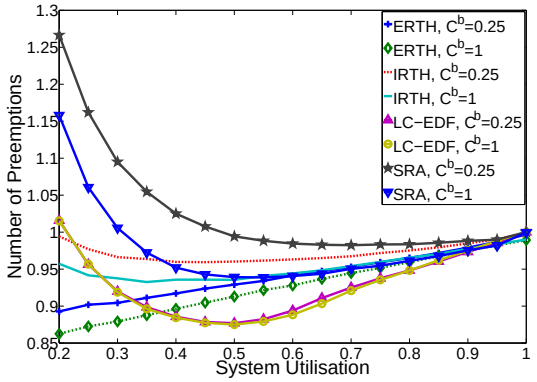


Figure A.35: Variation in C^b for $|\tau| = 10$ $(\Gamma_{0.2}, \xi_1)$

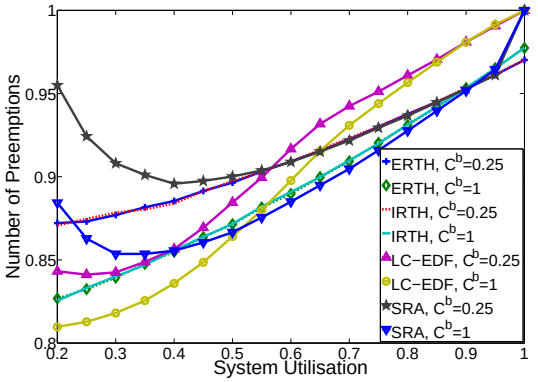


Figure A.36: Variation in C^b for $|\tau| = 50$ $(\Gamma_{0.2}, \xi_1)$

A.4.2.3 Effect of ξ

The variation in the distribution of task-set ξ also increase the number of pre-emption when the borrowing is allowed in the system. BE tasks that overrun demand extra execution and hence more pre-emptions compared to the normal system without borrowing.

Finally, it is observed, when it comes to number of pre-emptions, ERTH performs superior to IRTH, LWRTH and SRA for small task-set sizes. Nevertheless, it equally performs comparable to IRTH, SRA and LWRTH if not better for large task-set sizes. Though SRA performs better energy-wise but has the highest number of pre-emptions at low utilisations and sometimes it even exceeds those by plain EDF scheduler. The overhead associated to the number of pre-emptions saved through the use of sleep states can help to reduce the worst-case execution time of the tasks. This effect further extends the slack in the system and consequently provide an extra opportunity to save energy in the system or increase the system utilisation.

Bibliography

[AA05] T.A. AlEnawy and H. Aydin. Energy-aware task allocation for rate monotonic scheduling. In Proceedings of the 11th IEEE Real-Time and Embedded Technology and Applications Symposium, pages 213–223, 2005.

[AB98] Luca Abeni and Giorgio Buttazzo. Integrating multimedia applications in hard real-time systems. In Proceedings of the 19th IEEE Real-Time Systems Symposium, pages 4–13, Dec 1998.

[AB08] James Anderson and Sanjoy Baruah. Energy-efficient synthesis of edf-scheduled multiprocessor real-time systems. International Journal on Embedded Systems, 4(1), 2008.

[ABJ01] B. Andersson, S. Baruah, and J. Jonsson. Static-priority scheduling on multiprocessors. In Proceedings of the 22nd IEEE Real-Time Systems Symposium, pages 193–202, Dec 2001.

[AIS08] John Augustine, Sandy Irani, and Chaitanya Swamy. Optimal power-down strategies. Society for Industrial and Applied Mathematics Journal on Computing, 37(5):1499–1516, January 2008.

[AMMM01] H. Aydin, R. Melhem, D. Mosse, and Alvarez P. Mejia. Dynamic and aggressive scheduling techniques for power-aware real-time systems. In Proceedings of the 22nd IEEE Real-Time Systems Symposium, pages 95 – 105, dec. 2001.

[ANP11] Muhammad Ali Awan, Borislav Nikolic, and Stefan M. Petters. Comparing the schedulers and power saving strategies with sparts. In the RTSS@Work, Open Demo Session of Real-Time Techniques and Technologies, Proceedings of the 32nd IEEE Real-Time Systems Symposium, Vienna, Austria, November 2011. IEEE.

[AP11] Muhammad Ali Awan and Stefan M. Petters. Enhanced race-to-halt: A leakage-aware energy management approach for dynamic priority systems. In Proceedings of the 23rd Euromicro Conference on Real-Time Systems, pages 92–101. IEEE Computer Society, 2011.

[ARMa] ARM Ltd. big.LITTLE Processing. http://www.arm.com/products/processors/technologies/biglittleprocessing.php.

[ARMb] ARM Ltd. Cortex™-A Series. http://www.arm.com/products/processors/cortex-a/cortex-a17-processor.php.

[AT06] B. Andersson and E. Tovar. Multiprocessor scheduling with few preemptions. In Proceedings of the 12th IEEE Conference on Embedded and Real-Time Computing and Applications, pages 322–334, 2006.

[AY03] H. Aydin and Qi Yang. Energy-aware partitioning for multiprocessor real-time systems. In Proceedings of the 17nd IEEE Parallel and Distributed Processing Symposium, pages 9 pp.–, april 2003.

[AYP13] Muhammad Ali Awan, Patrick Meumeu Yomsi, and Stefan M. Petters. Optimal procrastination interval upon uniprocessors, CISTER-TR-130608, 2013. https://www.cister.isep.ipp.pt/people/Muhammad%2BAli%2BAwan/publications/.

[Bak05] T.P. Baker. An analysis of edf schedulability on a multiprocessor. IEEE Transactions on Parallel and Distributed Systems, 16(8):760–768, Aug 2005.

[Bap06] Philippe Baptiste. Scheduling unit tasks to minimize the number of idle periods: a polynomial time algorithm for offline dynamic power management. In Proceedings of the 17th ACM-SIAM Symposium on Discrete Algorithms, pages 364–367, Miami, Florida, January 2006. ACM.

[Bar06] SanjoyK. Baruah. The non-preemptive scheduling of periodic tasks upon multiprocessors. Journal of Real–Time Systems, 32(1-2):9–20, 2006.

[Bar14] Michael. Barr. Embedded Systems Glossary. BARR group, reterived on 20-02-2014 edition, 2014. http://www.barrgroup.com/Embedded-Systems/Glossary.

[BB06] S. Baruah and A Burns. Sustainable scheduling analysis. In Proceedings of the 27th IEEE Real-Time Systems Symposium, pages 159–168, Dec 2006.

[BBDM00] L. Benini, A. Bogliolo, and G. De Micheli. A survey of design techniques for system-level dynamic power management. IEEE Transactions on Very Large Scale Integration Systems, 8(3):299 –316, june 2000.

[BBLB03] Scott A. Brandt, Scott Banachowski, Caixue Lin, and Timothy Bisson. Dynamic integrated scheduling of hard real-time, soft real-time and non-real-time processes. In Proceedings of the 24th IEEE Real-Time Systems Symposium, page 396, Cancun, Mexico, December 2003.

[BCPV93] S. K. Baruah, N. K. Cohen, C. G. Plaxton, and D. A. Varvel. Proportionate progress: A notion of fairness in resource allocation. In Proceedings of the Twenty-fifth Annual ACM Symposium on Theory of Computing, STOC '93, pages 345–354, New York, NY, USA, 1993. ACM.

[BDM02] L. Benini and G. De Micheli. Networks on chips: a new soc paradigm. Computer, 35(1):70–78, Jan 2002.

[BDMM01] L. Benini, G. De Micheli, and E. Macii. Designing low-power circuits: practical recipes. IEEE Circuits and Systems Magazine, 1(1):6–25, First 2001.

[BDWZ12] A. Burns, R.I. Davis, P. Wang, and F. Zhang. Partitioned edf scheduling for multiprocessors using a c=d task splitting scheme. Journal of Real–Time Systems, 48(1):3–33, 2012.

[BET04] A.M Bernardes, D.C.R Espinosa, and J.A.S Tenório. Recycling of batteries: a review of current processes and technologies. Journal of Power Sources, 130(1–2):291 – 298, 2004.

[BG03] Sanjoy Baruah and Joël Goossens. Scheduling real-time tasks: Algorithms and complexity, 2003.

[Bin09] Enrico Bini. Modeling preemptive edf and fp by integer variables. In Proceedings of the 4th Multidisciplinary International Scheduling Conference, Dublin, Ireland, August 2009.

[BKP07] Nikhil Bansal, Tracy Kimbrel, and Kirk Pruhs. Speed scaling to manage energy and temperature. Journal of the ACM, 54(1):3:1–3:39, March 2007.

[BLOS95] A. Burchard, J. Liebeherr, Yingfeng Oh, and S.H. Son. New strategies for assigning real-time tasks to multiprocessor systems. IEEE Transactions on Computers, 44(12):1429–1442, Dec 1995.

[BRH90] Sanjoy K. Baruah, Louis E. Rosier, and Rodney R. Howell. Algorithms and complexity concerning the preemptive scheduling of periodic, real-time tasks on one processor. Journal of Real–Time Systems, 1990.

[BW09] A. Burns and A.J. Wellings. Real-Time Systems and Programming Languages: Ada, Real-Time Java and C/Real-Time POSIX. International Computer Science Series. Addison-Wesley, 2009.

[CC89] H. Chetto and M. Chetto. Some results of the earliest deadline scheduling algorithm. IEEE Transactions on Software Engineering, 15(10):1261–1269, 1989.

[CG05] Hui Cheng and Steve Goddard. Integrated device scheduling and processor voltage scaling for system-wide energy conservation. In Proceedings of the 2005 Workshop on Power Aware Real-time Computing, pages 24–29, September 2005.

[CG06] Hui Cheng and Steve Goddard. Online energy-aware I/O device scheduling for hard real-time systems. In Proceedings of the 43rd ACM/IEEE Conference on Design Automation Conference, pages 1055–1060, Leuven, Belgium, 2006. European Design and Automation Association.

[CH10] Aaron Carroll and Gernot Heiser. An analysis of power consumption in a smartphone. In Proceedings of the 2010 USENIX conference on USENIX annual technical conference, USENIXATC'10, pages 21–34. USENIX Association, 2010.

[Che04] W.K. Chen. The Electrical Engineering Handbook. Elsevier Science, 2004.

[Che08] Maryline Chetto. Results on the slack of a periodic task set. Technical report, ri 2008_5, Institut de Recherche en Communications et en Cybernétique de Nantes, june 2008.

[CHK07] Jian-Jia Chen, Chia-Mei Hung, and Tei-Wei Kuo. On the minimization fo the instantaneous temperature for periodic real-time tasks. In Proceedings of the 13th IEEE Real-Time and Embedded Technology and Applications Symposium, pages 236 –248, april 2007.

[CHQ10] V. Chaturvedi, Huang Huang, and Gang Quan. Leakage aware scheduling on maximum temperature minimization for periodic hard real-time systems. In IEEE 10th International Conference on Computer and Information Technology (CIT), pages 1802 –1809, July, 2010.

[CHT+09] Edward Chu, Tai-Yi Huang, Cheng-Han Tsai, Jian-Jia Chen, and Tei-Wei Kuo. A dvs-assisted hard real-time I/O device scheduling algorithm. Journal of Real–Time Systems, 41:222–255, 2009.

[CK06] Jian-Jia Chen and Tei-Wei Kuo. Procrastination for leakage-aware rate-monotonic scheduling on a dynamic voltage scaling processor. SIGPLAN Notices, 41:153–162, June 2006.

[CK07a] Jian-Jia Chen and Chin-Fu Kuo. Energy-efficient scheduling for real-time systems on dynamic voltage scaling (dvs) platforms. In Proceedings of the 13th IEEE Conference on Embedded and Real-Time Computing and Applications, pages 28 –38, aug. 2007.

[CK07b] Jian-Jia Chen and Tei-Wei Kuo. Procrastination determination for periodic real-time tasks in leakage-aware dynamic voltage scaling systems. In Proceedings of the International Conference on Computer Aided Design, pages 289 –294, November 2007.

[CKYK07] Jian-Jia Chen, Tei-Wei Kuo, Chia-Lin Yang, and Ku-Jei King. Energy-efficient real-time task scheduling with task rejection. In Proceedings of the 44th ACM/IEEE Conference on Design Automation Conference, pages 1–6, 2007.

[Cor] Nokia Corporation. http://www.nokia.com/global/.

[CQ11] Vivek Chaturvedi and Gang Quan. Leakage conscious dvs scheduling for peak temperature minimization. In Proceedings of the 16th Asia and South Pacific Design Automation Conference, 2011.

[CRJ06] Hyeonjoong Cho, B. Ravindran, and E.D. Jensen. An optimal real-time scheduling algorithm for multiprocessors. In Proceedings of the 27th IEEE Real-Time Systems Symposium, pages 101–110, Dec 2006.

[Cro] Crossbow Technology, Inc. Mica2 Mote, document part number: 6020-0042-08 rev a edition. www.investigacion.frc.utn.edu.ar/sensores/Equipamiento/Wireless/MICA2_Datasheet.pdf.

[CST09] Jian-Jia Chen, A. Schranzhofer, and L. Thiele. Energy minimization for periodic real-time tasks on heterogeneous processing units. In Proceedings of the IEEEInternational Symposium on Parallel & Distributed Processing 2009, pages 1 –12, 2009.

[CT08a] Jian-Jia Chen and L. Thiele. Energy-efficient task partition for periodic real-time tasks on platforms with dual processing elements. In 14th ICPADS, pages 161 – 168, dec. 2008.

[CT08b] Jian-Jia Chen and Lothar Thiele. Expected system energy consumption minimization in leakage-aware dvs systems. In Proceedings of the International Symposium on Low Power Electronics and Design, pages 315–320, Bangalore, India, 2008. ACM.

[CT09] Jian-Jia Chen and L. Thiele. Task partitioning and platform synthesis for energy efficiency. In Proceedings of the 15th IEEE Conference on Embedded and Real-Time Computing and Applications, pages 393–402, 2009.

[CWT09] Jian-Jia Chen, S. Wang, and L. Thiele. Proactive speed scheduling for real-time tasks under thermal constraints. In Proceedings of the 15th IEEE Real-Time and Embedded Technology and Applications Symposium, 2009.

[CYLK08] Jian-Jia Chen, Chuan-Yue Yang, Hsueh-I Lu, and Tei-Wei Kuo. Approximation algorithms for multiprocessor energy-efficient scheduling of periodic real-time tasks with uncertain task execution time. In Proceedings of the 14th IEEE Real-Time and Embedded Technology and Applications Symposium, pages 13–23, 2008.

[DA08a] Vinay Devadas and Hakan Aydin. On the interplay of dynamic voltage scaling and dynamic power management in real-time embedded applications. In Proceedings of the 8th International Conference on Embedded Software, pages 99–108, Atlanta, GA, USA, 2008. ACM.

[DA08b] Vinay Devadas and Hakan Aydin. Real-time dynamic power management through device forbidden regions. In Proceedings of the 14th IEEE Real-Time and Embedded Technology and Applications Symposium, pages 34–44, Washington, DC, USA, 2008. IEEE Computer Society.

[Dec98] M. Deck. Software reliability and the "cleanroom" approach: a position paper. In Reliability and Maintainability Symposium, 1998. Proceedings., Annual, pages 218–223, Jan 1998.

[DL78] Sudarshan K. Dhall and C. L. Liu. On a real-time scheduling problem. Operations Research, 26(1):127–140, 1978.

[DT97] George B. Dantzig and Mukund N Thapa. Linear programming: 1: Introduction. In Springer Verlag, 1997.

[Fre14] FreeScale. MPC8544E: PowerQUICC III Processor with DDR2, PCI, PCI Express®, SerDes, 1 GB Ethernet, SGMII, Security, document number: mpc8544fs, rev 2 edition, 2014. http://www.freescale.com/webapp/sps/site/prod_summary.jsp?code=MPC8536E.

[Fun10] Shelby Funk. Lre-tl: an optimal multiprocessor algorithm for sporadic task sets with unconstrained deadlines. Journal of Real–Time Systems, 46(3):332–359, 2010.

[FW11] Xing Fu and Xiaorui Wang. Utilization-controlled task consolidation for power optimization in multi-core real-time systems. In Proceedings of the 17th IEEE Conference on Embedded and Real-Time Computing and Applications, volume 1, pages 73–82, 2011.

[GFB03] Joël Goossens, Shelby Funk, and Sanjoy Baruah. Priority-driven scheduling of periodic task systems on multiprocessors. Journal of Real–Time Systems, 25(2-3):187–205, 2003.

[GH09] Laurent George and Jean-François Hermant. Characterization of the space of feasible worst-case execution times for earliest-deadline-first scheduling. In Journal of Aerospace Computing, Information, and Communication, volume 6:11, pages 604–623, 2009.

[Gro] Samsung Group. http://www.samsung.com/us/mobile/cell-phones/.

[HCK06a] Heng-Ruey Hsu, Jian-Jia Chen, and Tei-Wei Kuo. Multiprocessor synthesis for periodic hard real-time tasks under a given energy constraint. In Proceedings of the 43rd ACM/IEEE Conference on Design Automation Conference, 2006.

[HCK06b] Chia-Mei Hung, Jian-Jia Chen, and Tei-Wei Kuo. Energy-efficient real-time task scheduling for a dvs system with a non-dvs processing element. In Proceedings of the 27th IEEE Real-Time Systems Symposium, 2006.

[HK05] Pao-Ann Hsiung and Hsin-Chieh Kao. Device-centric low-power scheduling for real-time embedded systems. International Journal of Software Engineering and Knowledge Engineering, 15(2):461–466, 2005.

[HL94] Rhan Ha and J. W S Liu. Validating timing constraints in multiprocessor and distributed real-time systems. In Proceedings of the 14thInternational Conference on Distributed Computing Systems, pages 162–171, 1994.

[HQ11] Huang Huang and Gang Quan. Leakage aware energy minimization for real-time systems under the maximum temperature constraint. In Proceedings of the 48th ACM/IEEE Conference on Design Automation Conference, pages 479–484, 2011.

[HSC+09] Kai Huang, Luca Santinelli, Jian-Jia Chen, Lothar Thiele, and Giorgio C. Buttazzo. Adaptive dynamic power management for hard real-time systems. In Proceedings of the 30th IEEE Real-Time Systems Symposium, pages 23–32, Washington, DC, USA, 2009. IEEE Computer Society.

[HSC+11] Kai Huang, Luca Santinelli, Jian-Jia Chen, Lothar Thiele, and Giorgio C. Buttazzo. Applying real-time interface and calculus for dynamic power management in hard real-time systems. Journal of Real–Time Systems, 47(2):163–193, 2011.

[HTC07] Tai-Yi Huang, Yu-Che Tsai, and E.T.-H. Chu. A near-optimal solution for the heterogeneous multi-processor single-level voltage setup problem. In Proceedings of the IEEEInternational Symposium on Parallel & Distributed Processing 2007, pages 1–10, 2007.

[Hu10] C. Hu. Modern Semiconductor Devices for Integrated Circuits. Prentice Hall, 2010.

[IF00] D. Isovic and G. Fohler. Efficient scheduling of sporadic, aperiodic, and periodic tasks with complex constraints. In Real-Time Systems Symposium, 2000. Proceedings. The 21st IEEE, pages 207–216, 2000.

[Inf] Infineon. AURIX™ Family – TC27xT. http://www.infineon.com/cms/en/product/microcontroller/32-bit-tricore-tm-microcontroller/aurix-tm-family/channel.html?channel=db3a30433727a44301372b2eefbb48d9.

[Ins97] Texas Instruments. CMOS Power Consumption and Cpd Calculation. Texas Instruments Incorporated, scaa035b edition, June 1997. http://www.ti.com/lit/an/scaa035b/scaa035b.pdf.

[ISG07] Sandy Irani, Sandeep Shukla, and Rajesh Gupta. Algorithms for power savings. ACM Transactions on Algorithms, 3(4):41, 2007.

[ITR05] ITRS. International technology roadmap for semiconductors,2005, 2005.

[ITR10] ITRS. International technology roadmap for semiconductors, 2010 update, overview, 2010.

[ITR11] ITRS. International technology roadmap for semiconductors, 2011 edition, design, 2011.

[JCR07] Lei Ju, Samarjit Chakraborty, and Abhik Roychoudhury. Accounting for cache-related preemption delay in dynamic priority schedulability analysis. In Proceedings of the 10th Conference on Design Automation and Test in Europe, Nice, France, April 2007.

[JCS+10] V. Joshi, B. Cline, D. Sylvester, D. Blaauw, and K. Agarwal. Mechanical stress aware optimization for leakage power reduction. Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on, 29(5):722 –736, may 2010.

[JG04] Ravindra Jejurikar and Rajesh Gupta. Procrastination scheduling in fixed priority real-time systems. In Proceedings of the Conference on Language, Compiler and Tool Support for Embedded Systems'04, pages 57–66, Washington DC, 2004.

[JG05] Ravindra Jejurikar and Rajesh Gupta. Dynamic slack reclamation with procrastination scheduling in real-time embedded systems. In Proceedings of the 42nd Design Automation Conference, pages 111–116, Anaheim, 2005.

[JKC10] Heungjun Jeon, Yong-Bin Kim, and Minsu Choi. Standby leakage power reduction technique for nanoscale cmos vlsi systems. Instrumentation and Measurement, IEEE Transactions on, 59(5):1127 –1133, may 2010.

[JNW10] B. Jacob, S. Ng, and D. Wang. Memory Systems: Cache, DRAM, Disk. Elsevier Science, 2010.

[JPG04] Ravindra Jejurikar, Cristiano Pereira, and Rajesh Gupta. Leakage aware dynamic voltage scaling for real-time embedded systems. In Proceedings of the 41st Design Automation Conference, pages 275–280, San Diego, 2004.

[Kam03] R. Kamal. Embedded systems: architecture, programming and design. McGraw-Hill, 2003.

[KH01] Minyoung Kim and Soonhoi Ha. Hybrid run-time power management technique for real-time embedded system with voltage scalable processor. SIGPLAN Notices., 36(8):11–19, August 2001.

[KKLR11] A. Kandhalu, Junsung Kim, K. Lakshmanan, and R. Rajkumar. Energy-aware partitioned fixed-priority scheduling for chip multi-processors. In Proceedings of the 17th IEEE Conference on Embedded and Real-Time Computing and Applications, volume 1, pages 93 –102, aug. 2011.

[KY08] Shinpei Kato and Nobuyuki Yamasaki. Portioned edf-based scheduling on multiprocessors. In Proceedings of the 8th International Conference on Embedded Software, pages 139–148, New York, NY, USA, 2008. ACM.

[KY09] S. Kato and N. Yamasaki. Semi-partitioned fixed-priority scheduling on multiprocessors. In Proceedings of the 15th IEEE Real-Time and Embedded Technology and Applications Symposium, pages 23–32, April 2009.

[LB05] Caixue Lin and Scott A. Brandt. Improving soft real-time performance through better slack management. In Proceedings of the 26th IEEE Real-Time Systems Symposium, pages 410–421, Miami, FL, USA, December 2005.

[LBC+03] Hai Li, S. Bhunia, Y. Chen, T.N. Vijaykumar, and K. Roy. Deterministic clock gating for microprocessor power reduction. In High-Performance Computer Architecture, 2003. HPCA-9 2003. Proceedings. The Ninth International Symposium on, pages 113 – 122, feb. 2003.

[LBDM00] Yung-Hsiang Lu, Luca Benini, and Giovanni De Micheli. Low-power task scheduling for multiple devices. In Proceedings of the 8th International Workshop on Hardware/Software Codesign, CODES '00, pages 39–43, New York, NY, USA, 2000. ACM.

[LG11] Junyang Lu and Yao Guo. Energy-aware fixed-priority multi-core scheduling for real-time systems. In Proceedings of the 17th IEEE Conference on Embedded and Real-Time Computing and Applications, volume 1, pages 277–281, 2011.

[LHC11] Kai Lampka, Kai Huang, and Jian-Jia Chen. Dynamic counters and the efficient and effective online power management of embedded real-time systems. In Proceedings of the 9th International Conference on Hardware/Software Codesign and System Synthesis, Proceedings of the 9th International Conference on Hardware/Software Codesign and System Synthesis, pages 267–276, 2011.

[LHL05] Weiping Liao, Lei He, and K.M. Lepak. Temperature and supply voltage aware performance and power modeling at microarchitecture level. IEEE Transactions on CAD ICAS, 24(7):1042 – 1053, july 2005.

[LHS+98] Chang-Gun Lee, Joosun Hahn, Yang-Min Seo, Sang Lyne Min, Rhan Ha, Seongsoo Hong, Chang Yun Park, Minsuk Lee, and Chong San Kim. Analysis of cache-related preemption delay in fixed-priority preemtive scheduling. IEEE Transactions on Computers, 47(6):700–713, 1998.

[Liu00] J.W.S. Liu. Real-Time Systems. Prentice Hall, 2000.

[LJ02] Jiong Luo and Niraj K. Jha. Static and dynamic variable voltage scheduling algorithms for real-time heterogeneous distributed embedded systems. In ASP-DAC, 2002.

[LL73] C. Liu and J. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. Journal of the ACM, 20:46–61, 1973.

[LRK03] Yann-Hang Lee, K.P. Reddy, and C.M. Krishna. Scheduling techniques for reducing leakage power in hard real-time systems. In Proceedings of the 15th Euromicro Conference on Real-Time Systems, pages 105–112, Jul. 2003.

[LRL09] K. Lakshmanan, R. Rajkumar, and J.P. Lehoczky. Partitioned fixed-priority preemptive scheduling for multi-core processors. In Proceedings of the 21st Euromicro Conference on Real-Time Systems, pages 239–248, July 2009.

[LSC05] Xiaotao Liu, Prashant Shenoy, and Mark Corner. Chameleon: Application level power management with performance isolation. In Proceedings of the 13th Annual ACM International Conference on Multimedia, MULTIMEDIA '05, pages 839–848, New York, NY, USA, 2005. ACM.

[LSC08] Xiaotao Liu, P. Shenoy, and M.D. Corner. Chameleon: Application-level power management. Mobile Computing, IEEE Transactions on, 7(8):995 –1010, aug. 2008.

[LSD89] John P. Lehoczky, Lui Sha, and Y. Ding. The rate monotonic scheduling algorithm: Exact characterization and average case behavior. In Proceedings of the 10th IEEE Real-Time Systems Symposium, pages 166–171, 1989.

[LW82] Joseph Y.-T. Leung and Jennifer Whitehead. On the complexity of fixed-priority scheduling of periodic, real-time tasks. Performance Evaluation, 2(4):237 – 250, 1982.

[LZ10] Yongpeng Liu and Hong Zhu. A survey of the research on power management techniques for high-performance systems. Software: Practice and Experience, 40(11):943–964, 2010.

[mic11] microSSD(TM). micro Solid State Drive (microSSD) Products, rev. 0.1 edition, 2011. http://americas.micross.com/products-services/extended-temperature-plastic-packages/microssd.stml.

[Mok83a] A. K. Mok. Fundamental design problems of distributed systems for the hard-real-time environment. Technical report, Massachusetts Institute of Technology, Cambridge, MA, USA, 1983.

[Mok83b] Aloysius Ka-Lau Mok. Fundamental design problems of distributed systems for the hard-real-time environment. PhD thesis, Electrical Engineering and Computer Science Dept., Massachusetts Institute of Technology, 1983.

[Moo98] G.E. Moore. Cramming more components onto integrated circuits. Proceedings of the IEEE, 86(1):82–85, 1998.

[MSIGO13] M. Norazizi Sham Mohd Sayuti, Leandro Soares Indrusiak, and Alberto Garcia-Ortiz. An optimisation algorithm for minimising energy dissipation in noc-based hard real-time embedded systems. In Proceedings of the 21st Conference Real-Time and Networked Systems, RTNS '13, pages 3–12, New York, NY, USA, 2013. ACM.

[MSV98] M. Mehendale, S.D. Sherlekar, and G. Venkatesh. Extensions to programmable dsp architectures for reduced power dissipation. In VLSI Design, 1998. Proceedings., 1998 Eleventh International Conference on, pages 37–42, Jan 1998.

[NAP11a] Borislav Nikolic, Muhammad Ali Awan, and Stefan M. Petters. SPARTS: Simulator for power aware and real-time systems, 2011. http://www.cister.isep.ipp.pt/projects/sparts/.

[NAP11b] Borislav Nikolic, Muhammad Ali Awan, and Stefan M. Petters. SPARTS: Simulator for power aware and real-time systems. In Proceedings of the 8th IEEE International Conference on Embedded Software and Systems, pages 999–1004, Changsha, China, November 2011. IEEE.

[Nel11] Vincent Nelis. Energy-aware real-time scheduling in embedded multiprocessor systems. In PhD Dissertation, Universite Libre De Bruxelles, ULB, Belgium, 2010-2011.

[NG09] Vincent Nelis and Joël Goossens. Mora: An energy-aware slack reclamation scheme for scheduling sporadic real-time tasks upon multiprocessor platforms. In Proceedings of the 15th IEEE Conference on Embedded and Real-Time Computing and Applications, pages 210–215, Washington, DC, USA, 2009. IEEE Computer Society.

[NGDN08] V. Nelis, J. Goossens, R. Devillers, and N. Navet. Power-aware real-time scheduling upon identical multiprocessor platforms. In Proceedings of the 2rd IEEEInternational Conference on Sensor Networks, Ubiquitous and Trustworthy Computing, pages 209–216, 2008.

[NMA+12] M. Neukirchner, T. Michaels, P. Axer, S. Quinton, and R. Ernst. Monitoring arbitrary activation patterns in real-time systems. In Proceedings of the 33rd IEEE Real-Time Systems Symposium, pages 293–302, 2012.

[Noe05] T. Noergaard. Embedded Systems Architecture: A Comprehensive Guide for Engineers and Programmers. Electronics & Electrical. Elsevier/Newnes, 2005.

[NQ04] Linwei Niu and Gang Quan. Reducing both dynamic and leakage energy consumption for hard real-time systems. In Proceedings of the International Conference on Compilers, Architecture and Synthesis for Embedded Systems, pages 140–148, Washington DC, USA, 2004. ACM.

[nvi] nvidia. TEGRA K1—THE WORLD'S MOST ADVANCED MOBILE PROCESSOR. http://www.nvidia.com/object/tegra-k1-processor.html.

[NXP13] NXP. TJA1043: High-speed CAN transceiver, rev.3 edition, 2013. http://www.nxp.com/documents/data_sheet/TJA1043.pdf.

[ON 13] ON Semiconductor(TM). NCV7321:Stand-alone Local Interconnect Network Transceiver, rev.11 edition, 2013. http://www.onsemi.com/PowerSolutions/product.do?id=NCV7321.

[OS95] Yingfeng Oh and Sang H. Son. Allocating fixed-priority periodic tasks on multiprocessor systems. Journal of Real–Time Systems, 9(3):207–239, November 1995.

[PB00] Peter Puschner and Alan Burns. Guest editorial: A review of worst-case execution-time analysis. Real-Time Systems, 18(2-3):115–128, 2000.

[PC08] Rodolfo Pellizzoni and Marco Caccamo. M-cash: A real-time resource reclaiming algorithm for multiprocessor platforms. Journal of Real–Time Systems, 40:117–147, 2008.

[PL05] Rodolfo Pellizzoni and Giuseppe Lipari. Feasibility analysis of real-time periodic tasks with offsets. Journal of Real–Time Systems, 30:105–128, 2005.

[PLHE09] Stefan M. Petters, Martin Lawitzky, Ryan Heffernan, and Kevin Elphinstone. Towards real multi-criticality scheduling. In Proceedings of the 15th IEEE Conference on Embedded and Real-Time Computing and Applications, pages 155–164, Beijing, China, August 2009.

[PS01] Padmanabhan Pillai and Kang G. Shin. Real-time dynamic voltage scaling for low-power embedded operating systems. In Proceedings of the 18th ACM Symposium on Operating Systems Principles, October 2001.

[PSSG10] P.R. Panda, B.V.N. Silpa, A. Shrivastava, and K. Gummidipudi. Power-efficient System Design. Springer, 2010.

[QC10] Gang Quan and Vivek Chaturvedi. Feasibility analysis for temperature constraint hard rt periodic tasks. IEEE Transactions on Industrial Informatics, 2010.

[RCN03] J.M. Rabaey, A.P. Chandrakasan, and B. Nikolic. Digital integrated circuits: a design perspective. Prentice Hall electronics and VLSI series. Pearson Education, 2003.

[RGR08] Ahmed Rahni, Emmanuel Grolleau, and Michael Richard. Feasibility analysis of non-concrete real-time transactions with edf assignment priority. In Proceedings of the 16th Conference Real-Time and Networked Systems, page NA, October 2008.

[RMMM03] K. Roy, S. Mukhopadhyay, and H. Mahmoodi-Meimand. Leakage current mechanisms and leakage reduction techniques in deep-submicrometer cmos circuits. Proceedings of the IEEE, 91(2):305 – 327, feb 2003.

[SB02] Anand Srinivasan and Sanjoy Baruah. Deadline-based scheduling of periodic task systems on multiprocessors. Inf. Process. Lett., 84(2):93–98, October 2002.

[SC03] V. Swaminathan and K. Chakrabarty. Energy-conscious, deterministic I/O device scheduling in hard real-time systems. IEEE Transactions on CAD ICAS, 22(7):847 – 858, july 2003.

[SC05] Vishnu Swaminathan and Krishnendu Chakrabarty. Pruning-based, energy-optimal, deterministic I/O device scheduling for hard real-time systems. ACM Transactions on Embedded Computing Systems, 4:141–167, February 2005.

[SCI01] V. Swaminathan, K. Chakrabarty, and S.S. Iyengar. Dynamic I/O power management for hard real-time systems. In Proceedings of the 9th International Symposium on Hardware/Software Codesign, pages 237 –242, 2001.

[Sem] FreeScale Semiconductor. MPC8536E PowerQUICC III Integrated Processor Hardware Specifications. Number: MPC8536EEC,Rev. 5, 09/2011.

[Sil99] Maryline Silly. The edl server for scheduling periodic and soft aperiodic tasks with resource constraints. Journal of Real–Time Systems, 17(1):87–111, 1999.

[SLD12] S. Saha, Ying Lu, and J.S. Deogun. Thermal-constrained energy-aware partitioning for heterogeneous multi-core multiprocessor real-time systems. In Proceedings of the 18th IEEE Conference on Embedded and Real-Time Computing and Applications, pages 41–50, 2012.

[SLSPH09] David C. Snowdon, Etienne Le Sueur, Stefan M. Petters, and Gernot Heiser. Koala: A platform for OS-level power management. In Proceedings of the 4th EuroSys Conference, Nuremberg, Germany, April 2009.

[SMP+10] Luca Santinelli, Mauro Marinoni, Francesco Prosperi, Francesco Esposito, Gianluca Franchino, and Giorgio Buttazzo. Energy-aware packet and task co-scheduling for embedded systems. In Proceedings of the 10th International Conference on Embedded Software, pages 279–288. ACM, 2010.

[SPH07] David C. Snowdon, Stefan M. Petters, and Gernot Heiser. Accurate on-line prediction of processor and memory energy usage under voltage scaling. In Proceedings of the 7th International Conference on Embedded Software, pages 84–93, Salzburg, Austria, October 2007.

[STD94] C.-L. Su, Chi-Ying Tsui, and A.M. Despain. Saving power in the control path of embedded processors. Design Test of Computers, IEEE, 11(4):24–31, Winter 1994.

[TA03] L. Tian and T. Arslan. A genetic algorithm for energy efficient device scheduling in real-time systems. In Evolutionary Computation, 2003. CEC '03. The 2003 Congress on, volume 1, pages 242 – 247 Vol.1, dec. 2003.

[TCN00] L. Thiele, S. Chakraborty, and M. Naedele. Real-time calculus for scheduling hard real-time systems. In Proceedings of the 27th International Symposium on Computer Architecture, volume 4, pages 101 –104 vol.4, 2000.

[Tex] Texas Intrument. OMAP^{TM} 5 Application Processors. http://www.ti.com/lsds/ti/omap-applications-processors/products.page?paramCriteria=no.

[Tex07] Texas Instruments. A True System-on-Chip solution for 2.4 GHz IEEE 802.15.4 / ZigBee(TM), rev. f edition, 2007. http://www.ti.com/product/cc2430.

[TSR+98] V. Tiwari, D. Singh, S. Rajgopal, G. Mehta, R. Patel, and F. Baez. Reducing power in high-performance microprocessors. In Proceedings of the 36th Design Automation Conference, pages 732 –737, june 1998.

[TWS06] Lothar Thiele, Ernesto Wandeler, and Nikolay Stoimenov. Real-time interfaces for composing real-time systems. In Proceedings of the 6th International Conference on Embedded Software, EMSOFT '06, pages 34–43, New York, NY, USA, 2006. ACM.

[VB08] A. Valentian and E. Beigne. Automatic gate biasing of an sccmos power switch achieving maximum leakage reduction and lowering leakage current variability. Solid-State Circuits, IEEE Journal of, 43(7):1688 –1698, july 2008.

[VZG+10] Milena Vratonjić, Matthew Ziegler, GeorgeD. Gristede, Victor Zyuban, Thomas Mitchell, Ee Cho, Chandu Visweswariah, and VojinG. Oklobdzija. A new methodology for power-aware transistor sizing: Free power recovery (fpr). In José Monteiro and René Leuken, editors, Integrated Circuit and System Design. Power and Timing Modeling, Optimization and Simulation, volume 5953 of Lecture Notes in Computer Science, pages 307–316. Springer Berlin Heidelberg, 2010.

[WA11] D. Wolpert and P. Ampadu. Managing Temperature Effects in Nanoscale Adaptive Systems. Springer, 2011.

[WAB10] Shengquan Wang, Youngwoo Ahn, and Riccardo Bettati. Schedulability analysis in hard real-time systems under thermal constraints. Journal of Real–Time Systems, 46:160–188, 2010.

[WB08] Shengquan Wang and Riccardo Bettati. Reactive speed control in temperature-constrained real-time systems. Journal of Real–Time Systems, 39(1-3):73–95, August 2008.

[WCST09] Shengquan Wang, Jian-Jia Chen, Zhenjun Shi, and Lothar Thiele. Energy-efficient speed scheduling for real-time tasks under thermal constraints. In Proceedings of the 15th IEEE Conference on Embedded and Real-Time Computing and Applications, pages 201–209, 2009.

[WEE+08] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenström. The worst-case execution-time problem—overview of methods and survey of tools. ACM Transactions on Embedded Computing Systems, 7(3):36:1–36:53, May 2008.

[WKI+07] R. Watanabe, M. Kondo, M. Imai, H. Nakamura, and T. Nanya. Task scheduling under performance constraints for reducing the energy consumption of the gals multi-processor soc. In Proceedings of the 44th ACM/IEEE Conference on Design Automation Conference, pages 1–6, 2007.

[WLL+11] Yi Wang, Hui Liu, Duo Liu, Zhiwei Qin, Zili Shao, and Edwin H.-M. Sha. Overhead-aware energy optimization for real-time streaming applications on multiprocessor system-on-chip. ACM Trans. Des. Autom. Electron. Syst., 16(2):14:1–14:32, April 2011.

[WRD00] Liqiong Wei, K. Roy, and V.K. De. Low voltage low power cmos design techniques for deep submicron ics. In VLSI Design, 2000. Thirteenth International Conference on, pages 24–29, 2000.

[Wri] John Daintith Edmund Wright. A dictionary of computing.

[YAY+07] Yuri Yasuda, Yutaka Akiyama, Yasushi Yamagata, Yoshiro Goto, and Kiyotaka Imai. Design Methodology of Body-Biasing Scheme for Low Power System LSI With Multi Vth Transistors. IEEE Transactions on Electron Devices, 54:2946–2952, 2007.

[YCKT09] Chuan-Yue Yang, Jian-Jia Chen, Tei-Wei Kuo, and Lothar Thiele. An approximation scheme for energy-efficient scheduling of real-time tasks in heterogeneous multiprocessor systems. In Proceedings of the 46th ACM/IEEE Conference on Design Automation Conference, pages 694–699, 2009.

[YCTK10] Chuan-Yue Yang, Jian-Jia Chen, Lothar Thiele, and Tei-Wei Kuo. Energy-efficient real-time task scheduling with temperature-dependent leakage. In Proceedings of the 47th ACM/IEEE Conference on Design Automation Conference, pages 9–14, 2010.

[YLQ06] Lin Yuan, S. Leventhal, and Gang Qu. Temperature-aware leakage minimization technique for real-time systems. In Proceedings of the International Conference on Computer Aided Design, 2006.

[You82] Stephen J. Young. Real Time Languages: Design and Development. Halsted Press, New York, NY, USA, 1982.

[YP02] Yang Yu and V.K. Prasanna. Power-aware resource allocation for independent tasks in heterogeneous real-time systems. In Parallel and Distributed Systems, pages 341 – 348, 2002.

[ZC02] Fan 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, pages 235 – 245, 2002.

[ZYTT09] Gang Zeng, T. Yokoyama, H. Tomiyama, and H. Takada. Practical energy-aware scheduling for real-time multiprocessor systems. In Proceedings of the 15th IEEE Conference on Embedded and Real-Time Computing and Applications, pages 383–392, 2009.