COST MINIMIZATION ALGORITHMS FOR SCHEDULING PARALLEL, SINGLE-THREADED, HETEROGENEOUS, SPEED-SCALABLE PROCESSORS

by

Rashid Khogali

B.A.Sc., University of Toronto, 2009

A thesis presented to Ryerson University in partial fulfillment of the
requirements for the degree of Master of Applied Science (M.A.Sc.)
in the Program of Electrical and Computer Engineering
Toronto, Ontario, Canada, 2013

Author's Declaration

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I authorize Ryerson University to lend this thesis to other institutions or individuals for the purpose of scholarly research.

I further authorize Ryerson University to reproduce this thesis by photocopying or by other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research.

I understand that my thesis may be made electronically available to the public.

Rashid Khogali

COST MINIMIZATION ALGORITHMS FOR SCHEDULING PARALLEL, SINGLE-THREADED, HETEROGENEOUS, SPEED-SCALABLE PROCESSORS

Rashid Khogali
Master of Applied Science
Electrical and Computer Engineering
Ryerson University, 2013

Abstract

We synthesize online scheduling algorithms to optimally assign a set of arriving heterogeneous tasks to heterogeneous speed-scalable processors under the single threaded computing architecture. By using dynamic speed-scaling, where each processor's speed is able to dynamically change within hardware and software processing constraints, the goal of our algorithms is to minimize the total financial cost (in dollars) of response time and energy consumption (TCRTEC) of the tasks. In our work, the processors are heterogeneous in that they may differ in their hardware specifications with respect to maximum processing rate, power function parameters and energy sources. Tasks are heterogeneous in terms of computation volume, memory and minimum processing requirements. We also consider that the unit price of response time for each task is heterogeneous because the user may be willing to pay higher/lower unit prices for certain tasks, thereby increasing/decreasing their optimum processing rates. We model the overhead loading time incurred when a task is loaded by a given processor prior to its execution and assume it to be heterogeneous as well.

Under the single threaded, single buffered computing architecture, we synthesize the SBDPP algorithm and its two other versions. Its first two versions allow the user to specify the unit price of energy and response time for executing each arriving task. The algorithm's second version extends the functionality of the first by allowing the user or the OS of the computing device to further modify a task's unit price of time or energy in order to achieve a linearly controlled operation point that lies somewhere in the *economy-performanc*e *mode* continuum of a task's execution. The algorithm's third version operates exclusively on the latter. We briefly extend the algorithm and its versions to consider migration, where an unfinished task is paused and resumed on another processor.

The SBDPP algorithm is qualitatively compared against its two other versions. The SBDPP' dispatcher is analytically shown to perform better than the well known Round Robin dispatcher in terms of the TCRTEC performance metric. Through simulations we deduce a relationship between the arrival rate of tasks, number of processors and response time of tasks.

Under the Single threaded, multi-buffered computing architecture we have four contributions that constitute the SMBSPP algorithm. First, we propose a novel task dispatching strategy for assigning the tasks to the processors. Second, we propose a novel preemptive service discipline called Smallest remaining Computation Volume Per unit Price of response Time (SCVPPT) to schedule the tasks on the assigned processor. Third, we propose a dynamic speed-scaling function that explicitly determines the optimum processing rate of each task. Most of the simulations consider both stochastic and deterministic traffic conditions. Our simulation results show that SCVPPT outperforms the two known service disciplines, Shortest Remaining Processing Time (SRPT) and the First Come First Serve (FCFS), in terms of minimizing the TCRTEC performance metric. The results also show that the algorithm's dispatcher drastically outperforms the well known Round Robin dispatcher with cost savings exceeding 100% even when the processors are mildly heterogeneous. Finally, analytical and simulation results show that our speed scaling function performs better than a comparable speed scaling function in current literature.

Under a fixed budget of energy, we synthesize the SMBAD algorithm which uses the micro-economic laws of Supply and Demand (LSD) to heuristically adjust the unit price of energy in order to extend battery life and execute more than 50% of tasks on a single processor (under the single threaded, multi buffered computing architecture). By extending all our multiprocessor algorithms to factor independent (battery) energy sources that is associated with each processor, we analytically show that load balancing effects are induced on heterogeneous parallel processors. This happens when the unit price of energy is adjusted by the battery level of each processor in accordance with LSD. Furthermore, we show that a variation of this load balancing effect also occurs when the heterogeneous processors use a single battery as long as they operate at unconstrained processing rates.

Acknowledgements

I graciously thank my supervisor, Dr. Olivia Das for her genuine support, guidance and encouragement. I deeply appreciate her for granting me the opportunity to carry out formal research under her supervision. I am quite fortunate to have been supervised by a wise, easy going and perspicuous supervisor.

I am grateful to Prof. Venetsanopoulos, Prof. Anpalagan, Dr. Raahemifar, Dr. Yifeng and Prof. Gu for effectively exposing me to valuable graduate courses that provided me with sufficient background to carry out novel research projects. I greatly thank the members of my thesis committee: Prof. Anpalagan, Dr. Yang, and Dr. Jaseemuddin for taking time to review my thesis and for providing constructive feedback.

I thank Bruce Darwin for setting up and troubleshooting the laboratory workstations. I appreciate Ryerson University for financially supporting my efforts through awards and various teaching assistantship opportunities. I also acknowledge the Natural Sciences and Engineering Research Council of Canada (NSERC) for financially supporting publications stemming from this thesis.

I dedicate this thesis to my parents who have sacrificed so much for my siblings and I. To my wonderful mother (Mama) who continues to battle a severe case of type II Diabetes, yet she finds the strength to encourage, advice and assist my family in ways that are immeasurable. Mama, my queen, you are the pulse of my heart. To my father (Baba), an honorable man of great insight, work ethic and generosity. The effort invested in this thesis is but a drop compared to their ocean of devotion.

Last but not least, I cherish my magnificent siblings and friends for always being there and making life delightful.

Contents

Pg

1 Introduction 1

1.1 Motivation 1

1.2 Research Overview 2

1.3 Related works 4

1.4 Thesis Contribution 6

1.5 Thesis Outline 10

2. Background 11

2.1 Introduction 11

2.2 Speed Scaling 11

2.3 PDM (Under Static Speed Scaling) For Single Processors 12

2.3.1 PDM Problem Scenario 13

2.3.2 Competitive Analysis (Relevant to PDM) 14

2.3.3 PDM for Two States 14

2.3.4 PDM for Multiple States 14

2.4 Dynamic Speed Scaling (Single Processors) 15

2.4.1 Competitive Analysis (Relevant to Dynamic Speed Scaling) 16

2.5 Deadline Based Scheduling (Single Processor) 16

2.5.1 Overview of Yao et al's Framework, Algorithms and Related Extensions for Single Processor Systems. 17

2.5.2 Deadline Based Scheduling Constrained Speed (Single Processor) 18

2.6 Minimizing Temperature (Single Processor) 18

2.7 Minimizing Flow time (Single Processor) 19

2.8 Flow Time Plus Energy (FTPE) For Single Processors 19

2.8.1 FTPE - Unweighted 19

2.8.2 FTPE - Fractionally Weighed 21

2.8.3 FTPE - Weighed 22

2.8.4 Multithreading (Processor sharing) Extension 22

2.9 Dynamic Speed scaling Multiprocessor Algorithms **24**

2.9.1 Deadline Based Scheduling for Multiprocessors 24

2.9.2 Flow Time Plus Energy (FTPE) For Multi Processors 25

2.9.3 Flow Time Plus Energy (FTPE) For Heterogeneous Multi Processors 25

2.10 Limitations of Speed Scaling **25**

3.Theoretical Framework: Model and Notation **26**

3.1 A Task **27**

3.2 A User Profile **29**

3.3 A Processing Stream **30**

3.3.1 Stream Processor 31

3.3.2 Memory Queue 32

3.3.3 Parallel Processing Streams 32

3.4 Mobile Hardware Resources **32**

3.4.1 Mobile Hardware Parameters 32

3.4.2 Single or Multiple Energy Sources 33

3.5 Task's Processing Rate and Time Consumption **34**

3.5.1 Modeling Overhead Access Time 34

3.5.2 Modeling Processing Rate and Execution Time 34

3.6 A Task's Energy & Power Consumption **35**

3.7 Description of a Task's Computation Volume upon Execution. **36**

3.8 The Decision Algorithm **39**

3.8.1 Memory, Processing Rate and Energy Constraints 39

3.8.2 The Decision Algorithm 40

3.9 Performance Metrics **41**

3.9.1 Measuring Response Time in a Sequential Process 41

3.9.2 Criticisms of Performance Metrics Used in Current Literature 42

3.9.3 TCRTEC Performance Metric 43

3.9.4 Distinguishing our Model from Dynamic Speed Scaling Models Found in Current Literature (Major Differences) 43

3.9.5 Mapping Our work in Current Literature 44

3.10 Defining Traffic conditions 45

3.11 Conclusions 46

4. Cost Minimization of Scheduling Single-buffered Processors 47

4.1 Introduction 47

4.2 Problem Formulation 49

4.2.1 Processing Streams with Single Buffers 49

4.2.1 A Processing Stream Cost Function ` 51

4.2.3 Minimized Cost Function of the j-th processing stream 52

4.2.4 Minimized Constrained Cost Function of the j-th processing stream 53

4.3 Single-Buffer Decision & Parallel Processing Algorithm (SBDPP) 55

4.4 Calibrating the Ratio of Time and Energy Prices 56

4.4.1 Determining a Task’s Mode of Operation 58

4.4.2 Minimized Constrained Cost Function Using The Power Sensitivity Factor 60

4.5 Alternative Versions of the SBDPP Algorithm 62

4.5.1 Single Buffer Assisted Decision & Processing Algorithm (SBADPA) 62

4.5.2 Fixed Power Decision & Processing Algorithm (FPDPA) 64

4.5.3 Extending the Algorithms to Allow Migration 64

4.6 Analysis 65

4.6.1 Qualitative Comparison of Algorithms 65

4.6.2 Quantitative Comparison of Algorithm's Dispatcher to Round Robin 66

4.6 Simulations 68

4.6.1 MATLAB Simulations 68

4.6.1 Java Simulations & Insights 70

4.6 Conclusions 70

5. Cost Minimization of Single-threaded, Multi-buffered Processors 73

5.1 Introduction 73

5.2 Problem Formulation 75

5.2.1 Processing Streams with Multiple Buffers 75

5.2.2 The Cost Function of the j-th Processing Stream 76

5.2.3. The Minimized Cost Function of the j-th Processing Stream 78

5.2.4. The Minimized Constrained Cost Function of the j-th Processing Stream 81

5.3 Algorithms Description 82

5.3.1 SMBSPP Algorithm's Dispatcher (MMCVITPS) 82

5.3.2 SMBSPP Algorithm's Service Discipline/Policy (SCVPPT) 82

5.3.3 Single-threading Multi-buffer Scheduling & Processing Algorithm 83

5.4 Analytical Demonstration 85

5.4.1 A SMBSPP Robustness: Handles Dynamic Inclusion of Tasks 85

5.5 Simulations 88

5.5.1. Performance Metrics 88

5.5.2 Simulation I: Sensitivity of SMBSPP Alg. To Inter-arrival Periods 88

5.5.3 Simulation II: Comparing SMBSPP Algorithm's Dispatcher Versus Round Robin Dispatcher under FCFS, SRPT and SCVPPT Service Disciplines 91

5.5.4 Simulation III: Evaluating SMBSPP Algorithm's Dispatcher (MMCVITPS) under FCFS, SRPT and SCVPPT Service Disciplines 94

5.6 Comparing the SMBSPP Algorithm's Speed-Scaling Function to a Competitive Speed Scaling Function in Current Literature 95

5.6.1 Analytically Comparing OSTSSF to a Competitive Speed Scaling Function in Current Literature 95

5.6.2 Simulation IV: Comparing SMBSPP Algorithm's Speed-Scaling Function (OSTSSF) to $\tilde{p}(\beta n)^{-1}$ under the SRPT Service Discipline 98

5.7 Conclusions 101

6. Using the Laws of Supply and Demand to Extend Battery Life and Improve Load Balancing 104

6.1 Introduction 104

6.2 Synthesizing the STMBAD Algorithm 106

6.2.1 Introduction 106

6.2.2 Mobile Hardware Resources of A Single Processor 107

6.2.3 Managing the Remaining Battery Energy Percentage 108

6.2.4 Showing How Increased Supply Leads To Lower Price & Vise Versa 110

6.2.5 Problem Formulation 111

6.2.6 Cost Function 112

6.2.6 Minimized Cost Function 113

6.2.7 Minimized Constrained Cost Function 115

6.3 The STMBAD Algorithm 115

6.4 Simulating The STMBAD Algorithm 116

6.4.1 Performance Metrics 116

6.4.2 Simulation I: STMBAD Algorithm's EPARBEP Mode Versus UEP Mode While Processing N Homogenous Tasks 117

6.4.3 Simulation II: STMBAD Algorithm's EPARBEP Mode Versus UEP mode for N Heterogeneous Tasks 122

6.5 Multiple Energy Sources 124

6.5.1 Mobile hardware Parameters of multiple energy sources. 124

6.5.2 Single or Multiple Energy Sources 124

6.5.3 Defining operation modes for multiple energy sources 125

6.6 Extending The SBDPP Algorithm to Include EPARBEB Mode 125

6.6.1 A Processing Stream Cost Function 125

6.6.2 Minimized Constrained Cost Function of the jth processing stream 126

6.6.3 SBDPP Algorithm Under EPARBEP and UEP modes 127

6.7 Extending The SBADPA Algorithm to Include EPARBEB Mode 129

6.8 Extending The SMBSPP Algorithm to Include EPARBEB Mode 131

6.8.1 The Minimized Constrained Cost Function of the j-th Processing Stream under EPARBEP 131

6.8.2 SMBSPP Algorithm Under EPARBEP and UEP Modes 132

6.9 Effects of the EPARBEB and UEP Modes on the Speed Scaling Functions and Dispatchers of the Algorithms 134

6.9.1 Effects of the EPARBEB and UEP Modes on the Speed Scaling Functions of the Algorithms 134

6.9.2 Effects of the EPARBEB and UEP Modes on the Dispatchers of the Algorithms 136

6.10 Conclusion 141

7. Conclusion 144

7.1 Research Summary 144

7.1.1 Theoretical Framework 145

7.1.2 Single-buffered Processors 145

7.1.3 Multi-buffered Processors 146

7.1.4 Laws of Supply & Demand and Energy Sources 148

7.2 Research Limitations 150

7.2.1 Algorithmic Overhead 150

7.2.2 Overhead Energy 151

7.2.3 Scope of Analysis 151

7.2.4 System Calibration 152

7.3 Future Research 152

7.4 Closing Remarks 153

Bibliography 154

Appendices 162

Appendix I: Initial Modeling of A Task's Energy & Power Consumption 162

Appendix II: Calibrating the Ratio of Time and Energy Prices under EPARBEP Mode 163

Appendix III: Determining a Task's Mode of Operation with EPARBEP mode 166

List of Tables

Table 3.1: Other hardware parameters of the computing device 33

Table 3.2: Energy, power and execution time incurred (example scenario) 39

Table 4.1: Interpretation of power sensitivity factor 60

Table 4.2: Qualitative comparison of algorithms 65

Table 4.3: Dispatcher cost savings: SBDPP algorithm versus TEST 67

Table 5.1: Performance metrics 88

Table 5.2: Interpretation of inter-arrival periods 89

Table 6.1: Hardware parameters of a mobile device with a single processor 108

Table 6.2: Performance metrics 117

Table 6.3: Multiple energy sources 124

Table 6.4: Load balancing effect on dispatchers by EPARBEP modes 141

List of Figures

Fig. 1.1: Thesis outline 10

Fig. 2.1: Overview of speed scaling problems (an algorithmic perspective) 12

Fig. 2.2: Single processor literature review and research gap 23

Fig. 3.1: Interpretation and possible implementation of a User Profile 30

Fig. 3.2: Processing Stream under (a) Single-threading and (b) Multithreading computing architectures 31

Fig. 3.3: An example describing the remaining computation volume of a task during execution over a finite number of intervals 38

Fig. 3.4: Memory, processing rate and energy constraints 40

Fig. 3.5: Placing our problem relative to the single processor problems in literature 44

Fig. 4.1: Illustrating the parallel single-buffer scenario 50

Fig. 4.2: A task's operating mode and optimum processing rate as a function of user-defined unit prices 57

Fig. 4.3. Illustrating linear calibration of a task's operation mode by utilizing the processor's power consumption during execution 59

Fig. 4.4: Dispatcher cost savings: SBDPP algorithm versus TEST 68

Fig. 4.5. MATLAB GUI simulation validating all three algorithms 69

Fig. 5.1: The parallel multi-buffer scenario 76

Fig. 5.2: Example demonstrating how SMBSPP robustly handles dynamic inclusion of tasks 86

Fig. 5.3: Time analysis of the processor as it executes each of the two tasks in the example 87

Fig. 5.4: Average execution time for N homogeneous Tasks: effect of deterministic arrival periods (μ) 90

Fig. 5.5: Average cost of response time & energy consumption for N homogeneous tasks: Effect of deterministic arrival periods (μ) 90

Fig. 5.6: Average cost of response time & energy consumption versus average cost of system time & energy consumption for N homogenous tasks with Poisson arrivals (heavy traffic) 91

Fig. 5.7: MMCVITPS Versus Round Robin for N heterogeneous tasks with Poisson arrivals (heavy traffic) and Gaussian dist. CV; heterogeneous unit prices of response time under FCFS Service Discipline 92

Fig. 5.8: MMCVITPS Versus Round Robin for N heterogeneous tasks with Poisson arrivals (heavy traffic) and Gaussian dist. CV; heterogeneous unit prices of response time all under SRPT Service Discipline 93

Fig. 5.9: MMCVITPS Versus Round Robin for N heterogeneous tasks with Poisson arrivals (heavy traffic) and Gaussian dist. CV; heterogeneous unit prices of response time all under SCVPPT Service Discipline 93

Fig. 5.10: MMCVITPS Versus Round Robin for N homogeneous tasks under three main deterministic arrival periods with homogeneous unit prices of response time 94

Fig. 5.11: MMCVITPS Dispatcher performance under SCVPPT, SRPT and FCFS service disciplines for N heterogeneous tasks; Poisson arrivals; extreme traffic; Gaussian Distributed CV and response time pricing 95

Fig. 5.12: Constant Correction Factor between $\tilde{p}(\beta n)^{-1}$ and OSTSSF 98

Fig. 5.13: OSTSSF versus $\tilde{p}(\beta n)^{-1}$ for α = 1.01 99

Fig. 5.14: OSTSSF versus $\tilde{p}(\beta n)^{-1}$ for α = 1.25 99

Fig. 5.15: OSTSSF versus $\tilde{p}(\beta n)^{-1}$ for α = 1.5 99

Fig. 5.16: OSTSSF versus $\tilde{p}(\beta n)^{-1}$ for α = 1.75 99

Fig. 5.17: OSTSSF versus $\tilde{p}(\beta n)^{-1}$ for α = 2.25 100

Fig. 5.18: OSTSSF versus $\tilde{p}(\beta n)^{-1}$ for α = 2.5 100

Fig. 5.19: OSTSSF versus $\tilde{p}(\beta n)^{-1}$ for α = 2.75 100

Fig. 5.20: OSTSSF versus $\tilde{p}(\beta n)^{-1}$ for α = 3.0 100

Fig. 6.1:Remaining battery energy percentage of an iPhone 5 109

Fig. 6.2: Increased supply of a commodity leads to lower price 110

Fig. 6.3: Decreased supply of a commodity leads to higher price 111

Fig. 6.4: Remaining battery energy percentage $\varepsilon_{\%}$ after executing N tasks 118

Fig. 6.5: Average execution time of executing N homogeneous tasks 119

Fig. 6.6: Average response time for N homogeneous tasks 120

Fig. 6.7: Average energy consumption for executing N homogeneous tasks 121

Fig. 6.8: Average cost of response time and energy consumption for executing N homogeneous tasks 121

Fig. 6.9: Average cost of execution time and energy consumption for executing N homogeneous tasks 122

Fig. 6.10: Average cost of response time and energy consumption for executing N heterogeneous tasks (Gaussian distributed computation volumes) 123

Fig. 6.11: Average cost of execution time and energy consumption for executing N heterogeneous tasks (Gaussian distributed computation volumes) 123

Fig. 6.12: Attenuation factor induced by the EPARBEP mode on speed scaling functions 135

Fig. 6.13: Dilation factor induced by the EPARBEP mode on dispatchers under unconstrained processing rates 138

Fig. 6.14: Contour diagram of Figure 6.13 with a superimposed example 139

Fig. 6.15: Dilation factor induced by the EPARBEP mode on dispatchers under constrained processing rates 140

Fig. A1: A task's operating mode and optimum processing rate as a function of user-defined (time/energy) unit prices under EPARBEP mode 164

Fig. A.2: linear calibration of a task's operation mode using the processor's power consumption during execution under EPARBEP mode 167

List of Abbreviations

ACPI	**A**dvanced **C**onfiguration and **P**ower **I**nterface architecture
AMD	**A**dvanced **M**icro **D**evices: *a technology company*
AR	**A**verage **R**ate: *an algorithm*
BAL	**B**ampis, **A**ngel and **L**etios: *an algorithm*
BKP	**B**ansal, **K**imbrel and **P**ruhs: *an algorithm*
BPS	**B**ansal, **P**ruhs and **S**tein: *an algorithm*
CMOS	**C**omplementary **M**etal **O**xide **S**emiconductor
CPU	**C**entral **P**rocessing **U**nit
EDF	**E**arliest **D**eadline **F**irst: *a service discipline/policy*
EPARBEP	**E**nergy **P**rice **A**ffected by **R**emaining **E**nergy **P**ercentage: *a proposed operation mode in computing*
FCFS	**F**irst **C**ome **F**irst **S**erve: *a well known service discipline*
FPDPA	**F**ixed **P**ower **D**ecision & **P**rocessing **A**lgorithm: *a proposed algorithm*
FTPE	**F**low **T**ime **P**lus **E**nergy: *a dynamic speed scaling problem*
GUI	**G**raphical **U**ser **I**nterface
HDF	**H**ighest **D**ensity **F**irst: *a service discipline/policy*
HMO	**H**orizontal **M**igratory **O**peration: *a proposed operation to tackle migration*
IBM	**I**nternational **B**usiness **M**achines **C**orporation: *a technology company*
LSD	**L**aws of **S**upply and **D**emand (micro-economic)
MATLAB	**MAT**trix **LAB**oratory: *a numerical computing environment*
MMCVITPS	**M**inimum among **M**inimized **C**osts of **V**irtually **I**ntroducing the **T**ask to each **P**rocessing **S**tream: *a proposed dispatcher*
OA	**O**ptimum **A**vailable: *an algorithm*
OS	**O**perating **S**oftware
OSTSSF	**O**ptimum **S**ingle-**T**hreading **S**peed **S**caling **F**unction: *a proposed speed scaling function*
PDM	**P**ower **D**own **M**echanisms
PS	**P**rocessor **S**haring: *a well known computing architecture*
SBADPA	**S**ingle-**B**uffer **A**ssisted **D**ecision & **P**rocessing **A**lgorithm: *a proposed algorithm*

SBDPP	**S**ingle-**B**uffer **D**ecision & **P**arallel **P**rocessing: *a proposed algorithm*
SCVPPT	**S**mallest remaining **C**omputation **V**olume **P**er unit **P**rice of response **T**ime: *a proposed service discipline*
SMBSPP	**S**ingle-threading **M**ulti-**B**uffer **S**cheduling & **P**arallel **P**rocessing: *a proposed algorithm*
SRPT	**S**hortest **R**emaining **P**rocessing **T**ime: *a well known service discipline*
ST	**S**ystem **T**ime: *a performance metric*
STMBAD	**S**ingle-**T**hreading **M**ulti **B**uffer **A**djusted **D**ynamic speed-scaling algorithm: *a proposed algorithm*
TCRTEC	**T**otal **C**ost of **R**esponse **T**ime and **E**nergy **C**onsumption in dollars: *a proposed performance metric*
TCRTEC/N	Average **C**ost of **R**esponse **T**ime and **E**nergy **C**onsumption for executing **N** tasks: *a performance metric*
TEC	**T**otal **E**nergy **C**onsumption: *a performance metric*
TEC/N	**A**verage **E**nergy **C**onsumption for executing **N** tasks: *a performance metric*
TET	**T**otal **E**xecution **T**ime: *a performance metric*
TET/N	Average **E**xecution **T**ime of executing **N** tasks: *a performance metric*
TRT	**T**otal **R**esponse **T**ime: *a performance metric*
TRT/N	Average **R**esponse **T**ime of **N** tasks: *a performance metric*
TSSC	**T**otal cost of **S**ystem time and energy **C**onsumption: *a performance metric*
TSSC/N	Average cost of **S**ystem time and energy **C**onsumption for executing **N** tasks: *a performance metric*
UEP	**U**nadjusted **E**nergy **P**rice: *a proposed operation mode in computing*
XTG	**XTG** Technology: *a consumer electronics manufacturer*
YDS	**Y**ao, **D**emers, **S**henker: *an algorithm*

Chapter 1: Introduction

1.1 Motivation

Energy consumption is a major constraint in today's computing devices. A principal engineer at Google alerts us that in the next few years, power costs could substantially exceed (server) hardware costs under the current trend of performance and power consumption [16]. Portable/mobile computing devices e.g. laptops and mobile phones are a special class of computing devices in that they rely on batteries for energy. In portable computing devices, battery energy is indeed a scarce and essential resource. Desirable user experience, measured by sufficiently fast execution of tasks is equally important. Portable battery life can be extended by higher capacity batteries or through remote execution [55]. On the go, it can also be extended by portable energy restoration devices such as solar panel chargers produced and sold by XTG Technology [67]. An online article suggests that in 2009, Nokia worked on a technology to recharge their cellular phone battery by extracting energy emitted from ambient radio waves [62]. In that same year, another online article reports that Samsung worked to develop a prototype of a solar powered cellular phone [66]. It is evident that energy in portable computing devices is of great concern and companies that design or manufacture portable computing devices invest in battery or energy technology to remain competitive.

From an algorithmic perspective, computing devices can use variable speed processors to regulate the energy consumption and completion time of executing jobs/tasks. Intel, IBM and AMD provide a selection of multiprocessors that are indeed capable of operating at variable speeds. The ability of a processor to operate at a variable speed is known as *dynamic speed scaling*. Dynamic Speed scaling has been used as a strategy to reduce energy consumption [2, 4, 6, 7, 33, 68]. It has been used to manage a processor's temperature and energy consumption [12] as well as to mitigate processor heat failure [49]. Some speed scaling algorithms factor both time and energy consumption of tasks [1, 68].

Contemporary portable computing devices such as the recent versions of mobile phones, Tablets, iPads and gaming consoles (for example, the PSPVita [65]) utilize multiple processors. Multiple parallel processors are mostly used to improve overall processing performance needed for multi-media applications. In the domain of scheduling, considerable attention has been given to single processor architecture [1, 11-13, 15, 47, 56, 58, 68]. Fewer have considered multiprocessors [4, 7, 20, 42, 44]. Although current architectures mostly consist of homogenous collection of processors, several works suggest that future chip architectures would consist of heterogeneous processors e.g. [18, 53]. Gupta et al. [28] further suggest that scheduling heterogeneous processors is substantially more challenging than scheduling homogeneous processors.

This thesis primarily investigates how to (online) schedule arriving heterogeneous tasks to run on multiple, heterogeneous, speed-scalable processors with the goal of minimizing the financial cost of response time and energy consumption of tasks. The tasks are heterogeneous in terms of computation volume, memory and processing requirements. The processors are heterogeneous in terms of their hardware specifications with respect to maximum processing rate, power functions and energy sources. The user or OS is also allowed to dictate the unit price of response time per task so as to influence the priority of tasks. In a later chapter of this thesis, we also allow the unit price of energy for all tasks to be heuristically adjusted by the micro economic laws of demand and supply so as to conserve energy and improve load balancing on heterogeneous processors.

1.2 Research Overview

The energy consumption of a processor is commonly assumed to grow in proportion to s^{α} where s is the processor speed and α is a constant > 1 e.g. [4, 7, 19, 25, 68]. This implies that a high processing speed leads to fast execution, but incurs a high energy consumption. One way to reduce energy consumption is to employ dynamic speed-scaling (e.g. see [13, 69]), where the speed of the processors can be changed dynamically depending on the workload. The aim is to reduce processor speed at times of low workload.

Generally, the goal of any speed-scaled multiprocessor scheduling algorithm is: (i) to minimize the response time given energy as a budget, (e.g. [59]) or (ii) to minimize the energy consumption as long as the task deadlines are not violated [56, 58, 68], or (iii) to optimize a tradeoff between energy consumption and response time [6, 15]. The objective of our work is to synthesize parallel scheduling algorithms that use dynamic speed scaling to minimize the total cost (in terms of dollars) of energy and response time (**TCRTEC**). In our work, the user or OS determines unit price of response time per task. This allows the user to influence the degree of a task's execution in the economy-performance continuum. The user or OS can set the unit price of energy for all tasks depending on the actual unit price of energy in a given geographical region and time of day.

A brief summary of key assumptions made in this thesis are as follows.

- *Multiple heterogeneous processors:* Few speed scaling algorithms factor multiple processors e.g. [2, 4, 7]. Our scheduling algorithms consider heterogeneous processors that may differ in all their hardware specifications with respect to maximum processing rate, power function parameters and energy sources.
- *Heterogeneous tasks:* There are speed scaling algorithms that only consider homogenous tasks, e.g. [1, 11, 59]. We consider heterogeneous jobs/tasks that may differ in computation volume, memory and processing requirements.
- *Online*: Some speed scaling algorithms operate offline e.g. YDS algorithm in [68]. Our algorithms run in real time to schedule incoming heterogeneous tasks to run on heterogeneous processors.
- *Constrained processing rates*: We factor the maximum hardware processing rate of processors and the minimum software processing rate of tasks to regulate the execution of tasks as opposed to deadline based scheduling of tasks. Many speed scaling algorithms [2, 4, 7, 46, 68] utilize deadline based scheduling. Deadline based scheduling is not always practical in general because tasks that run in conventional operating systems such as Windows and Unix do not utilize it, but instead use minimum or recommended processing rates to regulate the smooth execution of a task or application. Although few speed scaling algorithms factor the maximum hardware processing rate e.g. [11, 71], our speed scaling algorithms are the only ones that explicitly factor both hardware and software processing constraints.
- *User or OS determines unit cost of energy and time of a task's execution:* Unlike any speed scaling algorithm, we explicitly factor the input of a user or OS with

respect to determining the unit price time for executing each task. This allows the user to influence the priority of tasks. The user or OS can set the unit price of energy for all tasks depending on the actual unit price of energy in a given geographical region and time of day.

- *Overhead access time of loading tasks:* We have not seen any dynamic speed scaling algorithm explicitly factors the overhead access time of loading and accessing a task by a given processor prior to execution.
- *Multiple energy sources*: Unlike any speed scaling algorithm, our algorithms allow each processor to have its independent energy source. In the future, each processor may have its own energy source to improve reliability and also to increase total energy of the mobile computing device. Our analysis effortlessly considers the single energy source as well.
- *Tasks' unit price of energy adjusted by battery energy level*: Unlike any speed scaling algorithm, we allow the unit price of energy for all tasks to be heuristically adjusted by the device's remaining battery (or batteries) energy level in accordance with the micro-economic laws of supply and demand. This is done so as to conserve energy and additionally done to improve load balancing.

1.3 Related Works

In this section, we provide a concise summary of prior related work that is most relevant to this thesis.

In the past, when energy was not a major concern, the objective of scheduling algorithms was to minimize the total response time (also called flow time) of all tasks where processors were running at fixed speeds (e.g. [10, 57]). The response time is the time elapsed since a task arrives until it is completed.

The study of energy-efficient speed-scaled scheduling was initiated by Yao et al. in [68]. They considered deadline-based scheduling for a single processor where the jobs need to complete by their given deadlines. The goal was to minimize energy consumption. Assuming the processor's power consumption ($P(s)$) is a convex function of processor speed (s), where $P(s) = s^{\alpha}$ for $\alpha > 1$, they considered scheduling a sequence of tasks on a single variable speed processor. Each task has a required deadline, release time and processing volume (analogous to the number of CPU cycles required to execute a task). They allow pre-emption, where a task is allowed to resume on the same processor after being interrupted. They proposed an optimal offline algorithm (YDS) to solve the task scheduling problem in polynomial time. In the same work, they further introduced two online algorithms, namely, Optimum Available (OA) and Average Rate (AR). They proved that AR has an energy competitive ratio of $(2\alpha)^{\alpha}/2$. Bansal, Kimbrel and Pruhs [12] worked on OA and proved it to have an energy competitive ratio of exactly α^{α}. To solve for multiprocessor case, Angel et al. [7] considered the problem of scheduling a set of tasks with deadlines, release dates and processing requirements, on parallel (speed scalable) processors so as to minimize the total energy consumption. They considered migration where a task is allowed to resume its execution on a different processor. They also allowed pre-emption. They name their optimal scheduling algorithm BAL which has a time complexity of $O(nf(n)\log U)$ where, n is the number of jobs, $f(|V|)$ is the computational complexity of solving a maximum flow in a layered graph with $O|V|$vertices and U is the range of all processor speed values divided by the targeted accuracy. Independently, Albers et al. [2] considered the same multiprocessor speed scaling problem with migration, and obtained an optimal scheduling algorithm that is fully combinatorial and has a time complexity of $O(n^2 f(n))$. Angel et al. [7] compared their BAL algorithm to the one of Albers et al. [2] and stated that when the target precision is sufficiently high, the algorithm of Albers et al. [2] is superior to BAL, otherwise if the target accuracy is relaxed, BAL's algorithm is indeed superior.

Among energy efficient scheduling algorithms, several studies have considered minimizing the response time of jobs given a set energy budget (e.g. [59]). In particular, Pruhs et al. [59] considered offline scheduling to minimize the average response time on a single processor, for a given amount of energy. They gave a polynomial time optimal algorithm for the special case when jobs are of unit size.

To better understand the tradeoff between response time and energy, Albers and Fujiwara [1] proposed minimizing the sum of total response time and energy for a single processor. They presented an online algorithm that is $8.3e\left(\frac{3+\sqrt{5}}{2}\right)^{\alpha}$ competitive for jobs of unit size. This result was improved by Bansal et al. [15] who showed that this algorithm is 4-competitive. Bansal et al. [15] also gave the first constant competitive algorithm for arbitrary size jobs. The multiprocessor case was first discussed by Bunde [20] that presented an offline approximation algorithm for unit size jobs. However, Lam et al. [44] presented the first constant competitive online algorithm for arbitrary job sizes. In [44], jobs are clustered and then round robin dispatched to the processors independently for each cluster. Then they apply the BPS online algorithm given by Bansal et al. [14-15] to each processor.

In this thesis, we present online (dynamic speed scaling) scheduling algorithms that minimizes the financial cost of response time plus energy for the heterogeneous multiprocessor case.

1.4 Thesis Contribution

The contributions of the thesis are as follows:

1. Propose a theoretical frame work to tackle the problem of dynamic speed scaling in a parallel heterogeneous processing environment. We do so by carrying out the following.
   a) Define and describe a task, its computation volume and minimum processing rate.
   b) Define and describe a *user profile*.
   c) Define and describe a *processing stream* under different computing architectures and briefly describe *parallel processing streams*.
   d) Define relevant *mobile hardware resource parameters* and describing how our framework handles single and multiple energy sources.
   e) Model the *overhead access time* and describing the theoretical processing rate and execution time of a task.
   f) Use formulas in current literature to deduce useful relationships pertaining to a task's computation volume, energy and power consumption.
   g) Analytically and graphically illustrate the effect of processing on a task's remaining computation volume as well as the energy and power consumed.

h) Describe the decision algorithm and summarizing relevant pre-processing constraints.
i) Justify the constituents of our target performance metric and briefly critique other performance metrics used in current literature.
j) Distinguish our model from other relevant models found in current literature and map our contributions in current literature.
k) Define traffic conditions to systematically analyze and simulate our algorithms.

2. Present the first, elaborate, analytical study on the use of dynamic speed scaling to schedule heterogeneous tasks on single-buffered, heterogeneous, parallel processors with the objective of reducing the total cost of response time and energy consumption.
We accomplish this by carrying out the following.

a) Use our theoretical framework to formulate the problem and to synthesize the "*Single-Buffer Decision & Parallel Processing* (SBDPP)" algorithm.
b) Achieve a linear calibration of a task's operation mode as a function of the (user-specified) unit prices of time and energy.
c) Construct and present two other versions of the SBDPP algorithm, namely *"Single Buffer Assisted Decision & Processing Algorithm (SBADPA)"* and *"Fixed Power Decision & Processing Algorithm (FPDPA)"*.
d) Briefly describe how the SBDPP algorithm handles migration.
e) Qualitatively compare the three versions of the SBDPP algorithm to each other.
f) Analytically show that the dispatcher of the SBDPP algorithm outperforms the Round Robin dispatcher under minimal traffic conditions.
g) Develop a MATLAB Graphical User Interface program to simulate the SBDPP, SBADPA and FPDPA algorithms and also validate the algorithms via discrete time based simulations written in Java.
h) Use the simulations to deduce a relationship between the arrival rate of tasks, number of processors and response time of tasks under the (parallel) single buffered computing architecture.
i) Provide insights on the limitations of the parallel single buffered computing architecture.

3. Study the use of dynamic speed scaling to schedule heterogeneous tasks on multi-buffered, heterogeneous, parallel processors with the objective of reducing the total cost of response time and energy consumption (TCRTEC) of tasks.

We achieve this by carrying out the following.

a) Synthesize and present the "*Single-threading Multi-Buffer Scheduling & Parallel Processing* (SMBSPP)" algorithm.

b) Present the (SMBSPP) algorithm's dispatcher which assigns heterogeneous tasks to a given heterogeneous processors.

c) Present the (SMBSPP) algorithm's dynamic speed-scaling function, which we name, "*Optimum Single-Threading Speed Scaling Function*" (OSTSSF).

d) Present the (SMBSPP) algorithm's service discipline which we name the "*Smallest remaining Computation Volume Per unit Price of response Time* (SCVPPT)".

e) Use a variety of performance metrics to validate the functionality of the SMBSPP algorithm by conducting discrete time based simulations written in Java (as well as analytical techniques).

f) Use simulations to show that our MMCVITPS dispatcher works well with heterogeneous processors and drastically outperforms the classic Round Robin dispatcher with cost savings exceeding 100% on average even when processors are mildly heterogeneous. This was done under various deterministic and stochastic traffic conditions.

g) Show that our SCVPPT scheduling discipline outperforms the two known service disciplines, Shortest Remaining Processing Time (SRPT) and the First Come First Serve (FCFS), in terms of minimizing the TCRTEC performance metric.

h) Analytically compare our dynamic speed scaling function (OSTSSF) to a comparable and most competitive speed scaling function found in current literature.

i) Corroborate this analytical comparison with elaborate simulations (written in Java) to show that our OSTSSF out performs this competitive speed scaling function in terms of the TCRTEC performance metric.

j) Offer a recommendation to improve the most competitive speed scaling function found in current literature in terms of minimizing the TCRTEC performance metric.

4. Use our theoretical framework and the Laws of Supply and Demand (LSD) to heuristically adjust the unit price of energy, extend battery life and improve load balancing in speed scalable processors of a mobile computing device. We do so by carrying out the following.

a) Use LSD to heuristically adjust the unit price of energy of tasks via the remaining energy percentage parameter.

b) Use the remaining energy percentage parameter and our theoretical framework to synthesize an online single processor (multi-buffered) speed-scaling algorithm (*Single-Threading Multi Buffer Adjusted Dynamic speed scaling* algorithm STMBAD).

c) Use discrete time based simulations (written in Java) to show that when the STMBAD algorithm factors the remaining energy percentage parameter, it completes more than 50% more jobs for both homogenous and heterogeneous tasks and ultimately allows the mobile computing device to last longer on the go.

d) Implement the remaining energy percentage parameter in the speed scaling functions of all algorithms presented in this thesis to analytically show that it is a heuristic controller that rations battery energy by slowing down the speed scaling functions of our algorithms (as the battery depletes).

e) Integrate the remaining energy percentage parameter to the dispatchers of all algorithms presented in this thesis to analytically show that it is a heuristic controller that induces load balancing when each heterogeneous processor has its independent energy source.

f) Shed light on the difference between optimum and robust speed scaling algorithms (speed scaling functions and coupled dispatchers) in the context of scheduling and processing heterogeneous tasks by heterogeneous processors with the goal of reducing response time and adjusted energy consumption.

Preliminary components of this thesis were peer-reviewed and accepted for publication in [39-41].

1.5 Thesis Outline

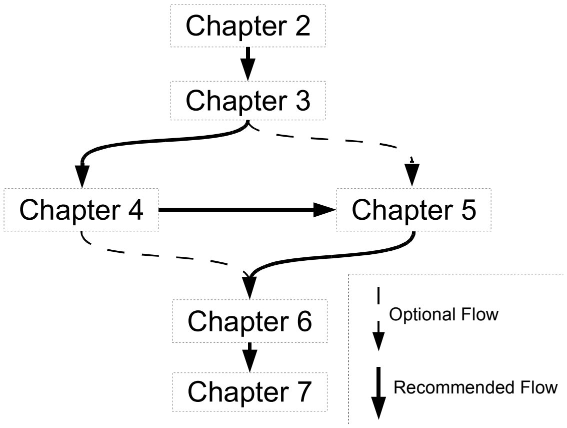


Fig. 1.1: Thesis outline

Chapter 2 provides a background of the relevant definitions, principles and models found in current literature that are pertinent to speed scaling. In chapter 3, we propose a theoretical frame work to tackle the problem of dynamic speed scaling in a parallel heterogeneous processing environment. This framework is used in all subsequent chapters of this thesis. In chapter 4, we present the first, elaborate, analytical study on the use of dynamic speed scaling to schedule heterogeneous tasks on single-buffered, heterogeneous, parallel processors with the objective of minimizing the total cost of response time and energy consumption. In Chapter 5 we study the use of dynamic speed scaling to schedule heterogeneous tasks on multi-buffered, heterogeneous, parallel processors (under the single-threaded computing architecture) with the objective of minimizing the total cost of response time and energy consumption of tasks. In Chapter 6, we use our theoretical framework and the *Laws of Supply and Demand* to heuristically adjust the unit price of energy, extend battery life and improve load balancing in speed scalable processors of a mobile computing device. Lastly, In chapter 7, we summarize the critical findings presented in this thesis, discuss the limitations of our findings, highlight interesting opportunities for future work and offer closing remarks.

2. Background

2.1 Introduction

In this chapter we present a concise overview of speed scaling algorithms that relate to the work in this thesis. Survey papers by Albers, S. [3] and Irani et al. [35] provide elaborated studies of these algorithms. Like much of the work in existing literature, this thesis concentrates on the system and device level to formulate and solve problems through an algorithmic perspective.

2.2 Speed Scaling

In existing literature, there are two types of speed scaling, Static and Dynamic speed scaling [6]. Static speed scaling can either involve two states or multiple states. A state is a discrete operation frequency or speed that a processor attains to consume some fixed power consumption. Static speed scaling is used to solve problems of *Power Down Mechanisms*. Dynamic speed scaling allows the processor to manipulate the entire speed/frequency spectrum. From an algorithmic perspective, dynamic speed scaling is used to solve four main problems[1]. They are as follows.

- Deadline Based Scheduling
- Minimizing Temperature
- Minimizing Flow Time
- (Minimizing) Flow Time Plus Energy

In subsequent sections we briefly go through the above-mentioned problems.

[1] There is a problem known as *Makespan Minimization* that is related to deadline based scheduling problem. Although we do not discuss it in this thesis, researchers such as [20] and [59] have solved the problem in single and multiprocessor environments. The *makespan* is the point in time where a schedule ends [3].

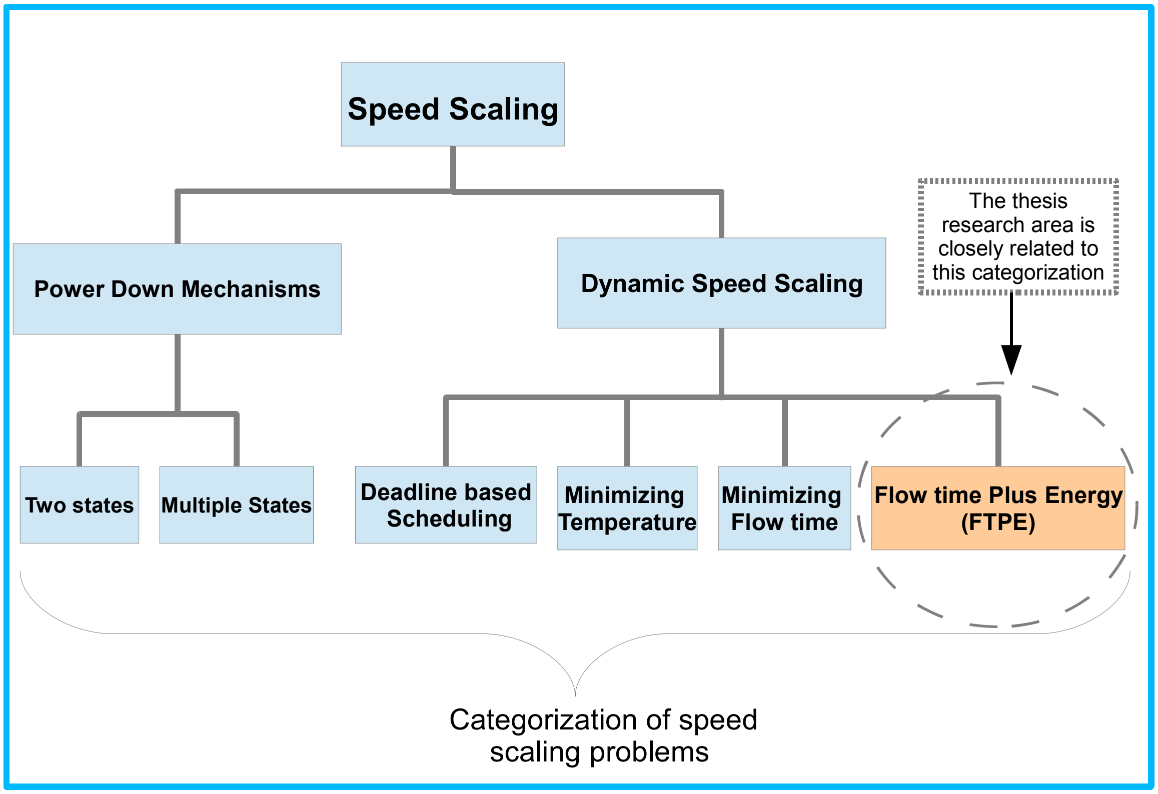

Speed Scaling
Power Down Mechanisms
Dynamic Speed Scaling
The thesis research area is closely related to this categorization
Two states
Multiple States
Deadline based Scheduling
Minimizing Temperature
Minimizing Flow time
Flow time Plus Energy (FTPE)
Categorization of speed scaling problems

Fig. 2.1: Overview of speed scaling problems (an algorithmic perspective)

2.3 PDM (Under Static Speed Scaling) For Single Processors

Power Down Mechanisms (PDM) is an omni present strategy to manage energy in computing devices, for instance we see that laptops switch between *off*, *sleep* and *awake* states to conserve energy [3]. Also, desktops running operating systems such as Linux *Ubuntu* or Windows *XP*, *7* etc. deactivate their monitor and/or cut off power to some other external peripherals when the computer has been inactive for a while. The idea is to temporarily switch off the computing device through a *sleep* state when (computing) service is expected to resume in the near future or to shut down the device (*off* state) when service is not required any time soon and lastly, to maintain an active or *awake* state when the device is actively computing. In practice, computing devices consume some energy while in *sleep* state because they need to provide power to their Random-access memory which stores the memory settings of an *awake* state prior to the *sleep* state [64]. These states are managed by the operating software of the computing device. The most essential parameter in PDM techniques is the *idleness threshold*, the overhead time interval required for the computing device to switch from an active state to a sleep state [35].

Power down mechanisms still dominate industry products because they mitigate the (processor's) current leaks that stem from the dynamic switching of processing speed [2] [52]. Power down mechanisms have been thoroughly studied by several researchers; from a stochastic perspective (e.g. [22]), an algorithmic perspective (e.g. [34]) as well as a learning-based perspective (e.g. [29]). Also, concentrated research from industry e.g. *Microsoft's Desktop PC Energy Savings for Enterprises* [50] and *Microsoft's Power Management and Driver Support* through *ACPI (Advanced Configuration and Power Interface Architecture*) [51] thoroughly explore and implement PDM. We refer the reader to an elaborate survey by Irani et al. [35]. In this survey, the authors comprehensively examine PDM under various approaches.

Next, we briefly mention PDM from an algorithmic perspective for two and multiple states[3].

2.3.1 PDM Problem Scenario

- The computing device can operate in more than one state e.g. *completely off*, *sleep*, *stand by*, *economy* and *performance* states.
- This is an online problem, implying that the computing device is not aware of future states. Also, for a given idle period, the system has no information when the period ends.
- Each state incurs a different power consumption.
- Energy consumption during *power up* (moving from a state of low power consumption to a state of higher power consumption) is substantial.
- Generally, the energy consumption during a *power down* between any two states is assumed to be insignificant.
- The goal is to minimize energy consumption.

[2] We speculate that this may soon change because leakage power is on the rise [38].

[3] In subsequent sections, we do not attempt to summarize all the algorithms pertaining to PDM because this thesis falls under *Dynamic Speed Scaling* and not PDM. [3, 34, 35, 37] go over PDM strategies in more scope and depth.

- The challenge is as we attempt to minimize energy consumption through sustaining residency in low states, the system is inactive, but the system needs to attain higher power state(s) to compute [3]. Furthermore, a power up will incur an energy penalty and we are also not aware of future states. It may sometimes not be justifiable to greedily reside in a low power state to save energy, just to be interrupted by a request that will lead to a penalty during a power up.

2.3.2 Competitive Analysis (Relevant to PDM)

- Competitive analysis is conducted to give a guarantee of worse case performance [3].
- In competitive analysis, a given algorithm ALG is compared to its optimal offline counterpart or adversary, OPT [60].
- OPT knows all future events, so it has an advantage to minimize energy through computing an offline state transition schedule [3].
- ALG is considered c-competitive for any input (*idle* periods), ALG's energy consumption is c times that of OPT [3].

2.3.3 PDM for Two States

Algorithm ALG-D is a 2 competitive deterministic algorithm that solves the PDM problem for two states [3]. Furthermore, [3] shows that no online deterministic algorithm achieves a competitive ratio lower than 2 for the two state PDM problem.

Algorithm ALG-R is a stochastic algorithm that improves on *Algorithm ALG-D* by using a probability density function to transition to the sleep state from the awake state. It was presented by [37] and was shown to achieve a competitive ratio approaching 1.58.

2.3.4 PDM for Multiple States

Algorithm Lower-Envelope was proposed by Irani et al. [34]. This is a deterministic algorithm that solves the PDM problem for the multi state scenario. The authors assume that the energy incurred during a power up is additive (not arbitrary) and proved that their *algorithm is* 2-competitive [3]. Furthermore, [3] asserts that no online deterministic algorithm achieves a competitive ratio lower than 2 for the multi state PDM problem.

2.4 Dynamic Speed Scaling (Single Processors)

Dynamic speed scaling or *dynamic voltage scaling* is the ability of a processor to operate at a variable speed. This is a relatively recent technique to save energy and achieve decent service by manipulating the full spectrum a processor's frequency (speed) [3]. Examples of modern processors that support dynamic speed scaling are the Intel's *SpeedStep* processor [32], IBM's *Power7* processor[4] [31] and the AMD's *PowerNow* processor[5] [5]. Dynamic Speed scaling has been used as a strategy to reduce energy consumption [2, 4, 6, 7, 33, 68]. It has been used to manage a processor's temperature and energy consumption [12] as well as to mitigate processor heat failure [49]. Some speed scaling algorithms factor both time and energy consumption of tasks [1, 6, 11, 68]. Under dynamic speed scaling, the energy consumption of a processor is commonly assumed to grow in proportion to s^{α} where s is the processor speed and α is a constant > 1 (e.g. [1, 4, 6, 7, 19, 25, 68]). This implies that a high processing speed leads to a fast execution, but unfortunately incurs a high energy consumption. Note that the well known cube-root rule e.g., as suggested by [3, 43] is that $\alpha = 3$ for a CMOS based processor[6]. The cube-root rule stems from the modeling of dynamic power in CMOS chips. According to [38], it is modeled as being proportional to $cv^2 f$, where c is the processor's capacitance, v is the voltage supplied and f is the frequency; but at high frequencies $f \propto v$. Surprisingly, Wierman et al. [63] carried out experiments to show that in today's CMOS based computing devices α is close to quadratic (i.e. they found out that a calibration of $\alpha = 1.8$ is more accurate). We speculate that this discrepancy in α is due to an improvement in technology. Anyhow, In the algorithmic literature pertaining

Abbreviations:

[4] IBM- International Business Machines Corporation.

[5] AMD - Advanced Micro Devices (Technology Company).

[6] CMOS - Complementary Metal Oxide Semiconductor.

to dynamic speed scaling, most researchers use a general $\alpha \in (1,3]$ and some assume the cube-root rule ($\alpha = 3$).

Under the single processor scenario, dynamic speed scaling gives rise to a variety of challenging problems because the scheduler needs to decide on the job/task to execute as well as the speed of processing [3]. Generally, this is more complicated in the multiprocessor environment and is even more challenging when processors are heterogeneous [28]. Typically, from an algorithmic perspective, we have four main problem categorizations that fall under dynamic speed scaling, they are: Deadline Based Scheduling, Minimizing Temperature, Minimizing Flow Time and Minimizing Flow Time Plus Energy. We briefly cover these problems in subsequent sections, but first we briefly touch on competitive analysis in application to dynamic speed scaling.

2.4.1 Competitive Analysis (Relevant to Dynamic Speed Scaling)

From an algorithmic perspective, the *offline* setting is defined in literature as the scenario where we have advance knowledge of jobs/tasks [3]. The online setting is when we have to make scheduling decisions in real time without any advance knowledge of jobs, i.e. we learn about jobs when as they arrive. Online strategies, just like in PDM, are assessed using competitive analysis [3]. An Online dynamic speed scaling algorithm (ALG) is considered c-competitive if for every input, ALG's objective function (usually energy, but could be both energy and response time or some other performance criteria) is c times that of the optimal offline solution/adversary [3].

2.5 Deadline Based Scheduling (Single Processor)

The study of energy-efficient speed-scaled scheduling was initiated in 1995 by Yao et al. [68]. They considered the deadline-based scheduling of a single processor where the jobs need to complete by their given deadlines. Using dynamic speed scaling, the goal was to construct a schedule that minimizes energy consumption. Yao et al.s' deadline based scheduling framework has been the most extensively studied framework in the context of dynamic speed scaling algorithms [3].

2.5.1 Overview of Yao et al's Framework, Algorithms and Related Extensions for Single Processor Systems.

Yao et al. [68] considered scheduling a sequence of tasks on a single variable speed processor (The processor is unbound in the sense that it has no maximum processing rate). Each task has a required deadline, release time and processing volume (analogous to the number of CPU cycles required to execute a task). They allow **preemption**, where a task is allowed to resume on the same processor after being interrupted[7]. They proposed an optimal offline algorithm[8] (YDS) to solve the task scheduling problem in polynomial time via iterations. A direct implementation of the YDS algorithm has a computational complexity of $O(n^3)$, where n is the number of jobs [3]. Li et al. [46] illustrate an alternative implementation of YDS with an improved computational complexity of $O(n^2 \log n)$ based on finding successive approximations of the optimal schedule. Furthermore, when the processor is assumed to have a d number of discrete voltage/speed levels, Li and Yao [47] propose an algorithm that improves the computational complexity of the offline YDS algorithm to $O(dn \log n)$.

In the same work, Yao et al. further introduced two online algorithms, namely, Optimum *Available (OA)* and *Average Rate (AR)*. They proved that AR has an energy competitive ratio[9] of $(2\alpha)^\alpha / 2$. Bansal, Kimbrel and Pruhs [12] worked on OA and proved it to have an energy competitive ratio of exactly α^α. Bansal et al. [13] present an online algorithm[10] (*BKP)* which sort of approximates the speeds of YDS in real time [3]. In the same work,

[7] The YDS algorithm makes use of a **preemptive** service discipline, *Earliest Deadline First (EDF)* service policy. Among the unfinished Jobs, this well-known service policy gives priority to jobs with the earliest deadline.

[8] YDS - Yao, Demers, Shenker.

[9] Recall that α is the exponent of a processor's power function (s^α), where s is the processor's speed and α is a constant > 1.

[10] BKP- Bansal, Kimbrel and Pruhs.

Bansal et al. proved that their BKP algorithm achieves an energy competitive ratio that is better than *Optimal Available* for large α values[11], i.e. for $\alpha \geq 5$.

2.5.2 Deadline Based Scheduling Under Maximum Processing Rate Constraints (Single Processor)

Under a constrained processing rate, a summary of the extended deadline based problem and results are as follows.

- The processor is scalable between a speed of zero and some maximum speed *T*.
- The constrained maximum processing rate of T potentially compromises the ability to find a feasible schedule.
- The revised objective is to maximize *throughput* i.e., the total processing volume of tasks that are successfully completed by their deadline.
- [11] give an online algorithm that is constant competitive for the energy consumed and is 4-competitive for throughput.
- [71] present an online constant competitive algorithm on both throughput and energy consumption[12].

What we have mentioned so far is not an exhaustive summary of all work related to deadline based scheduling in the context of dynamic speed scaling. For such work, see a survey paper by Albers [3]. Next, we move over to other dynamic speed scaling problems.

2.6 Minimizing Temperature (Single Processor)

Bansal et al. [13] initiate the study of using dynamic speed scaling to manage temperature, more specifically to simultaneously meet the objectives of maximizing temperature and minimizing energy consumption. These two objectives conflict because processors with high temperature incur high energy consumption. They assume the ambient environment temperature is fixed and the computing device cools according to Isaac Newton's law of

[11] The practical significance of this result is questionable since in conventional processors, $\alpha \in (1,3]$.

[12] [71] was the first to introduce the constrained speed model, where the speed is bounded from zero to *T*.

cooling [21]. They show that their BKP algorithm is O*(1)* competitive for all of the following: maximum speed, maximum temperature, maximum power, and total energy. Also, they interestingly show that algorithm OA (*Optimal Available*) does not achieve a temperature competitive ratio of O*(1)* even though it is known to have an energy competitive ratio of O*(1)*. They also show that Algorithm YDS achieves a constant temperature competitive ratio even though it is not optimal with respect to minimizing the maximum temperature/energy consumption.

2.7 Minimizing Flow time (Single Processor)

Flow time [1] or response time is the time elapsed since a task arrives until it is completed. Among energy efficient scheduling algorithms, several studies have considered minimizing the response time of jobs, given a set energy budget (e.g. [59]). In particular, Pruhs et al. [59] considered offline scheduling to minimize the average response time on a single processor, for a given amount of energy. They gave a polynomial time optimal algorithm for the special case when jobs are of unit size.

2.8 Flow Time Plus Energy (FTPE) For Single Processors

We concentrate more on this problem type because it is closely related to the theme of this thesis. All the FTPE problems are online in that tasks arrive in real time.

2.8.1 FTPE - Unweighted

Albers and Fujiwara [1] consider minimizing the combined objective function[13] (g) of both energy and flow time. In this objective function, the penalty or weight of each job's flow time is not only homogenous but is treated to have the same weight as that of a unit of energy. In other words they consider homogenously *unweighted* response time and energy consumption of jobs. They assume the following.

- They let g be the objective function or target performance metric.
- They let E be the energy consumption of jobs in the schedule.

[13] g is actually a function of s, the processor speed; i.e. $g(s)$.

- They consider a given schedule having n unit-sized jobs.
- They let the i^{th} job have a response time f_i.
- They define $\sum_{i=1}^{n} f_i$ as the flow time or response time of jobs in the schedule.
- Their target performance metric is $g = E + \sum_{i=1}^{n} f_i$

They formulate an online algorithm called algorithm *Phaseball* that processes jobs in phases. A verbatim quote of their algorithm is as follows.

> "**Algorithm**. Phasebal
> If $\alpha < \left(19 + \sqrt{161}\right)/10$ then $\mathrm{c} := \alpha - 1$; otherwise $\mathrm{c} := 1$. Let n_1 be the number of jobs arriving at time $t = 0$ and set $i = 1$. While $\mathrm{n}_1 > 0$, execute the following to steps: (1) For $\mathrm{j} = 1,...,\mathrm{n}_1$, process the j-th using a speed of $\sqrt[\alpha]{\left(\mathrm{n}_1 - j + 1\right)/c}$. We refer to this entire time interval as Phase i. (2) Let $\mathrm{n}_{\mathrm{i}+1}$ be the number of jobs that arrive in Phase i and set $i := i + 1$." [14]

In the same work, [1] showed that their Phaseball algorithm achieves a competitive ratio of $O\left(\left(\frac{3+\sqrt{5}}{2}\right)^{\alpha}\right)$. They also propose another offline algorithm that uses dynamic programming [61] and runs in polynomial time to find schedules for unit sized jobs that have minimal average flow times for all energy levels. Bansal et al. [11] solve the problem of Albers and Fujiwara by presenting an online algorithm that was shown to be 4-competitivein in terms of minimizing the total flow time plus energy for *unweighted* unit sized jobs. This was done[15] under a more realistic constraint where the maximum processing speed of the processor is bounded.

[14] Albers, S. and Fujiwara, H., ACM Transactions on Algorithms, Vol. 3, No. 4, Article 49, Pg.5, Publication date: November 2007.

[15] Bansal et al. [11] maintained the assumption of Albers and Fujiwara in [1] by assuming unit sized jobs.

Bansal et al. [70] improve algorithm *Phaseball* by presenting a 3-competitive speed scaling algorithm. They call this algorithm *Algorithm A*. Algorithm A uses a speed of $\sqrt[\alpha]{n+1}$, where n is the number of active jobs (of any arbitrary size). It also sets the speed of the processor to zero when there are no jobs.

2.8.2 FTPE - Fractionally Weighed

The objective function under this scenario is $g = E + \sum_{i=1}^{n} w_i f_i$

In this objective function, the $\left(\sum_{i=1}^{n} w_i f_i \right)$ term is the *fractional* flow time costs of a jobs.

It weighs each job's response time (f_i) by some weight (w_i). This weight is the remaining fraction of a job (the remaining work divided by the original work).

[11] provide an online preemptive algorithm (BPS) that works with constrained maximum processing speed (*T*). The algorithm operates at a speed of $\min\left\{ w_a(t)^{\frac{1}{\alpha}}, T \right\}$ where *T* is the maximum speed of the processor and $w_a(t)$ is the total remaining fraction of all the active jobs. The remaining fraction of a job is the remaining work divided by the original work. The algorithm was shown to have a competitive ratio of $\left(2 + O(1)\right)\alpha / \ln(\alpha)$. The algorithm uses the HDF (Highest density first) service policy.

The HDF gives highest priority of jobs based on the highest weight to original size ratio [28]. It is an online preemptive service discipline that is optimal for fractional weighed time [11]. Bansal et al. [14] considers a similar algorithm except they relax the maximum processing speed constraint. Their algorithm runs at power equal to the fractional weight of unfinished jobs by using the HDF service discipline. Using amortized local competitiveness, they show that their algorithm is $\left(O(\alpha / \log \alpha)\right)$ competitive with respect to the objective function (fractionally weighed flow time plus energy). Bansal et al. consider arbitrary weights and job sizes in [11, 14].

2.8.3 FTPE - Weighed

Under this scenario, the objective function is similar to the fractional weighted FTPE problem. The objective function (g) is $g = E + \beta\sum_{i=1}^{n} f_i$.

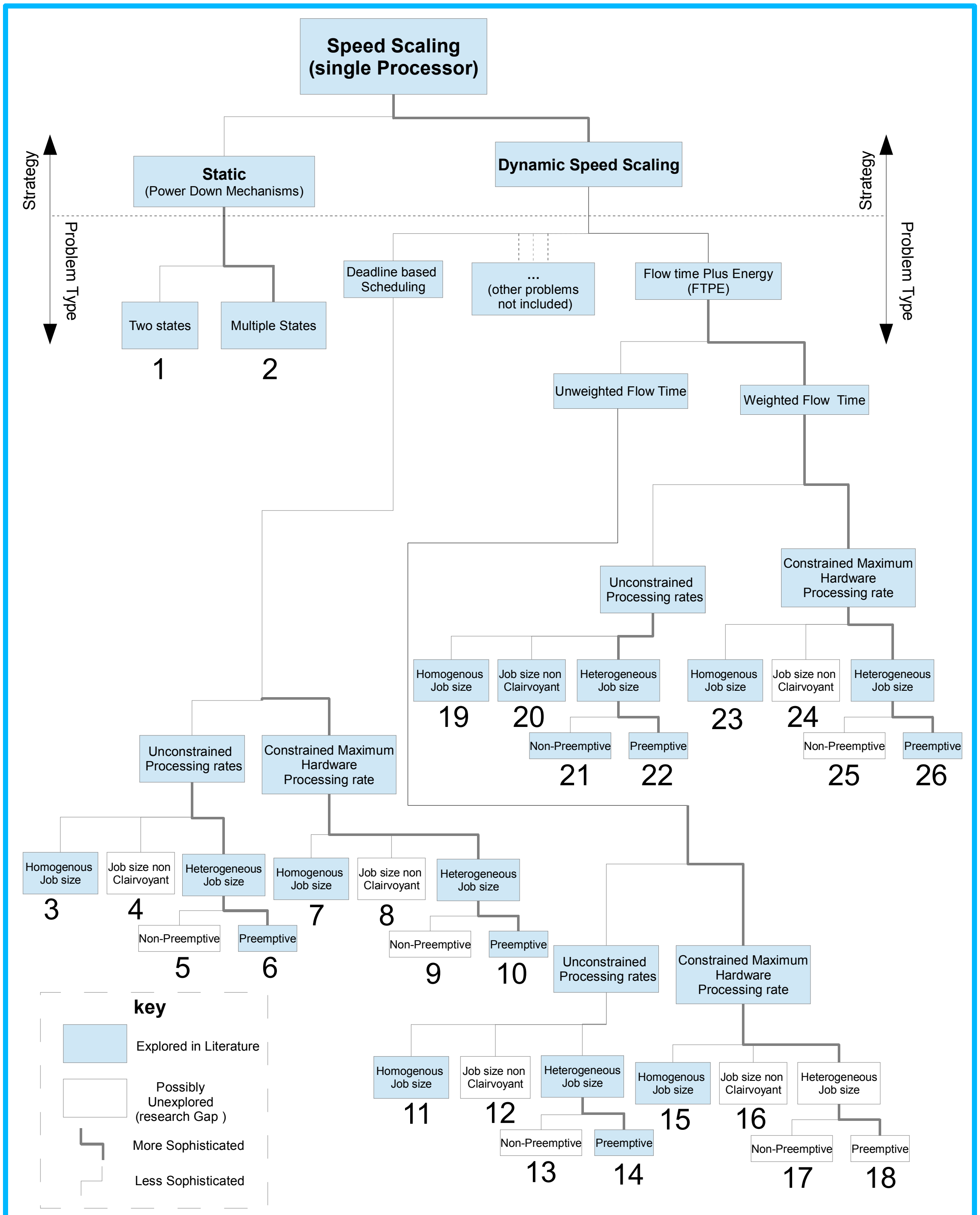
[63] explicitly defines β to be the relative cost of delay. Andrew et al. [6] consider the weighed FTPE problem. They carry out analysis and assert that the online speed scaling function with minimal competitive ratio under the SRPT service discipline is $p(\beta n)^{-1}$, where n is the number of active jobs. The SRPT (Shortest Remaining Processing Time) service discipline schedules tasks according to their least remaining work. In the same work, Andrew et al. show that (dynamic) speed scaling magnifies unfairness under SRPT and also for non preemptive service disciplines. The idea stems from the fact that the $p(\beta n)^{-1}$ speed scaling function[16] favors jobs that happen to be executed when the job occupancy (n) is large and is unfair to jobs that are processed when the occupancy is low.

2.8.4 Multithreading (Processor sharing) Extension

Andrew et al. [6] show that the $p(\beta n)^{-1}$ speed scaling function under Processor sharing (PS) is $O(1)$ competitive.

Wierman et al. [63] stochastically analyze dynamic speed scaling functions under the processor sharing computing architecture. They show that for a system with Poisson arrivals [26] of tasks, which runs at optimal speed under PS achieves a constant competitive ratio.

[16] Speed scaling function is simply the function that describes the speed of a given schedule.



1	[37]	**10**	[11, 47, 71]	**20**	[63]
2	[34]	**11**	[1, 70]	**21**	[6, 63]
3	[13, 68]	**14**	[14]	**22**	[6, 14]
6	[13, 68]	**15**	[11]	**23**	[11]
7	[11, 47, 71]	**19**	[6, 14]	**26**	[11]

Fig. 2.2: Single processor literature review and research gap

2.9 Dynamic Speed scaling Multiprocessor Algorithms

The multiprocessor case was first discussed by Bunde [20]. Bunde presented an offline approximation algorithm for unit size jobs. Bunde solves a *Makespan*[17] *Minimization problem* that is related to deadline based scheduling problem. From this point henceforth, for the multiprocessor scenario, we briefly go over two types of dynamic speed scaling problems: FTPE because it is mostly related to the thesis, and deadline based scheduling because it has been extensively studied.

2.9.1 Deadline Based Scheduling for Multiprocessors

To solve for multiprocessor case, Angel et al. [7] consider the problem of scheduling a set of tasks with deadlines, release dates and processing requirements, on parallel (speed scalable) processors so as to minimize the total energy consumption. They consider migration, where a task is allowed to resume its execution on a different processor. They also allowed pre-emption. They name their optimal scheduling algorithm BAL which has a time complexity of $O(nf(n)\log U)$ where, n is the number of jobs, $f(|V|)$ is the computational complexity of solving a maximum flow in a layered graph with $O|V|$vertices and U is the range of all processor speed values divided by the targeted accuracy. Independently, Albers et al. [2] considered the same multiprocessor speed scaling problem with migration, and obtained an optimal scheduling algorithm that is fully combinatorial and has a time complexity of $O(n^2 f(n))$. Angel et al. [7] compared their BAL algorithm to the one of Albers et al. [2] and stated that when the target precision is sufficiently high, the algorithm of Albers et al. [2] is superior to BAL, otherwise if the target accuracy is relaxed, BAL's algorithm is indeed superior.

Lam et al. [45] solve the deadline-based scheduling for dual processors. They realistically assume that the maximum speed of processors is bounded. Their objective is to maximize throughput while using the least amount of energy. They meet their objective by obtaining a constant competitive solution.

[17] The *makespan* is the point in time where a schedule ends [3].

2.9.2 Flow Time Plus Energy (FTPE) For Multi Processors

Lam et al. [44] presented the first constant competitive online algorithm for arbitrary job sizes. In [44], jobs are clustered and then Round Robin dispatched to the processors independently for each cluster. They then apply the BPS online algorithm[18] given by Bansal et al. [11, 15]

2.9.2 Flow Time Plus Energy (FTPE) For Heterogeneous Multi Processors

All the multiprocessor problems we have discussed so far only deal with homogenous processors. In 2012, Gupta et al. [27] present the first provably scalable non-clairvoyant algorithm on heterogeneous multi processors. This algorithm constitutes a variation of the *Late Arrival Processor sharing* scheduling algorithm [23] that is coupled with a non-obvious speed scaling function. This algorithm handles unweighted flow time plus energy and was shown to be scalable. Gupta et al. [27] formally define heterogeneous processors as those processors that have their own speed function with different power consumption. They also define non-clairvoyant schedulers as those that are unaware of job sizes and make decisions accordingly. Gupta et al. emphasize that scheduling heterogeneous multiprocessors is quite challenging. Furthermore, they believe the algorithms required for parallel heterogeneous processors should be different than those for homogenous multiprocessors.

2.10 Limitations of Speed Scaling

In the past, dynamic power dissipation that stems from the dynamic switching of processing speed has been dominant [52]. In recent technologies, current leaks that stem from gate leakage, sub-threshold leakage and other sources account for roughly 20% or more of power dissipation, and is on the rise [38]. Furthermore, [6] states that the polynomial power function that is used in dynamic speed scaling is not always appropriate because of the interference of additive white Gaussian noise over communication channels (they have exponential power functions).

[18] See section 2.82.

Chapter 3: Theoretical Framework: Model and Notation

In this chapter, we propose a theoretical frame work to tackle the problem of dynamic speed scaling in a parallel processing environment. The study of relevant computing parameters, their relationships and underlying assumptions enable us to systematically synthesize useful dynamic speed scaling algorithms. These algorithms are presented in succeeding chapters. In the context of dynamic speed scaling, the framework attempts to respect the major characteristics and limitations of computing devices as well as to ergonomically integrate relevant parameters that are to be provided by the user. Subsequent chapters mainly take advantage of this framework, but will include their own extensions where appropriate. In this chapter, we:

- Define and describe a *task* in section 3.1;
- Define and describe a *user profile* in section 3.2;
- Define and describe a *processing stream* under different computing architectures and briefly describe *parallel processing streams* in section 3.3;
- Define other relevant *mobile hardware resource parameters* and describe how our framework handles *multiple energy sources* in section 3.4;
- Model *overhead access time* and describe the *processing rate* and *execution time* of a task in section 3.5;
- Use formulas in current literature to deduce useful relationships pertaining to a task's computation volume, energy and power consumption in section 3.6;
- Analytically and graphically illustrate the effect of processing on a task's computation volume as well as the energy and power consumed in section 3.7;
- Describe the *decision algorithm* and summarize relevant constraints in section 3.8;
- Justify the constituents of our *target performance metric* and offer a brief critique of other performance metrics used in current literature in section 3.9; in this section we also distinguish and map our work in current literature; and
- Lastly define traffic conditions relevant for simulations in section 3.10.

3.1 A Task

A task comprises of a set of *base* instructions, usually with processing and memory requirements that are enforced in advance by the programmer during software architectural planning. Mathematically, we model a task, $T_k \in T$ as a vector with the following three parameters.

$T_k = (B_k, p_{\mu,k}, m_k)$

- B_k is the task's remaining computation volume in *base* instructions (*n*).
- $p_{\mu,k}$ is the task's minimum recommended processing rate in *base* instructions per second *(n.Hz).*
- m_k is the task's memory requirement in *bits*.

B_k is the T_k task's (expected) remaining computation volume or the amount of remaining (unprocessed) number of instructions measured in *base* instructions. B_k is measured in *base* instructions to consistently measure a task's raw instructions or remaining computation volume. For example, *multiplication* and *addition* operations are not treated as commensurable instructions, but is each translated to some number of *base* operations or floating point operations. In this example, the number of *base* instructions required for a *multiplication* operation generally exceeds that of an *addition* operation. Depending on the resolution or granularity of a *base* instruction, it can take any arbitrary number of fixed clock cycle/s. We assume a base instruction requires 10 Kilo clock cycles in many of our experiments.[19]

The main reason we use *base* instructions instead of regular clock cycles is because in a given application context, it may be more convenient to lump together common instruction types, and use them as a basis to measure other larger instruction types. Generally, the representation of a task's remaining computation in terms of a *base* instruction requires fewer number of digits to represent because a given *base* instruction could be comprised of a substantial number of clock cycles. This benefit is inherited in the measurement and representation of minimum, optimum and maximum processing

[19] Once we establish the magnitude of a single base instruction in terms of clock cycles, it is fixed.

rates. An obvious drawback of making a single *base* instruction too large is that it will lose its granularity to the extent where the representation of a tasks remaining computation volume may involve fractions or decimals, which is undesirable from a representation view point. Without any loss, a single *base* instruction can represent a single clock cycle so long ass all the relevant parameters in our model are calibrated with this in mind. The unit of a *base* instruction is *n*.

$p_{\mu,k}$, the T_k task's minimum recommended processing rate in base instructions per second *(n.Hz)*, is a software constraint imposed by the software designer. It is fixed and optional, but crucial in identifying the minimum processing rate of executing the task by a given processor. An example is when a task or a set of tasks make up a game. The game's refresh rate is heavily influenced by $p_{\mu,k}$ and if it is not satisfied, the game may be unplayable. We also enforce $p_{\mu,k} > 0$ because we want to eliminate the trivial zero-processing rate condition. The μ sub-script symbol in $p_{\mu,k}$ denotes *minimum* and has no relation to the *inter-arrival period* of arriving tasks[20].

m_k, the T_k task's memory requirement in *bits*, is a fixed requirement that needs to be satisfied by the hardware memory resources (disks, drives, flash) of a mobile device or workstation. If a base instruction consumes z bits, we can model uncompressed m_k using the following equation.

$$m_k = z.B_k + m_{k,r} + m_{k,p} \tag{3.1}$$

$m_{k,r}$ and $m_{k,p}$ are the raw and processed memory requirements of a task (respectively). For example, if a task comprises of encoding a segment of an audio file, B_k will be the number of base operations needed to accomplish the task of encoding $m_{k,r}$ raw bits into $m_{k,p}$ bits of processed data. The *r* and *p* subscripts in $m_{k,r}$ and $m_{k,p}$ denote *raw* and *processed* respectively, and are not indices.

[20] The inter-arrival period of tasks as denoted by μ and is introduced in chapter 5.

3.2 A User Profile

A User Profile comprises of a set of unit cost sensitivity factors or unit prices that are specified by the user through a profile setting integrated in the operating software of the computing device. This profile setting could be an energy saving profile, a performance intensive profile or any other custom profile that is specified by the user. If the user chooses not to specify a custom profile setting, a default setting can be implemented by the programmer that is a balanced tradeoff between an energy saving profile and a performance intensive profile.

Mathematically, we model a user profile vector $U_k \in U$ associated with a task $T_k \in T$ as $U_k = (u_\varepsilon, u_{t,k})$, where:

u_ε - Unit price of energy measured in *\$/Joule*, where $0 < u_\varepsilon < \infty$.

$u_{t,k}$ - Unit price of response time measured in *\$/Second*, where $0 < u_{t,k} < \infty$.

The ε and t subscripts in u_ε and $u_{t,k}$ are purely symbolic to denote *energy* and *time* (respectively).

One practical way to calibrate these unit cost sensitivity factors is to use the actual unit prices of energy and time in a given geographical region and time of day. For instance, in Ontario, Canada the regulated price of energy during peak hours is 12.4 $¢ / kWh$ [30] and the minimum wage of employment as of May 2013 in Ontario Canada is CD\$10.25/hour [54]. This translates to $u_\varepsilon = 3.\overline{4}x10^{-8}$ *\$/Joule* and $u_{t,k} = 2.847\overline{2}x10^{-3}$ *\$/Second*. This is merely a suggestion as we are not enforcing the notion that the unit price of time for a specific individual should always be dictated by his/her hourly pay. Ideally a given user should set $u_{t,k}$ to any price he/she can afford or believes is the price of a second of his/her life.

Figure 3.1 shows an interpretation of these unit cost sensitivity factors. As shown in the figure, they could possibly be implemented through a graphical user interface integrated in the OS of the computing device.

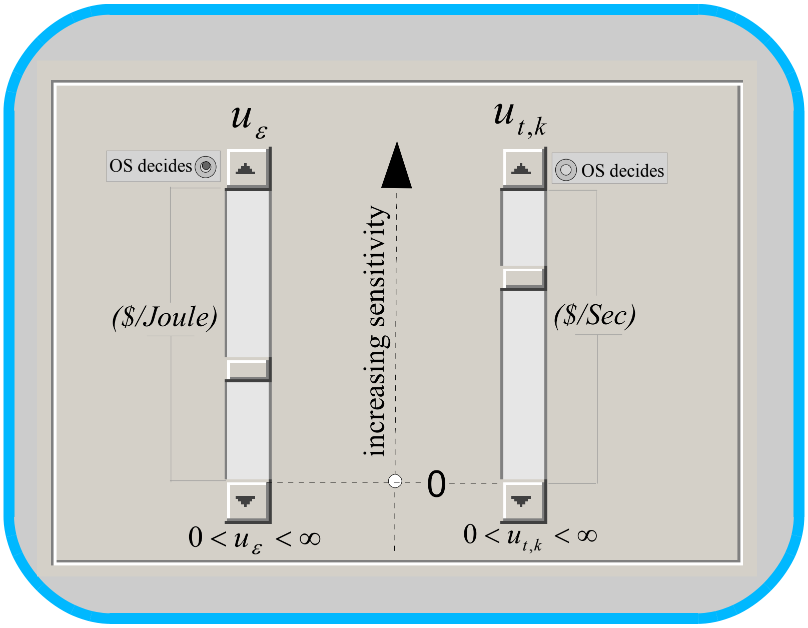


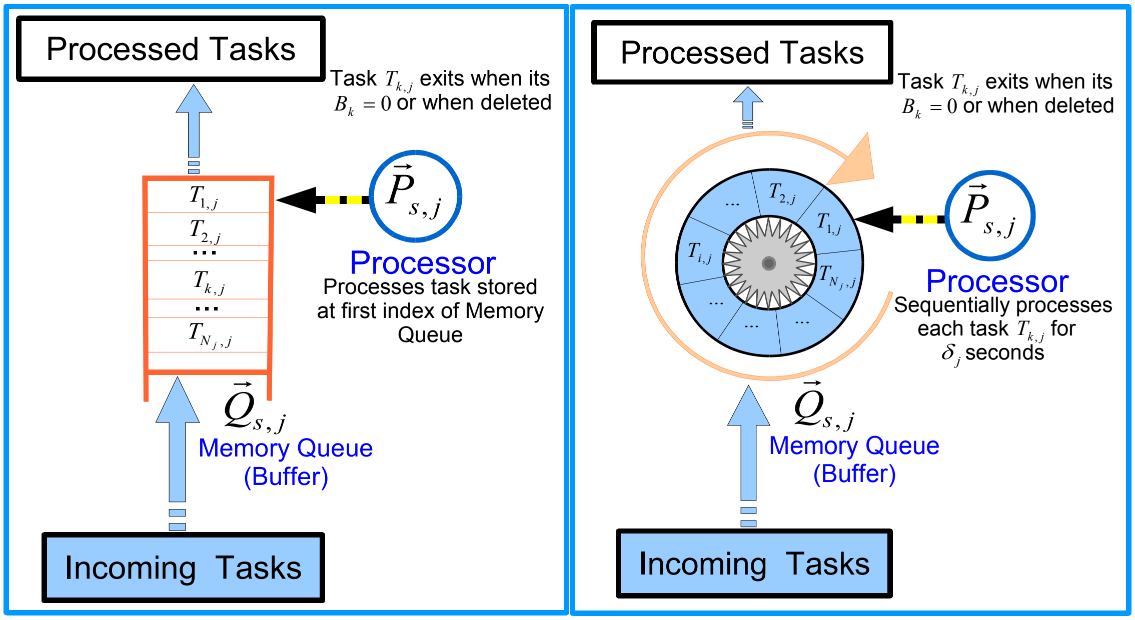
Fig. 3.1: Interpretation and possible implementation of a User Profile

Note that the unit price of energy (u_ε) for all tasks need not be different (this explains the missing *k* subscript in comparison to $u_{t,k}$) and can be set by the OS, but the unit price of response time for each task may be different because we allow the user to influence the priority of a task's through various ways discussed in subsequent chapters.

Furthermore, in a later chapter of this thesis, the unit price of energy is treated with more objectivity because it is adjusted by inversely relating it to the amount of battery life remaining in the computing device, while the unit price of response time ($u_{t,k}$) is indeed more subjective as it essentially depends on how patient the user is with respect to the task's completion time.

3.3 A Processing Stream

A processing stream as described in Fig. 3.2, consists of a (core) processor ($\vec{P}_{s,j}$) and a corresponding memory Queue ($\vec{Q}_{s,j}$). A processing stream is distinguished among other parallel processing streams by the j^{th} index, where $1 \le j \le m$. The vector notation in $\vec{P}_{s,j}$ and $\vec{Q}_{s,j}$ is purely symbolic to denote hardware. Likewise, the *s* subscript denotes *stream* and is not an index.



(a) (b)

Fig. 3.2: Illustrating a Processing Stream under (a) Single-threading and (b) Multi-threading computing architectures.

3.3.1 Stream Processor

Each processing stream's processor ($\vec{P}_{s,j}$) executes a given task at a processing rate of $P_{s,j}$ base instructions per second (*n.Hz*). We assume each and every stream processor can be dynamically speed-scaled. We have $p_{\mu,k} \le P_{s,j} \le P_{Max,j}$ where $P_{Max,j}$ is the maximum operating frequency in base instructions per second of the j^{th} processing stream's processor; it is a constraint imposed by the hardware specification of the computing device (processor). For a given task $T_k \in T$, its minimum processing rate, $p_{\mu,k}$, is a software constraint imposed by the software designer and is generally lower than $P_{Max,j}$ for analytical and practical purposes.

3.3.2 Memory Queue

A memory queue $\vec{Q}_{s,j}$ of the j^{th} processing stream stores N_j tasks at some instance in time. Therefore $0 \le N_j < \infty$. In other words, N_j is the occupancy of the j^{th} processing stream's memory queue[21].

- $N_j = 0$: denotes that the memory queue of the j^{th} processing stream is empty.
- Under single-threading, at any given time, the j^{th} stream processor processes a task stored in the first index[22] of the memory queue.
- Under the multi-threaded computing architecture, at any given time, the j^{th} stream processor sequentially processes each task stored in its memory queue for δ_j (time slice) seconds.

3.3.3 Parallel Processing Streams

Parallel processing streams are a set of processing streams configured in parallel. When our work is applied to mobile computing devices, the processing streams may or may not share the same energy (battery) source. Our analysis holds for either one of the following scenarios:

- All parallel processing streams share only one battery source.
- Each processing stream has its independent battery source of equal capacity but not necessarily equal energy level.

3.4 Mobile Hardware Resources

3.4.1 Mobile Hardware Parameters

Table 3.1 summarizes other hardware resource/parameters of the mobile device. We refer to Table 3.1 in subsequent sections.

[21] In chapter 4 we enforce the constraint $N_j \le 1$. We relax this constraint in subsequent chapters.

[22] This first index corresponds to a system index of $(1, j)$.

Table 3.1: Other hardware parameters of the computing device

Parameter	Meaning	SI Unit
M_m	Available memory of mobile device	bits
$E_{m,j}$	Battery energy level of j^{th} processing stream	Joules
$E_{\theta,j}$	Threshold energy level of j^{th} processing stream	Joules
$\left(E_{m,j} - E_{\theta,j}\right)$	Usable battery energy of j^{th} processing stream	Joules
$E_{cap,j}$	Maximum energy capacity of j^{th} processing stream (under full charge)	Joules
$\varepsilon_{\%,j}$	Remaining battery energy percentage of j^{th} processing stream, $\varepsilon_{\%,j} \in [0,1]$	dimensionless
$t_{\theta,k,j}$	Overhead access time of a task T_k to be accessed and loaded by processor $\vec{P}_{s,j}$ from Memory Queue $\vec{Q}_{s,j}$	Seconds

3.4.2 Single or Multiple Energy Sources

It is worth mentioning that the analysis done assumes each processing stream has its independent battery source of equal capacity, but not necessarily of equal energy level. In practice, a special case of this assumption is usually implemented where all parallel processing streams share only one battery source; an example is the iPhone 5 [8]. We can simply narrow the work to single energy sources by substituting each and every $\varepsilon_{\%,j}$ for $\varepsilon_{\%}$, i.e., $\varepsilon_{\%,j} = \varepsilon_{\%}, \forall_{j\in\{1,2...m\}}$.

If the mobile device is currently being re-charged (battery inflow energy exceeds current use) and it is known in advance that the mobile device will not be disrupted from recharging its battery/batteries until completion, then during the recharging period we can prematurely set $\varepsilon_{\%,j} = 1, \forall_{j\in\{1,2...m\}}$ since energy is temporarily not a scarce resource during foreseen battery recharge period. Also, all of the work presented in this thesis can be extended to non-mobile work stations or computing devices that have a reliable and unlimited power supply (but not free)by setting $\varepsilon_{\%,j} = 1, \forall_{j\in\{1,2...m\}}$.

3.5 Task's Processing Rate and Time Consumption

3.5.1 Modeling Overhead Access Time

In Table I, we defined $t_{\theta,k,j}$ as the overhead access time of a task T_k to be accessed and loaded by processor $\vec{P}_{s,j}$ from memory queue $\vec{Q}_{s,j}$. The θ subscript in $t_{\theta,k,j}$ is purely symbolic to denote *loading* and is not an index. The magnitude of $t_{\theta,k,j}$ mainly stems from digital delays of both the memory queue and activation of the processor. Other delays from the OS or hardware architecture that delay the execution of the task after its scheduled processing are included as part of $t_{\theta,k,j}$ with specific exception to waiting time for another task/s that is being executed ahead in line in the memory queue. It is important to include $t_{\theta,k,j}$ in our model especially if the task's computation volume (B_k) is small enough such that it has an execution time (t_k) that is of around comparable magnitude as $t_{\theta,k,j}$. If B_k is substantial enough where it has an execution time, t_k such that: $t_k >> t_{\theta,k,j}$, then we can safely eliminate $t_{\theta,k,j}$ from the model.

3.5.2 Modeling Processing Rate and Execution Time

p_k is a task's ($T_k \in T$) theoretical processing rate in base instructions per second (*n.Hz*). t_k is the task's expected execution time in seconds. We relate P_k to t_k in the next section. The overhead (processor) switching times during processing are assumed to be negligible in comparison to execution times of tasks. Furthermore, these switching times can not be deterministically modeled in the online scenario that considers a preemptive service discipline under the single threading computing architecture, e.g. SRPT, because the number of preemptions are unknown and rely on the properties of tasks arriving in real time. Under non-preemptive service disciplines, e.g. FCFS, these overhead switching times can be (deterministically) included as part of the overhead access times $(t_{\theta,k,j})$. We also have not come across any work that explicitly factors these overhead access times in the context of dynamic speed scaling.

3.6 A Task's Energy & Power Consumption

For a task: $T_k \in T$, let Pow_k be the task's expected power function in Watts and let ε_k be the task's expected energy function in Joules when processed by the j^{th} processor. Let us initially assume the task's (theoretical) processing rate (P_k) is time invariant or constant over its expected execution time (t_k).

$$Pow_k = \lambda_j (P_k)^{\alpha_j} \quad \textit{(Watts)} \tag{3.2}$$

Many researchers e.g. [4, 7, 19, 25, 68] use a variation of equation 3.2, but [6] presents an equation similar to it. See Appendix 1 for how we initially modeled the energy and power consumption of a task without the formal knowledge of dynamic speed scaling.

We define λ_j, measured in $(J.S^{\alpha_j - 1}.n^{-\alpha_j})$, as the energy inefficiency factor or the scaling factor of the j^{th} processor's power function[23] and we assume $\lambda_j > 0$.

α_j is the exponent of the j^{th} processor's power function and it is assumed to be a constant. [6] suggests that $\alpha_j = 1.8$ is a good approximation for CMOS based processors and that $\alpha_j \in (1,3]$ holds for most computer systems comprising of disks, processing chips and servers.

We know that power consumption is the rate of energy consumption; this implies the following.

$$\varepsilon_k = \int_0^{t_k} Pow_k dt = \int_0^{t_k} \lambda_j (P_k)^{\alpha_j} = \lambda_j (P_k)^{\alpha_j} t_k \quad \textit{(Joules)} \tag{3.3}$$

[23] We may choose to model λ_j as a function increasing by temperature (K^o) because the hotter a processor gets, the more current leaks occur which lead to more power inefficiency [9]. We assume $\lambda_j((K^o)) > 0$.

B_k relates t_k to P_k, and happens to be the task's remaining computation volume in *base* instructions (*n*).

$$t_k = \frac{B_k}{P_k} \qquad \textit{(Seconds)} \qquad (3.4)$$

Using (3.2) and (3.3), we deduce:

$$\varepsilon_k = \lambda_j B_k (P_k)^{\alpha_j - 1} \qquad \textit{(Joules)} \qquad (3.5)$$

We exclude the overhead energy consumed when processors switch speed and also assume the processors consume zero power when idle.

3.7 Description of a Task's Computation Volume upon Execution

Equations (3.3-3.5) are valid when a task is executed at a constant processing rate until completion[24]. These equations can be adjusted to consider situations where a task is executed at constant processing rates that differ over a finite number of time intervals. Consider two situations as follows.

First, let us consider a scenario where a task, $T_k \in T$ is executed by a processor $\vec{P}_{s,j}$. During execution[25], the task's computation volume (B_k) decreases at a constant rate (processing rate). When $B_k = 0$, the task execution is complete. Also, the task's execution will consume energy as dictated by the convexity of the power function (equation (3.2)). As mentioned in section 3.1, B_k is the remaining computation volume of a task T_k. It is ultimately a non-increasing function of global time $\left(B_k(t)\right)$ and a decreasing function of execution time.

[24] The use of these equations to model and formulate our online algorithms are justified because we can not predict preemptions that are caused by the future arrival of tasks.

[25] Prior to execution, the task first incurs a loading time or overhead access time $(t_{\theta,k,j})$.

To illustrate this, let the task T_k be executed at a constant processing rate $\left(P_k\right)$ over some finite time interval (a,b). Using the fundamental theorem of calculus.

$$B_k(t)\big|_a - B_k(t)\big|_b = \int_a^b \left(-\frac{\partial B_k(t)}{\partial_t} \right) dt \tag{3.6}$$

Using $P_k = \left| \frac{\partial B_k(t)}{\partial_t} \right|$, since $P_k \geq 0$ and $B_k(t) \geq 0$, it implies

$P_k = -\frac{\partial B_k(t)}{\partial_t}$ and assuming constant processing rates in (3.6),

$$B_k(t)\big|_a - B_k(t)\big|_b = \int_a^b P_k dt = (b-a)P_k \tag{3.7}$$

$$\Rightarrow B_k(t)\big|_b = B_k(t)\big|_a - (b-a)P_k$$

Equation (3.4) can be confirmed by (3.7) when $t_k = b - a$,

$B_k(t)\big|_a = B_k$ and $B_k(t)\big|_b = 0$ which gives

$$B_k = 0 + \int_a^b P_k dt = P_k t_k \Rightarrow B_k = P_k t_k$$

Let us consider a second example described by the Fig. 3.3.

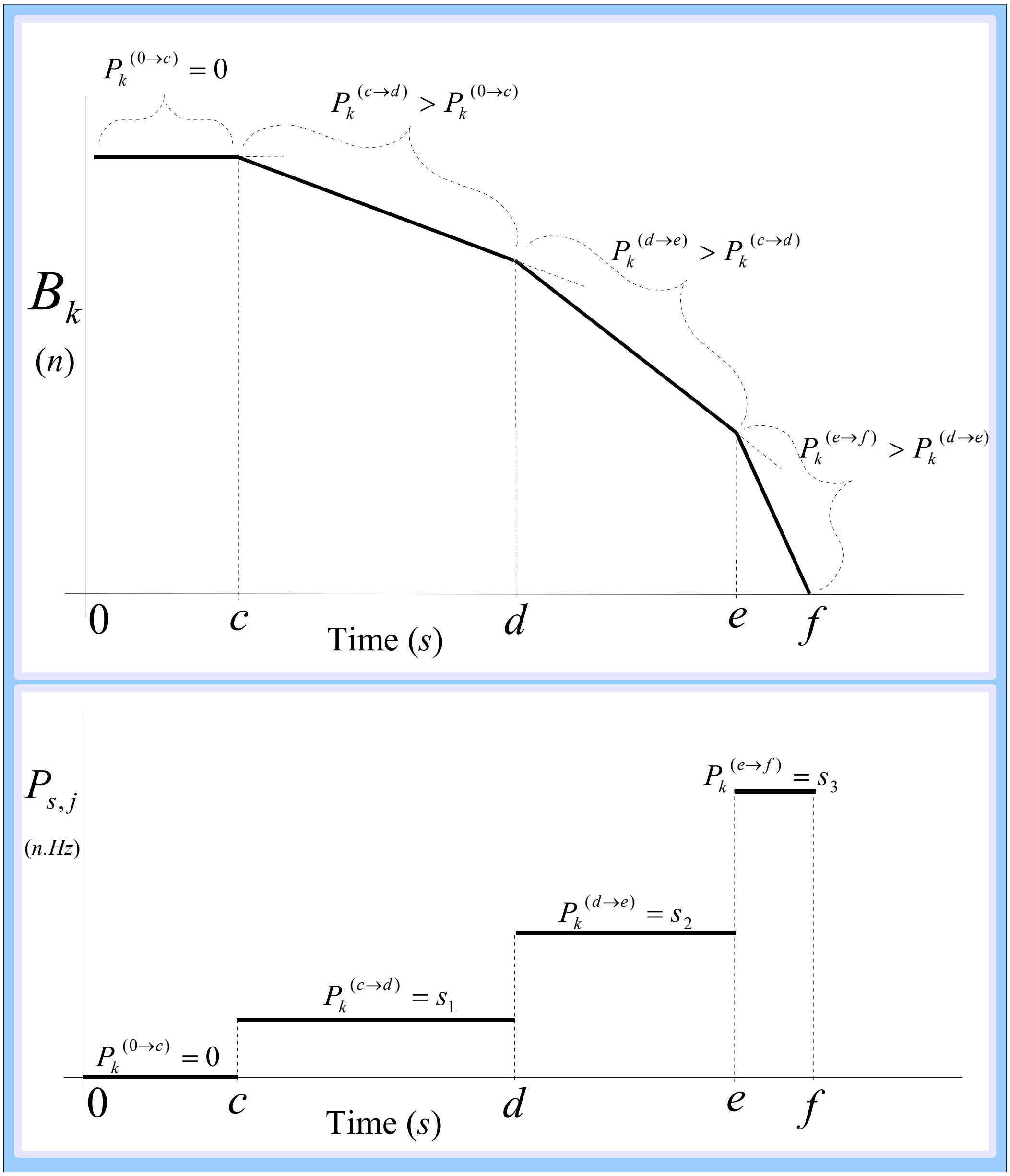


Fig. 3.3:An example describing the remaining computation volume of a task during (constant processing rate) execution over a finite number of intervals

Fig. 3.3 describes an example where a task's execution is paused over the time interval $(0,c)$. Over the time intervals: (c,d), (d,e) and (e,f), the task is executed at different constant processing rates of s_1, s_2 and s_3 respectively.

The table below summarizes the energy, power and execution time incurred during the execution of the task.

Table 3.2: Energy, power and execution time incurred (example scenario)

Time Interval	$(0,c)$	(c,d)	(d,e)	(e,f)
Processing Rate	0	s_1	s_2	s_3
Energy Consumed	0	$\lambda_j \left(B_k(t)\|_d^c\right)(s_1)^{\alpha_j-1}$	$\lambda_j \left(B_k(t)\|_e^d\right)(s_2)^{\alpha_j-1}$	$\lambda_j \left(B_k(t)\|_f^e\right)(s_3)^{\alpha_j-1}$
Execution time	0	$(d-c)$ $=\left(B_k(t)\|_d^c\right)/ s_1$	$(e-d)$ $=\left(B_k(t)\|_e^d\right)/ s_2$	$(f-e)$ $=\left(B_k(t)\|_f^e\right)/ s_3$
Power Consumed	0	$\lambda_j(s_1)^{\alpha_j}$	$\lambda_j(s_2)^{\alpha_j}$	$\lambda_j(s_3)^{\alpha_j}$

*power consumption at the exact time instances: *c*, *d* and *e* are undefined due to instantaneous speed changes.

Note that the ongoing depletion of a tasks remaining computation volume during execution is purely a property of how the processor operates (see equation 3.7).

3.8 The Decision Algorithm

3.8.1 Memory, Processing Rate and Energy Constraints

Fig. 3.4 is a Venn diagram that illustrates how a task has to simultaneously satisfy memory, processing rate and energy requirements with respect to a processing stream for it to be potentially executed along that processing stream. Generally, if the task's (remaining) computation volume (B_k) is substantial, it becomes difficult to simultaneously satisfy all three constraints.

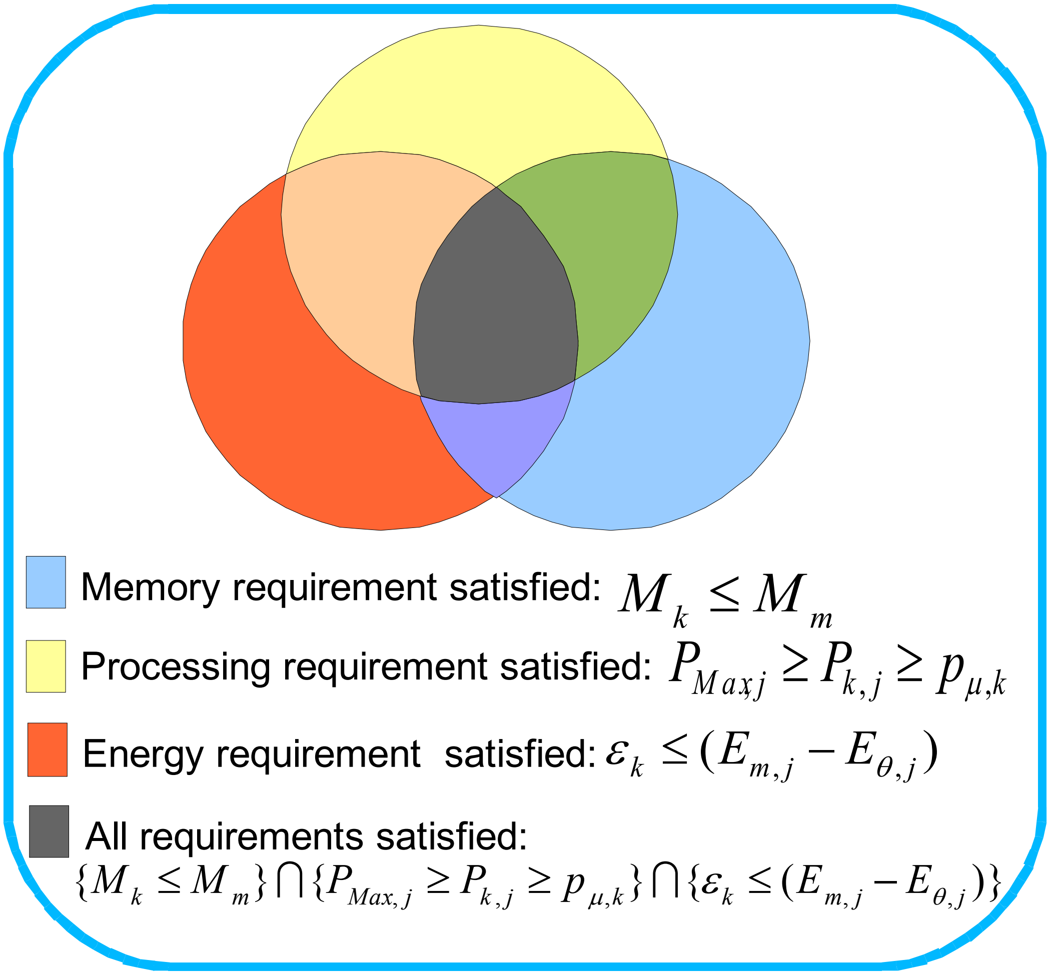

Memory requirement satisfied: $M_k \le M_m$
Processing requirement satisfied: $P_{Max,j} \ge P_{k,j} \ge p_{\mu,k}$
Energy requirement satisfied: $\varepsilon_k \le (E_{m,j} - E_{\theta,j})$
All requirements satisfied:
$\{M_k \le M_m\} \cap \{P_{Max,j} \ge P_{k,j} \ge p_{\mu,k}\} \cap \{\varepsilon_k \le (E_{m,j} - E_{\theta,j})\}$

Fig. 3.4: Memory, processing rate and energy constraints

3.8.2 The Decision Algorithm

Once we have a task or a set of tasks that satisfy the preliminary memory, processing and energy constraints for *m* number of processing streams, we generally have three major questions that need to be addressed by the decision algorithm.

The decision algorithm performs three main functions as follows:

1) *Dispatcher:* Addresses which processing stream among the *m* processing streams should process a given task.

2) *Service discipline / policy:* Specifies the order or discipline in which tasks should be serviced.

3) *Speed-scaling function:* Explicitly determines the optimum processing rate of executing a task/s.

The service discipline only applies to the multi-buffer, single-threading computing architecture. Under (single buffer) single threading and classical processor sharing (or multi-threading) computing architectures, the service disciplines do not matter.

3.9 Performance Metrics

3.9.1 Measuring Response Time in a Sequential Process

The response time (also known as flow time [1]) is the time elapsed since a task arrives until it is completed. Measuring response time is a bit of a convoluted procedure when delays are involved. Under the single threaded computing architecture, when we have a pre-existing "*traffic build-up*" of tasks, total execution time (time between execution of the first task and completion of last task) tends to under-represent the response time of a set of tasks. This happens because any common delay occurs simultaneously and can only be observed or measured once on a global timescale, while in reality, any delay should be multiplied by some integer *z*, where *z* is the number of tasks affected. The response time correctly factors time delays. To demonstrate this, consider the following analogical example. Assume we have a car B waiting behind a car A in traffic. Upon a launch of a green light, car A stalls for *t* seconds. If we examine this scenario by considering the response time perspective, the total time wasted is *2t*; *t* seconds for car A and *t* seconds for car B because car B's path is blocked by car A. If we examine this example using an execution time perspective, the total time incurred of car A stalling for *t* seconds is simply *t* seconds since this *t* seconds is evolving simultaneously for both cars. Now, let us pose a question with some options. If we had to delay one of the two cars for *t* seconds upon the launch of a green light, which car should we delay? The response time perspective suggests that car B should be delayed for *t* seconds rather than car A, because car A will remain unaffected, and the total time wasted will be *t* seconds instead of *2t*. In this scenario, the execution time perspective suggests that the time wasted is *t* seconds irrespective of the options posed. A mild extension is to observe that response time coincides with the execution time when there are no delays.

The response time perspective can be used to derive *greedy* time sensitive algorithms that are efficient in identifying and penalizing bottlenecks in sequential processes. Response time unlike execution time augments the time cost function of a sequential process (e.g.

single threading computing architecture) by appropriately factoring delay/waiting and execution times of each task and it can be used to strategically mitigate bottlenecks at the expense of energy by using *dynamic speed-scaling*. Execution time does not sufficiently capture the waiting time dependencies in a sequential process. This is why we chose to consider response time instead of execution time as part of our target performance metric. We formally present the conditions in which response time should be considered.

- There exist delays.
- We are to democratically treat each task as an independent entity.
- The execution of tasks is a sequential process e.g. single-threading computing architecture.

3.9.2 Criticisms of Performance Metrics Used in Current Literature

We briefly mention a few criticisms we have with existing models that address dynamic speed scaling problems from an algorithmic perspective.

Researchers such as Wierman et al. [63] provide a better way than [1] and [11] to integrate energy and flow time because they explicitly define a translator parameter β that converts energy to response time through relative pricing.

All the problems related with minimizing energy and flow times from an algorithmic perspective do not provide SI units. This has made it difficult to discern what quantities are actually being optimized, moreover what parameters are implicit or explicit.

Also, most of the dynamic speed scaling algorithms that we have come across model dynamic power as s^{α} where s is the speed of the processor and α is some positive constant. We know that dynamic power grows in proportional to s^{α} e.g. in CMOS processors it is modeled by [38] as being proportional to $cv^2 f$, where c, v, and f are the processor's capacitance, voltage and frequency/speed respectively. We are aware of this practical consideration so we model it as λs^{α} by explicitly factoring a slack parameter (λ) that accounts for other variables or phenomena, e.g. **capacitance, temperature** etc.

3.9.3 TCRTEC Performance Metric

We propose a (financial) performance metric called TCRTEC (Total Cost of Response Time and energy Consumption).

We defined in previous sections, each task T_k has an associated user profile $U_k = (u_\varepsilon, u_{t,k})$ and consumes energy ε_k.

Let us assume each task T_k incurs a response time R_k. Also let the vector $V_k = (\varepsilon_k, R_k)$

If we let a schedule Q have some tasks $T_k \in T$, Using dot product operations, we explicitly define TCRTEC for the schedule as TCRTEC = $\sum_{T_k \in Q}(U_k \bullet V_k)$.

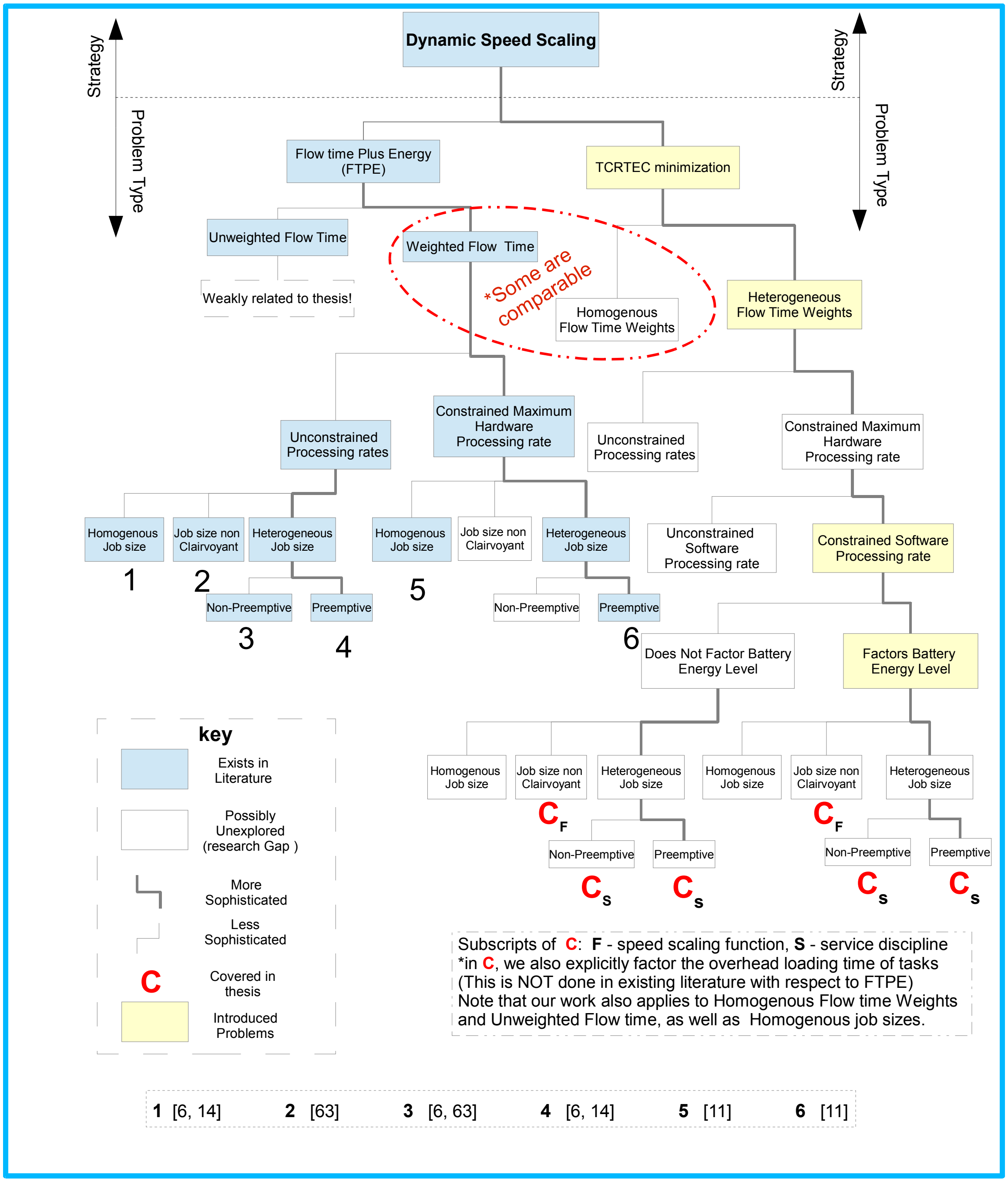
The TCRTEC performance metric stems from the amalgamation of the user profiles of tasks with the resource consumption (energy and response time) of schedules. This performance metric is convenient in the sense that it translates the energy and response time components of a schedule into dollars through the user (or OS)-specified pricing of energy and response time. It allows the unit price of response time for each task (in a schedule) to be different because the user is allowed to influence the priority of tasks. It is also an appropriate metric because it does not violate a fundamental law of physics pertaining to the addition of different SI units, namely a Joule and a Second.

3.9.4 Distinguishing our Model from Dynamic Speed Scaling Models Found in Current Literature (Major differences)

- We assume general power functions of the form λs^α, λ accounts for capacitance, temperature etc. Existing literature uses s^α.
- We explicitly model overhead loading times.
- We use the remaining computation volume to model energy consumption.
- We augment the processing requirement of tasks to include minimum software requirements in addition to hardware processing rates.
- We model energy sources (single and multiple).
- We incorporate the preferences of the user or OS through customized pricing (energy pricing and heterogeneous response time pricing).

- We use the proposed TCRTEC performance metric to formulate and evaluate our algorithms.
- We define all our parameters in standard SI units.

3.9.5 Mapping Our work in Current Literature



Subscripts of **C**: **F** - speed scaling function, **S** - service discipline
*in **C**, we also explicitly factor the overhead loading time of tasks
(This is NOT done in existing literature with respect to FTPE)
Note that our work also applies to Homogenous Flow time Weights and Unweighted Flow time, as well as Homogenous job sizes.

1 [6, 14] **2** [63] **3** [6, 63] **4** [6, 14] **5** [11] **6** [11]

Fig. 3.5: Placing our problem relative to the single processor problems in literature

Fig. 3.5 shows the relevant research gap and also illustrates the complexity of our assumptions with respect to the single processor scenario. The problems that we solve are significantly more challenging than what is shown. We consider the scheduling of heterogeneous parallel processors in unison to what is shown in Fig. 3.5.

3.10 Defining Traffic conditions

In assessing the performance of our algorithms, the arrival rate of tasks is indeed a crucial consideration. High arrival rates generally stress the performance of the algorithms potentially leading to the build up of task traffic congestion. Low arrival rates of tasks, especially in the multiprocessor environment is also not ideal because there is poor utilization of resources. The arrival of tasks is generally modeled as Poisson process [26]. For the sake of simplicity, let us first consider different classifications of deterministic arrival rates and define them based on some standard. They are as follows:

- Minimal traffic - we have an arrival rate of tasks such that at any given time, only one processor is actively processing a single and lone task in the system. This leads to minimum congestion, but poor system utilization.
- Ideal traffic - we have an arrival rate of tasks such that for the majority of the time, each processor is actively processing a task, but no arriving task awaits for service. This situation maximizes utilization as well as minimizes traffic congestion but is difficult to enforce in practice, especially in the online scenario.
- Heavy traffic - we have an arrival rate of tasks that falls in between ideal and extreme traffic. The occupancy of each processor exceeds 1 most of the time.
- Extreme traffic - tasks arrive as a batch. This maximizes stress on algorithmic performance.

We can extend these definitions to consider stochastic arrival rates (exponentially distributed) by using the deterministic arrival rates as input parameters in the exponential probability distributions that model the arrival rate of tasks. Doing so, will generally lead to higher traffic congestion as compared to that of their deterministic counterparts. This happens because the system requires time to recover from some randomly generated arrival rates that are higher than those defined by their deterministic counterparts.

We use these classifications of traffic conditions as a standard to evaluate the performance of our algorithms when carrying out analysis and simulations.

3.11 Conclusions

The theoretical frame work presented in this chapter is used in subsequent chapters to address the problem of dynamic speed scaling in a parallel processing environment.

Subsequent chapters mainly take advantage of this framework, but will include their own extensions where appropriate.

Chapter 4: Cost Minimization For Scheduling Single-buffered Processors

4.1 Introduction

This chapter synthesizes a scheduling and parallel processing algorithm named "*Single-Buffer Decision & Parallel Processing algorithm* (SBDPP)". It operates in real time to optimally assigns an incoming stream of heterogeneous tasks to run on multiple (single-buffered) heterogeneous processors in a mobile computing device or an *energy aware* work station. By using dynamic speed-scaling, where each processor's speed is able to change within hardware and software processing constraints, the algorithm also explicitly determines the optimum processing rate of executing each task residing in the single buffer of each processor. Tasks are heterogeneous in terms of computation volume, processing and memory requirements. The time and energy dimensions of executing an arriving task is modeled in a cost function that is each associated with a processing stream. The algorithm's dispatching strategy is to minimize this expected cost by using dynamic speed scaling and to select the least expensive processing stream. The algorithm has three versions. Its first two versions allow the user to specify the unit price of energy and response time for executing each arriving task. The algorithm's second version extends the functionality of the first by allowing the user or the OS of the computing device to further modify a task's unit price of time or energy in order to achieve a linearly controlled operation point that lies somewhere in the *economy-performanc*e *mode* continuum of a task's execution. The algorithm's third version operates exclusively on the latter. We initially focus on single buffer, single-threading where a single task is allocated to a given processor and is processed until its completion. We extend the algorithm and its versions to consider migration, where an unfinished task is paused and resumed on another processor. For diverse application, we also assume that the processors/cores are heterogeneous in that they may differ in their hardware specifications with respect to maximum processing rate and general power function parameters.

The SBDPP algorithm is qualitatively compared against its versions. The algorithm's dispatcher is analytically shown to perform better than the well known Round Robin dispatcher in terms of reducing the total cost of response time and energy consumption when traffic is minimal. Through simulations we deduce a relationship between the arrival rate of tasks, number of processors and response time of tasks under the (parallel) single buffered computing architecture. Although the dynamic speed scaling problem on multi-buffered (single) processors has been previously attempted (e.g. by [6]), this chapter presents the first elaborate, analytical study on the use of dynamic speed scaling to schedule heterogeneous tasks on single-buffered, heterogeneous, parallel processors with the objective of reducing the financial total cost of response time and energy consumption (of tasks).

The single-buffered computing architecture warrants a deep analysis because it encompasses the following characteristics.

1. When a task's overhead loading time is excluded, a tasks response time equals its execution time.
2. Traffic congestion is minimal as a result of constrained single buffers.
3. The service disciplines within processing streams do not apply due to single buffers.
4. It potentially leads to serious bottlenecks, i.e. if the rate of task arrival exceeds that of completion, the single buffers get clogged up. This condition is undesirable because it ultimately forces arriving tasks to be rejected.

In the scope of parallel scheduling of single buffered processors, the ideal scenario is that each of the single-buffers (associated with its corresponding processor) is fully occupied all the time but no task is rejected upon arrival. This maximizes system utilization, minimizes traffic congestion (in comparison to multi-buffered processors), but for this to be practically feasible, it unfortunately requires some control over the properties and rate of arriving tasks. Since such a control is unavailable in the online case, we can sacrifice consistently maximum system utilization for a lower probability of rejecting arriving tasks by enforcing any of the following:

- Increasing the lower bound on the arrival periods of tasks[26].
- Increasing the number of processors.
- Decreasing the response time of tasks.

These claims are corroborated by conducting simulations based on our model.

The major theme in this chapter is how to schedule arriving heterogeneous tasks on to heterogeneous single-buffered processors by utilizing dynamic speed-scaling. This chapter is organized as follows. Section 4.2 formulates the problem and provides sufficient background to construct the SBDPP algorithm. Section 4.3 describes the default version of the SBDPP algorithm. Section 4.4 focuses on how to achieve a linear calibration of a task's operation mode as a function of the (user-specified) unit prices of time and energy, and it also provides preliminary background for the next section. Section 4.5 uses the background presented in the previous section to construct the two other versions of the SBDPP algorithm, namely SBADPA and FPDPA. In this section we also briefly describe how the SBDPP algorithm can deal with migration. In section 4.6, we qualitatively compare the three versions of the algorithm to each other and quantitatively compare the dispatcher of the SBDPP algorithm to the Round Robin dispatcher. Section 4.7 provides a brief report of simulations conduced and lastly provides some insights that were extracted from simulating the algorithm(s).

4.2 Problem Formulation

4.2.1 Processing Streams with Single Buffers

Fig. 1 illustrates the single buffer scenario: each processing stream has a memory queue with a limited capacity of accommodating only one task at a time.

We are essentially trying to achieve two goals. For a given task, one goal is to find the optimized dispatcher that dictates which of the processing streams should process/execute the task at hand. The other goal is to determine the optimized processing rate of executing the task. The problem's major constraint is the single buffer scenario that is described as:

[26] This is equivalent to decreasing the upper bound on the arrival rate of tasks.

$\mathrm{N}_j \le 1, \forall_{1\le j\le m}$, where N_j is the occupancy of the j^{th} processing stream at some point in time.

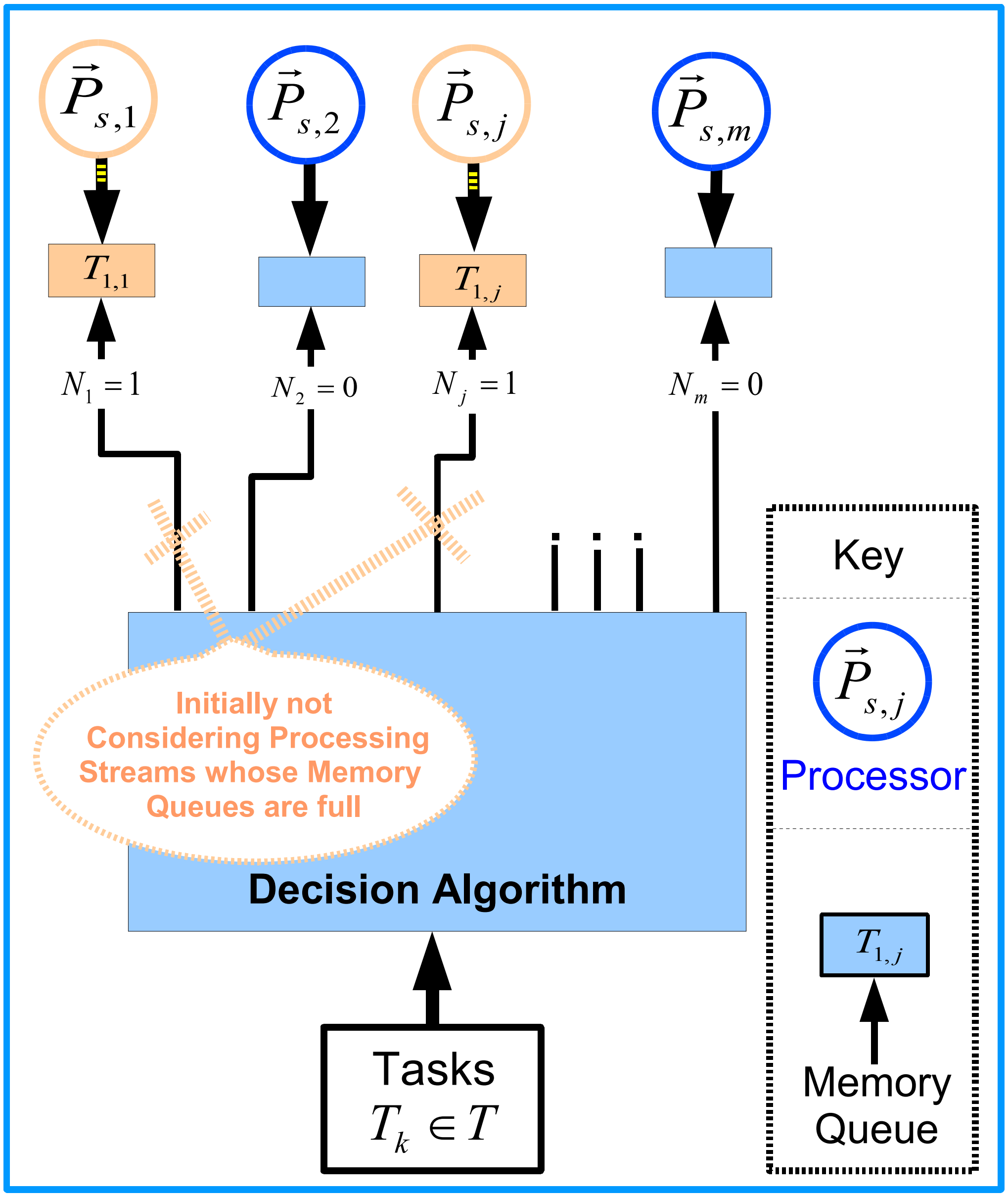


Fig. 4.1: Illustrating the parallel single buffer scenario

4.2.1 A Processing Stream Cost Function

Let $C_j = C_j(u_\varepsilon, u_{t,k}, t_k, t_{\theta,k,j}, \varepsilon_k)$ be the cost function that aggregates the cost of processing a task T_k in the j^{th} processing stream. Each memory queue of each processing stream is assumed to be initially empty and has the capacity to hold only a single task. Formally, we have: $\lfloor \mathrm{N}_{\mathrm{j}} = 0, \forall_{1\leq \mathrm{j} \leq \mathrm{m}} \rfloor$.

We are primarily trying to penalize the energy and response time requirements of a task. One reasonable definition of C_j is as follows.

$$C_j = u_\varepsilon \varepsilon_k + u_{t,k}(t_k + t_{\theta,k,j}) \qquad (4.1)$$

Substituting (3.4) and (3.5) into the cost function gives:

$$C_j = \underbrace{u_\varepsilon \overbrace{\lambda_j B_k (P_k)^{\alpha_j - 1}}^{\text{Task' s energy (J)}}}_{\text{Task' s energy cost (\$)}} + \underbrace{u_{t,k} \overbrace{\left(\frac{B_k}{P_k} + t_{\theta,k,j}\right)}^{\text{Task' s response time (s)}}}_{\text{Task' s response time cost (\$)}} \quad (\$) \qquad (4.2)$$

4.2.2 Optimizing the task's processing rate

In (4.2), the only dynamic parameter within our control is P_k

In order to optimize P_k, we suggest the following:

$$\frac{\partial C_j}{\partial P_k} = (\alpha_j - 1) u_\varepsilon \lambda_j B_k (P_k)^{\alpha_j - 2} - \frac{u_{t,k} B_k}{P_k^{\ 2}} = 0$$

Solving for a critical point we get:

$$P*_k = \left[\frac{u_{t,k}}{(\alpha_j - 1) u_\varepsilon \lambda_j}\right]^{\frac{1}{\alpha_j}} \quad (n.Hz) \qquad (4.3)$$

We confirm that this critical point is indeed a minima as follows.

$$\left.\frac{\partial^2 C_j}{\partial P^2{}_k}\right|_{P^*{}_k} = \left.\left[(\alpha_j - 1)(\alpha_j - 2)u_\varepsilon \lambda_j B_k (P_k)^{\alpha_j - 3} + 2\frac{u_{t,k}B_k}{P_k^{\;3}}\right]\right|_{P^*{}_k}$$

$$= \left.\left[\frac{1}{P_k^{\;3}}\Big((\alpha_j - 1)(\alpha_j - 2)u_\varepsilon \lambda_j B_k (P_k)^{\alpha_j} + 2u_{t,k}B_k\Big)\right]\right|_{P^*{}_k}$$

$$= \left(\frac{(\alpha_j - 1)u_\varepsilon \lambda_j}{u_{t,k}}\right)^{\alpha_j/3}\left((\alpha_j - 1)(\alpha_j - 2)u_\varepsilon \lambda_j B_k \frac{u_{t,k}}{(\alpha_j - 1)u_\varepsilon \lambda_j} + 2u_{t,k}B_k\right)$$

$$= \left(\frac{(\alpha_j - 1)u_\varepsilon \lambda_j}{u_{t,k}}\right)^{\alpha_j/3}\Big((\alpha_j - 2)u_{t,k}B_k + 2u_{t,k}B_k\Big)$$

$$= \Big(\alpha_j u_{t,k}B_k\Big)\left(\frac{(\alpha_j - 1)u_\varepsilon \lambda_j}{u_{t,k}}\right)^{\alpha_j/3} > 0$$

Which confirms that this critical point is indeed a minima for $\alpha_j \in (1,3]$.

4.2.3 Minimized Cost Function of the j^{th} processing stream

We previously concluded that $P^*{}_k = \left(\frac{u_{t,k}}{(\alpha_j - 1)u_\varepsilon \lambda_j}\right)^{\frac{1}{\alpha_j}}$ minimizes our cost function (C_j). It could easily be implemented in the OS of the mobile device whenever a task is to be processed along the j^{th} processing stream. An interesting observation is that the task's computation volume and loading time does not affect its optimum processing rate. Let $C^*{}_{j\min}$ be the optimized (minimized) unconstrained cost function of processing a task in the j^{th} processing stream.

$$
\begin{aligned}
C*_{j\min} &= C_j\Big|_{P*_k} \\
&= u_\varepsilon \lambda_j B_k (P*_k)^{\alpha_j - 1} + u_{t,k}\left(\frac{B_k}{P*_k} + t_{\theta,k,j} \right) \\
&= \left(\frac{u_\varepsilon \lambda_j u_{t,k}}{(\alpha_j - 1) u_\varepsilon \lambda_j} + u_{t,k} \right) B_k \left(\frac{u_{t,k}}{(\alpha_j - 1) u_\varepsilon \lambda_j} \right)^{-\frac{1}{\alpha_j}} + u_{t,k} t_{\theta,k,j} \\
&= \left(\frac{u_{t,k}}{(\alpha_j - 1)} + u_{t,k} \right) B_k \left(\frac{u_{t,k}}{(\alpha_j - 1) u_\varepsilon \lambda_j} \right)^{-\frac{1}{\alpha_j}} + u_{t,k} t_{\theta,k,j} \\
&= \left(\frac{\alpha_j u_{t,k}}{(\alpha_j - 1)} \right) B_k \left(\frac{u_{t,k}}{(\alpha_j - 1) u_\varepsilon \lambda_j} \right)^{-\frac{1}{\alpha_j}} + u_{t,k} t_{\theta,k,j} \\
&= \alpha_j B_k \left(u_\varepsilon \lambda_j \right)^{\frac{1}{\alpha_j}} \left(\frac{u_{t,k}}{(\alpha_j - 1)} \right)^{\frac{\alpha_j - 1}{\alpha_j}} + u_{t,k} t_{\theta,k,j} \qquad (4.4)
\end{aligned}
$$

We use this result (4.4) in the synthesis of the algorithm, but we first have to incorporate the minimum and maximum processing constraints mentioned in the previous chapter (Chapter 3, Section 3.3.1)

4.2.4 Minimized Constrained Cost Function of the j^{th} processing stream

Let us factor the task's and processor's processing constraints mentioned earlier. We enforce $P_{Maxj} \geq P_k \geq p_{\mu,k}$ where, $p_{\mu,k}$ is the task's minimum recommended execution rate in base instructions per second (*n.Hz.*) and $P_{Max,j}$ is the maximum processing rate of the j^{th} processing stream. For a task $T_k \in T$, the minimum constrained cost function that factors the processing constraints is as follows.

$$C_{j,\min} = \left\{ \begin{array}{c} \alpha_j B_k \left(u_\varepsilon \lambda_j\right)^{\frac{1}{\alpha_j}} \left(\dfrac{u_{t,k}}{(\alpha_j - 1)}\right)^{\frac{\alpha_j - 1}{\alpha_j}} + u_{t,k} t_{\theta,k,j}, \text{ if } \mathrm{P}_{Max\,j} \geq \left(\dfrac{u_{t,k}}{(\alpha_j - 1) u_\varepsilon \lambda_j}\right)^{\frac{1}{\alpha_j}} \geq p_{\mu,k} \\ u_\varepsilon \lambda_j B_k (p_{\mu,k})^{\alpha_j - 1} + u_{t,k} (\dfrac{B_k}{p_{\mu,k}} + t_{\theta,k,j}), \text{ if} \left(\dfrac{u_{t,k}}{(\alpha_j - 1) u_\varepsilon \lambda_j}\right)^{\frac{1}{\alpha_j}} < p_{\mu,k} \\ u_\varepsilon \lambda_j B_k (P_{Max\,j})^{\alpha_j - 1} + u_{t,k} (\dfrac{B_k}{P_{Max\,j}} + t_{\theta,k,j}), \text{ if} \left(\dfrac{u_{t,k}}{(\alpha_j - 1) u_\varepsilon \lambda_j}\right)^{\frac{1}{\alpha_j}} > P_{Max\,j} \end{array} \right\}$$

for $\alpha_j \in (1,3]$.

If we assume loading times of **all** tasks are negligible ($t_{\theta,k,j} \approx 0$) the cost function reduces to:

$$C^{(2)}{}_{j,\min} = \left\{ \begin{array}{c} \alpha_j \left(u_\varepsilon \lambda_j\right)^{\frac{1}{\alpha_j}} \left(\dfrac{u_{t,k}}{(\alpha_j - 1)}\right)^{\frac{\alpha_j - 1}{\alpha_j}}, \text{ if } \mathrm{P}_{Max\,j} \geq \left(\dfrac{u_{t,k}}{(\alpha_j - 1) u_\varepsilon \lambda_j}\right)^{\frac{1}{\alpha_j}} \geq p_{\mu,k} \\ u_\varepsilon \lambda_j (p_{\mu,k})^{\alpha_j - 1} + \dfrac{u_{t,k}}{p_{\mu,k}}, \text{ if} \left(\dfrac{u_{t,k}}{(\alpha_j - 1) u_\varepsilon \lambda_j}\right)^{\frac{1}{\alpha_j}} < p_{\mu,k} \\ u_\varepsilon \lambda_j (P_{Max\,j})^{\alpha_j - 1} + \dfrac{u_{t,k}}{P_{Max\,j}}, \text{ if} \left(\dfrac{u_{t,k}}{(\alpha_j - 1) u_\varepsilon \lambda_j}\right)^{\frac{1}{\alpha_j}} > P_{Max\,j} \end{array} \right\}$$

This reduction above is not an equivalency reduction, but is rather a classification reduction because we use this function as a *discriminant* or for minimum comparison, and not for absolute value.

Observe that when a task's loading time is negligible, its computation volume does not influence its assignment to a given processor. We now have sufficient information to describe the SBDPP algorithm.

4.3 Single-Buffer Decision & Parallel Processing Algorithm (SBDPP)

1. User or OS specifies u_ε for all tasks and may specify different $u_{t,k}$ for each $T_k \in T$.
2. For an arriving task $T_k \in T$ we evaluate and compare the minimum processing cost ($C_{j\min}$)) of processing the task in each of the available processing streams. A task $T_k \in T$ should follow a stream $j*$ such that $C_{j*,\min} = \min\limits_{1\le j\le m}\{C_{j,\min} \mid N_j = 0\}$ thereby it acquires the label $T_{k,j*}$ and is processed by the $\vec{P}_{s,j*}$ processor at the optimum processing rate.

   If all the tasks' loading times are insignificant use $C^{(2)}{}_{j\min}$ instead of $C_{j\min}$. If all processors are homogenous and loading times are homogenous, ignore step 2 and utilize Round Robin dispatching.
3. Task $T_{k,j*}$ is executed by $\vec{P}_{s,j*}$ at the optimum processing rate:

$$P_{s,j*} = \left\{ \begin{array}{c} P*_k, \text{ if } P_{Max,j*} \ge P*_k = \left(\dfrac{u_{t,k}}{(\alpha_{j*}-1)u_\varepsilon \lambda_{j*}}\right)^{\frac{1}{\alpha_{j*}}} \ge p_{\mu,k} \\ p_{\mu,k}, \text{ if } \left(\dfrac{u_{t,k}}{(\alpha_{j*}-1)u_\varepsilon \lambda_{j*}}\right)^{\frac{1}{\alpha_{j*}}} < p_{\mu,k} \\ P_{Max,j*}, \text{ if } \left(\dfrac{u_{t,k}}{(\alpha_{j*}-1)u_\varepsilon \lambda_{j*}}\right)^{\frac{1}{\alpha_{j*}}} > P_{Max,j*} \end{array} \right\}$$

4. If $T_{k,j*}$ is to be cancelled/deleted or when it is completed, set $P_{s,j*} = 0$ and $N_{j*} = 0$.

The algorithm's dispatcher is described in steps 2. Step 3 is the algorithm's speed scaling function.

4.4 Calibrating the Ratio of Time and Energy Prices

Let us calibrate the ratio of unit prices ($u_{t,k} / u_{\varepsilon}$) that happen to correlate with the optimum processing rate and power consumption of a given task T_k . Generally, for a given $U_k = (u_{\varepsilon}, u_{t,k})$, associated with the task T_k, we want a one to one correspondence with $P*_k$ or $P_{s,j}$ which introduces the issue of calibration.

$$P_{Max,j} \geq P_{s,j} = P*_k = \left(\frac{u_{t,k}}{(\alpha_j - 1) u_{\varepsilon} \lambda_j} \right)^{\frac{1}{\alpha_j}} \geq p_{\mu,k}$$

$$\Rightarrow \left\{ \left(P_{Max,j} \right)^{\alpha_j} \geq \frac{u_{t,k}}{(\alpha_j - 1) u_{\varepsilon} \lambda_j} \geq \left(p_{\mu,k} \right)^{\alpha_j} \right\}$$

$$\Rightarrow \left\{ (\alpha_j - 1) \lambda_j \left(P_{Max,j} \right)^{\alpha_j} \geq \frac{u_{t,k}}{u_{\varepsilon}} \geq (\alpha_j - 1) \lambda_j \left(p_{\mu,k} \right)^{\alpha_j} \right\} \tag{4.5}$$

The relation (4.5) is consistent with minimum and maximum processing constraints.

Fig. 4.2 illustrates the optimum processing rate of a task as a function of the ratio of time and energy prices. For a given task, if a user wants the task's mode of operation to escape the economy region, he/she should be willing to spend more on time (increase $u_{t,k}$) or spend less on energy[27] (decrease u_{ε}) or rather accept a higher time cost relative to energy (increase $u_{t,k} / u_{\varepsilon}$).

[27] If the price of energy is determined by the OS based on time of day, a decrease in energy price can result from a transition between peak hours and off-peak hours.

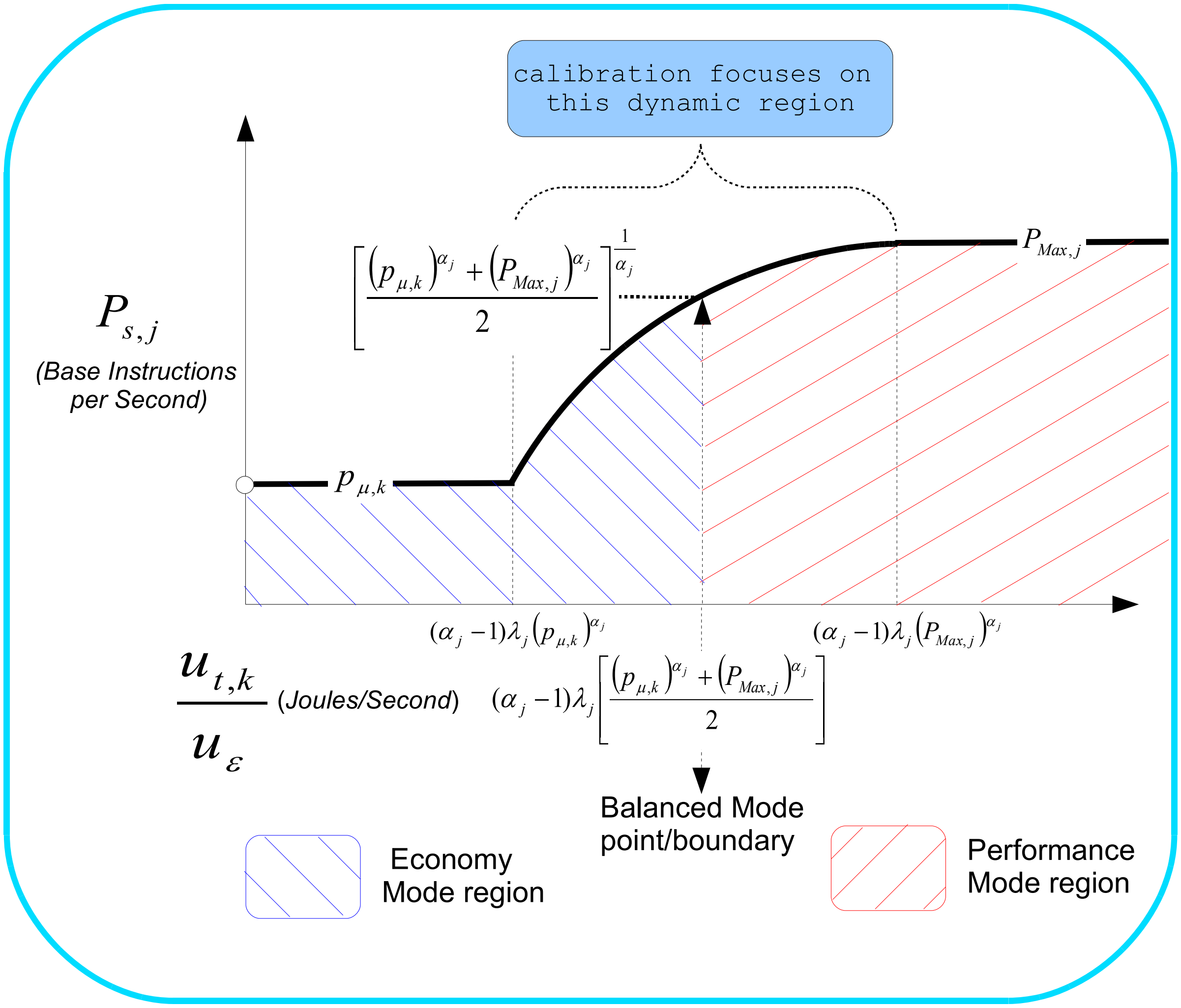


Fig. 4.2: A task's operating mode and optimum processing rate as a function of user-defined (time/energy) unit prices

Likewise, if a user wants the task's mode of operation to escape the performance region, he/she should be willing to spend less on time (decrease $u_{t,k}$) or spend more on energy (increase u_{ε}) or rather accept a lower time cost relative to energy (decrease $u_{t,k}/u_{\varepsilon}$). If an advanced user has a deep understanding of $u_{t,k}$ or u_{ε}, he or she would specify it, and allow the SBDPP algorithm to operate on the appropriate mode. Alternatively, a user may want to know the actual extent of a task's mode of operation, and may want to make a decision based on that rather than just the actual values of $u_{t,k}$ or u_{ε}. To do so in a consistent fashion, we need to use a metric that is a linear function of $\left(u_{t,k}/u_{\varepsilon}\right)$. Referring to Fig. 4.2, in order to achieve a linear calibration of the task's processing rate as a function of $\left(u_{t,k} / u_{\varepsilon}\right)$, we first identify each constant range (flat line portions of the economy and performance mode regions) in the graph and map each of these regions to a point value. We also need to *linearize* the curved portion of the figure (calibration region) via a non-linear transformation.

4.4.1 Determining a Task's Mode of Operation

In order to consistently determine a task's mode of operation we linearly calibrate the ratio of the user defined prices $\left(u_{t,k} / u_{\varepsilon}\right)$ by non-linearly transforming the task's processing rate. We achieve this by using the task's power consumption instead of the task's processing rate.

$\left(u_{t,k} / u_{\varepsilon}\right)$ is defined as the ratio of unit time price *($/Second*) and unit energy price *($/Joule*). It is convenient that the resulting dimension of $\left(u_{t,k} / u_{\varepsilon}\right)$ is indeed *Joule/Second* or *Watt*. According to equation (3.2), we see that $\left(u_{t,k} / u_{\varepsilon}\right)$ is the power consumption of a task multiplied by a constant factor of $(\alpha_j - 1)$.

$$\text{Let } \frac{u_{t,k}}{u_{\varepsilon}} = (\alpha_j - 1)\lambda_j \left[\left(p_{\mu,k}\right)^{\alpha_j} + \left(\left(P_{Max,j}\right)^{\alpha_j} - \left(p_{\mu,k}\right)^{\alpha_j} \right) S_j \right] \qquad (4.6)$$

where $S_j \in [0,1]$.

In Fig. 4.2, we see that a task's optimum processing rate as a function of $\left(u_{t,k} / u_{\varepsilon}\right)$ does not linearly determine the operation mode of a task. In Fig. 4.3, a task's power consumption as a function of $\left(u_{t,k} / u_{\varepsilon}\right)$ does indeed linearly determine the operation mode of a task.

This works because a task's power consumption is a non-linear transformation of its processing rate. In extension, observe that in Figs. 4.2 and 4.3, the balanced mode of a task's execution is identified by average of its minimum and maximum power consumption and not the average of its minimum and maximum processing rate.

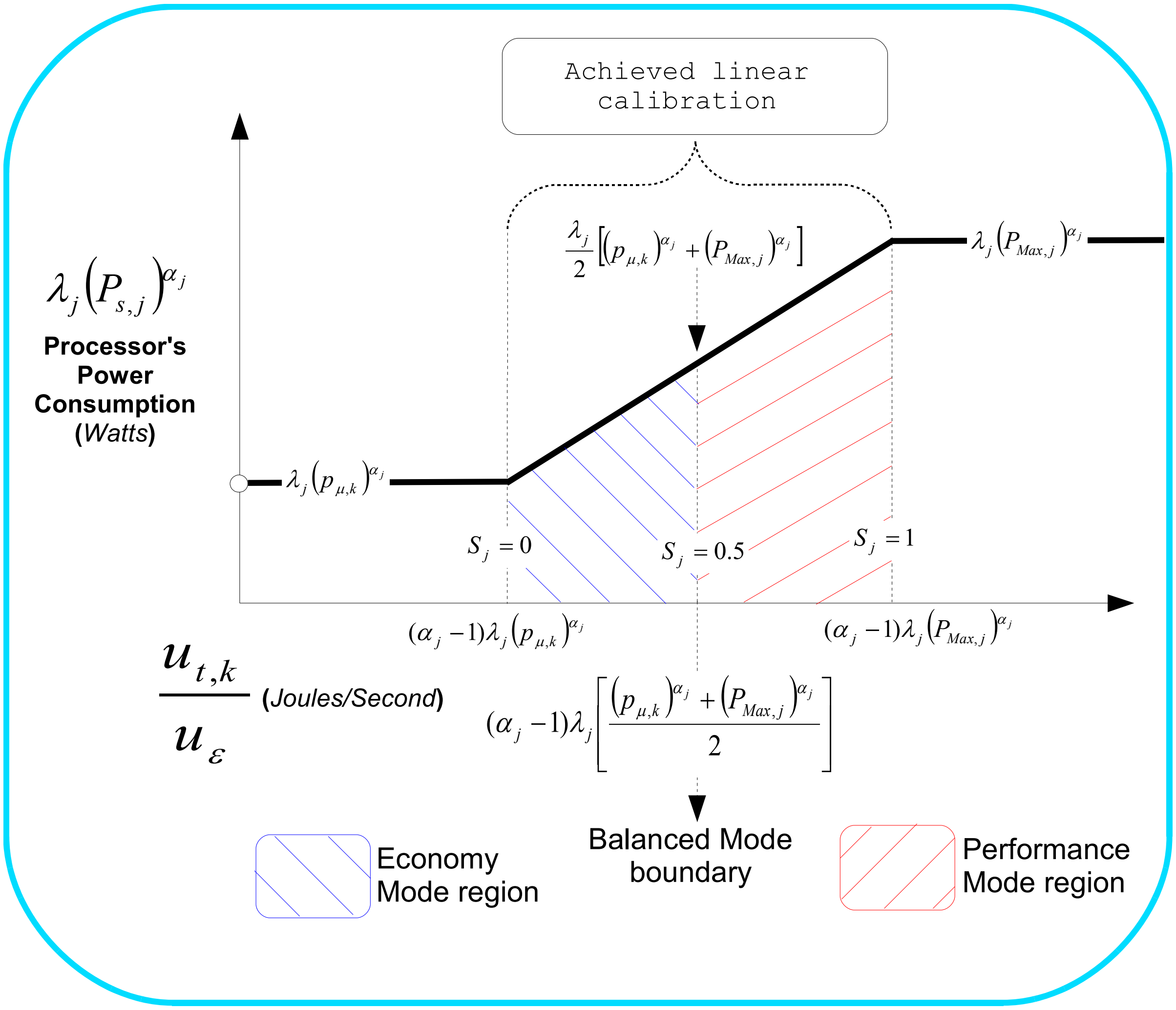

Achieved linear calibration
$\lambda_j(P_{s,j})^{\alpha_j}$
Processor's Power Consumption (Watts)
$\frac{\lambda_j}{2}\left[(p_{\mu,k})^{\alpha_j}+(P_{Max,j})^{\alpha_j}\right]$
$\lambda_j(P_{Max,j})^{\alpha_j}$
$\lambda_j(p_{\mu,k})^{\alpha_j}$
$S_j=0$
$S_j=0.5$
$S_j=1$
$(\alpha_j-1)\lambda_j(p_{\mu,k})^{\alpha_j}$
$(\alpha_j-1)\lambda_j(P_{Max,j})^{\alpha_j}$
$\frac{u_{t,k}}{u_\varepsilon}$ (Joules/Second)
$(\alpha_j-1)\lambda_j\left[\frac{(p_{\mu,k})^{\alpha_j}+(P_{Max,j})^{\alpha_j}}{2}\right]$
Economy Mode region
Balanced Mode boundary
Performance Mode region

Fig. 4.3. Illustrating linear calibration of a task's operation mode by utilizing the processor's power consumption during execution

In equation (4.6) and in Fig. 4.3, we define the auxiliary parameter S_j as the (*user specified) power sensitivity factor*. In Fig. 4.3, S_j is used to linearly parameterize a task's power consumption over the calibration region (spanned by ($u_{t,k}/u_\varepsilon$)). S_j informs us on the actual extent of power consumption while executing a task under software and hardware processing constraints, and it also linearly determines a task's mode of operation. Table 4.1 illustrates this.

Table 4.1: Interpretation of power sensitivity factor

S_j	**Interpretation**
0	Extreme Economy mode
0.25	75% Economy mode & 25% Performance mode (classified as Economy mode)
0.5	Balanced mode
0.75	25% Economy mode & 75% Performance mode (classified as Performance mode)
1	Extreme Performance mode

Using (4.6), it is quite convenient that the optimum processing rate that factors processing constraints reduces elegantly to:

$$P_{s,j} = \left(\frac{u_{t,k}}{(\alpha_j - 1)u_\varepsilon \lambda_j}\right)^{\frac{1}{\alpha_j}} = \left[(p_{\mu,k})^{\alpha_j} + \left((P_{Max,j})^{\alpha_j} - (p_{\mu,k})^{\alpha_j}\right)S_j\right]^{\frac{1}{\alpha_j}} \text{, for } S_j \in [0,1].$$

When $S_j \in [0,1]$, we get $P_{Max,j} \geq P_{s,j} = \left(\frac{u_{t,k}}{(\alpha_j - 1)u_\varepsilon \lambda_j}\right)^{\frac{1}{\alpha_j}} \geq p_{\mu,k}$ (as desired).

4.4.2 *Minimized Constrained Cost Function Using The Power Sensitivity Factor*

Recall the unconstrained cost function is as follows.

$$C*_{j\min} = C_j\Big|_{P*_k} = \alpha_j B_k \left(u_\varepsilon \lambda_j\right)^{\frac{1}{\alpha_j}} \left(\frac{u_{t,k}}{(\alpha_j - 1)}\right)^{\frac{\alpha_j - 1}{\alpha_j}} + u_{t,k} t_{\theta,k,j}$$

Using a variation of (4.6) we have

$$u_\varepsilon = \frac{u_{t,k}}{(\alpha_j - 1)\lambda_j}\left[(p_{\mu,k})^{\alpha_j} + \left((P_{Max,j})^{\alpha_j} - (p_{\mu,k})^{\alpha_j}\right)S_j\right]^{-1} \tag{4.7}$$

Substituting (4.7) into $C^*_{j\min}$ gives

$$\alpha_j B_k \left(\lambda_j \frac{u_{t,k}}{(\alpha_j - 1)\lambda_j} \left[\left(p_{\mu,k}\right)^{\alpha_j} + \left(\left(P_{Max,j}\right)^{\alpha_j} - \left(p_{\mu,k}\right)^{\alpha_j} \right) S_j \right]^{-1} \right)^{\frac{1}{\alpha_j}} \left(\frac{u_{t,k}}{(\alpha_j - 1)} \right)^{\frac{\alpha_j - 1}{\alpha_j}}$$

$$+ u_{t,k} t_{\theta,k,j}$$

$$= \alpha_j B_k \left(\left[\left(p_{\mu,k}\right)^{\alpha_j} + \left(\left(P_{Max,j}\right)^{\alpha_j} - \left(p_{\mu,k}\right)^{\alpha_j} \right) S_j \right]^{-\frac{1}{\alpha_j}} \right) \left(\frac{u_{t,k}}{(\alpha_j - 1)} \right)^{\frac{\alpha_j - 1}{\alpha_j} + \frac{1}{\alpha_j}} + u_{t,k} t_{\theta,k,j}$$

$$= \alpha_j B_k \left(\frac{u_{t,k}}{(\alpha_j - 1)} \right) \left(\left[\left(p_{\mu,k}\right)^{\alpha_j} + \left(\left(P_{Max,j}\right)^{\alpha_j} - \left(p_{\mu,k}\right)^{\alpha_j} \right) S_j \right]^{-\frac{1}{\alpha_j}} \right) + u_{t,k} t_{\theta,k,j}$$

$$= u_{t,k} \left[\left(\frac{\alpha_j B_k}{(\alpha_j - 1)} \right) \left(\left[\left(p_{\mu,k}\right)^{\alpha_j} + \left(\left(P_{Max,j}\right)^{\alpha_j} - \left(p_{\mu,k}\right)^{\alpha_j} \right) S_j \right]^{-\frac{1}{\alpha_j}} \right) + t_{\theta,k,j} \right] \quad (4.8)$$

In terms of classification accuracy, we can drop the $u_{t,k}$ term in (4.8) because it is a common multiplicative factor when comparing all processing streams. Again, this reduction is not an equivalency reduction with respect to value, but is equivalent in terms of classification ability (finding the comparative minimum).

Therefore the (reduced) constrained cost function, ($\hat{C}_{j\min}$) that factors the power sensitivity factor is follows.

$$\hat{C}_{j\min} = \left(\frac{\alpha_j B_k}{(\alpha_j - 1)} \right) \left(\left[\left(p_{\mu,k}\right)^{\alpha_j} + \left(\left(P_{Max,j}\right)^{\alpha_j} - \left(p_{\mu,k}\right)^{\alpha_j} \right) S_j \right]^{-\frac{1}{\alpha_j}} \right) + t_{\theta,k,j}$$

for $S_j \in [0,1]$

Notice that the unit prices of energy and response time are explicitly absent from this expression above, further more, if **all** loading times of all tasks are negligible we can reduce $\hat{C}_{j\min}$ to $\hat{C}^{(2)}{}_{j\min}$ by eliminating both the $t_{\theta,k,j}$ and B_k terms.

$$\hat{C}^{(2)}{}_{j\min} = \left(\frac{\alpha_j}{(\alpha_j - 1)}\right)\left(\left[(p_{\mu,k})^{\alpha_j} + \left((P_{Max,j})^{\alpha_j} - (p_{\mu,k})^{\alpha_j}\right)S_j\right]^{\frac{1}{\alpha_j}}\right)$$

for $S_j \in [0,1]$

We now have sufficient background to synthesize the two other versions of the SBDPP algorithm.

4.5 Alternative Versions of the SBDPP Algorithm

Using S_j, we present the "*Single Buffer Assisted Decision & Processing Algorithm (SBADPA)*" that extends the functionality of SBDPP by allowing the user or the OS of the mobile device/work station to further modify a task's unit cost of time/energy in order to achieve a desired (linearly controlled) mode of operation i.e., (economy/performance mode).

4.5.1 Single Buffer Assisted Decision & Processing Algorithm (SBADPA)

1. User or OS specifies u_ε for all tasks and may specify different $u_{t,k}$ for each $T_k \in T$.
2. For an arriving task $T_k \in T$, solve

   $$S_j = \frac{1}{\left((P_{Max,j})^{\alpha_j} - (p_{\mu,k})^{\alpha_j}\right)}\left[\frac{u_{t,k}}{(\alpha_j - 1)\lambda_j u_\varepsilon} - (p_{\mu,k})^{\alpha_j}\right], \forall_{1\le j\le m}.$$

   For each processing stream, If $S_j < 0$, set $S_j = 0$ and If $S_j > 1$, set $S_j = 1$ (satisfying processing constraints).

3. User or OS of mobile device can eliminate considering streams whose S_j values are undesirable **(optional)**.
4. For the given tasks $T_k \in T$, we evaluate and compare the minimum modified cost function of processing a task ($\hat{C}_{j\min}(.)$) in each of the available processing streams, where:

$$\hat{C}_{j,\min} = \left(\frac{\alpha_j B_k}{(\alpha_j - 1)} \right) \left(\left[(p_{\mu,k})^{\alpha_j} + \left((P_{Max,j})^{\alpha_j} - (p_{\mu,k})^{\alpha_j} \right) S_j \right]^{\frac{1}{\alpha_j}} \right) + t_{\theta,k,j}$$

If **all** task loading times are negligible use

$$\hat{C}_{j,\min} = \left(\frac{\alpha_j}{(\alpha_j - 1)} \right) \left(\left[(p_{\mu,k})^{\alpha_j} + \left((P_{Max,j})^{\alpha_j} - (p_{\mu,k})^{\alpha_j} \right) S_j \right]^{\frac{1}{\alpha_j}} \right)$$

5. A task $T_k \in T$ should follow a stream $j*$ such that $\hat{C}_{j,\min} = \min_{1 \le j \le m} \left\{ \hat{C}_{j,\min} \mid N_j = 0 \right\}$ thereby it acquires the label $T_{k,j*}$ and is processed by the $\vec{P}_{s,j*}$ processor at the optimum processing rate.

6. The optimum processing rate of the $\vec{P}_{s,j*}$ processor is

$$P_{s,j*} = \left[(p_{\mu,k})^{\alpha_j} + \left((P_{Max,j*})^{\alpha_j} - (p_{\mu,k})^{\alpha_j} \right) S_{j*} \right]^{\frac{1}{\alpha_j}}.$$

7. If $T_{k,j*}$ is to be cancelled/deleted or when it is completed, set $P_{s,j*} = 0$ and $N_{j*} = 0$.

If all processors are homogenous and loading times are homogenous, ignore step 4 and 5 and instead use Round Robin dispatching.

The algorithm's dispatcher is comprised of steps (4 & 5). Step 6 is the algorithm's speed scaling function.

4.5.2 Fixed Power Decision & Processing Algorithm (FPDPA)

We may want to consistently process a task based on the user specified power sensitivity factor (S_j) without explicitly requiring the user to provide a task's unit prices of time and energy. S_j Shows the degree of power consumption of executing each task and also linearly dictates the operation mode of a task's execution. This may be desirable because it reduces the burden of assigning the unit prices of time and energy of processing each task/s where only one parameter is assigned (S_j). On the other hand, it can be viewed as less flexible for advanced users because it does not explicitly factor each processor's power function parameters. In accordance with the abovementioned assumptions, we describe the *Single Buffer Fixed Power Decision & Processing Algorithm* (FPDPA) as follows.

- For each incoming task $T_k \in T$, $S_j \in [0,1]$ is specified by the user or through the computing device' OS (power setting).
- Carry out steps 3 through 7 of the SBADPA algorithm.
- If all processors are homogenous and loading times are homogenous, ignore step 4 and 5, and instead use Round Robin dispatching.

4.5.3 Extending the Algorithms to Allow Migration

Let as define a *Horizontal Migratory Operation* (HMO) as follows:

Among the tasks residing in the single buffers of each processing stream, re-arrange and migrate tasks to processing streams such that the tasks with the least remaining computation volumes are executed by processing streams with the lowest minimum (constrained) cost functions respectively. If processors are homogenous, there is no need to carry out HMOs.

If we assume a fixed or constant number of parallel processors, an HMO operation has a constant computational complexity. Moreover, we suggest that these horizontal migratory operations be conducted whenever a single buffer becomes vacant, or when all buffers are occupied for a sustained period of time. Investigation on other instances, or how frequently we should conduct HMOs deserves further attention but is not pursued as it falls beyond the scope of this thesis.

4.6 Analysis

4.6.1 Qualitative Comparison of Algorithms

The table below qualitatively compares the three versions of the SBDPP Algorithm by summarizing their relative strengths and weaknesses.

Table 4.2: Qualitative comparison of algorithms

Algorithm	**Pros**	**Cons**
SBDPP (Default)	-Low computational requirements. -Explicitly factors Processor's parameters. -Allows advanced users to specify unit prices of energy and response time.	-Cannot explicitly determine operation mode (power, balanced or economy) of tasks.
SBADPA	-Explicitly determines operation mode (power, balanced or economy) of tasks. -Explicitly factors Processor's parameters. -Allows advanced users to specify unit prices of energy and response time.	-Slightly more computationally expensive than SBDPP and FPDPA.
FPDPA	-Least Computationally expensive. -Allows user to determine operation mode (power, balanced or economy) of tasks. -Simplest to use.	-Does not explicitly allow the user to specify unit prices of time and energy. -User can not easily estimate amount of energy and response time consumed when tasks and processors are heterogeneous. -Unfair for fast processors.

4.6.2 Quantitative Comparison of Algorithm's Dispatcher to Round Robin

We would like to compare the performance of the algorithms to a comparable speed-scaling algorithm that at least considers most of the critical assumptions and preliminary modeling in which the algorithms are based on, but unfortunately to the best of our knowledge, no such algorithm exists. Let us instead assume there are N homogenous tasks that are to be processed (using single-buffered, single-threading) by two heterogeneous processors. We choose two processors, but the analysis can be extended to factor more processors. Assuming there exist an algorithm, let as call it TEST, that determines the optimum processing rate of a task by minimizing both the energy and time consumption, but uniformly distributes or assigns tasks to processors. In other words, it uses the Round Robin dispatcher and the SBDPP algorithm's speed scaling function. We would like to assess the dispatching performance of the SBDPP algorithm with respect to cost savings of both energy and response time that result from the intelligent assignment of tasks to heterogeneous processors. For simplicity, we also assume the following:

- Homogeneous tasks with equal computation volumes ($B_k = B$).
- A homogenous unit price of time and energy for all tasks.
- Negligible loading times of tasks($t_{\theta,k,j} = 0$).
- Unconstrained processing rates, i.e the optimum processing rates fall within the maximum hardware processing rates and the minimum software processing rates.
- Minimal traffic conditions .
- The first processor's power function parameters are $\alpha = 1.8$ and $\lambda = 1$.
- The second processor's function parameters are $\alpha_2 = 2$ and $\lambda_2 = 1$.
- Both the SBDPP and TEST algorithms use equally optimum speed scaling functions.

Following the above mentioned assumptions and using equations (3.4) and (3.5) as well as the algorithms dispatcher, Table 4.2 illustrates the cost savings.

Table 4.3: Dispatcher cost savings: SBDPP algorithm versus TEST

	SBDPP	**TEST**
Energy Consumption *(Joules)*	$NB\lambda_1(P_{s,1})^{\alpha_1-1}$	$0.5NB\lambda_1(P_{s,1})^{\alpha_1-1}$ $+0.5NB\lambda_2(P_{s,2})^{\alpha_2-1}$
Energy Cost *($)*	$u_\varepsilon NB\lambda_1(P_{s,1})^{\alpha_1-1}$	$0.5u_\varepsilon NB\lambda_1(P_{s,1})^{\alpha_1-1}$ $+0.5u_\varepsilon NB\lambda_2(P_{s,2})^{\alpha_2-1}$
Execution time *(Seconds)*	$NB(P_{s,1})^{-\alpha_1}$	$0.5NB(P_{s,1})^{-\alpha_1}$ $+0.5NB(P_{s,2})^{-\alpha_2}$
Execution Cost *($)*	$u_t NB(P_{s,1})^{-\alpha_1}$	$0.5u_t NB(P_{s,1})^{-\alpha_1}$ $+0.5u_t NB(P_{s,2})^{-\alpha_2}$
Total Cost Savings (Energy & Time) *($)*	$0.5u_\varepsilon NB\left(\lambda_2(P_{s,2})^{\alpha_2-1}-\lambda_1(P_{s,1})^{\alpha_1-1}\right)$ $+0.5u_t NB\left((P_{s,2})^{-\alpha_2}-(P_{s,1})^{-\alpha_1}\right)$	
Total Cost Savings Under Optimum Processing Rates *($)*	$0.5NB\left[\alpha_2(u_\varepsilon\lambda_2)^{\frac{1}{\alpha_2}}\left(\frac{u_t}{(\alpha_2-1)}\right)^{\frac{\alpha_2-1}{\alpha_2}}-\alpha_1(u_\varepsilon\lambda_1)^{\frac{1}{\alpha_1}}\left(\frac{u_t}{(\alpha_1-1)}\right)^{\frac{\alpha_1-1}{\alpha_1}}\right]$ $=NB\left[(u_\varepsilon.u_t)^{0.5}-0.9(u_\varepsilon)^{\frac{1}{1.8}}\left(\frac{u_t}{0.8}\right)^{0.8/1.8}\right]$	

The total cost savings (under optimal processing rates) is graphically illustrated in Fig. 4.4. According to equation (4.3), the ratio of energy price to time price dictates the optimum processing rate of a given task. Contrast this with the implication of Fig. 4.4; it shows us that optimum dispatching leads to cost savings that are dictated by the absolute values of **both** energy and time (prices) and not there ratio.

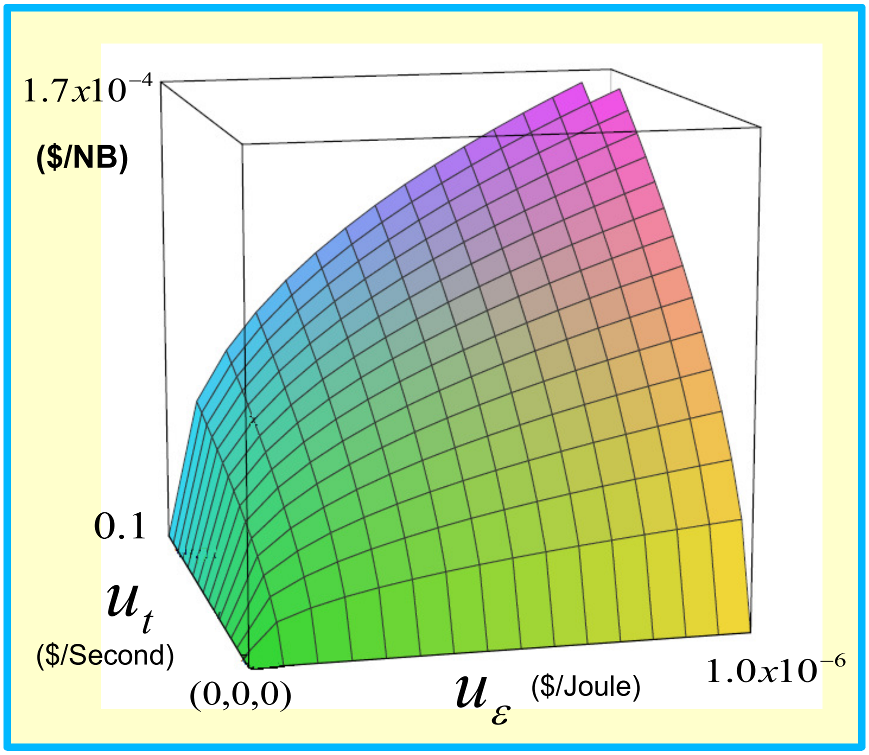


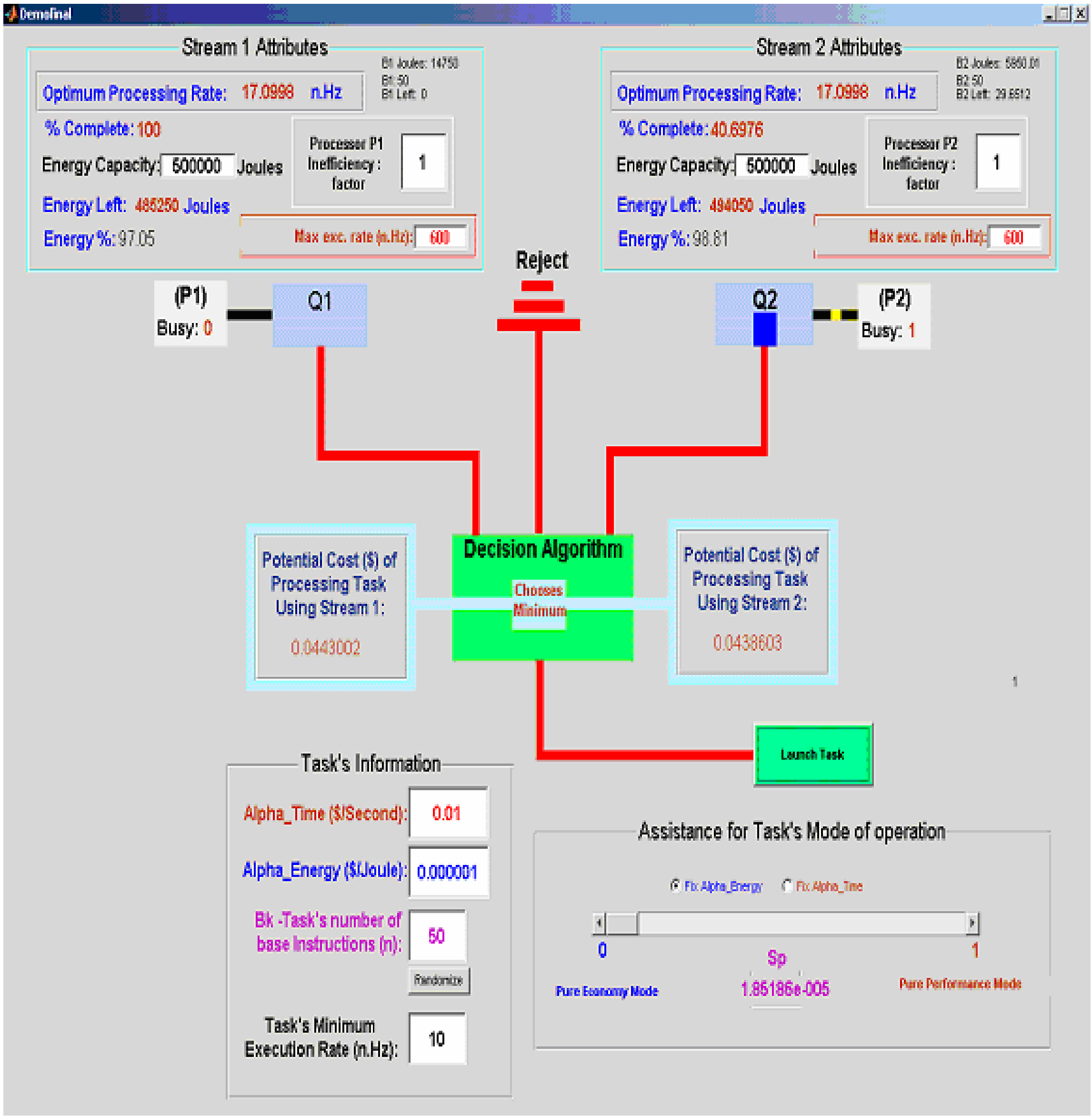
Fig. 4.4: Dispatcher cost savings: SBDPP algorithm versus TEST

This analytical comparison can be extended to factor more processing streams, complicated traffic conditions, and heterogeneous tasks that differ in computation volumes, time pricing and processing constraints. We emphasize that in this analysis, the TEST algorithm assigns half of the *N* tasks to each processor while our algorithms assign all the *N* tasks to the least expensive processor, and that traffic conditions are minimal. Also, for a fair comparison, both SBDPP and TEST algorithms utilize equally optimal speed-scaling functions.

4.6 Simulations

4.6.1 MATLAB Simulations

We initially simulated the algorithms in a program that was written using MATLAB (GUI). The algorithms were validated using a common graphical interface where we were able to numerically confirm the behavior of all the formulas used in context of all the assumptions made. For the sake of brevity, we initially simulated a simple case of two processing streams where a user has the ability (in real time) to launch one hypothetical task at a time. We factored all relevant processing and energy parameters. These parameters could be modified in real time.



Stream 1 Attributes

B1 Joules: 14750
B1: 50
B1 Left: 0

Optimum Processing Rate: 17.0998 n.Hz

% Complete: 100

Energy Capacity: 500000 Joules

Processer P1 Inefficiency : factor: 1

Energy Left: 485250 Joules

Energy %: 97.05

Max exc. rate (n.Hz): 600

Busy: 0

Stream 2 Attributes

B2 Joules: 5860.01
B2: 50
B2 Left: 29.6512

Optimum Processing Rate: 17.0998 n.Hz

% Complete: 40.6976

Energy Capacity: 500000 Joules

Processor P2 Inefficiency : factor: 1

Energy Left: 494050 Joules

Energy %: 98.81

Max exc. rate (n.Hz): 600

Busy: 1

Potential Cost ($) of Processing Task Using Stream 1: 0.0443002

Potential Cost ($) of Processing Task Using Stream 2: 0.0438603

Task's Information

Alpha_Time ($/Second): 0.01

Alpha_Energy ($/Joule): 0.000001

Bk -Task's number of base Instructions (n): 50

Task's Minimum Execution Rate (n.Hz): 10

Assistance for Task's Mode of operation

● Fix Alpha_Energy ○ Fix Alpha_Time

0 Pure Economy Mode

Sp 1.85186e-005

1 Pure Performance Mode

Fig. 4.5. MATLAB GUI simulation validating all three algorithms

In accordance with the rules of the algorithms, the user is allowed to specify the unit cost of energy and time of each hypothetical task prior to launch. The minimum processing rate of each task and the maximum processing rate of each processor are modifiable as well. The user is also allowed to either randomize the computation volume of task or to specify one. We realized that if initially two tasks are consecutively launched, and if a third task is launched before any of the first two tasks has been processed, both algorithms are forced to reject the third task. This limitation inspired us to consider the multi-buffer scenario where the memory queues of each processing stream have the capacity to queue up a finite arbitrary number of tasks.

4.6.1 Java Simulations & Insights

We extended this simulation to include more processors by conducting a discrete time based simulation written in Java and we gathered some insights (reported below).

> Let R be the response time of the average task (with average properties) be executed by the average single-buffered processor with (average hardware parameters). Through simulation findings, it turns out that if we have m parallel processors, the ideal deterministic arrival period (in the long run) that maximizes system utilization is roughly R/m for heterogeneous processors/tasks and is exactly R/m for homogenous processors and tasks.

In other words, m/R is the maximum deterministic arrival rate that prevents rejections for homogenous tasks running in the long run on homogenous, single-buffered processors. We call R/m the ideal (deterministic) inter-arrival period. We use this finding as benchmark or criterion to evaluate findings on the multi-buffered scenario presented in the next chapter.

4.6 Conclusions

We have synthesized and simulated the SBDPP algorithm and its variations. They can be used for optimized local parallel heterogeneous computing of mobile devices or energy aware work stations. We focused on single buffer, single threading where no processor executes more than a single task at any given time. We also assumed the constraint of imposing a maximum limit of one task in each memory queue for each corresponding processor (single buffer case). The algorithm and its variations run in real time to optimally dictate which processor among a multiple set of parallel processors should process an incoming task, and they also explicitly determine the optimum processing rate of executing each tasks residing in each processor's single-buffer. The three versions of the algorithm are conceptually similar, but differ on their application and they each have dispatchers and dynamic speed-scaling functions of constant computational complexity.

The algorithms provide some insights. They all inform us that a task's computation volume (B_k) does influence its processing cost when the loading times of tasks are not negligible, which in turn influences the actual processing stream that will process the task. But counter-intuitively, the optimum processing rate of a task is neither a function of its computation volume nor is it a function of its loading time. Moreover, when the loading times tasks are negligible, a tasks computation volume does not influence the actual processing stream that will process the task.

The algorithms and their variations were extended to allow migration. This was suggested through carrying out migration operations (*HMO*) of constant computational complexities (assuming a constant number of parallel processors) but a deep analysis on this front was not pursued.

The optimum processing rate of a task under the single buffer scenario was found to be a function of the unit price of time divided by that of energy as well as the processors power function parameters. Further more, through a simple analytical example, it was shown that our algorithm's dispatcher can outperform the Round Robin dispatcher with cost savings correlated with the absolute values of both the energy and time prices.

Through simulations we observed and constructed a relationship between the average response time of a given task and the ideal deterministic inter-arrival period that maximizes system utilization; i.e. if we let R be the response time of the average task (with average properties) be executed by the average single-buffered processor with (average hardware parameters). It turns out that if we have m parallel processors, the ideal deterministic arrival period that maximizes system utilization is approximately R/m for heterogeneous processors/tasks and is exactly R/m for homogenous processors and tasks. In extension, let γ_{in} be the rate at which tasks enter the decision algorithm. Also, let γ_{out} be the aggregate rate at which processed tasks exit the parallel streams. When $\gamma_{out} \geq \gamma_{in}$, the algorithm and its variations work well[28]. In practice, this will likely not be the case because if the rate of incoming tasks grows unpredictably, all parallel streams will quickly get clogged up (due to limited memory queue capacity - single buffers), and soon we will have to either reject incoming tasks or we will have to queue them up before the decision stage. Either way, this leads to undesired queuing delays that compromise the functionality and optimality of the algorithms[29]. One way to mitigate this problem is to reduce the response time of arriving tasks by limiting their computation volumes, but this strategy falls beyond the scope of this thesis. A second way is to reduce the response time of tasks by increasing their optimum processing rate, but that would imply that the user should be willing to accept a higher price of response time relative to that of energy[30]. The practical way to mitigate the problem is to arbitrarily increase the memory queue capacity for each processing stream (*multi-buffered* processors). This reduces the number of task rejections and will additionally provide more time for a decision algorithm to appropriately allocate arriving tasks to processors. In the next chapter we consider the multi buffered scenario where the memory queues of each processor have the capacity to store an arbitrary number of tasks: $N_j \geq 1, \forall_{1 \leq j \leq m}$ (multi- buffer case).

[28] Simulations suggest that if we are to prevent rejections, the maximum value of γ_{in} is m / R (in the long run) for a system with homogenous tasks and single buffered processors.

[29] Theoretically, increasing the number of processing streams also resolves the problem but is generally not feasible. Once the hardware of a mobile device or work station is built or fabricated, increasing the number of processing cores (or processing streams) is generally impractical if not substantially expensive.

[30] The power function parameters of the processors are assumed to be given and fixed. From a design perspective, utilizing processors with modest power consumption functions (generally, small values of α_j and λ_j) will lead to increased optimum processing rates that reduce execution and response times of tasks during processing (see the speed scaling function of the SBDPP algorithm).

Chapter 5: Cost Minimization of Single-threaded, Multi-buffered Processors

5.1 Introduction

This chapter introduces a multiprocessor speed-scaled scheduling algorithm named "*Single-threading Multi-Buffer Scheduling & Parallel Processing Algorithm* (**SMBSPP**)". The goal of this algorithm is to minimize the performance metric, the total cost of response time and energy consumption of tasks (**TCRTEC**). By utilizing the single-threaded (multi-buffered) computing architecture, the **SMBSPP** algorithm makes three key contributions:

- A novel task dispatcher which assigns a task to a given processor based on the *Minimum among Minimized Costs of Virtually Introducing the Task to each Processing Stream* (**MMCVITPS).** It dictates which of the heterogeneous processors should process each arriving task/s based on classifying a set of minimized potential aggregate cost functions that is each associated with a processing stream.
- A novel dynamic speed-scaling function, which we name, "*Optimum Single-Threading Speed Scaling Function*" (**OSTSSF**) that explicitly determines the optimum processing rate of a given processor as a function of the following:
    - The parameters of the processors power function.
    - The unit price of energy.
    - The sum of the unit prices of response time of all tasks residing in the processor's buffer.
- A novel preemptive service discipline called Smallest remaining Computation Volume Per unit Price of response Time (**SCVPPT**) to schedule the tasks on the assigned processor. This discipline minimizes the **TCRTEC** performance metric and also conveniently allows the user to dynamically upgrade or degrade the priority of tasks.

The first two contributions are achieved through solving a set of multidimensional convex optimization problems.

In this work, we focus on multi-buffer, single-threading where a set of tasks is allocated to a given processor, but only one task is processed at a time until completion unless preemption is dictated by the service discipline. In order to practically find the optimal speed of a processor, the maximum allowable rate of the processor and the minimum recommended rate of execution for a task are considered as constraints.

We validated the performance of the **SMBSPP** algorithm by conducting discrete time based simulations (as well as analytical techniques). In this front, we briefly report on three major findings. Firstly, our simulation results show that our MMCVITPS dispatcher works well with heterogeneous processors and drastically outperforms the classic Round Robin dispatcher with cost savings exceeding 100% on average even when processors are mildly heterogeneous[31]. Secondly, simulation results also show that our SCVPPT scheduling discipline outperforms the two known service disciplines, Shortest Remaining Processing Time (SRPT) and the First Come First Serve (FCFS), in terms of minimizing the TCRTEC performance metric. SRPT policy always selects for service the task that has the least remaining service time and it is a preemptive policy. FCFS, on the other hand, is a non-preemptive policy that selects the tasks for service in order of their arrivals. Lastly, we analytically compare our dynamic speed scaling function (**OSTSSF**) to a comparable and competitive speed scaling function found in current literature ($\tilde{p}(\beta n)^{-1}$). We corroborated this analytical comparison with elaborate simulations to show that our **OSTSSF** out performs this competitive speed scaling function[32] in terms of the TCRTEC performance metric. Furthermore, we offer a recommendation to improve this speed scaling function ($\tilde{p}(\beta n)^{-1}$).

This chapter is organized as follows. Section 5.2 utilizes expressions found in section 3.6 (chapter 3) to formally state the problem and synthesize the SMBSPP algorithm.

[31] Power function parameters were conservatively chosen to differ from the mean by at most 8% from average.

[32] There is a special condition in which OSTSSF and $\tilde{p}(\beta n)^{-1}$ achieve equal results. See Section 5.6.2.

Section 5.3 describes the SMBSPP algorithm in detail. Section 5.4 provides a simple example that analytically demonstrates the ability of the algorithm to robustly handle the dynamic inclusion of tasks. Section 5.5 provides simulation results that evaluate the overall performance of the algorithm using a variety of performance metrics. Also in this section, we demonstrate the performance of the algorithm's dispatcher in comparison to the Round Robin dispatcher under three service disciplines and various traffic conditions. In section 5.6, we use analysis and simulations to show that our speed scaling function (OSTSSF) achieves better results than a comparable speed scaling function found in current literature, and further more, we offer a recommendation of improvement.

5.2 Problem Formulation

5.2.1 Processing Streams with Multiple Buffers

Fig. 5.1 illustrates the parallel multi-buffer scenario: each processing stream has a memory queue that has a capacity to store a arbitrary finite number of tasks. For a set of arriving tasks, we are essentially trying to find the optimum dispatcher, speed scaling function and service discipline that minimizes the total cost of response time and energy consumption (**TCRTEC**) of executing these tasks where the unit price of response time is heterogeneous. The unit price of response time for each task may be different because we allow the user to dynamically influence the priority of a task's execution through the following ways:

- If a user is willing to pay more for a task's response time, the algorithm's speed scaling function (**OSTSCF**) increases hence executing the task at a faster rate at the expense of energy and vice versa.
- Under our proposed service discipline, **SCVPPT** (which is a generalized version of SRPT) will prioritize the task accordingly to the smallest remaining computation volume per unit price of response time. Therefore a user can maintain or even improve the priority of a large task by accepting higher unit price of response time or even degrade the priority of a small non-urgent task by setting a sufficiently small unit price of response time.

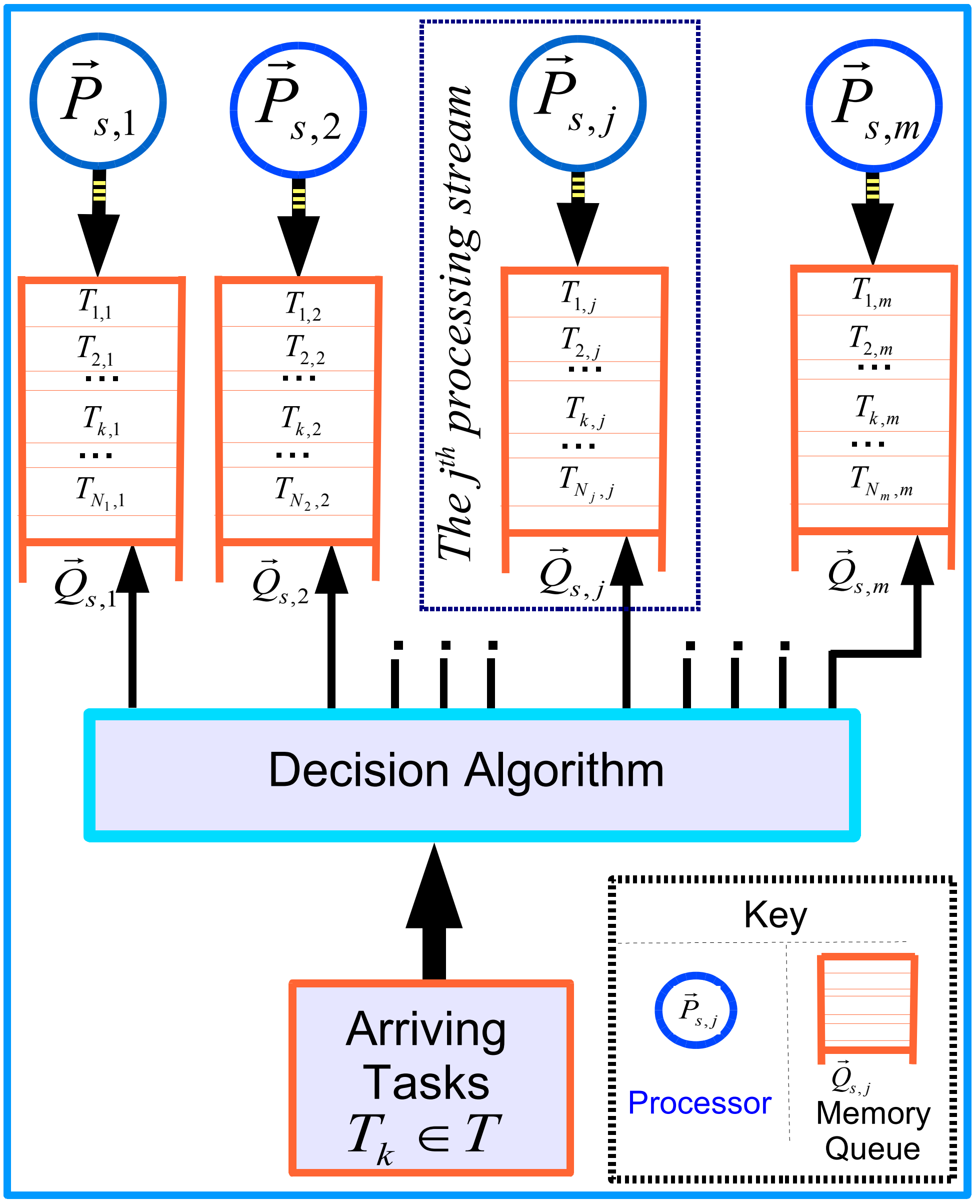


Fig. 5.1: The parallel multi-buffer scenario

5.2.2 The Cost Function of the j^{th} Processing Stream

Let us assume that the j^{th} processing stream has $\mathrm{N_j}$ tasks already queued up in its corresponding memory queue $(\vec{Q}_{s,j})$. Let us also assume that the aggregate cost function of the j^{th} processing stream be C_j. This cost function aggregates the total cost of response time and energy consumption of these $\mathrm{N_j}$ tasks. Also let $C_{k,j}$ be the cost of response time and energy consumption of the task stored at the k^{th} index of the $\vec{Q}_{s,j}$ memory queue/multi-buffer.

Using vector notation and dot product operations, we have:

$$C_j = \sum_{k=1}^{\mathrm{N_j}} C_{k,j} = \sum_{k=1}^{\mathrm{N_j}} \left\{ U_k \bullet \left[\varepsilon_k , \sum_{r=1}^{k} (t_r + t_{\theta,r,j}) \right] \right\}$$

More explicitly, using (3.4) and (3.5) from chapter 3 (section 3.6) we have:

$$C_j = \sum_{k=1}^{\mathrm{N_j}} C_{k,j} = \underbrace{u_\varepsilon \overbrace{\lambda_j B_1 (P_1)^{\alpha_j -1}}^{\text{Task 1 energy (J)}}}_{\text{Task 1 energy cost (\$)}} + \overbrace{u_{t,1} \underbrace{\left(\frac{B_1}{P_1} + t_{\theta,1,j} \right)}_{\text{Task 1 response time (S)}}}^{\text{Task 1 response time cost (\$)}}$$

$$+ \underbrace{u_\varepsilon \overbrace{\lambda_j B_2 (P_2)^{\alpha_j -1}}^{\text{Task 2 energy (J)}}}_{\text{Task 2 energy cost (\$)}} + \overbrace{u_{t,2} \underbrace{\left[\underbrace{\left(\frac{B_2}{P_2} + t_{\theta,2,j} \right)}_{\text{Task1 time(S)}} + \underbrace{\left(\frac{B_1}{P_1} + t_{\theta,1,j} \right)}_{\text{Task 2 time(S)}} \right]}_{\text{Task 2 response time(S)}}}^{\text{Task 2 response time cost (\$)}}$$

$$+ u_\varepsilon \lambda_j B_3 (P_3)^{\alpha_j -1} + u_{t,3} \left(\frac{B_3}{p_3} + \frac{B_2}{P_2} + \frac{B_1}{P_1} + t_{\theta,3,j} + t_{\theta,2,j} + t_{\theta,1,j} \right)$$

$$+ \ldots$$

$$+u_{\varepsilon}\lambda_{j}B_{\mathrm{N_j}}(P_{\mathrm{N_j}})^{\alpha_j-1}+u_{t,\mathrm{N_j}}\left(\frac{B_{\mathrm{N_j}}}{P_{\mathrm{N_j}}}+\left(\sum_{r=1}^{\mathrm{N_j}}t_{\theta,r,j}\right)+\left(\sum_{r=1}^{\mathrm{N_j}-1}\frac{B_r}{P_r}\right)\right)$$

$$=\sum_{\ell=1}^{\mathrm{N_j}}\left\{u_{\varepsilon}\lambda_{j}B_{\ell}(P_{\mathrm{N_j}})^{\alpha_j-1}+u_{t,\ell}\left(\sum_{r=1}^{\ell}\left(\frac{B_r}{P_r}+t_{\theta,r,j}\right)\right)\right\} \tag{5.1}$$

Rearranging the terms of (5.1), we have:

$$C_j=\sum_{k=1}^{\mathrm{N_j}}C_{k,j}=u_{\varepsilon}\lambda_{j}B_{1}(P_{1})^{\alpha_j-1}+u_{t,1}\left(\frac{B_1}{P_1}+t_{\theta,1,j}\right)+\left(\frac{B_1}{P_1}\sum_{r=2}^{\mathrm{N_j}}u_{t,r}\right)$$

$$+\,u_{\varepsilon}\lambda_{j}B_{2}(P_{2})^{\alpha_j-1}+u_{t,2}\left(\frac{B_2}{P_2}+t_{\theta,2,j}+t_{\theta,1,j}\right)+\left(\frac{B_2}{P_2}\sum_{r=3}^{N_{\mathrm{j}}}u_{t,r}\right)$$

$$+\ldots$$

$$+u_{\varepsilon}\lambda_{j}B_{N_j}(P_{N_j})^{\alpha_j-1}+u_{t,N_j}\left(\frac{B_{N_j}}{P_{N_j}}+\sum_{r=1}^{\mathrm{N_j}}t_{\theta,r,j}\right)$$

$$=\sum_{k=1}^{\mathrm{N_j}}\left\{u_{\varepsilon}\lambda_{j}B_{k}(P_{k})^{\alpha_j-1}+\left(\frac{B_k}{P_k}\sum_{r=k}^{\mathrm{N_j}}u_{t,r}\right)+\left(u_{t,k}\sum_{r=1}^{k}t_{\theta,r,j}\right)\right\} \tag{5.2}$$

5.2.3. The Minimized Cost Function of the j^{th} Processing Stream

For each j^{th} stream, we have an "N_j" dimensional optimization problem. The adjustable parameters are the theoretical processing rates (P_k) of the tasks: $T_k \in T \mid k \in \{1,2...N_j\}$ as well as their service sequence in the j^{th} processing stream. We optimize C_j as defined by (7) as follows.

$$\frac{\partial C_j}{\partial P_k} = (\alpha_j - 1)u_\varepsilon \lambda_j B_k (P_k)^{\alpha_j - 2} - \frac{B_k}{{P_k}^2}\sum_{r=k}^{\mathrm{N_j}} u_{t,r} = 0 \text{ for } k \in \{1,2...N_j\}.$$

The solution of our optimization problem is:

$$P_k = P'_k = \left(\frac{1}{(\alpha_j - 1)u_\varepsilon \lambda_j}\sum_{r=k}^{\mathrm{N_j}} u_{t,r}\right)^{\frac{1}{\alpha_j}} \text{ for } k \in \{1,2..N_j\} \text{ and } \alpha_j \in (1,3]$$

$$\left.\frac{\partial^2 C_j}{\partial {P_k}^2}\right|_{P_k = P'_k} = \left.\left\{(\alpha_j - 2)(\alpha_j - 1)u_\varepsilon \lambda_j B_k (P_k)^{\alpha_j - 3} + 2\frac{B_k}{{P_k}^3}\sum_{r=k}^{\mathrm{N_j}} u_{t,r}\right\}\right|_{P_k = P'_k}$$

$$= \left.\left\{\frac{B_k}{{P_k}^3}\left((\alpha_j - 2)(\alpha_j - 1)u_\varepsilon \lambda_j (P_k)^{\alpha_j} + 2\sum_{r=k}^{\mathrm{N_j}} u_{t,r}\right)\right\}\right|_{P_k = P'_k}$$

$$= \alpha_j B_k \sum_{r=k}^{\mathrm{N_j}} u_{t,r}\left(\frac{(\alpha_j - 1)u_\varepsilon \lambda_j}{\sum_{r=k}^{\mathrm{N_j}} u_{t,r}}\right)^{\frac{3}{\alpha_j}} > 0 \text{ for } k \in \{1,2...N_j\}$$

and $\alpha_j \in (1,3]$

In order to confirm a global minima of C_j, we generate and examine the Hessian (H) matrix [24].

$$\text{Let } f_1 = \frac{\partial C_j}{\partial P_1},\ f_2 = \frac{\partial C_j}{\partial P_2} \ldots f_{N_j} = \frac{\partial C_j}{\partial P_{N_j}}$$

$$\left(\left.\frac{\partial C_j}{\partial P_1}\right|_{P_1 = P'_1} = 0\right), \left(\left.\frac{\partial C_j}{\partial P_2}\right|_{P_2 = P'_2} = 0\right) \cdots \left(\left.\frac{\partial C_j}{\partial P_{N_j}}\right|_{P_{N_j} = P'_{N_j}} = 0\right)$$

The Hessian (H) is defined as:

$$H = \left.\begin{bmatrix} \frac{\partial f_1}{\partial P_1} & \frac{\partial f_1}{\partial P_2} & \cdots & \frac{\partial f_1}{\partial P_{N_j}} \\ \frac{\partial f_2}{\partial P_1} & \frac{\partial f_2}{\partial P_2} & \cdots & \frac{\partial f_2}{\partial P_{N_j}} \\ \cdots & \cdots & \cdots & \cdots \\ \frac{\partial f_i}{\partial P_1} & \frac{\partial f_i}{\partial P_2} & \cdots & \frac{\partial f_i}{\partial P_{N_j}} \end{bmatrix}\right|_{(P_1,P_2,\ldots P_{N_j})=(P'_1,P'_2,\ldots P'_{N_j})}$$

Implementing the definition above, we obtain the following.

$$H = \begin{bmatrix} \alpha_j B_1 \sum_{r=1}^{\mathrm{N_j}} u_{t,r} \left(\frac{(\alpha_j - 1) u_\varepsilon \lambda_j}{\sum_{r=1}^{\mathrm{N_j}} u_{t,r}} \right)^{\frac{3}{\alpha_j}} & 0 & 0 & 0 \\ 0 & \alpha_j B_2 \sum_{r=2}^{\mathrm{N_j}} u_{t,r} \left(\frac{(\alpha_j - 1) u_\varepsilon \lambda_j}{\sum_{r=2}^{\mathrm{N_j}} u_{t,r}} \right)^{\frac{3}{\alpha_j}} & 0 & 0 \\ 0 & 0 & \cdots & 0 \\ 0 & 0 & 0 & \alpha_j B_{\mathrm{N_j}} u_{t,\mathrm{N_j}} \left(\frac{(\alpha_j - 1) u_\varepsilon \lambda_j}{u_{t,\mathrm{N_j}}} \right)^{\frac{3}{\alpha_j}} \end{bmatrix}$$

Since the main diagonal of H has all non-negative entries i.e.:

$\alpha_j B_k \sum_{r=k}^{\mathrm{N_j}} u_{t,r} \left(\frac{(\alpha_j - 1) u_\varepsilon \lambda_j}{\sum_{r=k}^{\mathrm{N_j}} u_{t,r}} \right)^{\frac{3}{\alpha_j}} > 0$ for $k \in \{1,2 \ldots N_j\}$ and all the off-diagonal entries are all zero, we conclude that $P_k = \left(\frac{1}{(\alpha_j - 1)u_\varepsilon \lambda_j} \sum_{r=k}^{\mathrm{N_j}} u_{t,r} \right)^{\frac{1}{\alpha_j}}$ for $k \in \{1,2..N_j\}$ and $\alpha_j \in (1,3]$ globally minimizes C_j.

5.2.4. *The Minimized Constrained Cost Function of the j^{th} Processing Stream*

$\forall T_k \in T \mid T_k \in \vec{Q}_{s,j}$, let us factor in the task and processor stream processing constraints mentioned earlier (Chapter 3, Section 3.3.1).

We enforce $P_{Max,j} \geq P_k \geq p_{\mu,k}$ where, $p_{\mu,k}$ is the task's minimum recommended execution rate in base instructions per second (*n.Hz.*). The minimum constrained cost function that factors the processing constraints is:

$$C_{j_{\min}}(N_j) = \sum_{k=1}^{\mathrm{N_j}} \left(u_\varepsilon \lambda_j B_k (P*_k)^{\alpha_j - 1} + \left(\frac{B_k}{P*_k} \sum_{r=k}^{\mathrm{N_j}} u_{t,r} \right) + \left(u_{t,k} \sum_{r=1}^{k} t_{\theta,r,j} \right) \right) \quad \$ \qquad (5.3)$$

and

$$P*_k = \left\{ \begin{array}{c} \left(\frac{1}{(\alpha_j - 1)u_\varepsilon \lambda_j} \sum_{r=k}^{\mathrm{N_j}} u_{t,r} \right)^{\frac{1}{\alpha_j}}, \text{ if } \mathrm{P}_{Max,j} \geq \left(\frac{1}{(\alpha_j - 1)u_\varepsilon \lambda_j} \sum_{r=k}^{\mathrm{N_j}} u_{t,r} \right)^{\frac{1}{\alpha_j}} \geq p_{\mu,k} \\ p_{\mu,k}, \text{ if} \left(\frac{1}{(\alpha_j - 1)u_\varepsilon \lambda_j} \sum_{r=k}^{\mathrm{N_j}} u_{t,r} \right)^{\frac{1}{\alpha_j}} < p_{\mu,k} \\ P_{Max,j}, \text{ if} \left(\frac{1}{(\alpha_j - 1)u_\varepsilon \lambda_j} \sum_{r=k}^{\mathrm{N_j}} u_{t,r} \right)^{\frac{1}{\alpha_j}} > P_{Max,j} \end{array} \right\} \quad n.Hz \qquad (5.4)$$

for $k \in \{1,2...N_j\}$ & $\alpha_j \in (1,3]$

$P*_k$ is the optimum constrained processing rate of potentially executing the task stored in the k^{th} index of the $\vec{Q}_{s,j}$ memory queue.

5.3 Algorithms Description

This section describes the SMBSPP algorithm. First we describe our MMCVITPS dispatcher and our SCVPPT scheduling policy. Then we present our algorithm.

*5.3.1 SMBSPP Algorithm's Dispatcher (**MMCVITPS**)*

Before presenting the complete algorithm description (V.C), let us describe in words how its dispatcher **(**MMCVITPS) works. For an arriving task, MMCVITPS hypothetically or virtually assumes the potential aggregate cost of virtually introducing the task (according to a service discipline) to each of the processing streams. It then virtually minimizes the aggregate cost function of each processing stream by again virtually re-adjusting the processing rates of all tasks in the queues (of each processing stream) including the task in question. It then finally decides on the processing stream with the lowest potential (minimized) aggregate cost. This decision will likely dynamically affect the speed function of the chosen processing stream's processor. We mathematically describe the speed scaling function in section 5.3.3.

*5.3.2 SMBSPP Algorithm's Service Discipline/Policy **(SCVPPT)***

In this service discipline, arriving tasks are sorted in each processing stream's memory queue or multi-buffer from the lowest index (highest priority) according to their smallest remaining computation volume per unit price of response time $\left(B_k \,/\, u_{t,k}\right)$.

5.3.3 Single-threading Multi-buffer Scheduling & Processing Algorithm (SMBSPP)

1. User or OS specifies u_ε for all tasks and may specify different $u_{t,k}$ for each $T_k \in T$.
2. For an arriving task, $T_k \in T$, we evaluate and compare the minimum potential processing cost, $C_{j\min}(N_j + 1)$ of virtually introducing and processing the arriving task in each of the available processing streams $(1 \le \mathrm{j} \le m)$. The task virtually acquires a position index according to $B_k \,/\, u_{t,k}$ (**SCVPPT**) in each of the processing streams.
3. Using equations (5.3) and (5.4), the task should follow a stream $j*$ such that $C_{j*\min}(N_{j*} + 1) = \min\limits_{1 \le j \le m} \{C_{j\min}(N_j + 1)\}$ thereby it acquires the position index according to $(B_k \,/\, u_{t,k})$ **(SCVPPT)** and will be processed by the $\vec{P}_{s,j*}$ processor at some adjusted optimum processing rate.
4. Update N_{j*} .
5. The task stored at system index $(1, j*)$ i.e., the task $T_{1,j*}$, is executed by the $\vec{P}_{s,j*}$ processor at the optimum adjusted processing rate defined below: (e)

$$P_{s,j*} = \left\{ \begin{array}{c} \left(\dfrac{1}{(\alpha_{j*} - 1) u_\varepsilon \lambda_{j*}} \displaystyle\sum_{r=1}^{\mathrm{N}_{j*}} u_{t,r} \right)^{\frac{1}{\alpha_{j*}}}, \text{ if } \mathrm{P}_{Maxj*} \ge \left(\dfrac{1}{(\alpha_{j*} - 1) u_\varepsilon \lambda_{j*}} \displaystyle\sum_{r=1}^{\mathrm{N}_{j*}} u_{t,r} \right)^{\frac{1}{\alpha_{j*}}} \ge p_{\mu,1} \\ p_{\mu,1}, \text{ if} \left(\dfrac{1}{(\alpha_{j*} - 1) u_\varepsilon \lambda_{j*}} \displaystyle\sum_{r=1}^{\mathrm{N}_{j*}} u_{t,r} \right)^{\frac{1}{\alpha_{j*}}} < p_{\mu,1} \\ P_{Maxj*}, \text{ if} \left(\dfrac{1}{(\alpha_{j*} - 1) u_\varepsilon \lambda_{j*}} \displaystyle\sum_{r=1}^{\mathrm{N}_{j*}} u_{t,r} \right)^{\frac{1}{\alpha_{j*}}} > P_{Maxj*} \end{array} \right\}$$

6. Repeat steps 4 & 5 whenever a task/s is either dynamically introduced or deleted in $\vec{Q}_{s,j*}$.(b)

7. Once the execution of the task $T_{1,j*}$ is complete or terminated, the indices of all tasks in memory queue $\vec{Q}_{s,j*}$ are shifted down by one creating room for another task.[(a), (b)]
8. If any task or tasks in $\vec{Q}_{s,j*}$ are deleted/cancelled, each alive task in $\vec{Q}_{s,j*}$ is shifted to the minimum available slot starting from the first index to preserve task priority. [(a), (b)]
9. If we are to enforce FCFS queuing service policy or we are not allowed to exercise preemption, whenever a task enters the queue of a processing stream it acquires the Smallest Empty Index (SEI), also in step 2, while calculating the virtual cost of introducing the task to each processing stream, the arriving task virtually acquires the SEI. [(c)]
10. Ignore steps 2 & 3 when processors are homogeneous and instead utilize Round Robin dispatching. [(d)]

Steps 2 & 3 summarize the SMBSPP algorithm's default dispatcher (**MMCVITPS**) under the **SCVPPT** service discipline.
Step 5 describes the speed scaling function (**OSTSSF**).

Notes pertaining to algorithm's description:

[(a)] Steps 7 and 8 are maintenance operations that facilitate the long-run functionality of the algorithm.

[(b)] Steps 3, 4, 7 and 8 can be implemented by *adjuster* modules that dynamically make changes and keep track of the memory queue environment of each processing stream.

[(c)] Step 9 may degrade the performance of the algorithm.

[(d)] Step 10 improves the algorithm's computational complexity when processors are homogeneous, but should not be conducted when processors substantially differ in terms of their energy/power consumption or maximum processing rates.

[(e)] In Step 5, if the processor speed can only be set to integer values, set the optimum processing speed to its floor or ceiling, or better yet, alternate between the two.

Furthermore, if the theoretical optimum speed is un achievable, set the processors speed to a speed that is closest to it.

5.4 Analytical Demonstration

5.4.1 A Simple Example Demonstrating the Robustness of the SMBSPP Algorithm: Handles Dynamic Inclusion of Tasks

We analytically present the SMBSPP algorithm's ability to robustly handle the dynamic inclusion of an incoming task into a processing stream by making optimum adjustments to the execution rate of the currently processed task. Let us consider a simple scenario described as follows. Let us assume two tasks have been optimally dispatched by the SMBSPP algorithm to the j^{th} processing stream. A task 2 is introduced into a given (j^{th}) processing stream after a task 1 is already being processed. In general, the currently processed task 1 is no longer being executed at the optimum rate because the inclusion of task 2 augments the aggregate cost function of the j^{th} processing stream, thereby changing the optimization problem. In order to rectify (optimize) task 1's processing rate, we follow the direction of the SMBSPP algorithm by carrying out its step 6. Step 6 of the algorithm explicitly dictates an optimal change in processing rate of the currently processed task whenever one or more tasks are introduced or deleted from the same processing stream. The figure 5.2 illustrates this scenario by demonstrating the robustness of the SMBSPP Algorithm with respect to handling the dynamic Inclusion of task 2 prior to the full completion of task 1.

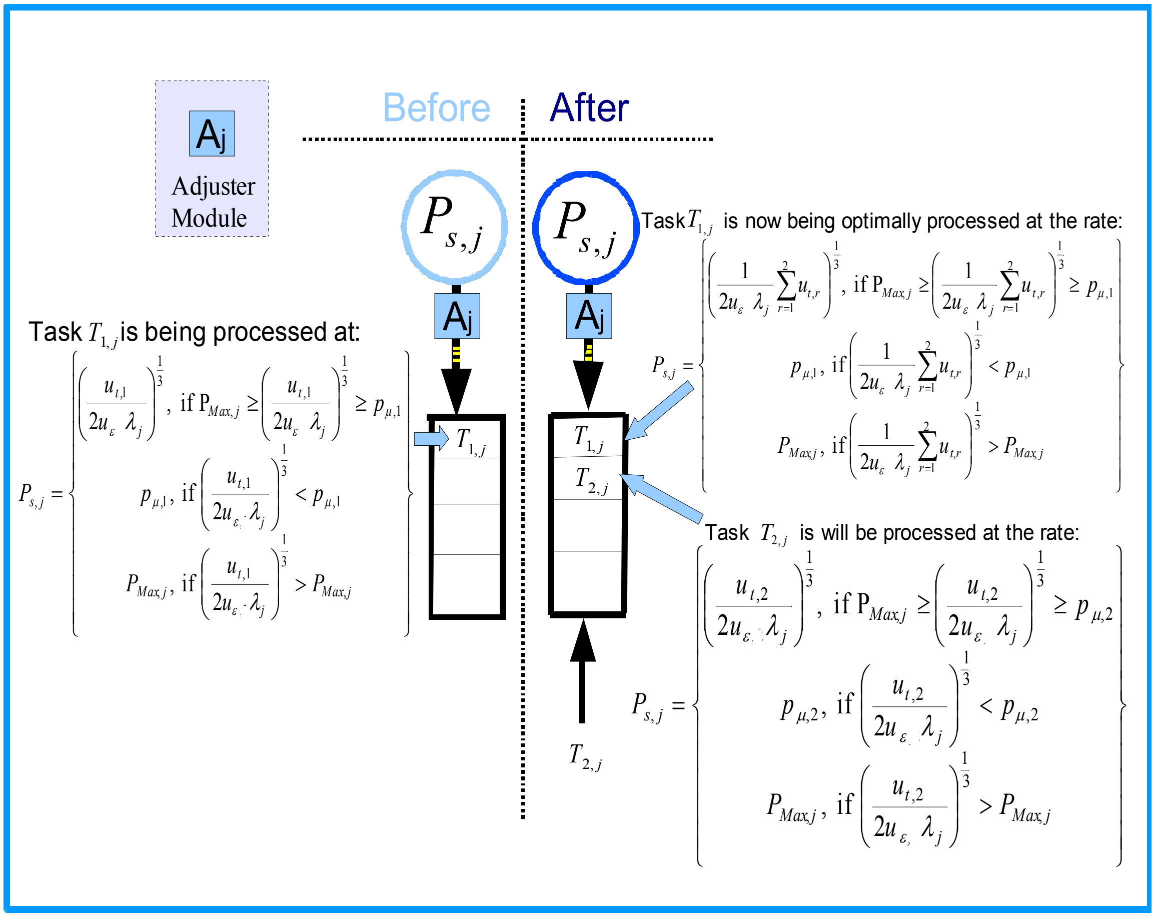
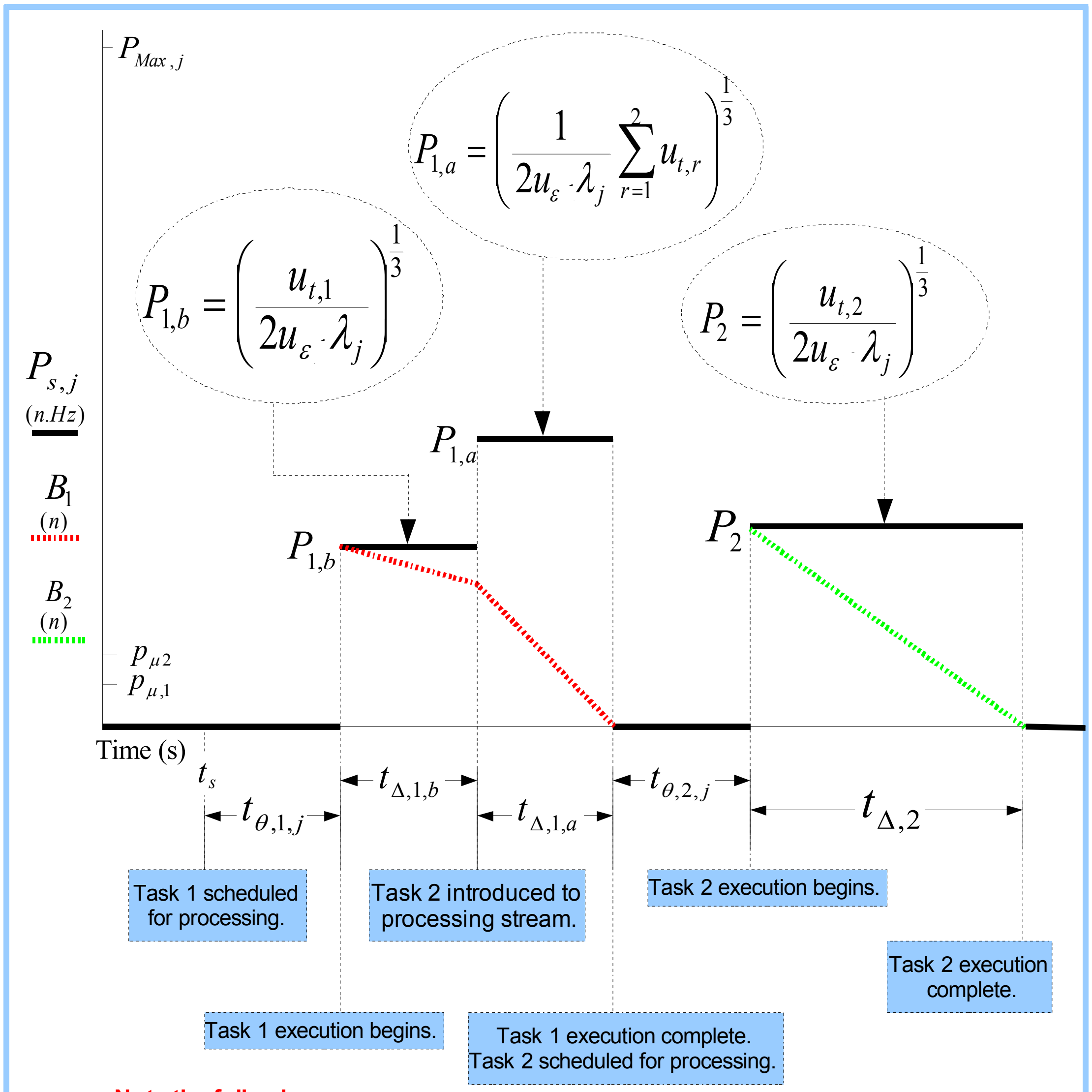

Aj
Adjuster Module
Before
After
$P_{s,j}$
$P_{s,j}$
Aj
Aj
Task $T_{1,j}$ is being processed at:
$P_{s,j} = \begin{cases} \left(\frac{u_{t,1}}{2u_\varepsilon \lambda_j}\right)^{\frac{1}{3}}, \text{ if } P_{Max,j} \geq \left(\frac{u_{t,1}}{2u_\varepsilon \lambda_j}\right)^{\frac{1}{3}} \geq p_{\mu,1} \\ p_{\mu,1}, \text{ if } \left(\frac{u_{t,1}}{2u_\varepsilon \cdot \lambda_j}\right)^{\frac{1}{3}} < p_{\mu,1} \\ P_{Max j}, \text{ if } \left(\frac{u_{t,1}}{2u_\varepsilon \cdot \lambda_j}\right)^{\frac{1}{3}} > P_{Max,j} \end{cases}$
$T_{1,j}$
$T_{1,j}$
$T_{2,j}$
$T_{2,j}$
Task $T_{1,j}$ is now being optimally processed at the rate:
$P_{s,j} = \begin{cases} \left(\frac{1}{2u_\varepsilon \lambda_j}\sum_{r=1}^{2} u_{t,r}\right)^{\frac{1}{3}}, \text{ if } P_{Max j} \geq \left(\frac{1}{2u_\varepsilon \lambda_j}\sum_{r=1}^{2} u_{t,r}\right)^{\frac{1}{3}} \geq p_{\mu,1} \\ p_{\mu,1}, \text{ if } \left(\frac{1}{2u_\varepsilon \lambda_j}\sum_{r=1}^{2} u_{t,r}\right)^{\frac{1}{3}} < p_{\mu,1} \\ P_{Max j}, \text{ if } \left(\frac{1}{2u_\varepsilon \lambda_j}\sum_{r=1}^{2} u_{t,r}\right)^{\frac{1}{3}} > P_{Max j} \end{cases}$
Task $T_{2,j}$ is will be processed at the rate:
$P_{s,j} = \begin{cases} \left(\frac{u_{t,2}}{2u_\varepsilon \lambda_j}\right)^{\frac{1}{3}}, \text{ if } P_{Max j} \geq \left(\frac{u_{t,2}}{2u_\varepsilon \lambda_j}\right)^{\frac{1}{3}} \geq p_{\mu,2} \\ p_{\mu,2}, \text{ if } \left(\frac{u_{t,2}}{2u_\varepsilon \lambda_j}\right)^{\frac{1}{3}} < p_{\mu,2} \\ P_{Max j}, \text{ if } \left(\frac{u_{t,2}}{2u_\varepsilon \lambda_j}\right)^{\frac{1}{3}} > P_{Max j} \end{cases}$

Fig. 5.2: Example demonstrating how SMBSPP robustly handles dynamic inclusion of tasks

Fig. 5.3 is a time analysis of the $\vec{P}_{s,j}$ processor's activity as it executes each of the two tasks. The analysis begins at time t_s as shown. The tasks are executed at constant optimum processing rates as shown in Fig 5.3. An assumption we are making is each of the tasks are processed at optimum processing rates that are not constrained i.e. they are not being processed at the maximum or minimum processing rates. Note that the presented analysis changes if a third task is included into the j^{th} processing stream before either task 1 or task 2 has been fully processed. From an engineering design perspective, *Adjuster* modules comprised of ad hoc digital circuitry may be used to dynamically keep track of the memory queue environment (sequencing, inclusion and deletion of tasks) of each processing stream and to compute as well as to update the optimum execution rates of tasks accordingly.


$P_{Max,j}$
$P_{1,a} = \left(\frac{1}{2u_\varepsilon \cdot \lambda_j} \sum_{r=1}^{2} u_{t,r} \right)^{\frac{1}{3}}$
$P_{1,b} = \left(\frac{u_{t,1}}{2u_\varepsilon \cdot \lambda_j} \right)^{\frac{1}{3}}$
$P_2 = \left(\frac{u_{t,2}}{2u_\varepsilon \cdot \lambda_j} \right)^{\frac{1}{3}}$
$P_{s,j}$
$(n.Hz)$
B_1
(n)
B_2
(n)
$P_{1,a}$
$P_{1,b}$
P_2
$p_{\mu 2}$
$p_{\mu,1}$
Time (s)
t_s
$t_{\theta,1,j}$
$t_{\Delta,1,b}$
$t_{\Delta,1,a}$
$t_{\theta,2,j}$
$t_{\Delta,2}$
Task 1 scheduled for processing.
Task 2 introduced to processing stream.
Task 2 execution begins.
Task 2 execution complete.
Task 1 execution begins.
Task 1 execution complete. Task 2 scheduled for processing.

Note the following:

- For simplicity, we are assuming each task is executed at optimum processing rates that are unconstrained.
- $t_{\Delta,1,a} = \frac{B_1\big|^{t_0} - t_{\Delta,1,b}.P_{1,b}}{P_{1,a}}$
- $t_{\Delta,2} = \frac{B_2\big|^{t_0}}{P_2}$
- Drawing not to scale: usually $\{t_{\theta,1,j}, t_{\theta,2,j}\} << \{t_{\Delta,2}, (t_{\Delta,1,b} + t_{\Delta,1,a})\}$

The subscripts “b” and “a” denote “before” and “after” respectively.

Fig. 5.3: Time analysis of the $\vec{P}_{s,j}$ processor as it executes each of the two tasks in the example.

In this example, we assumed $\alpha_j = 3$ and FCFS service discipline for simplicity.

5.5 Simulations

5.5.1. Performance Metrics

Table I provides a list (with abbreviations and standard units) of some performance metrics.

Table 5.1: Performance metrics

METRIC	DEFINITION	UNITS
TET	Total execution time of executing N tasks	ms
TET/N	Average execution time of executing N tasks	ms/task
TCRTEC	Total cost of response time and energy consumption for executing N tasks	CDN$
TCRTEC/N	Average cost of response time and energy consumption for executing N tasks	CDN$/task
ST	System time of executing N tasks: amount of time that at least one processor is active	ms
TSSC	Total cost of system time and energy consumption for executing N tasks	CDN$
TSSC/N	Average cost of system time and energy consumption for executing N tasks	CDN$/task

In Table 5.1, the metrics in bold are used to evaluate the algorithm.

5.5.2 Simulation I: Sensitivity of SMBSPP Algorithm To Inter-arrival Periods

The preliminary simulation assumptions are as follows:

- We have an *N* number of homogenous tasks each with a computation volume of 100 base instructions.
- We have three processors with the following processor power inefficiency coefficients:

$\lambda_1 = 1.08J.S^{\alpha_1 - 1}.n^{-\alpha_1}$, $\lambda_2 = 1.0J.S^{\alpha_2 - 1}.n^{-\alpha_2}$ and $\lambda_3 = 0.92J.S^{\alpha_3 - 1}.n^{-\alpha_3}$

- These three processor have the following corresponding power constants: $\alpha_1 = \alpha_2 = \alpha_3 = 1.8$
- In this simulation, the computation volumes, loading times and unit price of response times for all tasks are homogenous so as to eliminate the effect of service disciplines, i.e. FCFS, SRPT and SCVPPT all behave in the same way.
- The unit price of energy is $u_\varepsilon = 3.\overline{4}x10^{-8}$ *\$/Joule* and the unit price of response time is $u_t = 2.847\overline{2}x10^{-3}$ *\$/Second* (see Chapter 2, section ? for details).
- The task loading time ($t_{\theta,k,j}$) is fixed to $3.\overline{4}x10^{-9}$ seconds for all tasks.
- A tasks base instruction is assumed to be comprised of 10,000 clock cycles.
- For each simulation iteration, we utilize the TET/N, TCRTEC/N and TSSC/N performance metrics to evaluate the effect of deterministic and stochastic arrival periods.
- All this was repeated for growing values of *N* (simulation iterations).
- Results were confirmed using discrete-time based simulations written in Java.

Following these assumptions, the figures below summarize the simulation results.

Table 5.2: Interpretation of inter-arrival periods

INTER-ARIVAL PERIOD	INTERPRETATION
μ = 0ms	Extreme (batch arrivals)
μ = 26.1ms	Heavy
μ = 50ms	Almost ideal
μ ≥ 156.4ms	Minimal (no traffic)

Fig. 5.4 exhibits how the SMBSPP algorithm utilizes dynamic speed-scaling to adapt to various traffic conditions. The reason why the average execution time of a task falls under high traffic in comparison to low traffic conditions is because as a processing stream gets clogged up, the algorithm's speed scaling function increases therefore executing the tasks at a high rate.

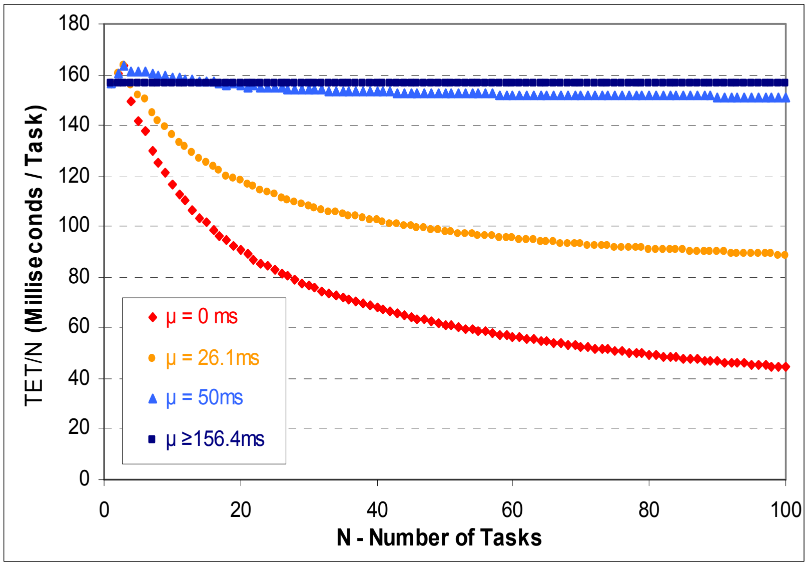


Fig. 5.4: Average Execution Time for N Homogeneous Tasks: Showing Effect of Deterministic Arrival Periods (μ).

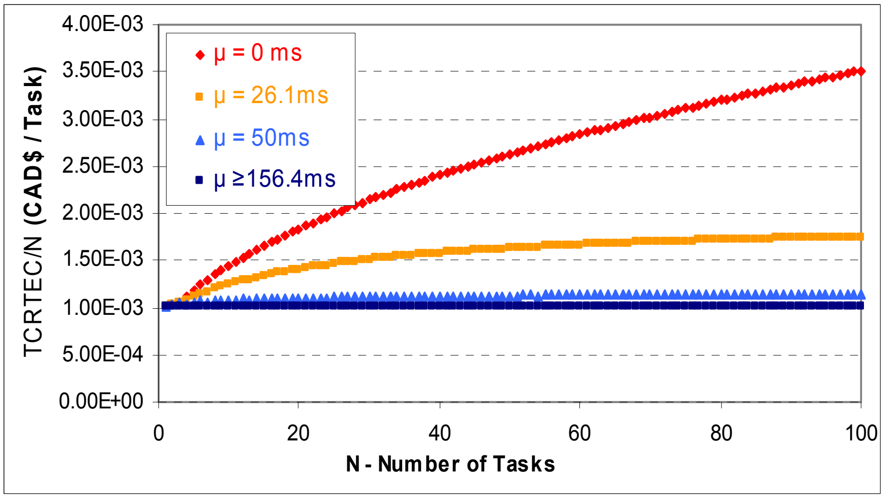


Fig. 5.5: Average Cost of Response Time & Energy Consumption for N Homogeneous Tasks: Showing Effect of Deterministic Arrival Periods (μ).

In Fig. 5.6, the TSSC/N performance metric is a convenient metric in the sense that it is actually the amount in dollars per task that it costs to lease out computation services. The reason why the TSSC/N curve falls way below the TCRTEC/N metric is due to multiple processors working in parallel; where the TSSC/N metric charges the global timescale as can be experienced by a user while TCRTEC/N factors response times of each task leading to multiple aggregation of delays. The fact that the algorithm has a fairly constant TSSC/N curve under heavy stochastic traffic conditions reveals its robustness.

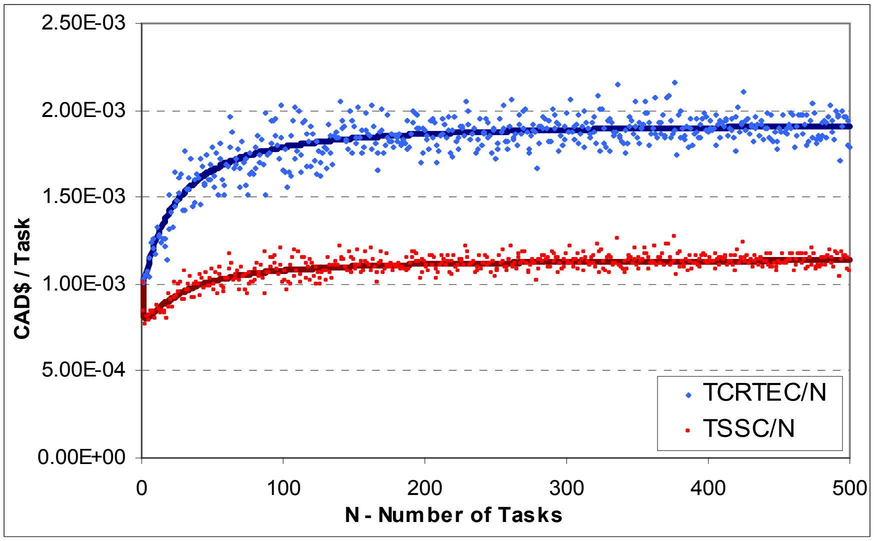


Fig. 5.6: Average Cost of Response Time & Energy Consumption Versus Average Cost of System Time & Energy Consumption for N Homogeneous Tasks under Exponentially Distributed Arrival Periods with a Mean of 1/μ (μ =26ms: heavy traffic) (The results for deterministic arrival periods is interpolated by continuous curves).

5.5.3 Simulation II: Comparing SMBSPP Algorithm's Dispatcher (MMCVITPS) Versus Round Robin Dispatcher under FCFS, SRPT and SCVPPT Service Disciplines

The preliminary simulation assumptions are as follows:

- We have an *N* number of heterogeneous tasks whose computation volumes is Gaussian distributed with a mean of 100 base instructions and a standard deviation of 20% mean.
- We have three processors with the following processor inefficiency coefficients:

$\lambda_1 = 1.08 J.S^{\alpha_1 - 1}.n^{-\alpha_1}$, $\lambda_2 = 1.0 J.S^{\alpha_2 - 1}.n^{-\alpha_2}$ and $\lambda_3 = 0.92 J.S^{\alpha_3 - 1}.n^{-\alpha_3}$

These three processor have the following corresponding power constants: $\alpha_1 = 1.944$, $\alpha_2 = 1.8$ and $\alpha_3 = 1.656$. According to [6], power constants equal to around 1.8 is a good approximation for CMOS based processors. The power function parameters were conservatively chosen to differ from the mean by at most 8%. Presumably, this 8% deviation can be attributed to the manufacturing error of fabricating homogeneous processors, failing to achieve equal temperature environments for all processors or the **intentional** fabrication of heterogeneous processors due to design budget constraints.

- The unit price of energy is $u_{\varepsilon} = 3.\overline{4}x10^{-8}$ *\$/Joule* and the unit price of response time is Gaussian distributed with a mean of $u_t = 2.84\overline{72}x10^{-3}$ *\$/Second* and a standard deviation of 25 % of the mean.
- For each simulation iteration, the SMBSPP Algorithm runs using its default Dispatcher (MMCVITPS) and independently runs using the Round Robin Dispatcher using the **same input data** for various service disciplines.
- All this is repeated for growing values of *N* (simulation iterations).
- We assume heavy traffic conditions with exponentially distributed arrival periods.

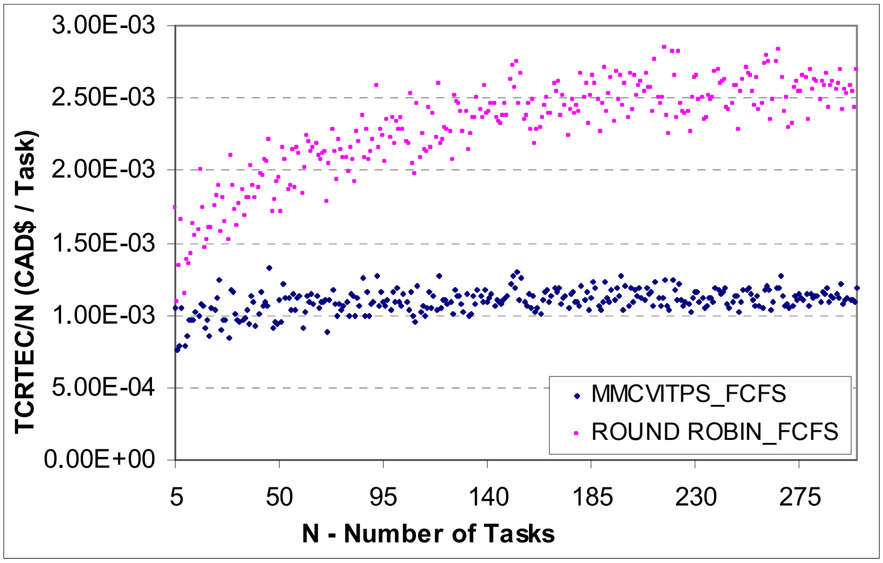


Fig. 5.7: MMCVITPS Versus Round Robin for N Heterogeneous Tasks under Exponentially Distributed Arrival Periods (heavy traffic) with Heterogeneous Unit Prices of Response Time under FCFS.

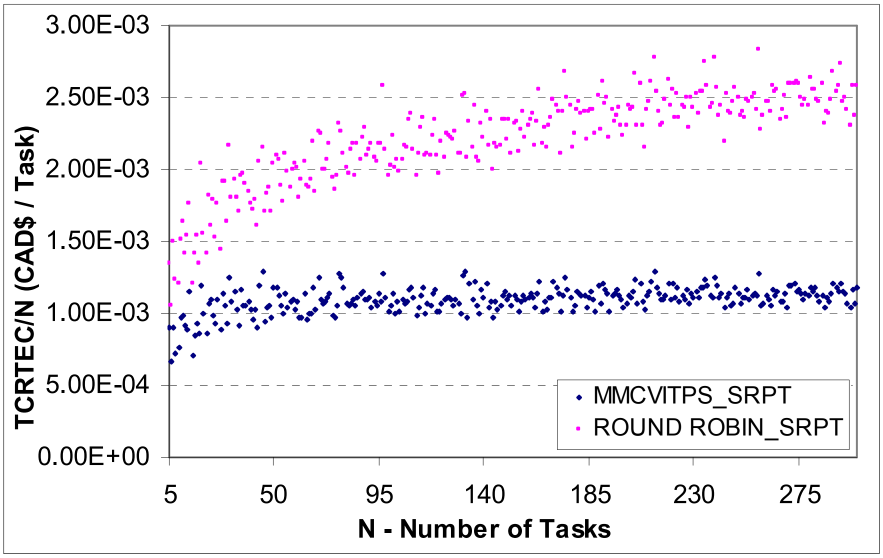


Fig. 5.8: MMCVITPS Versus Round Robin for N Heterogeneous Tasks under Exponentially Distributed Arrival Periods (heavy traffic) with Heterogeneous Unit Prices of Response Time under SRPT Service Discipline.

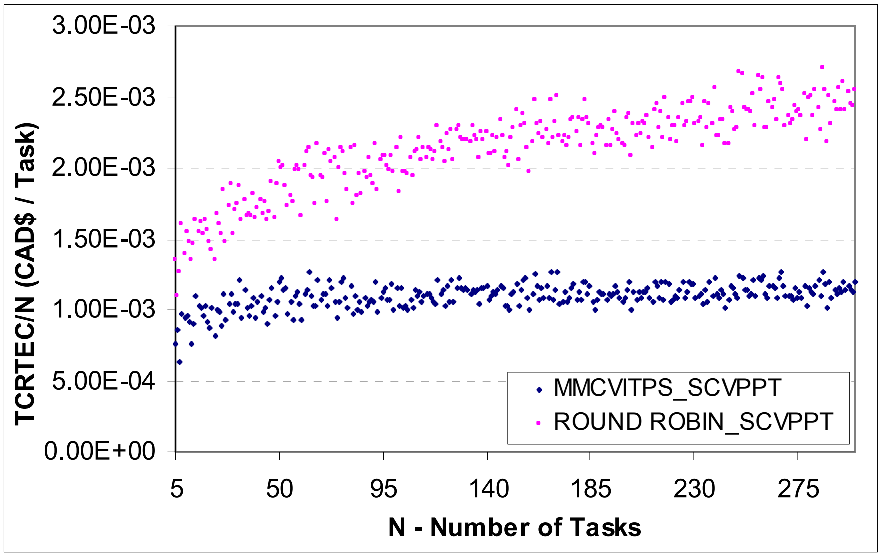


Fig. 5.9: MMCVITPS Versus Round Robin for N Heterogeneous Tasks under Exponentially Distributed Arrival Periods (heavy traffic) with Heterogeneous Unit Prices of Response Time under SCVPPT Service Discipline.

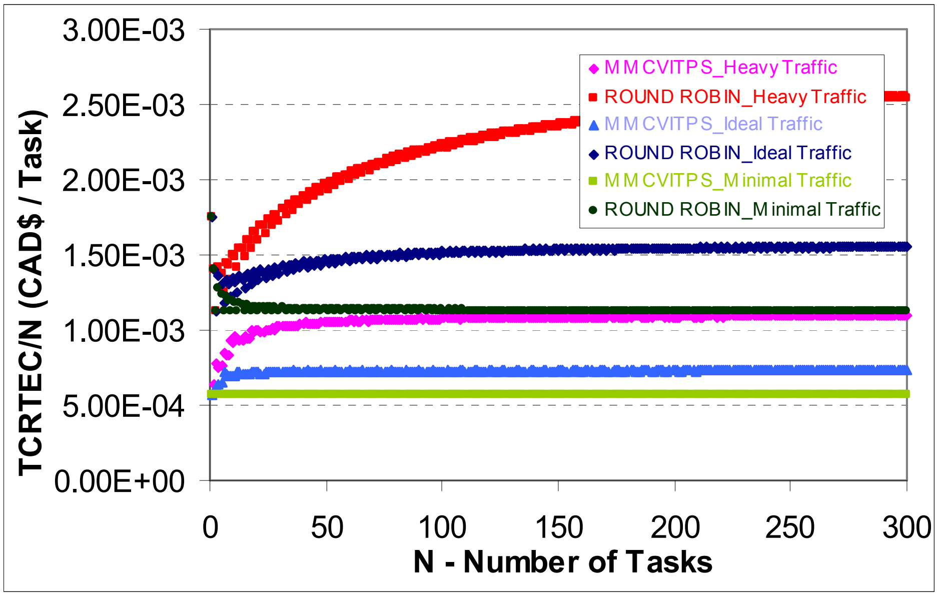


Fig. 5.10: MMCVITPS Versus Round Robin for N Homogeneous Tasks under Three Main Deterministic Arrival Periods with Homogeneous Unit Prices of Response Time. (The three service disciplines are equivalent and have no effect in this scenario).

In Figs. 5.7-5.9 we show that the algorithms dispatcher (MMCVITPS) out performs the Round Robin dispatcher under the FCFS, SRPT and SCVPPT service disciplines under heavy stochastic traffic conditions (with heterogeneous computation volumes of tasks and heterogeneous unit prices of response time).

Fig. 5.10 shows that the MMCVITPS dispatcher outperforms the Round Robin dispatcher under three main deterministic arrival periods that correspond to very heavy, ideal and minimal traffic conditions. If we had further assumed that heterogeneity of the processors was more substantial, the MMCVITPS dispatcher would drastically outperform the Round Robin dispatcher.

5.5.4 Simulation III: Evaluating SMBSPP Algorithm's Dispatcher (MMCVITPS) under FCFS, SRPT and SCVPPT Service Disciplines.

Using the assumptions of Simulation II, we compare the MMCVITPS dispatcher under the three service disciplines.

Fig. 5.11 shows that the SCVPPT service discipline minimizes TCRTEC making it the most ideal for the SMBSPP algorithm with its default dispatcher. We recommend that the SCVPPT service discipline be implemented in any online speed-scaling algorithm that aims to minimize TCRTEC and considers tasks with heterogeneous unit prices of response time.

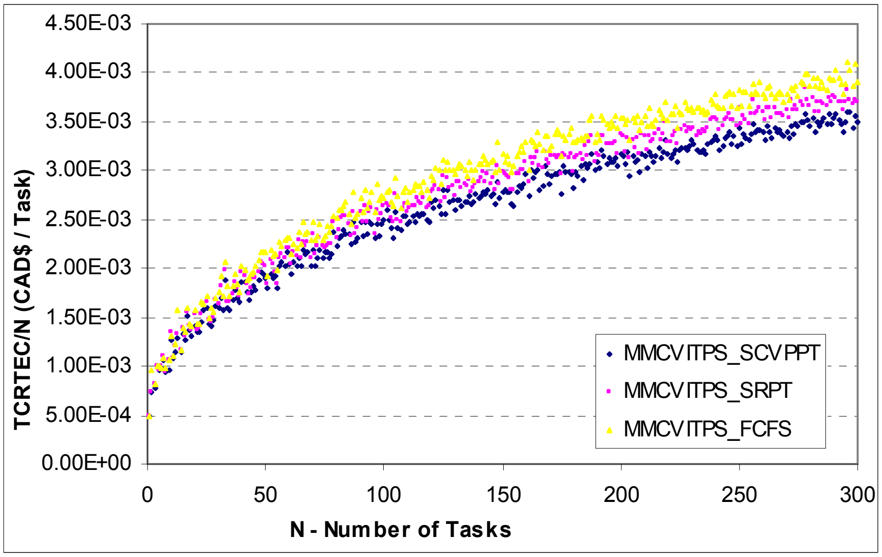


Fig. 5.11: MMCVITPS Dispatcher Performance under SCVPPT, SRPT and FCFS Service Disciplines for N Heterogeneous Tasks that have Exponentially Distributed Arrival Periods with a Mean of 1/μ (almost extreme traffic of μ = 2ms) with Heterogeneous Unit Prices of Response Time (Gaussian distributed).

5.6 Comparing the SMBSPP Algorithm's Speed-Scaling Function (OSTSSF) to a Competitive Speed Scaling Function Found in Current Literature

5.6.1 Analytically Comparing OSTSSF to a Competitive Speed Scaling Function in Current Literature

In this section, we analytically compare the *(*OSTSSF*)* to a comparable and competitive speed scaling function found in current literature ($\tilde{p}(\beta n)^{-1}$). In the next section we validate this comparison via simulations and complete the analysis. We also offer a recommendation to rectify the optimality of the $\widetilde{p}(\beta n)^{-1}$ speed scaling function.

Recall that our speed scaling function (OSTSSF) of the j^{th} processor is:

$$P_{s,j} = \left\{ \begin{array}{c} \left(\frac{1}{(\alpha_j - 1) u_\varepsilon \lambda_j} \sum_{r=1}^{\mathrm{N_j}} u_{t,r} \right)^{\frac{1}{\alpha_j}}, \text{ if } \mathrm{P}_{Max,j} \geq \left(\frac{1}{(\alpha_j - 1) u_\varepsilon \lambda_j} \sum_{r=1}^{\mathrm{N_j}} u_{t,r} \right)^{\frac{1}{\alpha_j}} \geq p_{\mu,1} \\ p_{\mu,1}, \text{ if} \left(\frac{1}{(\alpha_j - 1) u_\varepsilon \lambda_j} \sum_{r=1}^{\mathrm{N_j}} u_{t,r} \right)^{\frac{1}{\alpha_j}} < p_{\mu,1} \\ P_{Max,j}, \text{ if} \left(\frac{1}{(\alpha_j - 1) u_\varepsilon \lambda_j} \sum_{r=1}^{\mathrm{N_j}} u_{t,r} \right)^{\frac{1}{\alpha_j}} > P_{Max,j} \end{array} \right\}$$

and if we assume non-constrained processing rates, our speed scaling function (OSTSSF) reduces to: $P_{s,j} = \left(\frac{1}{(\alpha_j - 1) u_\varepsilon \lambda_j} \sum_{r=1}^{\mathrm{N_j}} u_{t,r} \right)^{\frac{1}{\alpha}}$

If we further assume a homogeneous unit price of response time for all tasks ($u_{t,k} = u_t \big| \forall_{k \in \{1,2 \ldots N_j\}}$), OSTSSF reduces to:

$$P_{s,j} = \left(\frac{\mathrm{N_j} u_t}{(\alpha_j - 1) u_\varepsilon \lambda_j} \right)^{\frac{1}{\alpha_j}} \tag{5.5}$$

Since we are only dealing with a single processor, we drop the j^{th} index in all relevant parameters of (5.5) and we have the following.

$$P_s = \left(\frac{\mathrm{N} u_t}{(\alpha - 1) u_\varepsilon \lambda} \right)^{\frac{1}{\alpha}} \tag{5.6}$$

In current literature, [6] states that the online speed scaling function with minimal competitive ratio under the SRPT service discipline is[33] $\tilde{p}(\beta n)^{-1}$. Where $\tilde{p}(.)^{-1}$ denotes the inverse of $\tilde{p}(s)$.

[6] mentions that $\tilde{p}(s) = s^{\alpha}$ is the power function i.e., the power needed to run at processing speed (s) in a system with a single processor and $\alpha \in (1,3]$ holds for most computer systems.

In the above-mentioned statement, *n* (not to be confused for the unit symbol of a base instruction) is the occupancy of jobs. Under our notation, the occupancy is $\mathrm{N_j}$ in (5.5) and N in (5.6)). Let us generalize the result provided by [6] to include the energy inefficiency coefficient. We have $\tilde{p}(s) = \lambda s^{\alpha}$. In [6], the $\tilde{p}(\beta n)^{-1}$ speed scaling function considers a homogenous unit price of response time that is equal for all tasks. We translate this speed scaling function under our notation and deduce:

$$\tilde{p}(\beta n)^{-1} = \left(\beta \frac{N}{\lambda}\right)^{\frac{1}{\alpha}} = \left(\frac{\mathrm{N}u_t}{u_\varepsilon \lambda}\right)^{\frac{1}{\alpha}} \tag{5.7}$$

Through inspection[34], $\beta = u_t / u_\varepsilon$ where u_t and u_ε are the constant (and homogenous) unit prices of response time and energy, respectively. We assert that these two speed scaling functions; (5.6) and (5.7), differ by a *Constant Correction Factor* (*CCF*) of

$CCF = \left(\frac{1}{(\alpha - 1)}\right)^{\frac{1}{\alpha}}$ and are equivalent when $\alpha = 2$.

[33] The actual notation used in [6] is $p(\beta n)^{-1}$ instead of $\tilde{p}(\beta n)^{-1}$, but we do not want to confuse the reader since p looks similar to a task's theoretical processing rate under our notation.

[34] In [63], β was defined to be the relative cost of delay.

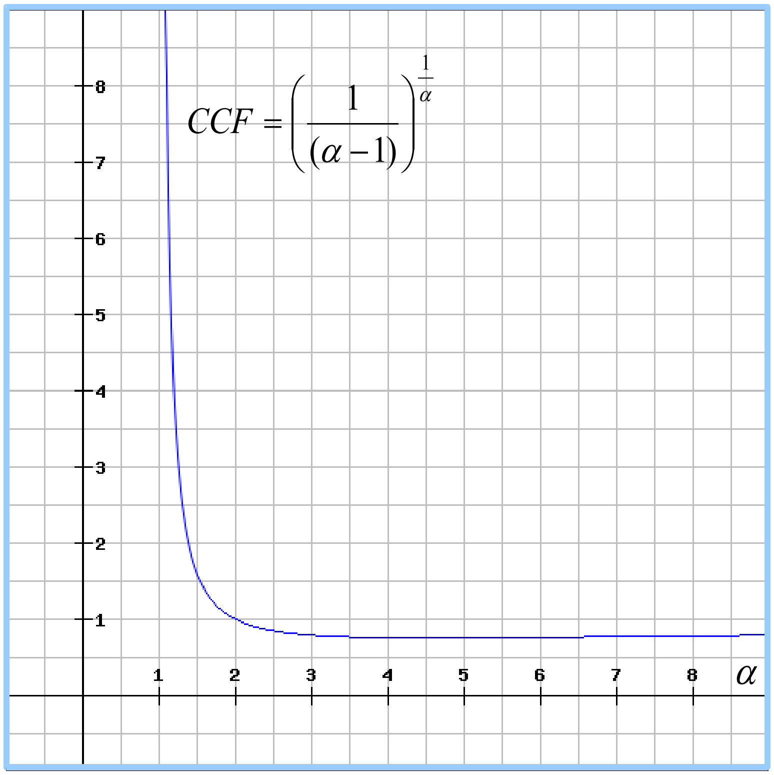


Fig. 5.12: Constant Correction Factor between $\tilde{p}(\beta n)^{-1}$ and OSTSSF

In Fig. 5.12, we see that as we closely approach $\alpha=1$ from the right, the disparity between $\tilde{p}(\beta n)^{-1}$ and OSTSSF grows enormously.

5.6.2 Simulation IV: Comparing SMBSPP Algorithm's Speed-Scaling Function (OSTSSF) to $\tilde{p}(\beta n)^{-1}$ under the SRPT Service Discipline.

We now compare the performance of OSTSSF versus $\tilde{p}(\beta n)^{-1}$ via simulation. Since we are dealing with a single processor, we simulate as a function of occupancy (N) which coincides with the number of arrived tasks as we are assuming no inter-arrival periods between tasks, where they arrive as a batch. The preliminary simulation assumptions are as follows:

- We have an occupancy of N number of homogenous tasks each with a computation volume of 100 base instructions.
- We have a single processor with a power inefficiency coefficient of $\lambda = 1.0 J.S^{\alpha-1}.n^{-\alpha}$ and a corresponding power constant of α .

- In this simulation, the computation volumes of all tasks are homogenous so as to eliminate the effect of service disciplines, i.e. FCFS and SRPT coincide.
- The unit price of energy is $u_\varepsilon = 3.\overline{4}x10^{-8}$ *\$/Joule* and the unit price of response time is $u_t = 2.847\overline{2}x10^{-3}$ *\$/Second* for all tasks in order to conduct a fair comparison because $\tilde{p}(\beta n)^{-1}$ considers homogenous unit prices of response time and energy consumption.
- For each simulation iteration, we utilize the TCRTEC/N performance metric to evaluate both the speed scaling functions.
- All this was repeated for growing values of (occupancy) N (simulation iterations) and different values of $\alpha \in (1,3]$).

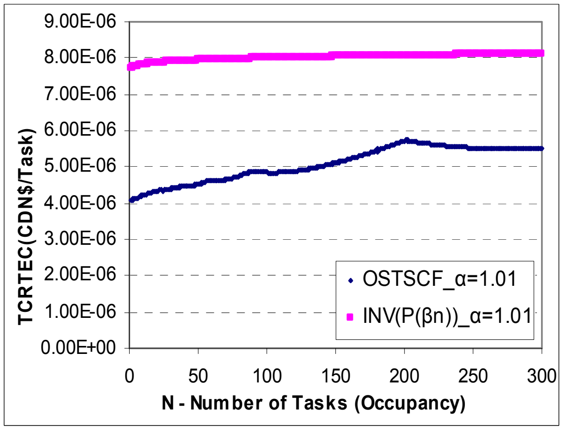


Fig. 5.13: OSTSSF versus $\tilde{p}(\beta n)^{-1}$ for $\alpha = 1.01$

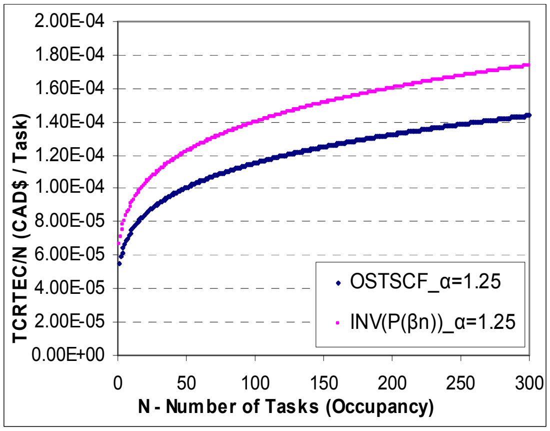


Fig. 5.14: OSTSSF versus $\tilde{p}(\beta n)^{-1}$ for $\alpha = 1.25$

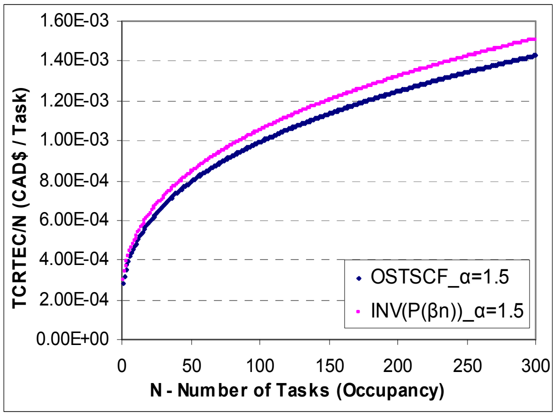


Fig. 5.15: OSTSSF versus $\tilde{p}(\beta n)^{-1}$ for $\alpha = 1.5$

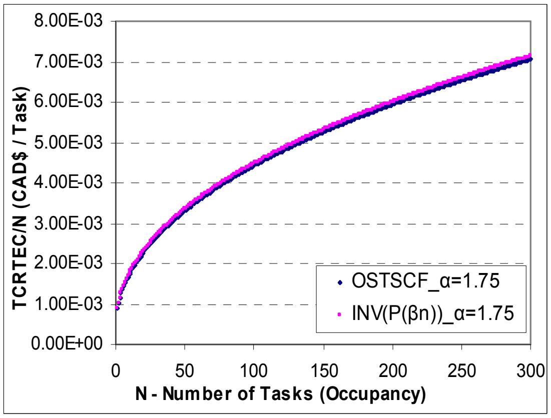


Fig. 5.16: OSTSSF versus $\tilde{p}(\beta n)^{-1}$ for $\alpha = 1.75$

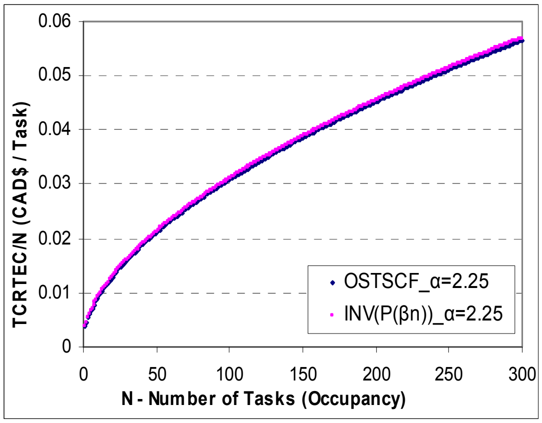


Fig. 5.17: OSTSSF versus $\tilde{p}(\beta n)^{-1}$ for $\alpha = 2.25$

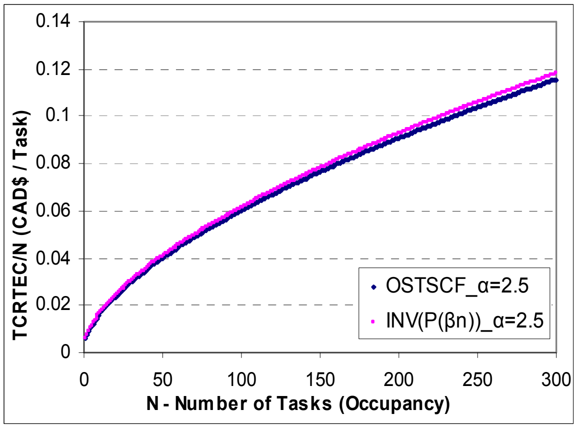


Fig. 5.18: OSTSSF versus $\tilde{p}(\beta n)^{-1}$ for $\alpha = 2.5$

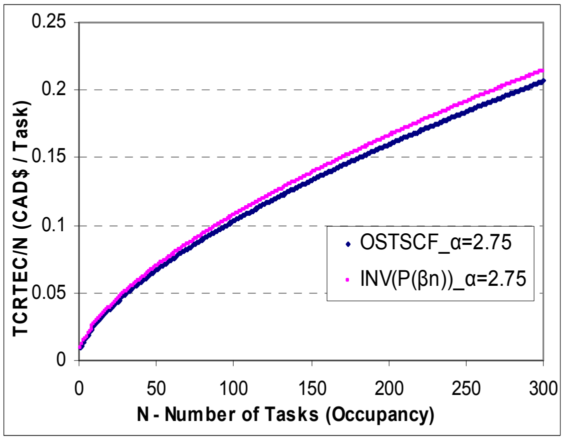


Fig. 5.19: OSTSSF versus $\tilde{p}(\beta n)^{-1}$ for $\alpha = 2.75$

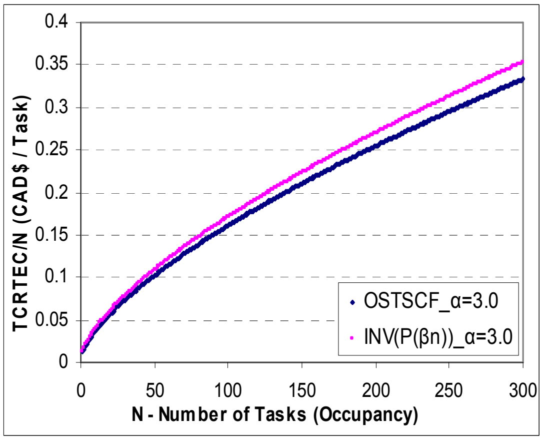


Fig. 5.20: OSTSSF versus $\tilde{p}(\beta n)^{-1}$ for $\alpha = 3.0$

In figures: 5.13-5.20, we see that for $\alpha \neq 2$ the OSTSSF speed scaling function achieves better results than $p(\beta n)^{-1}$ in terms of the TCRTEC/N performance metric and the disparity is more prominent the further away α is from the value of 2. When $\alpha = 2$ both speed scaling functions achieve equivalent performance.

By using a notation that is almost identical to that in current literature, we conclude that when the unit price of response time is homogeneous, the optimum speed scaling function under the SRPT service discipline is actually:

$$\tilde{p}\left(\frac{\beta n}{\alpha-1}\right)^{-1}, \text{ where } \beta = u_t / u_\varepsilon \quad \text{or} \quad \tilde{p}(\beta n)^{-1}, \text{ where } \beta = \frac{u_t}{u_\varepsilon}\left(\frac{1}{(\alpha-1)}\right)^{\frac{1}{\alpha_j}}$$

In the latter, we are correcting or better yet, improving the resolution of β. Without this correction, $\widetilde{p}(\beta n)^{-1}$ is suboptimal on the performance side for $\alpha > 2$, it is suboptimal on the economy side for $\alpha < 2$, and it is optimal when $\alpha = 2$.

Observe that our speed scaling function encompasses this correct result and is valid for the general case where the unit price of response time is heterogeneous in that it could vary per task. Unlike $\widetilde{p}(\beta n)^{-1}$, our speed-scaling function also considers the appropriate hardware and software processing constraints which is more realistic when implemented on actual hardware.

All of the simulation results presented in this chapter are consistently scalable in terms of considering tasks with substantially larger computation volumes, but the simulation run times will take longer and will require a calibration of the inter-arrival periods (and their categorizations i.e. extreme, heavy, ideal and minimal traffic conditions).

5.7 Conclusions

We have synthesized and simulated an online multiprocessor scheduling algorithm (SMBSPP) for optimum parallel computing of portable devices or energy-aware workstations. We focused on single threading where no processor executes more than a single task at any given time until completion unless preemption is dictated by the service discipline e.g. SCVPPT. In the near future, we aim to relax this assumption by considering multithreading. The SMBSPP algorithm provides some insights. It tells us that the optimum processing rate of a task is not a function of the task's computation volume and neither is it a function of the tasks loading time ($t_{\theta,k,j}$). It also tells us once a task is dynamically included into a given memory queue of a processing stream, the optimum processing rate of the currently processed task (stored at the first index of the queue) is likely to change. The processing rate changes because the aggregate cost function of all tasks in the queue has changed and there exists a time dependency among tasks in the processing stream's memory queue due to single-threading. The algorithm explicitly finds a globally optimum solution for each aggregate cost function associated with each processing stream. This globally optimum solution minimizes the total cost of both energy consumption and response time of tasks in each processing stream. The solution explicitly obtains the optimum processing rates of each task in all memory queues. We believe this robustness of the algorithm being able to handle dynamic inclusion of heterogeneous tasks at run-time makes it appealing among hardware architectural planers and software programmers of portable computing devices.

Assuming each processing stream has roughly *n* tasks queued up, the algorithm's default dispatcher (MMCVITPS) has a worse case computational complexity of $O(n^2)$ with heterogeneous response time pricing and $O(n)$ with homogenous response time pricing, and when it uses the Round Robin dispatcher, it has a worse case computational complexity of $O(1)$. In terms of the TCRTEC/N metric, we demonstrated that the algorithms default dispatcher (MMCVITPS) significantly out performs the Round Robin dispatcher under the FCFS, SRPT and SCVPPT service disciplines for various stochastic and deterministic traffic conditions where the degree of processor heterogeneity was mild (power function parameters were conservatively chosen to differ from the mean by at most 8%) yet the MMCVITPS dispatcher drastically outperformed the Round Robin dispatcher with cost savings exceeding 100% on average. In terms of the TCRTEC/N metric, we demonstrated that the algorithms default dispatcher (MMCVITPS) significantly out performs the Round Robin dispatcher under the FCFS, SRPT and SCVPPT service disciplines for various stochastic and deterministic traffic conditions. In fact, we do not recommend the use of the Round Robin dispatcher in systems that utilize heterogeneous processors. If the SMBSPP algorithm is to be implemented in devices with homogeneous processors, the Round Robin dispatcher would be more ideal to use because it would produce results equal to MMCVITPS, but with a lower worse case computational complexity as mentioned previously.

Through simulation, we demonstrated that the SMBSPP algorithm with its default dispatcher (MMCVITPS), service discipline (SCVPPT) and speed-scaling function (OSTSSF) has a fairly constant TSSC/N curve under heavy stochastic traffic conditions; this reveals the algorithm’s robustness. It makes it suitable to be implemented in energy aware work stations or *green* computational devices that utilize parallel processors and want to maintain a fairly stable (constant) operation cost under unpredictable heavy traffic conditions.

The proposed SCVPPT service discipline always matches or outperforms the FCFS and SRPT service disciplines as evaluated by the TCRTEC performance metric. When implemented in the algorithm, the SCVPPT and SRPT service disciplines each have computational complexities of $O(\log N_j)$. where N_j is the occupancy of a given processor. SCVPPT behaves exactly like SRPT when the unit price of response time is fixed and equivalent for all tasks; thereby it minimizes total response time. SCVPPT is sort of a generalized version of SRPT but is flexible. It allows a user to maintain or even improve the priority of a large task by accepting to set/pay a higher unit price of response time or even degrade the priority of a small non-urgent task by setting a sufficiently small unit price of response time. This is a dynamic feature that is absent in both FCFS and SRPT service disciplines. We recommend that the SCVPPT service discipline be implemented in any online speed-scaling algorithm that aims to minimize TCRTEC and considers tasks with heterogeneous unit prices of response time.

Finally, for $\alpha \neq 2$, simulation results show that our speed scaling function (OSTSSF) performs better than the $\left\{\tilde{p}(\beta n)^{-1}, SRPT\right\}$ speed scaling function. We suggest improving this speed scaling function to $\left\{\tilde{p}\left(\frac{\beta n}{\alpha_j - 1}\right)^{-1}, SRPT\right\}$ in order to achieve better results as dictated by the TCRTEC/N performance metric. When the unit price of response time and energy is fixed for all tasks, both of these speed scaling functions have a worse case computational complexity of *O(1)*. Unlike $\left\{\tilde{p}(\beta n)^{-1}, SRPT\right\}$, OSTSSF is valid for the general case where the unit price of response time is heterogeneous in that it could vary per task (this was done to influence the priority of task execution as mentioned previously). Also, OSTSSF unlike $\left\{\tilde{p}(\beta n)^{-1}, SRPT\right\}$, considers the appropriate hardware and software processing constraints, making it more appealing in an application context.

Chapter 6: Using the Laws of Supply and Demand to Extend Battery Life and Improve Load Balancing

6.1 Introduction

So far in this thesis we have studied algorithms that use dynamic speed scaling to reduce the total cost of response time and energy consumption when heterogeneous tasks are executed by heterogeneous processors under the single-threading computing architecture. These algorithms can be used for computing devices that have an unlimited (but not free) supply of energy. A special class of computing devices that are portable and have their own battery source (a.k.a. mobile computing devices) complicate the analysis because the total available energy becomes a budget. Up to this point, the undesired consequence of using dynamic speed scaling in mobile computing devices (under the single threading computing architecture) is that it does not explicitly factor the remaining battery energy level of the mobile computing device. This means that if we had a mobile computing device with low battery level and one with a fully charged battery, the *optimum* processing rate is the same. This is not *robust* because it violates intuition as well as the natural law pertaining to the scarcity of a resource (energy). It can be resolved using the micro-economic laws of demand and supply. In extension, if we had a mobile device with multiple independent energy sources (batteries) that is each associated with a processing stream, the dispatcher should also be affected by the remaining battery energy level of each processing stream.

In this chapter we use the laws of supply and demand to heuristically adjust the unit price of energy of tasks by using the remaining energy percentage. The remaining energy percentage is a dimensionless parameter available in most mobile computing devices. It gives an indication of the amount of remaining energy in the device. We use it as a heuristic controller to ration or preserve the resource of scarce energy in two ways

- It attenuates speed scaling functions (slows down processor speed) as the battery depletes.

- Under independent energy sources associated with each processing steam, it behaves like a load balancer.

To address the first point, we introduce the *Single-Threading Multi Buffer Adjusted Dynamic* (STMBAD) speed scaling algorithm. This online speed-scaling algorithm is used to determine either the optimum or robust processing rate of executing a set of *N* jobs by a single processor of a mobile computing device under the single-threading (multi-buffered) computing architecture. We consider heterogeneous tasks that could differ in computation volume and processing requirements. For simplicity, we assume the unit price of energy and response time is fixed for all tasks and the overhead loading times of tasks prior to their execution are negligible. By using speed-scaling, where the processor's speed is able to dynamically change within hardware and software processing constraints, the algorithm explicitly determines the robust processing rate of executing each task. This robust[35] processing rate was found to be a function of task occupancy, the remaining battery energy percentage, the processor's power function parameters, the unit price of response time and lastly, the unit price of energy. The algorithm allows the user or OS to specify the unit cost of energy and response time for executing all tasks. The algorithm has an operation mode where all tasks' unit cost of energy is also heuristically affected by the device' remaining battery energy percentage in accordance with the micro-economic laws of demand and supply. We synthesize the algorithm by analytically minimizing the total cost of response time and total adjusted cost of energy consumption of tasks. We also consider other conventional performance metrics to evaluate the algorithm. Using numerical simulations, we show that when the remaining battery energy percentage is factored (**EPARBEP**[36] mode), the algorithm: performs slightly slower[37] (mildly more slower when the battery is almost drained out), but consumes far less energy (in many cases more than 30%), can complete significantly more jobs i.e., more than 50% more jobs for both homogenous and heterogeneous tasks (Gaussian distributed computation volumes) and ultimately allows the mobile computing device to last longer on the go.

[35] Robust becomes optimum when the energy percentage is fixed to a value of one.

[36] **EPARBEP** stands for *Energy Price Affected by Remaining Energy Percentage*.

[37] it performs slower than **UEP** mode; UEP stands for *Unadjusted Energy Price*.

To address the second point, we extend the analysis that was conducted in synthesizing the STMBAD algorithm to all the parallel processing algorithms that were previously presented in this thesis. We do this to analytically show that the remaining energy percentage not only affects the speed scaling functions of our algorithms, but affects the dispatchers in such a way that it leads to load balancing when each processor has its independent energy supply (that is scarce).

The analysis in this chapter also sheds light on the difference between optimum and robust speed scaling algorithms (speed scaling functions and coupled dispatchers) in the context of scheduling and processing heterogeneous tasks by heterogeneous processors with the goal of reducing response time and energy consumption.

This chapter is organized as follows. In Section 6.2, we use the remaining energy percentage and the microeconomic laws of demand and supply to synthesize the STMBAD algorithm under the EPARBEB and UEP modes. We presents the STMBAD algorithm in section 6.3. We simulate the STMBAD algorithm under various performance metrics in section 6.4. In section 6.5, we introduce multiple energy sources and extend the definition of the EPARBEP mode under multiple energy sources. In sections 6.6, 6.7 and 6.8, we extend the SBDPP, SBADPA and SMBSPP algorithms to include EPARBEB and UEP modes (respectively). In section 6.9, we use the EPARBEP and UEP mode extensions of our algorithms to describe the effect of the remaining energy percentage on dynamic speed scaling functions as well as on dispatchers. We conclude the chapter in section 6.10.

6.2 Synthesizing the STMBAD Algorithm

6.2.1 Introduction

Let us consider a scenario where we have a mobile device with a single processor and a memory queue (multi-buffer) that stores an arbitrary finite number of *N* tasks (in other words *N* is the potential occupancy of the single processor). We focus on single threading where the processor executes no more than a single task at any given time (until completion). These *N* tasks may be heterogeneous in terms of the minimum software processing rate and computation volume. For simplicity, we assume the overhead loading time of tasks prior to processing is negligible.

The energy and response time dimensional costs of processing these *N* tasks by the single processor is aggregated in a cost function. In this cost function the user or OS defines the unit price of energy and response time for all tasks. The unit price of energy for processing all these *N* tasks is adjusted by the remaining battery energy percentage in accordance with the micro-economic laws of demand and supply. The cost function also factors the hardware/software processing constraints and the power function parameter of the processor. Using dynamic speed scaling, we focus on controlling/optimizing the processing rate of the processor to minimize the total cost of both response time and (adjusted) energy consumption of *N* tasks.

In this section of the chapter, we synthesize the *Threading Multi Buffer Adjusted Dynamic speed Scaling* Algorithm (**STMBAD**) that achieves two objectives.

- It explicitly determines the processing rate of executing each of these N tasks.
- It operates in two modes: **EPARBEP** and **UEP**.

The first objective is achieved through solving an *N* multidimensional convex optimization problem.

The second objective is achieved by utilizing the micro-economic laws of supply and demand to allow or disallows the battery energy percentage (a common parameter found in most modern mobile computing devices) to heuristically influence the price of energy while executing these *N* tasks. **EPARBEP** stands for Energy Price Adjusted by Remaining Battery Energy Percentage and **UEP** stands for Unadjusted Energy Price - i.e., the battery energy percentage does not affect the price of energy.

6.2.2 Mobile Hardware Resources of A Single Processor

Table 6.1 summarizes all the hardware resource/parameters of the mobile computing device with a single processor. We refer to the contents of Table 6.1 in subsequent sections.

Table 6.1: Hardware parameters of a mobile device with a single processor

Parameter	Meaning	SI Unit
E_m	Battery energy level of mobile device	Joules
E_θ	Threshold energy level of mobile device	Joules
$(E_m - E_\theta)$	Usable battery energy of mobile device	Joules
E_{cap}	Maximum energy capacity of level of mobile device (under full charge)	Joules
$\varepsilon_{\%}$	Remaining battery energy percentage of mobile device $\varepsilon_{\%} \in [0,1]$	dimensionless
$\vec{P}_m$	Single processor of mobile device	dimensionless
P_m	Operating processing rate of processor	nHz
P_{Max}	Maximum operating processing rate of processor	nHz
$\vec{Q}_m$	Multi-buffer of processor.	dimensionless

6.2.3 Managing the Remaining Battery Energy Percentage.

In Table 6.1, we defined $\varepsilon_{\%}$ as the remaining battery energy percentage. This parameter is conventionally found in most modern mobile computing devices. An example can be seen at the top right corner of Fig. 6.1.

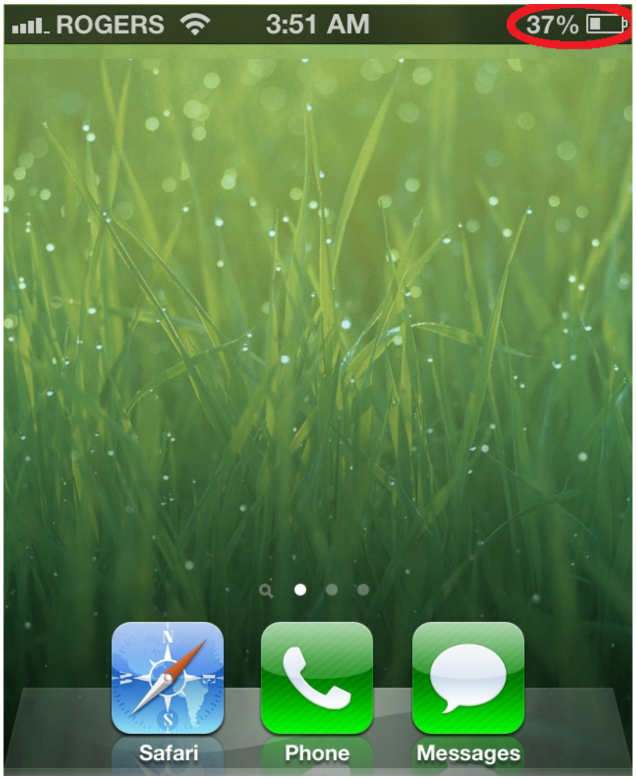


Fig. 6.1:Remaining battery energy percentage of an *iPhone 5* (circled in red)

The fact that $\varepsilon_{\%}$ is visible to the user through a graphical interface suggests that it should be accessible by the OS of the mobile computing device. If the mobile computing device is currently being re-charged (inflow energy meets or exceeds current use) and it is known in advance that the mobile device will not be disrupted from recharging its battery until completion, then during the recharging period we can ignore this value from the OS and prematurely set our $\varepsilon_{\%} = 1$ in our cost function (section 6.26). This is done because energy is temporarily not a scarce resource during foreseen battery-recharge period. Also all of the work presented in this chapter can be extended to non-mobile work stations or computing devices that have a reliable and unlimited power supply by setting $\varepsilon_{\%} = 1$ as well.

6.2.4 Showing how increased supply of a commodity leads to lower price and vise versa using demand and supply curves

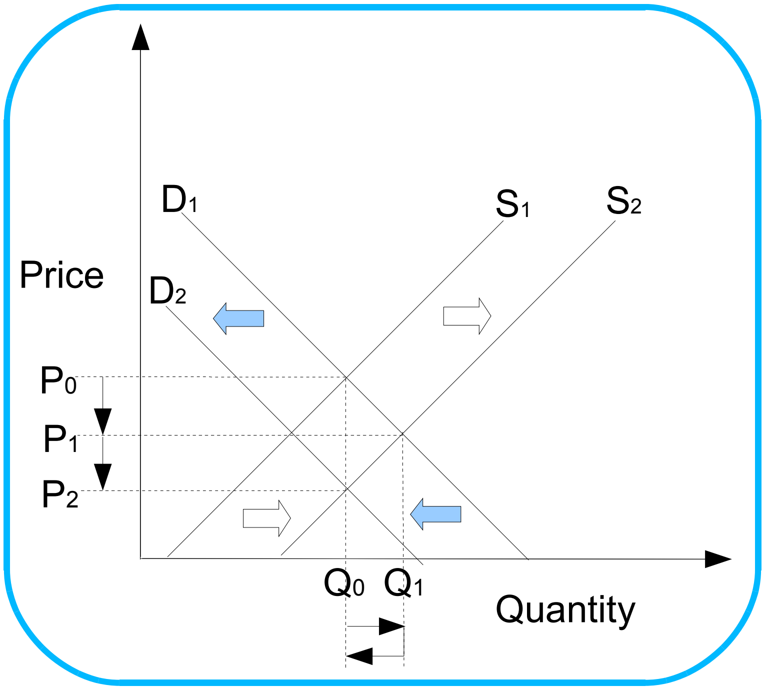


Fig. 6.2: Increased supply of a commodity leads to lower price

Let us assume the commodity of interest is the remaining energy in a battery of a mobile computing device. Consider the Fig. 6.2. Let us start at the equilibrium point where the supply curve 1 and demand curve 1 intersect (Q_0, P_0). Let the commodity's supply increase (battery recharge), this leads to a right shift of the supply curve 1 to supply curve 2. Our new equilibrium point is (Q_1, P_1). We already see a price drop (from P_0 to P_1) that suffices for arguments sake. Furthermore, the price drop is much more significant because the new equilibrium point has more quantity than was originally demanded (task's energy consumption) and we need to get back to our original equilibrium quantity. So the market forces prevail and the demand reduces to make this adjustment by left-shifting the demand curve 1 to demand curve 2. Now we are at operation point (Q_0, P_2). The aggregate price drop is now from (P_0 to P_2), which shows the effect of increased supply. In other words, when the battery of a mobile device is recharging, its price of energy should be decreasing.

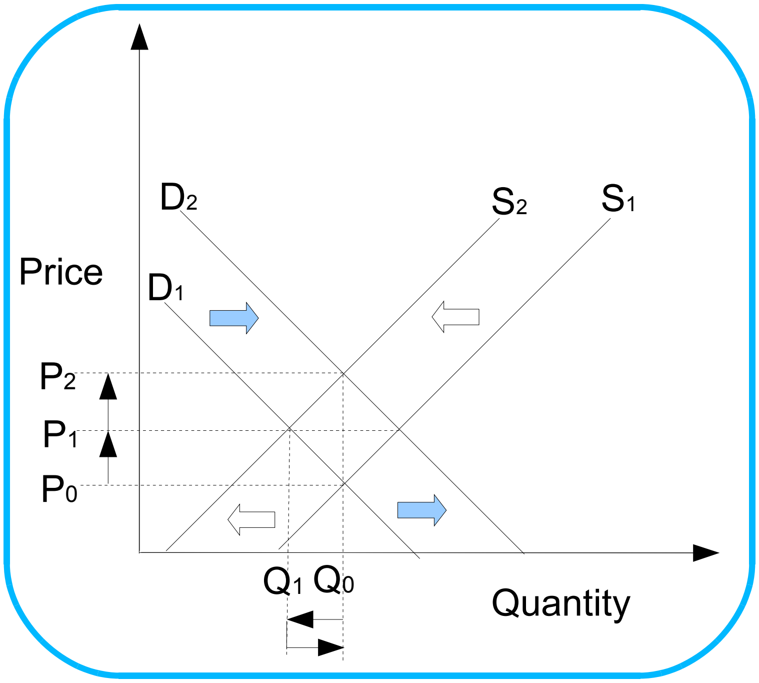


Fig. 6.3: Decreased supply of a commodity leads to higher price

The same argument in reverse is applied to Fig. 6.3. It shows that a decreased supply of a commodity leads to an increased price/value. This implies that when the battery energy of a mobile device is depleting (e.g. under use), its price of energy should be increasing. Supply and Demand are well established topics in micro-economics. Refer to [17, 36] for further elaboration.

6.2.5 Problem Formulation

Assume the mobile computing device has a memory queue buffer, $\vec{Q}_m$ that has the capacity to store a finite arbitrary number of (N) tasks. We are essentially trying to minimize a cost metric. This cost metric is the total cost of response time and total adjusted cost of energy consumption of N tasks where the remaining battery energy percentage heuristically adjusts the unit price of energy of all tasks in accordance with the micro-economic laws of supply and demand. We minimize this cost metric by using dynamic speed scaling, where we explicitly find the robust or optimum processing rates of all tasks in closed form.

6.2.6 Cost Function

Let us assume that the mobile computing device' memory queue buffer currently holds *N* tasks. Let c_s denote the total cost of response time and total adjusted cost of energy consumption of processing these *N* tasks by a single processor.

Using vector notation and dot product operations, we have:

$$C_s = \sum_{k=1}^{\mathrm{N}} \left\{ U_j \bullet \left(\frac{\varepsilon_k}{\varepsilon_{\%}}, \sum_{r=1}^{k} t_r \right) \right\}$$

More explicitly using equations (3.4) and (3.5) from chapter 3 we have the following[38].

$$C_j = \underbrace{\frac{u_\varepsilon}{\varepsilon_{\%}} \overbrace{\lambda (P_1)^{\alpha-1} B_1}^{\text{Task 1 energy (J)}}}_{\text{Task 1 (Adjusted) energy cost (\$)}} + \overbrace{u_t \left(\underbrace{\frac{B_1}{P_1}}_{\text{Task 1 time (S)}} \right)}^{\text{Task 1 time cost (\$)}}$$

$$+ \underbrace{\frac{u_\varepsilon}{\varepsilon_{\%}} \overbrace{\lambda (P_2)^{\alpha-1} B_2}^{\text{Task 2 energy (J)}}}_{\text{Task 2 (Adjusted) energy cost (\$)}} + u_t \overbrace{\left(\underbrace{\left(\frac{B_2}{P_2}\right)}_{\text{Task 1 time (S)}} + \underbrace{\left(\frac{B_1}{P_1}\right)}_{\text{Task 2 time (S)}} \right)}^{\text{Task 2 time cost (\$)}}$$

$$+ \ldots$$

$$+ \frac{u_\varepsilon \lambda (P_{\mathrm{N}})^{\alpha-1} B_{\mathrm{N}}}{\varepsilon_{\%}} + u_t \sum_{r=1}^{\mathrm{N}} \frac{B_r}{P_r}$$

[38] We drop the j^{th} index in those equations because we are dealing with a single processor.

$$= \sum_{k=1}^{N}\left[\frac{u_{\varepsilon}\lambda(P_k)^{\alpha-1}B_k}{\varepsilon_{\%}} + u_t(N+1-k)\frac{B_k}{P_k}\right] \quad (\$) \qquad (6.1)$$

In equation (6.1), $\varepsilon_{\%}$ heuristically adjusts the cost of our energy terms. It exists due to the micro-economic principles of demand and supply; these micro-economic laws confirm natural laws of resources which correlate the scarcity of a commodity with its value (monetary or otherwise). As the battery depletes, $\varepsilon_{\%}$ reduces which in turn inflates the price of our energy terms in our cost function as desired. This was discussed in more detail in section 6.2.4.

6.2.6 Minimized Cost Function

We have an *N* dimensional optimization problem. Using speed scaling, the adjustable parameters are the theoretical processing rates (P_k) of the tasks: $T_k \in T \mid k \in \{1,2...N\}$

Let us optimize C_j .

$$\frac{\partial C_s}{\partial P_k} = \frac{(\alpha-1)u_{\varepsilon}\lambda(P_k)^{\alpha-2}B_k}{\varepsilon_{\%}} - u_t(N+1-k)\frac{B_k}{P_k^{\,2}} = 0 \quad \text{for } k \in \{1,2...N\}.$$

Note that we have made a critical assumption that needs to be justified; we assumed $\varepsilon_{\%}$ does not significantly vary or is more or less a constant function of P_k which is valid under a specific condition as explained next.

Let us explicitly denote the time dependency of $\varepsilon_{\%}$ as $\varepsilon_{\%}(t)$

We have

$\varepsilon_{\%}(t) = \dfrac{E_m(t) - E_{\theta}}{E_{cap}}$. When the processor is executing a task T_k, we have

$$\varepsilon_{\%}(t)=\frac{\varepsilon_{\%}(t_0)E_{cap}-\lambda(P_k)^{\alpha}(t-t_0)-\varepsilon_{SBL}(t-t_0)}{E_{cap}},for\ \ t_k\geq t>t_0$$

$\varepsilon_{SBL}(t-t_0)$ is the battery energy stand-by loss[39] over the time interval $t-t_0$.

$$\left|\frac{\partial\varepsilon_{\%}(t)}{\partial P_k}\right|=\left|\frac{-\alpha\lambda(P_k)^{\alpha-1}(t-t_0)}{E_{cap}}\right|\leq\frac{\left|-\alpha\lambda(P_k)^{\alpha}t_k\right|}{E_{cap}}\leq\frac{\left|-\alpha\varepsilon_k\right|}{E_{cap}}=\frac{\left|3\varepsilon_k\right|}{E_{cap}}=\frac{3\varepsilon_k}{E_{cap}}$$

The assumption is valid as long as the condition: $\varepsilon_k << E_{cap}$ is satisfied i.e. the energy consumption of a single task is insignificant compared to the energy capacity of the battery.

Getting back to optimizing our cost function, we solve (6.1) and get:

$$P_{k,crit}=\left(\frac{\varepsilon_{\%}u_t(N-k+1)}{(\alpha-1)u_{\varepsilon}\lambda}\right)^{\frac{1}{\alpha}}\text{ for } k\in\{1,2...N\}\text{ .and } \alpha\in(1,3]$$

Using a Hessian matrix [24], it can be shown that this set of critical processing rates minimizes C_j.

[39] Initially in this thesis, we assumed the processor incurs a zero stand by energy loss when idle, we suspend this assumption in this particular context because we are trying to analytically model the behavior of a battery under practical use. As an aside, energy of batteries in mobile devices decay with time even during sleep mode and [49] shows that a battery's stand by current drain can be mitigated by a DC-DC converter.

6.2.7 Minimized Constrained Cost Function

$\forall T_k \in T \mid T_k \in \vec{Q}_m$, let us include the processing constraints mentioned earlier in this thesis (Chapter 3, Section 3.3.1)

We enforce $p_{\mu,k} \leq P_m \leq P_{Max}$ where, $p_{\mu,k}$ is the task's minimum recommended execution rate in base instructions per second (n.Hz.). Assuming $\varepsilon_k << E_{cap}$, the (theoretical) constrained (robust) processing rates of the tasks $\{T_1, T_2 ... T_k ... T_N\} \in T$ is:

$$P_k = \left\{ \begin{array}{c} \left(\frac{\varepsilon_{\%} u_t (N-k+1)}{(\alpha-1) u_\varepsilon \lambda} \right)^{\frac{1}{\alpha}}, \text{ if } \mathrm{P}_{Max} \geq \left(\frac{\varepsilon_{\%} u_t (N-k+1)}{(\alpha-1) u_\varepsilon \lambda} \right)^{\frac{1}{\alpha}} \geq p_{\mu,k} \\ p_{\mu,k}, \text{ if} \left(\frac{\varepsilon_{\%} u_t (N-k+1)}{(\alpha-1) u_\varepsilon \lambda} \right)^{\frac{1}{\alpha}} < p_{\mu,k} \\ \mathrm{P}_{Max}, \text{ if} \left(\frac{\varepsilon_{\%} u_t (N-k+1)}{(\alpha-1) u_\varepsilon \lambda} \right)^{\frac{1}{\alpha}} > \mathrm{P}_{Max} \end{array} \right\} \quad \text{for } k \in \{1,2...N\}.$$

We have sufficient information to describe the STMBAD Algorithm.

6.3 The STMBAD Algorithm

1. User or OS specifies u_ε and u_t for all tasks $T_k \in T$.

2. Fix $\varepsilon_{\%} = 1$ when energy is not a scarce resource (**UEP** mode) otherwise acquire $\varepsilon_{\%}$ from OS (**EPARBEP** mode[40]) .

3. Before processing the task stored at the first index (T_1), update *N* (number of 'alive' tasks)

4. The task T_1 is executed by the mobile computing device' processor, $\vec{P}_m$ at the optimum processing rate defined below:

[40] Use **EPARBEP** mode when $\varepsilon_k << E_{cap}$.

$$P_m = \left\{ \begin{array}{c} \left(\dfrac{\varepsilon_{\%} u_t N}{(\alpha - 1) u_\varepsilon \lambda}\right)^{\frac{1}{\alpha}}, \text{ if } \mathrm{P}_{Max} \geq \left(\dfrac{\varepsilon_{\%} u_t N}{(\alpha - 1) u_\varepsilon \lambda}\right)^{\frac{1}{\alpha}} \geq p_{\mu,k} \\ p_{\mu,k}, \text{ if } \left(\dfrac{\varepsilon_{\%} u_t N}{(\alpha - 1) u_\varepsilon \lambda}\right)^{\frac{1}{\alpha}} < p_{\mu,k} \\ P_{Max}, \text{ if } \left(\dfrac{\varepsilon_{\%} u_t N}{(\alpha - 1) u_\varepsilon \lambda}\right)^{\frac{1}{\alpha}} > P_{Max} \end{array} \right\}$$

5. Whenever a task joins or leaves the memory queue buffer, update *N* and repeat step 4.

6. If we are allowed to violate FCFS service policy and permit preemption, rearrange tasks from the lowest index according to smallest remaining computation volume (equivalent to SRPT).

7. Repeat steps: 2-6 until N = 0 (No tasks left), in which case $P_m = 0$.

By default, the STMBAD algorithm operates on a mode where the price of energy is heuristically influenced by the remaining battery energy percentage in accordance with the micro-economic laws of demand and supply; we abbreviate this operation mode as **EPARBEP** (Energy Price Adjusted by Remaining Battery Energy Percentage). The algorithm can also operate on a mode where the remaining battery energy percentage does not influence the price of energy by permanently setting $\varepsilon_{\%} = 1$; we abbreviate this mode as **UEP** (Unadjusted Energy Price).

6.4 Simulating The STMBAD Algorithm

6.4.1 Performance Metrics

Table 6.2 provides a list (with abbreviations and standard units) of some performance metrics. In this table, the metrics in bold are used to evaluate the STMBAD algorithm.

Table 6.2: Performance metrics

METRIC	**DEFINITION**	**UNITS**
TET	Total execution time of executing N tasks	ms
TET/N	**Average execution time of executing N tasks**	**ms/task**
TRT	Total response time of executing N tasks (factors delays and execution time for each ask)	ms
TRT/N	**Average response time of N tasks**	**ms/task**
TEC	Total Energy consumption for executing N tasks	Joules
TEC/N	**Average Energy consumption for executing N tasks**	**Joules/task**
TCRTEC	Total cost of response time and energy consumption for executing N tasks	CDN$
TCRTEC/N	**Average cost of response time and energy consumption for executing N tasks**	**CDN$/task**
TCETEC	Total cost of execution time and energy consumption for executing N tasks	CDN$
TCETEC/N	**Average cost of execution time and energy consumption for executing N tasks**	**CDN$/task**

6.4.2 Simulation I: STMBAD Algorithm's EPARBEP Mode Versus UEP Mode While Processing N Homogenous Tasks

The preliminary simulation assumptions are as follows:

- We have an N number of homogenous tasks each with a computation volume of 500 base instructions.
- The processor's power function parameters are $\alpha = 3$ and $\lambda = 1.0x10^{-9}(J.S^2 / n^3)$.
- The unit price of energy is $u_\varepsilon = 3.\overline{4}x10^{-8}$ \$/Joule and the unit price of response time is $u_t = 2.84\overline{72}x10^{-3}$ \$/Second (see Chapter 3 section 3.2 for details).
- We have a 900 Kilo Joule battery with 5% energy capacity reserved for OS maintenance. ($5\% \le \varepsilon_{\%} \le 1$).

- Prior to an iteration of the simulation, for each different value of N, it is assumed that the battery is fully charged and the simulation iteration terminates when the processing of all N tasks is complete.
- For each simulation iteration, the two modes of the STMBAD algorithm are independently simulated using the same input data.
- Simulation data is rejected when the mobile device runs out of energy before completing all these *N* tasks. This is done to draw an objective comparison between the two modes of the algorithm since a partial execution of N tasks complicates and skews the comparison.
- The service discipline employed is FCFS for practical reasons (clairvoyance) .
- All this was repeated for growing values of N (simulation iterations).
- Simulation results were confirmed using a discrete-time based simulation written in Java.

Following these assumptions, the graphs below summarize the simulation results.

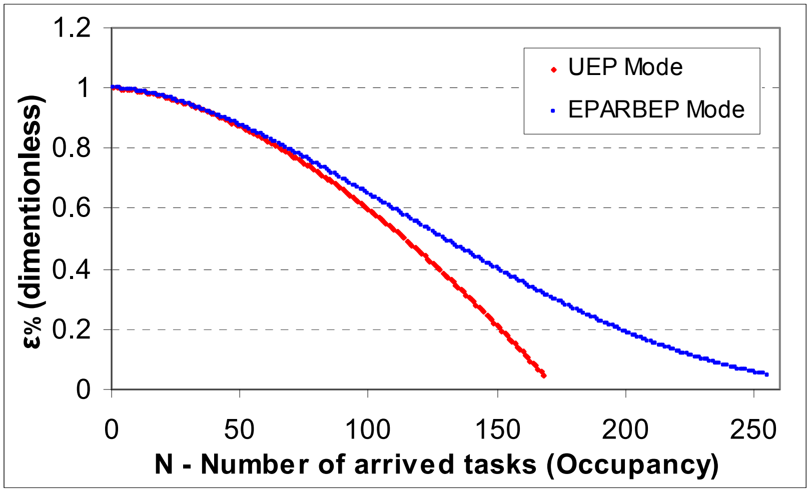


Fig. 6.4: Remaining battery energy percentage $(\varepsilon_{\%})$ after executing N tasks

In Fig. 6.4, when more 50 tasks (or about 70 tasks) are executed, we clearly see that the laws of demand and supply are countering the effect of optimum dynamic speed scaling as a function of occupancy. This is explained as follows. Under both modes, as the occupancy (N) of the processor increases, the dynamic speed scaling function tends to increase as well to reduce energy and response time costs[41]

Also, the battery energy depletes at a high rate with increased N because of more tasks and increased processing rates (that are dictated by the speed scaling function).

The UEP mode operates in the absence or knowledge of scarce energy where it finds the optimum speed scaling function that minimizes response time and unadjusted energy costs. As N increases, the battery depletes, and the EPARBEP mode slows down the processor's speed (attenuates it by a factor of $(\varepsilon_{\%})^{1/\alpha}$ in comparison to the optimum) because energy becomes more scarce, thereby it minimizes response time and adjusted[42] energy cost. Under the EPARBEP mode, this adjusted processing rate (that is attenuated by a factor of $(\varepsilon_{\%})^{1/\alpha}$ in comparison to the optimum processing rate) is defined to be robust.

As illustrated in Fig. 6.4, for a fixed amount of energy, The EPARBEP mode executes significantly more tasks than the UEP mode because the EPARBEP mode has been aware of the scarcity of energy whilst the battery has been depleting, and therefore has made a robust adjustment to the speed scaling function of the processor by reducing it accordingly.

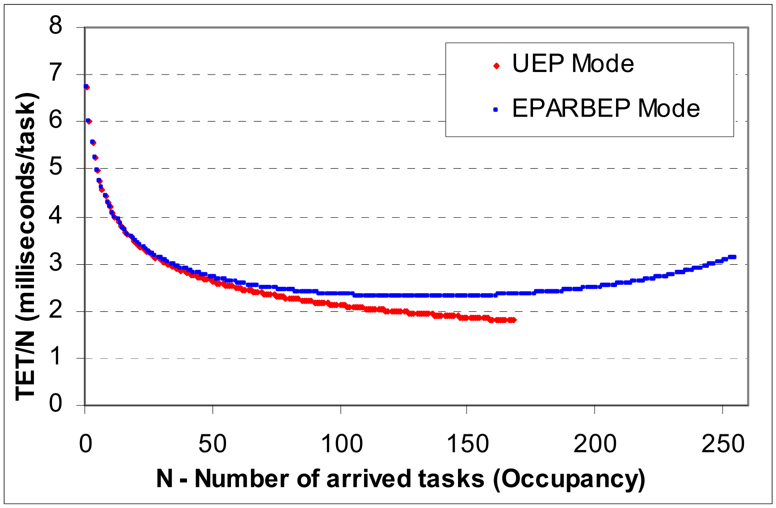


Fig. 6.5: Average execution time of executing N homogeneous tasks

[41] Refer to the speed scaling function of the STMBAD algorithm.

[42] *Adjusted* energy means that the unit price of energy is adjusted by the laws of demand and supply through the *remaining energy percentage* parameter.

In Fig. 6.5, initially as the occupancy of the processor increases, the speed scaling function increases under both modes (when occupancy is less than 168 tasks). This explains why the execution time decreases with increased occupancy. Under the EPARBEP mode, as the battery depletes, the processor's speed scaling function is dominantly countered or reduced by the remaining energy percentage parameter in accordance with the laws of demand and supply, therefore it executes tasks at slower (suboptimal[43] but robust) processing rates that lead to an increase in execution time as can be seen in Fig. 6.5.

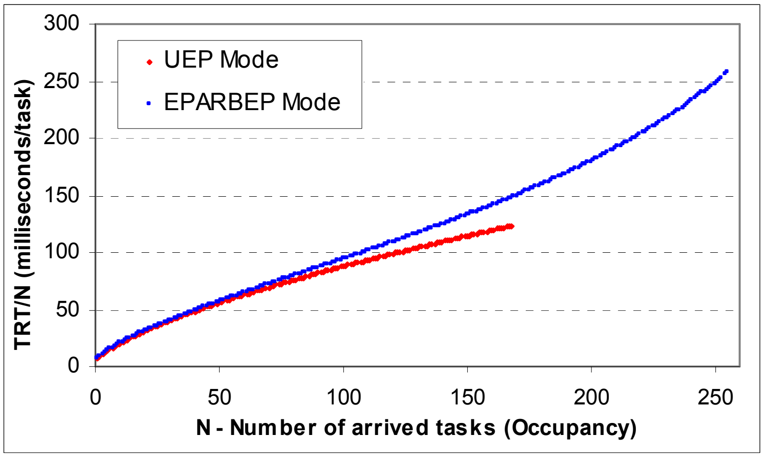


Fig. 6.6: Average response time for N homogeneous tasks

In Fig. 6.6, under both modes the response time increases with occupancy because of the simultaneous service time delays of tasks under the single-threading computing architecture. The reason why the EPABEP mode has a further increase in response time in comparison to the UEP mode is because the speed scaling function is countered or reduced by the remaining energy percentage parameter in accordance with the laws of demand and supply, therefore it executes tasks at slower (suboptimal) processing rates. Since higher processing rates incur a higher energy consumption as dictated by the convexity of power functions, the same argument is used to explain Fig. 6.7.

[43] The EPARBEP mode leads to robust but suboptimal processing rates in terms of the TCRTEC performance metric. If we evaluate the algorithms based on the TRTEC metric where the price of energy was hyperbolically reduced by a factor $\varepsilon_{\%}$, then the EPARBEP mode would lead to *optimum* processing rates. We do not pursue this line of reasoning in order to avoid confusion.

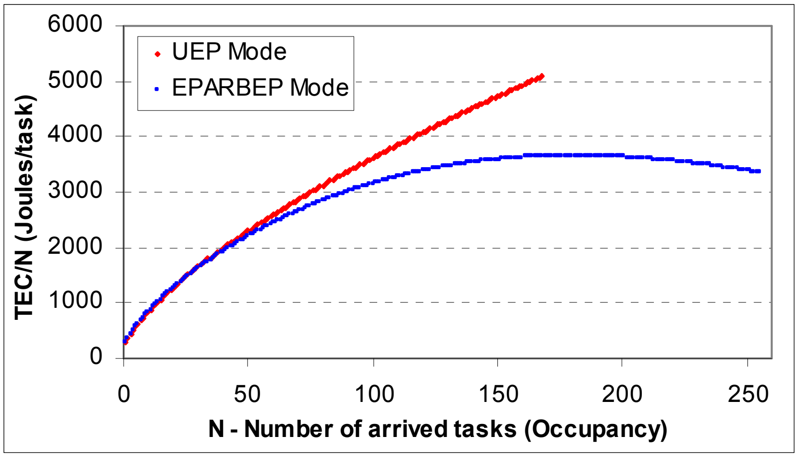


Fig. 6.7: Average energy consumption for executing N homogeneous tasks

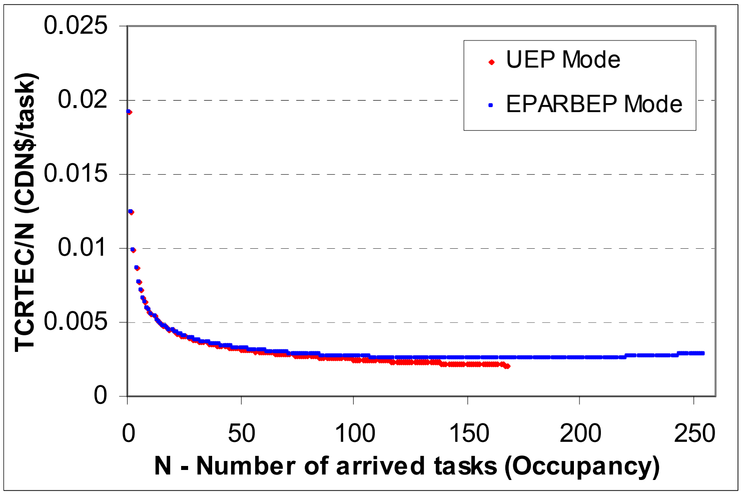


Fig. 6.8: Average cost of response time and energy consumption for executing N homogeneous tasks

In Fig.6.8, the TCRTEC is the appropriate performance metric that was used to synthesize and evaluate the algorithm because it factors response time and energy consumption. Notice how the EPARBEP mode is suboptimal compared to the UEP mode because it incurs a higher TCRTEC cost, but it is more robust because it budgets energy better and there by executes more tasks.

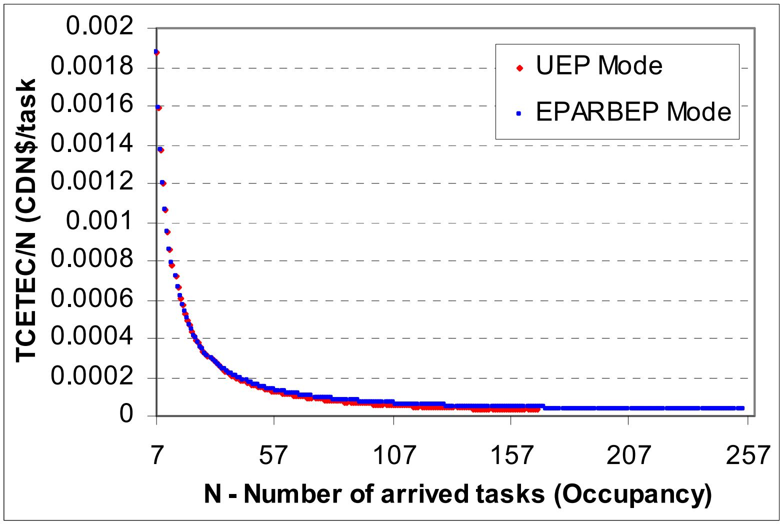


Fig. 6.9: Average cost of execution time and energy consumption for executing N homogeneous tasks

In Fig 6.9, the TCTEC performance metric uses execution time instead of response time making it advantageous in leasing out computational resources because execution time (unlike response time) for all tasks, can be conveniently measured by a global time scale. The TCETEC and TCRTEC performance metrics both confirm that the EPARBEP mode is suboptimal but more robust in comparison to the UEP mode.

Figures 6.4 - 6.9 show that the UEP mode prematurely drains the battery by only competing a maximum of (all) 168 tasks under a full battery energy budget, while the EPARBEP mode completes 255 tasks (approximately 52% more tasks).

6.4.3 Simulation II: STMBAD Algorithm's EPARBEP Mode Versus UEP mode for N Heterogeneous Tasks

We repeat Simulation I, but now consider tasks with heterogeneous computation volumes. We assume the computation volume of tasks is Gaussian distributed with a mean of 500 base instructions and a standard deviation of 100 base instructions (20%). Following these assumptions Fig. 6.10 and Fig. 6.11 summarize the results.

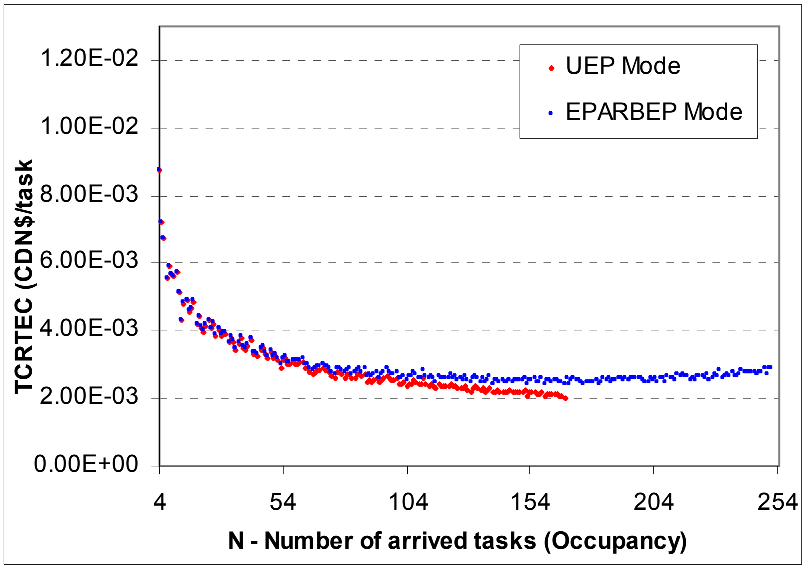


Fig. 6.10: Average cost of response time and energy consumption for executing *N* heterogeneous tasks (Gaussian distributed computation volumes)

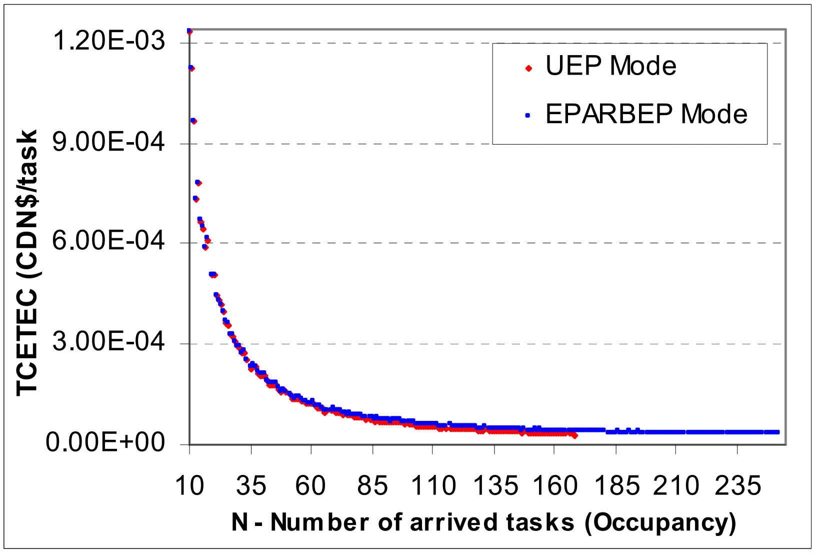


Fig. 6.11: Average cost of execution time and energy consumption for executing *N* heterogeneous tasks (Gaussian distributed computation volumes)

Figs. 6.10 and 6.11 both illustrate that the UEP mode prematurely drains the battery by only fully completing a maximum of 168 tasks under a full battery charge while the EPARBEP mode completes 252 tasks (50% more).

6.5 Multiple Energy Sources

Let as assume the mobile computing device has *m* multiple processors In addition, let us initially assume each processor has its independent energy source.

6.5.1 Mobile Hardware Parameters For Multiple Energy Sources.

Table 6.3 summarizes other hardware resource/parameters of the mobile device. The j^{th} index is from one to *m*. These parameters corresponds with each processing stream's power source.

Table 6.3: Multiple Energy Sources

Parameter	Meaning	SI Unit
$E_{m,j}$	Battery energy level of j^{th} processing stream	Joules
$E_{\theta,j}$	Threshold energy level of j^{th} processing stream	Joules
$(E_{m,j} - E_{\theta,j})$	Usable battery energy of j^{th} processing stream	Joules
$E_{cap,j}$	Maximum energy capacity of j^{th} processing stream (under full charge)	Joules
$\varepsilon_{\%,j}$	Remaining battery energy percentage of j^{th} processing stream, $\varepsilon_{\%,j} \in [0,1]$	dimensionless

6.5.2 Single or Multiple Energy Sources

It is worth mentioning that the analysis done assumes each processing stream has its independent battery source of equal capacity, but not necessarily of equal energy level. In practice, a special case of this assumption is usually implemented where all parallel processing streams share only one battery source; an example is the iPhone 5. We can simply narrow the work to single energy sources by substituting each and every $\varepsilon_{\%,j}$ for $\varepsilon_{\%}$, i.e., $\varepsilon_{\%,j} = \varepsilon_{\%}, \forall_{j \in \{1,2 \ldots m\}}$. We call this operation mode *homogenous EPARBEP mode*. As mentioned previously, if the mobile device is currently being re-charged (battery inflow energy exceeds current use) and it is known in advance that the mobile device will not be disrupted from recharging its battery/batteries until completion, then during the recharging period we can prematurely set $\varepsilon_{\%,j} = 1, \forall_{j\in\{1,2...m\}}$ since energy is temporarily not a scarce resource during foreseen battery recharge period. Also, all of the work presented in this thesis can be extended to non-mobile work stations or computing devices that have a reliable and unlimited power supply (but not free) by setting $\varepsilon_{\%,j} = 1, \forall_{j\in\{1,2...m\}}$ .We define this operation mode as UEP mode.

6.5.3 Defining operation modes for multiple energy sources

Homogenous EPARBEP mode set $\varepsilon_{\%,j} := \varepsilon_{\%}(\mathit{from\ OS}), \forall_{j\in\{1,2...m\}}$ - this means that the mobile computing device has multiple processors that utilize a single energy source.
Heterogeneous EPARBEP mode set $\varepsilon_{\%,j} := \varepsilon_{\%,j}(\mathit{from\ OS}), \forall_{j\in\{1,2...m\}}$ - this means that the mobile computing device has an independent energy source associated with each processor[44].
UEP mode $\varepsilon_{\%,j} = 1, \forall_{j\in\{1,2...m\}}$ - this implies that energy is not a budget. It is useful when the mobile device, with multiple (or single) energy sources is currently being recharged or is applicable to work stations that have a steady (but not free) supply of energy.
So far, All the algorithms presented in this thesis operate under UEP mode. We would like to extend them to operate under homogenous and heterogeneous EPARBEP modes. We do so in the next few sections in order to draw some insights on the effects of the UEP and EPARBEP modes.

6.6 Extending The SBDPP Algorithm to Include EPARBEB Mode

6.6.1 A Processing Stream Cost Function

Recall that the SBDPP is the *Single Buffer Decision and Parallel Processing* algorithm that was synthesized in Chapter IV.
Its modified cost function that includes the remaining energy percentage is as follows.

[44] We assume each battery (that is associated with each processor) has equal energy capacity.

$$C_j = \underbrace{\frac{u_\varepsilon}{\varepsilon_{\%,j}} \overbrace{\lambda_j B_k (P_k)^{\alpha_j -1}}^{\text{Task' s energy (J)}}}_{\text{Task' s (Adjusted) energy cost (\$)}} + \underbrace{u_{t,k} \overbrace{(\frac{B_k}{P_k} + t_{\theta,k,j})}^{\text{Task' s response time (s)}}}_{\text{Task' s response time cost (\$)}} \qquad (\$)$$

6.6.2 *Minimized Constrained Cost Function of the jth processing stream*

For a task $T_k \in T$, the minimum constrained cost function that factors the processing constraints and the remaining energy percentage is as follows.

$$C_{j,\min} = \left\{ \begin{array}{c} \alpha_j B_k \left(\frac{u_\varepsilon}{\varepsilon_{\%,j}}\lambda_j\right)^{\frac{1}{\alpha_j}} \left(\frac{u_{t,k}}{(\alpha_j -1)}\right)^{\frac{\alpha_j -1}{\alpha_j}} + u_{t,k} t_{\theta,k,j},\ \text{if}\ \mathrm{P}_{Maxj} \ge \left(\frac{\varepsilon_{\%,j} u_{t,k}}{(\alpha_j -1) u_\varepsilon \lambda_j}\right)^{\frac{1}{\alpha_j}} \ge p_{\mu,k} \\ \frac{u_\varepsilon}{\varepsilon_{\%,j}} \lambda_j B_k (p_{\mu,k})^{\alpha_j -1} + u_{t,k}(\frac{B_k}{p_{\mu,k}} + t_{\theta,k,j}),\ \text{if} \left(\frac{\varepsilon_{\%,j} u_{t,k}}{(\alpha_j -1) u_\varepsilon \lambda_j}\right)^{\frac{1}{\alpha_j}} < p_{\mu,k} \\ \frac{u_\varepsilon}{\varepsilon_{\%,j}} \lambda_j B_k (P_{Maxj})^{\alpha_j -1} + u_{t,k}(\frac{B_k}{P_{Maxj}} + t_{\theta,k,j}),\ \text{if} \left(\frac{\varepsilon_{\%,j} u_{t,k}}{(\alpha_j -1) u_\varepsilon \lambda_j}\right)^{\frac{1}{\alpha_j}} > P_{Maxj} \end{array} \right\}$$

for $\alpha_j \in (1,3]$.

If we assume loading times of **all** tasks are negligible ($t_{\theta,k,j} \approx 0$) the cost function reduces to:

$$C^{(2)}{}_{j,\min}=\left\{\begin{array}{l}\alpha_j\left(\frac{u_\varepsilon}{\varepsilon_{\%,j}}\lambda_j\right)^{\frac{1}{\alpha_j}}\left(\frac{u_{t,k}}{(\alpha_j-1)}\right)^{\frac{\alpha_j-1}{\alpha_j}}\text{, if }\mathrm{P}_{Maxj}\geq\left(\frac{\varepsilon_{\%,j}u_{t,k}}{(\alpha_j-1)u_\varepsilon\lambda_j}\right)^{\frac{1}{\alpha_j}}\geq p_{\mu,k}\\ \qquad\frac{u_\varepsilon}{\varepsilon_{\%,j}}\lambda_j(p_{\mu,k})^{\alpha_j-1}+\frac{u_{t,k}}{p_{\mu,k}}\text{, if}\left(\frac{\varepsilon_{\%,j}u_{t,k}}{(\alpha_j-1)u_\varepsilon\lambda_j}\right)^{\frac{1}{\alpha_j}}<p_{\mu,k}\\ \qquad\frac{u_\varepsilon}{\varepsilon_{\%,j}}\lambda_j(P_{Maxj})^{\alpha_j-1}+\frac{u_{t,k}}{P_{Maxj}}\text{, if}\left(\frac{\varepsilon_{\%,j}u_{t,k}}{(\alpha_j-1)u_\varepsilon\lambda_j}\right)^{\frac{1}{\alpha_j}}>P_{Maxj}\end{array}\right\}$$

for $\alpha_j \in (1,3]$.

We now have sufficient information to describe the SBDPP algorithm under EPARBEB and UEP modes.

6.6.3 Single-Buffer Decision & Parallel Processing Algorithm (SBDPP) Under EPARBEP and UEP modes.

1. User or OS specifies u_ε for all tasks and may specify different $u_{t,k}$ for each $T_k \in T$.
2. For an arriving task $T_k \in T$ we evaluate and compare the minimum processing cost ($C_{j\min}$)) of processing the task in each of the available processing streams. A task $T_k \in T$ should follow a stream j^* such that $C_{j^*,\min} = \min_{1\leq j\leq m}\{C_{j,\min} \mid N_j = 0\}$ thereby it acquires the label T_{k,j^*} and is processed by the $\vec{P}_{s,j^*}$ processor at the optimum processing rate.
3. Task T_{k,j^*} is executed by $\vec{P}_{s,j^*}$ at the optimum processing rate:

$$P_{s,j^*} = \left\{ \begin{array}{c} P^*_k, \text{ if } P_{Max,j^*} \geq P^*_k = \left(\dfrac{\varepsilon_{\%,j} u_{t,k}}{(\alpha_{j^*} - 1) u_\varepsilon \lambda_{j^*}} \right)^{\frac{1}{\alpha_{j^*}}} \geq p_{\mu,k} \\ p_{\mu,k}, \text{ if } \left(\dfrac{\varepsilon_{\%,j} u_{t,k}}{(\alpha_{j^*} - 1) u_\varepsilon \lambda_{j^*}} \right)^{\frac{1}{\alpha_{j^*}}} < p_{\mu,k} \\ P_{Max,j^*}, \text{ if } \left(\dfrac{\varepsilon_{\%,j} u_{t,k}}{(\alpha_{j^*} - 1) u_\varepsilon \lambda_{j^*}} \right)^{\frac{1}{\alpha_{j^*}}} > P_{Max,j^*} \end{array} \right\}$$

4. If T_{k,j^*} is to be cancelled/deleted or when it is completed, set $P_{s,j^*} = 0$ and $N_{j^*} = 0$.

In Step 2, If all the task loading times are insignificant use $C^{(2)}{}_{j\min}$ instead of $C_{j\min}$. If all processors are homogenous and loading times are homogenous, ignore step 2 and utilize Round Robin dispatching.

Algorithm Notes

- For **Homogenous EPARBEP** mode, acquire $\varepsilon_{\%,}$ from the one and only battery source and then set $\varepsilon_{\%,j} := \varepsilon_{\%,}, \forall_{j \in \{1,2...m\}}$. (use EPARBEP mode when $\varepsilon_k << E_{cap}$).
- For **Heterogeneous EPARBEP** mode acquire $\varepsilon_{\%,j}$ from each processing stream's battery (respectively). (use EPARBEP mode when $\varepsilon_k << E_{cap}$.)
- For **UEP** mode set $\varepsilon_{\%,j} := 1, \forall_{j \in \{1,2...m\}}$
- The algorithm's **dispatche**r is described in steps 2.
- Step 3 specifies the algorithm's **speed scaling function**.

Recall in Chapter 4 (section 4.41), we defined S_j to be the (*user specified) power sensitivity factor.*

The modified definition of S_j, under EPARBEP mode is defined as follows.

$$\frac{u_{t,k}}{u_{\varepsilon}} = \frac{(\alpha_j - 1)}{\varepsilon_{\%,j}} \lambda_j \left[\left(p_{\mu,k}\right)^{\alpha_j} + \left(\left(P_{Max,j}\right)^{\alpha_j} - \left(p_{\mu,k}\right)^{\alpha_j} \right) S_j \right] \text{ where } S_j \in [0,1].$$

Using this modified S_j, we present the Single Buffer Assisted Decision & Processing Algorithm (SBADPA) under the EPARBEP mode that extends the functionality of the SBDPP algorithm (under EPARBEP mode) by allowing the user or the OS of the mobile device to further modify a task's unit cost of time/energy in order to achieve a desired (linearly controlled) mode of operation (economy/performance mode).

See **Appendix II** for the calibration of the ratio of time and energy prices under EPARBEP Mode. Also refer to **Appendix III** for determining a task's mode of operation (economy/performance) with this modified definition of S_j.

6.7 Extending The SBADPA Algorithm to Include EPARBEB Mode

1. User or OS specifies u_{ε} for all tasks and may specify different $u_{t,k}$ for each $T_k \in T$.
2. For an arriving task $T_k \in T$, solve

   $$S_j = \frac{1}{\left(\left(P_{Max,j}\right)^{\alpha_j} - \left(p_{\mu,k}\right)^{\alpha_j} \right)} \left[\frac{\varepsilon_{\%,j} u_{t,k}}{(\alpha_j - 1)\lambda_j u_{\varepsilon}} - \left(p_{\mu,k}\right)^{\alpha_j} \right], \forall_{1 \le j \le m}$$

   for each processing stream. If $S_j < 0$, set $S_j = 0$ and If $S_j > 1$, set $S_j = 1$ (satisfying processing constraints).
3. User or OS of mobile device can eliminate considering streams whose S_j values are undesirable **(optional)**.
4. For the given tasks $T_k \in T$, we evaluate and compare the minimum modified cost function of processing a task ($\hat{C}_{j\min}(.)$) in each of the available processing streams, where:

$$\hat{C}_{j,\min} = \left(\frac{\alpha_j B_k}{(\alpha_j - 1)} \right) \left(\left[(p_{\mu,k})^{\alpha_j} + \left((P_{Max,j})^{\alpha_j} - (p_{\mu,k})^{\alpha_j} \right) S_j \right]^{-\frac{1}{\alpha_j}} \right) + t_{\theta,k,j}$$

If **all** task loading times are negligible use

$$\hat{C}_{j,\min} = \left(\frac{\alpha_j}{(\alpha_j - 1)} \right) \left(\left[(p_{\mu,k})^{\alpha_j} + \left((P_{Max,j})^{\alpha_j} - (p_{\mu,k})^{\alpha_j} \right) S_j \right]^{-\frac{1}{\alpha_j}} \right)$$

5. A task $T_k \in T$ should follow a stream $j*$ such that $\hat{C}_{j,\min} = \min_{1 \le j \le m} \{ \hat{C}_{j,\min} \mid N_j = 0 \}$ thereby it acquires the label $T_{k,j*}$ and is processed by the $\vec{P}_{s,j*}$ processor at the optimum processing rate.
6. The optimum processing rate of the $\vec{P}_{s,j*}$ processor is

$$P_{s,j*} = \left[(p_{\mu,k})^{\alpha_j} + \left((P_{Max,j*})^{\alpha_j} - (p_{\mu,k})^{\alpha_j} \right) S_{j*} \right]^{\frac{1}{\alpha_j}}.$$

7. If $T_{k,j*}$ is to be cancelled/deleted or when it is completed, set $P_{s,j*} = 0$ and $N_{j*} = 0$.

Algorithm Notes

a. Use **EPARBEP** mode when $\varepsilon_k << E_{cap}$.

b. For **Homogenous EPARBEP** mode, acquire $\varepsilon_{\%,}$ from the one and only battery source and then set $\varepsilon_{\%,j} := \varepsilon_{\%,}, \forall_{j \in \{1,2...m\}}$.

c. For **Heterogeneous EPARBEP** mode acquire $\varepsilon_{\%,j}$ from each processing stream's battery (respectively).

d. For **UEP** mode set $\varepsilon_{\%,j} := 1, \forall_{j \in \{1,2...m\}}$.

e. If all processors are homogenous and loading times are homogenous, ignore step 4 and 5 and instead use Round Robin dispatching.

f. The algorithm's dispatcher is comprised of steps (4 & 5).

g. Step 6 is the algorithm's speed scaling function.

In the next section, we extend the *Single-threading Multi-buffer Scheduling & Processing* algorithm (SMBSPP) to include the EPARBEP mode.

6.8 Extending The SMBSPP Algorithm to Include EPARBEB Mode

6.8.1 The Minimized Constrained Cost Function of the j^{th} Processing Stream under EPARBEP

The minimum constrained cost function that that includes homogenous and heterogeneous EPARBEP modes is as follows:

$$C_{j\,\min}(N_j)=\sum_{k=1}^{\mathrm{N_j}}\left(\frac{u_{\varepsilon}}{\varepsilon_{\%,j}}\lambda_j B_k(P*_k)^{\alpha_j-1}+\left(\frac{B_k}{P*_k}\sum_{r=k}^{\mathrm{N_j}}u_{t,r}\right)+\left(u_{t,k}\sum_{r=1}^{k}t_{\theta,r,j}\right)\right) \qquad (6.2)$$

and

$$P*_k=\left\{\begin{array}{c}\left(\frac{\varepsilon_{\%,j}}{(\alpha_j-1)u_{\varepsilon}\lambda_j}\sum_{r=k}^{\mathrm{N_j}}u_{t,r}\right)^{\frac{1}{\alpha_j}},\ \text{if }\mathrm{P}_{Max,j}\geq\left(\frac{\varepsilon_{\%,j}}{(\alpha_j-1)u_{\varepsilon}\lambda_j}\sum_{r=k}^{\mathrm{N_j}}u_{t,r}\right)^{\frac{1}{\alpha_j}}\geq p_{\mu,k}\\ p_{\mu,k},\ \text{if}\left(\frac{\varepsilon_{\%,j}}{(\alpha_j-1)u_{\varepsilon}\lambda_j}\sum_{r=k}^{\mathrm{N_j}}u_{t,r}\right)^{\frac{1}{\alpha_j}}<p_{\mu,k}\\ P_{Max,j},\ \text{if}\left(\frac{\varepsilon_{\%,j}}{(\alpha_j-1)u_{\varepsilon}\lambda_j}\sum_{r=k}^{\mathrm{N_j}}u_{t,r}\right)^{\frac{1}{\alpha_j}}>P_{Max,j}\end{array}\right\} \qquad (6.3)$$

for $k\in\{1,2\ldots N_j\}$ & $\alpha_j\in(1,3]$

$P*_k$ is the optimum constrained processing rate of potentially executing the task stored in the k^{th} index of the $\vec{Q}_{s,j}$ memory queue.

6.8.2 *Single-threading Multi-buffer Scheduling & Processing Algorithm (SMBSPP) under EPARBEP and UEP modes*

1. User or OS specifies u_ε for all tasks and may specify different $u_{t,k}$ for each $T_k \in T$.

2. For an arriving task, $T_k \in T$, we evaluate and compare the minimum potential processing cost, $C_{j\min}(N_j + 1)$ of virtually introducing and processing the arriving task in each of the available processing streams $(1 \le \mathrm{j} \le m)$. The task virtually acquires a position index according to $B_k / u_{t,k}$ (**SCVPPT**) in each of the processing streams.

3. Using equations (6.2) and (6.3), the task should follow a stream j^* such that $C_{j^*\min}(N_{j^*} + 1) = \min\limits_{1 \le j \le m}\left\{C_{j\min}(N_j + 1)\right\}$ thereby it acquires the position index according to $\left(B_k / u_{t,k}\right)$ **(SCVPPT)** and will be processed by the $\vec{P}_{s,j^*}$ processor at some adjusted optimum processing rate.

4. Update N_{j^*} .

5. The task stored at system index $\left(1, j^*\right)$ i.e., the task T_{1,j^*}, is executed by the $\vec{P}_{s,j^*}$ processor at the optimum adjusted processing rate defined below:

$$P_{s,j^*} = \left\{ \begin{array}{c} \left(\dfrac{\varepsilon_{\%,j}}{(\alpha_{j^*} - 1)u_\varepsilon \lambda_{j^*}} \sum\limits_{r=1}^{\mathrm{N_{j^*}}} u_{t,r}\right)^{\frac{1}{\alpha_{j^*}}}, \text{ if } \mathrm{P}_{Maxj^*} \ge \left(\dfrac{\varepsilon_{\%,j}}{(\alpha_{j^*} - 1)u_\varepsilon \lambda_{j^*}} \sum\limits_{r=1}^{\mathrm{N_{j^*}}} u_{t,r}\right)^{\frac{1}{\alpha_{j^*}}} \ge p_{\mu,1} \\ p_{\mu,1}, \text{ if} \left(\dfrac{\varepsilon_{\%,j}}{(\alpha_{j^*} - 1)u_\varepsilon \lambda_{j^*}} \sum\limits_{r=1}^{\mathrm{N_{j^*}}} u_{t,r}\right)^{\frac{1}{\alpha_{j^*}}} < p_{\mu,1} \\ P_{Maxj^*}, \text{ if} \left(\dfrac{\varepsilon_{\%,j}}{(\alpha_{j^*} - 1)u_\varepsilon \lambda_{j^*}} \sum\limits_{r=1}^{\mathrm{N_{j^*}}} u_{t,r}\right)^{\frac{1}{\alpha_{j^*}}} > P_{Maxj^*} \end{array} \right\}$$

6. Repeat steps 4 & 5 whenever a task/s is either dynamically introduced or deleted in $\vec{Q}_{s,j*}$.
7. Once the execution of the task $T_{1,j*}$ is complete or terminated, the indices of all tasks in memory queue $\vec{Q}_{s,j*}$ are shifted down by one creating room for another task.
8. If any task or tasks in $\vec{Q}_{s,j*}$ are deleted/cancelled, each alive task in $\vec{Q}_{s,j*}$ is shifted to the minimum available slot starting from the first index to preserve task priority.
9. If we are to enforce FCFS queuing service policy or we are not allowed to exercise preemption, whenever a task enters the queue of a processing stream it acquires the Smallest Empty Index (SEI), also in step 2, while calculating the virtual cost of introducing the task to each processing stream, the arriving task virtually acquires the SEI.
10. Ignore steps 2 & 3 when processors are homogeneous and instead utilize Round Robin dispatching.

Notes pertaining to algorithm's description

a. Steps 2 & 3 summarize the SMBSPP algorithm's default dispatcher (**MMCVITPS**) under the SCVPPT service discipline.
b. Step 5 describes the speed scaling function (**OSTSSF**).
c. Use (homogenous/heterogeneous) **EPARBEP** mode when $\varepsilon_k << E_{cap}$.
d. For **Homogenous EPARBEP** mode, acquire $\varepsilon_{\%,}$ from the one and only battery source and then set $\varepsilon_{\%,j} := \varepsilon_{\%,}, \forall_{j\in\{1,2...m\}}$
e. For **Heterogeneous EPARBEP** mode acquire $\varepsilon_{\%,j}$ from each processing stream's battery (respectively).
f. For **UEP** mode set $\varepsilon_{\%,j} := 1, \forall_{j\in\{1,2...m\}}$

6.9 Effects of the EPARBEB and UEP Modes on the Speed Scaling functions and Dispatchers of the Algorithms

6.9.1 Effects of the EPARBEB and UEP Modes on the Speed Scaling functions of the Algorithms

Through inspection, the unconstrained speed scaling function of each and every algorithm can be written in this form.

$$P_{s,j} = f_j .(\varepsilon_{\%,j})^{\frac{1}{\alpha_j}} \tag{6.4}$$

Where f_j is the optimum speed of the j-th processor under the UEP mode. Upon closer examination, f_j depends on many other parameters such as the current occupancy of the processing stream, the j-th processors power function parameters, and the user profile parameters of the active task/s.

We clearly see that under the heterogeneous EPARBEP mode, equation (6.4) suggests that each processor's speed scaling function is attenuated by a dynamic factor of $(\varepsilon_{\%,j})^{\frac{1}{\alpha_j}}$ relative to that of the optimum[45]. This is an attenuation and not a dilation since $0 < \varepsilon_{\%,j} \leq 1$, and for CMOS based processors, $1 < \alpha_j \leq 3$, hence implying $0 < (\varepsilon_{\%,j})^{\frac{1}{\alpha_j}} \leq 1$.

Let us define $(\varepsilon_{\%,j})^{\frac{1}{\alpha_j}}$ as the attenuation factor. $\varepsilon_{\%,j}$, the remaining energy percentage of the j-th processor actually varies with time. It decreases in the long run when the j-th battery is under use and it increases when the j-th battery is recharging. Fig. 6.12 illustrates this.

[45] The optimum here considers only unconstrained processing rates, which is the dominating condition most of the time.

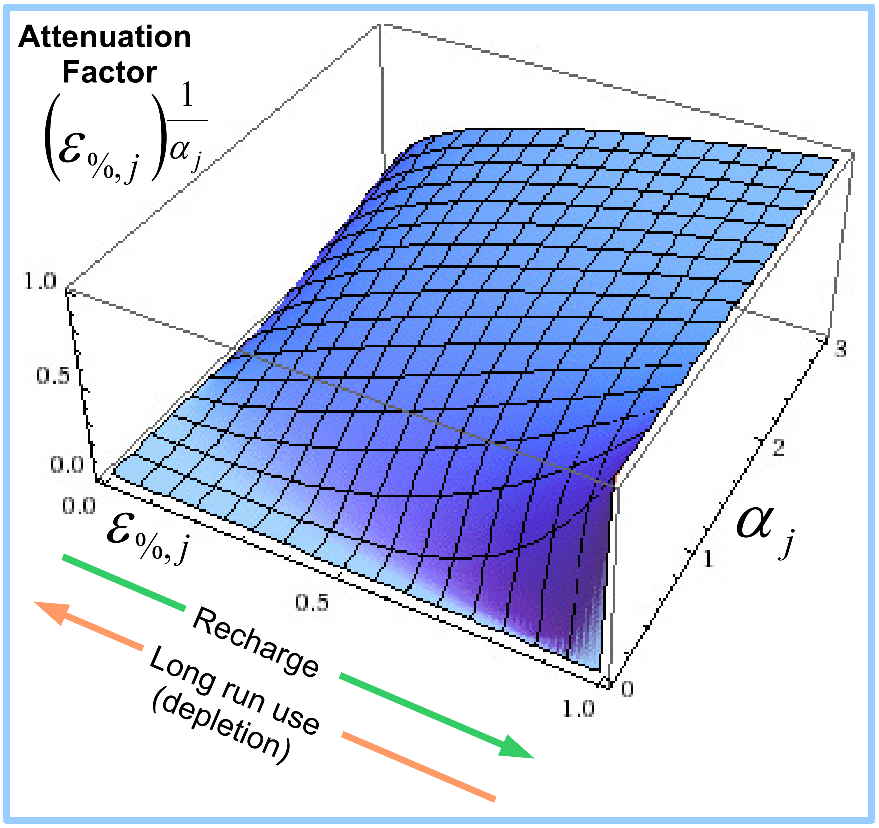


Fig. 6.12: Attenuation factor induced by the EPARBEP mode on speed scaling functions

Figure 6.12 clearly shows that a processor with a small α value is more susceptible to this attenuation (lower values of the attenuation factor). It also illustrates that low battery energy level/s substantially attenuate the speed scaling functions of the processor/s compared to that of the optimum (UEP mode).

Under the homogenous EPARBEP mode, we have $\varepsilon_{\%,j} := \varepsilon_{\%,}, \forall_{j\in\{1,2...m\}}$. The attenuation effect is not homogenous among the speed scaling functions of processors because although all processors share the one and only battery source (same $\varepsilon_{\%,}$), the attenuation factor is still affected by the exponent of each processor's power function (α_j). Moreover, equation (6.4) reduces to $P_{s,j} = f_j.(\varepsilon_{\%})^{\frac{1}{\alpha_j}}$ and the attenuation factor reduces to $(\varepsilon_{\%,})^{\frac{1}{\alpha_j}}$ in this mode.

To complete the argument, we acknowledge that the speed scaling functions of the algorithms can also operate at minimum or maximum constrained processing rates.

During these rare special cases, the EPARBEP mode coincides with that of the optimum UEP mode. In other words, the remaining energy percentage of each[46] of the j-th processor $\left(\varepsilon_{\%,j}\right)$ has no effect on the speed scaling functions. See the speed scaling functions for verification.

Let us sum up the finding of this section. The remaining energy percentage parameter/s attenuate the optimum (unconstrained) speed scaling functions of processors, i.e. they slow down the speed of each processor. In previous sections, this slowed down speed was defined to be robust. As the battery/batteries energy level/s decrease under use, the processors achieve robust processing rates that are slower compared to the optimum. This allows the computing device to save more critical energy especially when the battery/batteries are almost drained out. Finally, when the processors operate at constrained processing rates, the remaining energy percentage parameter/s have no effect on the processing rates of processors.

6.9.2 Effects of the EPARBEB and UEP Modes on the Dispatchers of the Algorithms

Referring to step 2 of the SBDPP and *SBADPA* algorithms as well as step 3 of the SMBSPP algorithm, It is quite clear that when each processing stream has its independent energy source[47], the dispatchers are dynamically affected by $\varepsilon_{\%,j}$. The j-th processing stream becomes more expensive the more $\varepsilon_{\%,j}$ depletes and vise versa.

Let us first provide some contextual details. Through inspection and after carrying out a mild algebraic manipulation, we see that when the processing rates are unconstrained, the optimum energy cost component (lets call it $\breve{C}_j$) of the j-th processing stream for each of our algorithms can be expressed by equation (6.5).

[46] Or for all under homogenous EPARBEP.

[47] Energy source of equal capacity but not necessarily of equal energy level.

$$\breve{C}_j = \frac{g_j}{\left(\varepsilon_{\%,j}\right)^{\frac{1}{\alpha_j}}} \tag{6.5}$$

Where g_j is the optimum energy cost component of the j-th processing stream (under UEP mode)

Likewise, when the processing rates are constrained, equation (6.5) transforms to equation (6.6). This assertion is easily verified by inspecting the dispatchers of the algorithms.

$$\breve{C}_j = \frac{g_j}{\left(\varepsilon_{\%,j}\right)} \tag{6.6}$$

Let us first consider the heterogeneous EPARBEP mode.

g_j, the optimum energy cost component of the j-th processing stream under the UEP mode is multiplied by some dynamic factor. We define this dynamic factor a dilation factor because it is always greater or equal to one.

When the processing rates are unconstrained and under the heterogeneous EPARBEP mode, the dilation factor according to (6.6) is a function of each processing stream's remaining energy percentage as well as the exponent of each processor's power function. When the processing rates are constrained and under heterogeneous EPARBEP mode, the dilation factor according to (6.6) is only a function of each processing stream's remaining energy percentage. Figures 6.13 and 6.14 illustrate this.

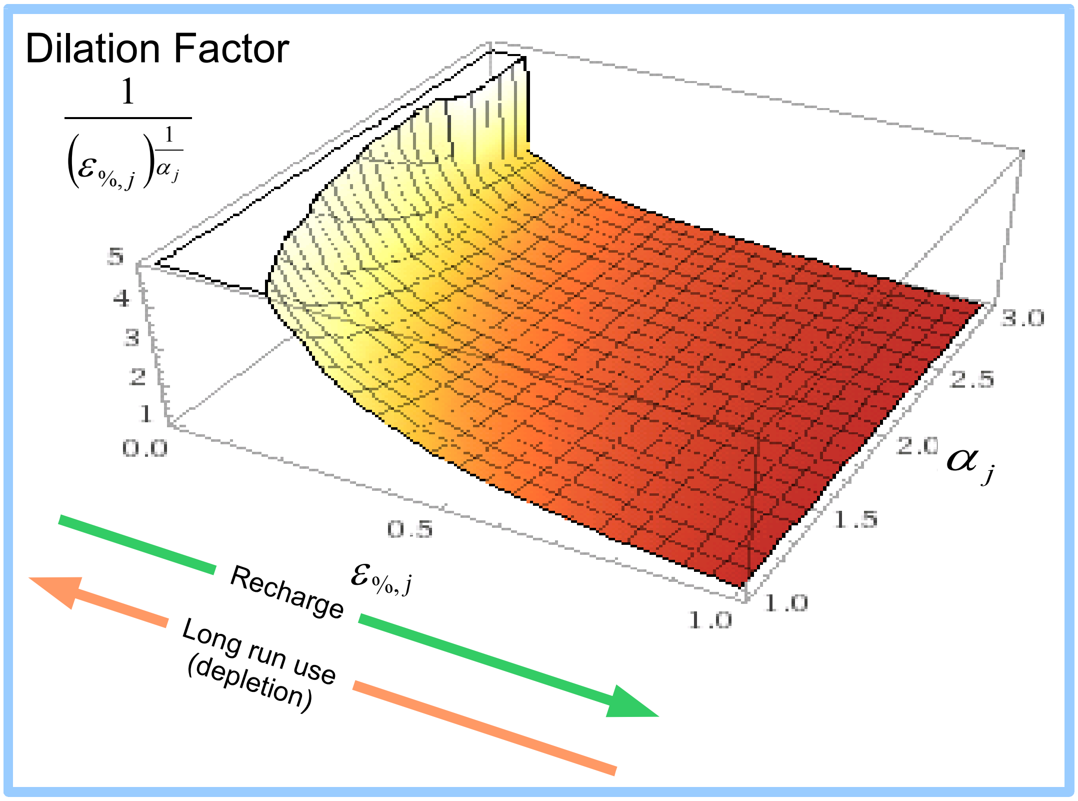


Fig 6.13: Dilation factor induced by the EPARBEP mode on dispatchers under unconstrained processing rates

Fig. 6.13 informs us that when processing rates are unconstrained, the processors with small exponents of their power functions incur a large dilation factor under the EPARBEP mode.

To demonstrate the effect of load balancing, let as examine the contour diagram, Fig. 6.13, and superimpose in it a simple example. In this example, let us have a processor 1 with a power function exponent of $\alpha_1 = 1.5$ and a processor 2 with a power function exponent of $\alpha_2 = 2$. Under the UEP mode and considering all other conditions being equal (e.g. occupancy, user profile parameters, etc) as well as having unconstrained processing rates, the dispatchers will select processor 1 over processor 2 because it is cheaper. In other words the UEP mode is inherently biased because it always attempts to optimally process task/s along the cheapest processing stream (see the dispatchers of the algorithms under the UEP mode).

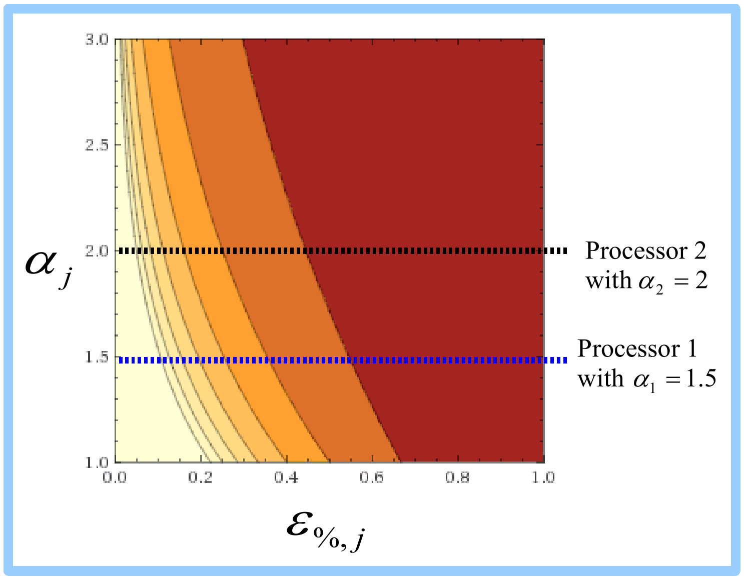


Fig. 6.14: Contour diagram of Fig. 6.13 with a superimposed example

Fig. 6.14 is interesting. It informs us that under the EPARBEP mode, the processors that are efficient (have small values of α_j and are always favored by the UEP mode, e.g. processor 1) actually incur large dilation factors of their energy cost terms. This means that the EPARBEP induces a load balancing effect when the processor are heterogeneous interims of the exponents of their power functions. This load balancing effect is further accentuated by independent energy sources because as the UEP mode selects processor 1 for processing, the remaining energy percentage of processor 1 ($\varepsilon_{\%,1}$) decreases in the long run, making it more expensive under the EPARBEP mode to further execute tasks by processor 1.

In the rare case that we operate at minimum or maximum constrained processing rates, the load balancing effect still occurs as long as we have independent energy sources (heterogeneous EPARBEP mode). This is corroborated by equation (6.6) and Fig. 6.15.

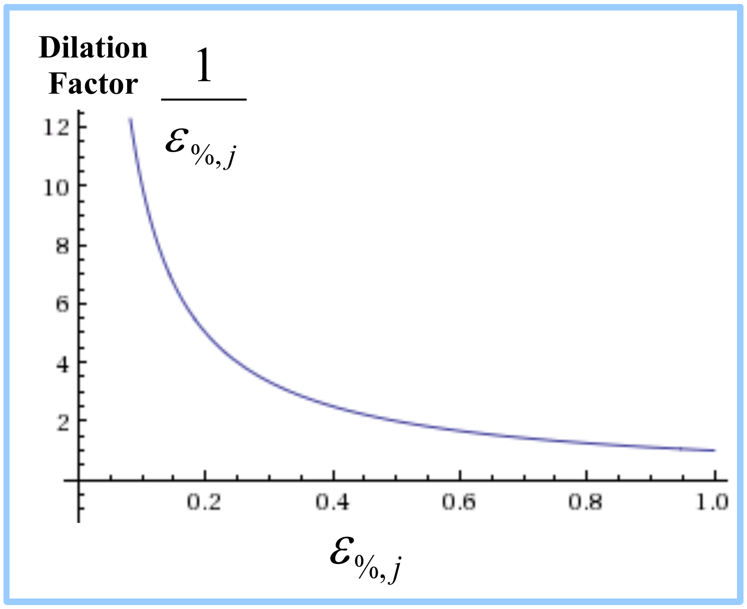


Fig. 6.15: Dilation factor induced by the EPARBEP mode on dispatchers under constrained processing rates

Fig. 6.15 demonstrates that when processors operate at minimum or maximum constrained processing rates, the energy component of the algorithm's cost functions are hyperbolically inflated by each processing stream's battery energy level (heterogeneous EPARBEP mode). Under the UEP mode, the dispatchers are not affected by the battery energy levels. This suggests that the efficient processing streams that are favored by the UEP mode become more expensive under the EPARBEP mode once their corresponding (and independent) battery energy levels decrease due to disproportionate use. Hence the load balancing effect is induced by the (independent) remaining battery energy level of each processor.

Under the homogenous EPARBEP mode, equation (6.5) reduces to equation (6.7) and equation (6.6) reduces to equation (6.8)

$$\breve{C}_j = \frac{g_j}{\left(\varepsilon_{\%}\right)^{\frac{1}{\alpha_j}}} \tag{6.7}$$

$$\breve{C}_j = \frac{g_j}{\left(\varepsilon_{\%}\right)} \tag{6.8}$$

We clearly see that when processing rates are unconstrained, load balancing effect is still induced by the homogenous EPAREP mode, but this type of load balancing is only influenced by the heterogeneity in the exponents of the processors' power functions

When the processors operate at the minimum or maximum processing constraints under the homogenous EPARBEP mode, equation (6.8) suggests that each processing stream's energy cost terms are dilated by the same dynamic parameter. This dynamic parameter is simply the reciprocal of the one and only battery energy level. Therefore, we can not speculate on any existence of load balancing under this scenario.

Table 6.4 summarizes the findings of this section.

Table 6.4: Load balancing effect on dispatchers by EPARBEP modes

	Unconstrained Processing Rates	**Constrained Processing Rates**
Homogeneous EPARBEP Mode	Load balancing effect induced by the heterogeneous exponent of each processor's power function	Inconclusive
Heterogeneous EPARBEP Mode	Load balancing effect induced by each processor's independent battery energy level as well as the heterogeneous exponent of each processor's power function	Load balancing effect induced by each processor's independent battery energy level

6.10 Conclusion

The STMBAD algorithm provides some insights. It tells us that the optimum processing rate of a task is not a function of its computation volume (B_k). It also tells us once a task is dynamically included into the computing device' memory buffer, the optimum processing rate of the currently processed task increases. This processing rate increases because the aggregate cost function (that factors response time and energy consumption of all tasks in the multi-buffer) has increased and there exists a response time dependency among tasks due to single-threading. The algorithm has an operation mode where all tasks' unit cost of energy is heuristically affected by the device' remaining battery energy percentage in accordance with the micro-economic laws of demand and supply.

Using numerical simulations, we showed that when the remaining battery energy percentage is factored (EPARBEP mode), the algorithm: performs slightly slower (mildly more slower when the battery is almost drained out), but consumes far less energy (in many cases more than 30%), can complete significantly more jobs (about 52% more jobs for homogenous deterministic tasks and more than 50% more jobs for heterogeneous tasks with Gaussian distributed computation volumes) and ultimately allows the mobile computing device to last longer on the go. The algorithm explicitly finds a globally optimum (minimum) solution for the cost of response time and energy consumption of all active tasks in the device' buffer. We believe this robustness of the algorithm being able to handle dynamic inclusion of heterogeneous tasks in real time and it being able to take advantage of the remaining battery energy percentage also at run-time makes it appealing among hardware architectural planers and software programmers of mobile computing devices. The STMBAD algorithm can also be implemented in non-mobile work stations or computing devices that have a reliable and unlimited (but not free) supply of power by permanently setting the battery energy percentage parameter to one. Assuming we have *N* tasks queued up for processing, the algorithm has worse case computational complexities of O(*1*) and O(log(*N*)) under FCFS and SRPT service policies (respectively).

We extended all the previously constructed algorithms of this thesis to include the EPARBEP mode and analytically showed that when processors have their independent energy sources, the EPARBEP mode induces a load balancing effect by dilating the energy cost terms (of a given schedule). The EPAREP mode strategically slows down speed scaling functions as long as the processing rates are unconstrained. This slowdown or attenuation in processing rate is inversely correlated with the amount of remaining energy. Therefore the EPAREP mode strategically saves the critical energy needed for a computing device to last longer on the go. The UEP mode always leads to optimum speed scaling functions and dispatchers but is not always robust in the context of energy preservation.

In regard to the dispatching of tasks on to processors, the UEP mode inevitably leads to a biased selection of efficient processors over inefficient processors in order to optimally minimize both energy and response time costs. Comparatively, the EPARBEP mode is suboptimal, but when each processing stream has its own independent energy supply, the EPARBEP mode induces a load balancing effect on dispatchers that counters the selection bias of the UEP mode. Furthermore, under the EPARBEP mode, this load balancing effect was also shown to be induced by the heterogeneous exponent of each processor's power function even if the processors shared a single energy source as long as they operated under unconstrained processing rates. A limitation that should not be overlooked is that the EPARBEP mode is valid when the energy consumption of tasks is negligible compared to the energy capacity of the battery.

Chapter 7: Conclusion

7.1 Research Summary

In this thesis we synthesized, analyzed and simulated online scheduling algorithms to optimally assign a set of arriving heterogeneous tasks to heterogeneous speed-scalable processors under the single threaded computing architecture. We used dynamic speed-scaling (where each processor's speed is able to dynamically change within hardware and software processing constraints) to minimize the total cost of response time and energy consumption (TCRTEC) of the tasks. In our work, the processors were assumed to be heterogeneous in that they may have differed in their hardware specifications with respect to maximum processing rate, power function parameters and energy sources. Tasks were heterogeneously modeled in terms of computation volume, memory and minimum processing requirements. We also considered that the unit price of response time for each task is heterogeneous because the user may be willing to pay higher/lower unit prices for certain tasks, thereby increasing/decreasing their optimum processing rates. We modeled the overhead loading time incurred when a task is loaded by a given processor prior to its execution and assumed it to be heterogeneous as well. We constructed a theoretical model that was used to synthesize the parallel processing algorithms for the single buffered and multi buffered processors. We also used the micro-economic Laws of Supply and Demand (LSD) to heuristically adjust the unit price of energy in order to extend battery life through a proposed multi buffered, single processor algorithm. Further more, we extended all the multi processor algorithms to include single or multiple independent energy sources associated with each processor, where we analytically showed that load balancing is induced in heterogeneous processors when the unit price of energy is adjusted by the battery level of each processor in accordance with LSD. All the algorithms provide a common insight. They all inform us that the optimum processing rate of a given task is neither a function of its computation volume nor is it a function of its loading time. All the algorithms in this thesis could be used for optimized local parallel (heterogeneous) computing of mobile devices or energy aware work stations.

7.1.1 Theoretical Framework

We constructed a theoretical framework to mainly model heterogeneous tasks and processors. In this framework, we: defined some relevant mobile parameters including multiple energy source parameters; proposed *user profiles* to incorporate the preference of the user with respect to energy and response time pricing; discussed multiprocessor computing scenarios based on the potential maximum occupancy; and used formulas in current literature to deduce useful relationships pertaining to a task's computation volume, energy and power consumption. These relationships were corroborated with a detailed example. In this framework we also proposed and justified a financial performance metric, namely the cost of response time and energy consumption (TCRTEC) in dollars. This performance metric stems from the integration of the user (pricing) profiles of tasks with the resource consumption of schedules. The framework also described the relevant pre-processing constraints and defined traffic conditions as a benchmark to systematically simulate all the parallel processing algorithms in this thesis.

7.1.2 Single buffered Processors

We presented the first, elaborate, analytical study on the use of dynamic speed scaling to schedule heterogeneous tasks on single-buffered, heterogeneous, parallel processors with the objective of reducing the total cost of response time and energy consumption.

We synthesized and simulated the SBDPP algorithm and its variations (SBADPA and FPDPA). The algorithm and its variations run in real time to optimally dictate which processor among a multiple set of (single-buffered) parallel processors should process an incoming task, and they also explicitly determine the optimum processing rate of executing each tasks residing in each processor's single-buffer. The three versions of the algorithm are conceptually similar, but differ on their application and they each have dispatchers and dynamic speed-scaling functions of constant computational complexity.

These algorithms informed us that a task's computation volume influences its processing cost when the loading times of tasks are not negligible, which in turn influences the actual processing stream that will process the task. Moreover, when the loading times tasks are negligible, a tasks computation volume does not influence the actual processing stream that will process the task.

The algorithms were extended to allow migration. This was suggested through carrying out migration operations (*HMO*) of constant computational complexities (assuming a constant number of parallel processors) but a deep analysis on this front was not pursued. The optimum processing rate of a task under the single buffer scenario was found to be a function of the unit price of time over that of energy as well as the processors power function parameters. Further more, through a simple analytical example, it was shown that our algorithm's dispatcher outperformed the Round Robin dispatcher with cost savings correlated with the absolute values of both the energy and time prices.

Through simulations, we observed and constructed a very useful relationship between the average response time of a given task and the ideal deterministic inter-arrival period that maximizes system utilization for systems with parallel, single buffered processors.

7.1.3 Multi buffered Processors

We synthesized and simulated a novel online multiprocessor scheduling algorithm (SMBSPP) that schedules arriving heterogeneous tasks on to multi-buffered, heterogeneous, parallel processors. This algorithm constitutes a dispatcher (MMCVITPS), a service discipline (MMCVITPS) and a speed scaling function (SCVPPT). We assumed the single threading computing architecture where no processor executes more than a single task at any given time until completion unless preemption is dictated by the service discipline The SMBSPP algorithm informed us that once a task is dynamically included into a given memory queue of a processing stream, the optimum processing rate of the currently processed task (stored at the first index of the queue) is likely to change. The processing rate changes because the aggregate cost function of all tasks in the queue has changed and there exists a time dependency among tasks in the processing stream's memory queue due to single-threading. The algorithm explicitly finds a globally optimum solution for each aggregate cost function associated with each processing stream. This globally optimum solution minimizes the total cost of both energy consumption and response time of tasks in each processing stream. The solution explicitly obtains the optimum processing rates of each task in all memory queues of all processors.

Assuming each processing stream has roughly *n* tasks queued up, the algorithm's default dispatcher (MMCVITPS) was found to have a worse case computational complexity of $O(n^2)$ with heterogeneous response time pricing and $O(n)$ with homogenous response time pricing, and when it used the Round Robin dispatcher, it had a worse case computational complexity of $O(1)$. In terms of the TCRTEC/N metric, we demonstrated that the algorithms default dispatcher (MMCVITPS) significantly out performs the Round Robin dispatcher under the FCFS, SRPT and SCVPPT service disciplines for various stochastic and deterministic traffic conditions where the degree of processor heterogeneity was mild (power function parameters were conservatively chosen to differ from the mean by at most 8%) yet the MMCVITPS dispatcher drastically outperformed the Round Robin dispatcher with cost savings exceeding 100% on average. In terms of the TCRTEC/N metric, we demonstrated that the algorithms default dispatcher (MMCVITPS) significantly out performed the Round Robin dispatcher under the FCFS, SRPT and SCVPPT service disciplines for various stochastic and deterministic traffic conditions. In fact, we did not recommend the use of the Round Robin dispatcher in systems that utilize heterogeneous processors.

Through simulation, we demonstrated that the SMBSPP algorithm with its default dispatcher (MMCVITPS), service discipline (SCVPPT) and speed-scaling function (OSTSSF) had a fairly constant TSSC/N curve under heavy stochastic traffic conditions; this revealed the algorithm's robustness. It made it suitable to be implemented in energy aware work stations or *green* computational devices that utilize parallel processors and want to maintain a fairly stable (constant) operation cost under unpredictable heavy traffic conditions.

The proposed SCVPPT service discipline always matched or outperformed the FCFS and SRPT service disciplines as evaluated by the TCRTEC performance metric. When implemented in the algorithm, the SCVPPT and SRPT service disciplines each have computational complexities of $O(\log N_j)$. where N_j is the occupancy of a given processor. SCVPPT was found to behave exactly like SRPT when the unit price of response time is fixed and equivalent for all tasks; thereby it minimized the total response time of tasks. SCVPPT is sort of a generalized version of SRPT but is flexible. It allows a user to maintain or even improve the priority of a large task by accepting to set/pay a higher unit price of response time or even degrade the priority of a small non-urgent task by setting a sufficiently small unit price of response time. This is a dynamic feature that is absent in both FCFS and SRPT service disciplines. We recommended that the SCVPPT service discipline be implemented in any online speed-scaling algorithm that aims to minimize TCRTEC and considers tasks with heterogeneous unit prices of response time.

Finally, for $\alpha \neq 2$, simulation results showed that our speed scaling function (OSTSSF) outperformed the $\left\{\tilde{p}(\beta n)^{-1}, SRPT\right\}$ speed scaling function. We suggested improving this speed scaling function to $\left\{\tilde{p}\left(\frac{\beta n}{\alpha_j - 1}\right)^{-1}, SRPT\right\}$ in order to achieve better results as dictated by the TCRTEC/N performance metric. When the unit price of response time and energy is fixed for all tasks, both of these speed scaling functions have a worse case computational complexity of $O(1)$. Unlike $\left\{\tilde{p}(\beta n)^{-1}, SRPT\right\}$, OSTSSF is valid for the general case where the unit price of response time is heterogeneous in that it could vary per task (we did this to influence the priority of task execution as mentioned previously). Also, OSTSSF unlike $\left\{\tilde{p}(\beta n)^{-1}, SRPT\right\}$, considers the appropriate hardware and software processing constraints, making it more appealing in an application context.

7.1.4 Laws Of Supply & Demand and Energy Sources

We used the micro-economic laws of *Supply and Demand* to heuristically adjust the unit price of energy in order to extend battery life and also to induce load balancing effects. We achieved the first objective by synthesizing and simulating a single processor, multi-buffered algorithm (STMBAD). This algorithm has an operation mode where all tasks' unit cost of energy is heuristically affected by the device' remaining battery energy percentage in accordance with the micro-economic laws of demand and supply (EPARBEP mode).

Using numerical simulations, we showed that when the remaining battery energy percentage is factored (EPARBEP mode), the algorithm: performs slightly slower (mildly more slower when the battery is almost drained out), but consumes far less energy (in many cases more than 30%), can complete significantly more jobs (about 52% more jobs for homogenous deterministic tasks and more than 50% more jobs for heterogeneous tasks with Gaussian distributed computation volumes) and ultimately allowed the mobile computing device to last longer. The algorithm explicitly finds a globally optimum (minimum) solution for the cost of response time and energy consumption of all active tasks in the device' buffer. Like all the previously synthesized algorithms, the STMBAD algorithm handles the dynamic inclusion of heterogeneous tasks in real time. We suggested that the STMBAD algorithm be implemented in non-mobile work stations or computing devices that have a reliable and unlimited (but not free) supply of power by permanently setting the battery energy percentage parameter to one (UEP mode). Assuming we have *N* tasks queued up for processing, the algorithm has worse case computational complexities of O(*1*) and O(log(*N*)) under FCFS and SRPT service policies (respectively).

We extended all the previously constructed algorithms of this thesis to factor single or multiple energy sources through the (homogenous or heterogeneous) EPARBEP mode. This mode was defined to be the scenario when the energy price of a given schedule is heuristically adjusted by the remaining batter energy level/s in accordance with the laws of demand and supply. In contrast, we also maintained the UEP mode, which is the scenario where the price of energy is un adjusted.

We analytically showed that when processors have their independent energy sources, the EPARBEP mode induces a load balancing effect by dilating the energy cost terms (of a given schedule). The EPAREP mode strategically slows down speed scaling functions as long as the processing rates are unconstrained. This slowdown or attenuation in processing rate is inversely correlated with the amount of remaining battery energy. Therefore the EPAREP mode strategically saves the critical energy needed for a computing device to last longer on the go. The UEP mode always leads to optimum speed scaling functions and dispatchers but was found to not always be robust in the context of energy preservation. In regard to the dispatching of tasks on to processors, the UEP mode inevitably leads to a biased selection of efficient processors over inefficient processors in order to optimally minimize both energy and response time costs. Comparatively, the EPARBEP mode was analyzed to be suboptimal, but when each processing stream has its own independent energy supply, the EPARBEP mode was shown to induce a load balancing effect on dispatchers that counters the selection bias of the UEP mode. Furthermore, under the EPARBEP mode, this load balancing effect was also shown to be induced by the heterogeneous exponent of each processor's power function even if the processors shared a single energy source as long as they operated under unconstrained processing rates. A limitation that should not be overlooked is that the EPARBEP mode is valid when the energy consumption of tasks is negligible compared to the energy capacity of the battery.

7.2 Research Limitations

The following are the research limitations of this thesis.

7.2.1 Algorithmic Overhead

Generally, the algorithms make decisions on three major fronts. These decisions are fundamentally categorical. They are as follows.

- a dispatcher to assign tasks on to processors.
- a service discipline to dictate the order of servicing tasks within each processor.
- a speed scaling functions to specify the speed of each processor.

Each of these decisions incurs a computational penalty both in time and energy. We classify this type of computational overhead as the algorithmic overhead. All the single buffered (multiprocessor) algorithms do not have an algorithmic overhead with respect to service discipline due to single buffers. They also have mild algorithmic overheads for both their speed scaling functions and their dispatchers because those decisions were shown to be of constant computational complexity.

In the (multiprocessor) multi-buffer scenario, the computational complexity of the MMCVITPS dispatcher is indeed substantial. It was shown to have a worse case computational complexity of $O(n^2)$, where n is the number of tasks in each processor's multi-buffer. In the same scenario, the proposed service discipline (SCVPPT) and speed scaling function (OSTSSF) have worse case computational complexities of $O(\log n)$ and $O(n)$ respectively. The computational complexity of the service discipline can substantially be mitigated by using a non-preemptive service discipline such as First Come First Serve, but doing so was shown to achieve sub-optimal TCRTEC performance. The OSTSSF speed scaling function can be reduced to a constant computational complexity as long as the unit price of response time is homogenous. The drawback of doing so only impacts the flexibility of the user. We lightly suggested the implementation of the MMCVITPS dispatcher in ad-hoc hardware to guarantee performance, but the actual algorithmic overhead cost of doing so warrants further investigation. However, we are currently working on enhancing its computational complexity as a means to reduce its algorithmic overhead.

7.2.2 Overhead Energy

Like all of the closely related work in existing literature[48], the energy consumption during loading times (overhead energy) was assumed to be negligible. This was justified since the unit price of response time generally exceeds that of energy. In addition, these loading times can be mitigated by an improvement in technology, i.e. faster digital switching technologies. In practice, this can also be overcome by processing tasks with computation volumes that incur response times that are sufficiently larger than their loading times.

Arguably, It is possible that the relative price of energy could increase in the future. Nevertheless, factoring the overhead energy in the analysis will only affect the dispatchers of our algorithms because this overhead does not influence the computation of optimum processing speeds. In short, the speed scaling functions of our algorithms will not change, but the dispatchers will be slightly different. Consequently, this may open up the possibility of finding a more optimum service discipline (better than our proposed SCVPPT service discipline) if indeed the overhead loading times are not only heterogeneous but are also comparable to the response times of tasks.

7.2.3 Scope of Analysis

While constructing our algorithms, the boundary of analysis begins when tasks arrive, over the time interval in which the tasks are dispatched to processors, and terminates when all the tasks are fully processed. Beyond this boundary of analysis is to consider and stochastically model the arrival of tasks as a Poisson process [26]. Although we

[48] Related work in existing literature do not explicitly factor overhead loading times nor do they explicitly factor overhead energy. We factored overhead loading times but not their energy counterparts (overhead energy).

considered this stochastic model while simulating the relevant algorithms, we did not consider it in the formulation and derivation of the algorithms. Extending the boundary of analysis to encompass this stochastic dimension will not affect the single buffered scenario as long as no task rejections are observed, but in the (multiprocessor) multi-buffer scenario, this consideration may prove to be a suitable avenue to derive more efficient algorithms.

7.2.4 System Calibration

The performance of all the algorithms heavily depend on the calibration of two key parameters. These parameters (α_j and λ_j) are the power function parameters of each processor's power function. Before implementing the algorithms on actual hardware, we suggest running preliminary experiments to extract sufficiently accurate values of these parameters. We suggest more effort be invested in identifying a higher resolution of α_j over λ_j because in general, α_j influences the performance of the algorithms to a greater extent. With respect to the polynomial modeling of the power functions of processors, [6] states that this model is not always appropriate because of the interference of additive white Gaussian noise over communication channels that induce exponential power functions. We alleviate this effect in most of our algorithms by conservatively (infrequently) updating the speed of processors.

7.3 Future Research

We outline examples of research work that is centered around the use of dynamic speed scaling to minimize the cost of response time and energy consumption. Considering that {tasks, loading times, processors and unit price of response time} are all heterogeneous, some examples of future research are as follows.

- Consider migration in the single threaded, multi-buffered computing architecture. Migration has been solved for the deadline based scheduling problem [2, 7] but it is an open problem in the context of the *energy and flow time cost minimization* problem. This open problem is quite challenging given the assumptions of our model where almost all the parameters are heterogeneous. Although we briefly discussed how migration can be addressed in the single buffer computing architecture, a detailed analysis on this front could help extract a solution for the multi-buffered case.

- Study the multithreading or processor sharing computer architecture under our model and furthermore, to consider migration as well.
- Possibly use the Lloyd Max algorithm [48] to address the following question. How are we to implement dynamic speed scaling algorithms in those conventional processors that do not support dynamic speed scaling?
- Analyze task synthesizers which break tasks by assigning or distributing their computation volumes. It would be interesting to investigate how tasks should be distributed as a function of arrival times, occupancy of processing streams, number of processors, power function parameters of processors, traffic conditions, etc. with the goal of minimizing the total cost of response time and energy consumption.
- Address some or all of the research limitations that were previously discussed.

7.4 Closing Remarks

In this thesis we have synthesized, analyzed and simulated various parallel processing algorithms. These algorithms use dynamic speed scaling to schedule heterogeneous tasks onto heterogeneous processors in real time. The algorithms are compatible with homogenous processors as well as homogenous tasks. They are also compatible with none, single or multiple battery energy sources. These versatilities make the algorithms appealing for both mobile and stationary computing environments. The common objective among all the algorithms is to minimize the financial cost of response time and energy consumption. Attaching this financial cost to computing services is quite convenient for those that lease these services. Furthermore, the algorithms may prove to be valuable in the near future because experts in the computer architecture field have speculated on the advent of conventional heterogeneous computing.

Bibliography

[1] Albers, S. and Fujiwara, H., "*Energy-efficient algorithms for flow time minimization*", Proc. 23rd Annual Symposium on Theoretical Aspects of Computer Science (STACS), Springer LNCS 3884, pp. 622–633, 2006.

[2] Albers, S., Antoniadis, A. and Greiner, G., "*On Multi-Processor Speed Scaling with Migration*", SPAA, pp. 279–288, 2011.

[3] Albers, S., "*Energy-Efficient Algorithms*", Communications of the ACM, Vol. 53 No. 5, Pages 86-96, May, 2010.

[4] Albers, S., Muller, F. and Schmelzer, S., "*Speed Scaling on Parallel Processors*", SPAA, pp. 289-298, 2007.

[5] AMD. (2013) "AMD PowerNow™ Technology" [Online]. Available: http://www.amd.com/us/products/technologies/amd-powernow-technology/Pages/amd-powernow-technology.aspx.

[6] Andrew, L.L.H., Lin, M., Wierman, A., "*Optimality, fairness, and robustness in speed scaling designs*", SIGMETRICS '10 Proceedings of the ACM SIGMETRICS international conference on Measurement and modeling of computer systems, Pages 37-48, 2010.

[7] Angel, E., Bampis, E., Kacem, F. and Letsios, D., "*Speed Scaling on Parallel Processors with Migration**", Euro-Par, pp.128-140, 2012.

[8] Apple. (2013) "iPhone" [Online].
Available: http://www.apple.com/asia/iphone/specs.html.

[9] Asanović, K., et al., "*The Landscape of Parallel Computing Research: A View from Berkeley*", EECS Department, University of California, Berkeley, pp.22, Tech. Rep. UCB/EECS-2006-183, December 2006.

[10] Avrahami, N. and Azar, Y., "*Minimizing total flow time and total completion time with immediate dispatching*", SPAA, pp. 11–18, 2003.

[11] Bansal, N., Chan, H.-L., Lam, T.-W., Lee, K.-L., "*Scheduling for speed bounded processors*", In Proceedings of the 35th International Colloquium on Automata, Languages and Programming, Springer LNCS 5125, 409–420, 2008.

[12] Bansal, N., Kimbrel, T. and Pruhs, K., "*Dynamic speed scaling to manage energy and temperature*", Proc. 45th Annual IEEE Symposium on Foundations of Computer Science, pp. 520–529, 2004.

[13] Bansal, N., Kimbrel, T. and Pruhs, K., "*Speed scaling to manage energy and temperature*", J. ACM 54 (1) , pp. 1–39, 2007.

[14] Bansal, N., Pruhs, K., Stein, C., "*Speed scaling for weighted flow time*", In SIAM Journal on Computing 1294-1308*,* 2009.

[15] Bansal, N., Pruhs, K., Stein, C., "*Speed scaling for weighted flow time*", In: Proc. of 18th Annual ACM-SIAM Symp. on Discrete Algorithms (SODA'07), pp. 805–813, 2007.

[16] Barroso, *L.A.,* "*The price of performance*"*,* ACM Queue 3 (2005).

[17] Baumol, W., *Microeconomics: principles and policy*, 1st Canadia Edition, Toronto, Nelson Education, 2009.

[18] Bower, F.A., Sorin, D.J. and Cox, L.P., "*The impact of dynamically heterogeneous multicore processors on thread scheduling*", Micro, IEEE, 28(3), pp. 17 –25, 2008.

[19] Brooks, D.M., Bose, P., Schuster, S.E., Jacobson, H., Kudva, P.N., Buyuktosunoglu, A., Wellman, J.-D., Zyuban, V., Gupta, M., Cook, P.W.,"*Power-aware microarchitecture: design and modeling challenges for next-generation microprocessors*", IEEE MICRO 20(6), pp. 26–44, 2000.

[20] Bunde, D.P., "*Power-aware scheduling for makespan and flow*", SPAA, pp. 190–196, 2006.

[21] Das, S. *Fundamentals of heat and mass transfer*, Oxford, U.K. : Alpha Science International, 2010.

[22] Dautovic, S., Malbasa, V., "*Dynamic Power Management of a System With a Two-Priority Request Queue Using Probabilistic-Model Checking*", In Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on Feb 2008.

[23] Edmonds, J. Pruhs, K., "*Scalably scheduling processes with arbitrary speedup curves*", In ACM-SIAM Symposiumon Discrete Algorithms, pages 685–692, 2009.

[24] Gradshteyn, I. S. and Ryzhik, I. M., *Hessian Determinants,* §14.314 in Tables of Integrals, Series, and Products, 6th ed. San Diego, CA: Academic Press, pp. 1069, 2000.

[25] Greiner, G. , Nonner, T. and Souza, A., "*The bell is ringing in speed-scaled multiprocessor scheduling*", SPAA, pp. 11-18, 2009.

[26] Grimmett, G. and Stirzaker, D., *Probability and Random Processes*, 3rd ed. Oxford University Press, Jul 2010.

[27] Gupta, A., Im, S., Krishnaswamy, R., Moseley, B. and Pruhs, K., "*Scheduling heterogeneous processors isn't as easy as you think*", Association for Computing Machinery. Proceeding of the ACM-SIAM Symposium on Discrete Algorithms: 1242-1253. Philadelphia: Society for Industrial and Applied Mathematics. (2012).

[28] Gupta, A., Im, S., Krishnaswamy, R., Moseley, B. and Pruhs, K., "*Scheduling heterogeneous processors isn't as easy as you think*", Proc. of the Twenty-Third Annual ACM-SIAM Symp. on Discrete Algorithms pp. 1242-1253. 2011.

[29] Hwang, C. H., Wu, H., "*A predictive system shutdown method for energy saving of event-driven computation*", in ACM Transactions on Design Automation of Electronic Systems (TODAES) , Volume 5 Issue 2, Pages 226 - 241, April 2000.

[30] Hydro One. (2013, May). "BUILDING YOUR BILL: prices & rates" [Online].Available:http://www.hydroone.com/RegulatoryAffairs/RatesPrices/Pages/Default.aspx Access on 2013, June 19.

[31] IBM. (2013) "Power Systems Energy Management" [Online]. Available: http://www-03.ibm.com/systems/power/software/energy/about.html.

[32] Intel. (2013) "Enhanced Intel SpeedStep® Tech. - How To Document" [Online]. Available:http://www.intel.com/cd/channel/reseller/asmo-na/eng/203838.htm.

[33] Irani, S., Shukla, S. and Gupta, R., "*Algorithms for power savings*", Proc. 14th Annual ACM-SIAM Symposium on Discrete Algorithms, pp.37–46, 2003.

[34] Irani, S., Shukla, S.K., Gupta, R.K., "*Online strategies for dynamic power management in systems with multiple power-saving states*", ACM Trans. Embedded Comput. Syst. 2 325–346, 2003.

[35] Irani, S., Singh, G., Shukla, S.K., Gupta, R.K., "*An overview of the competitive and adversarial approaches to designing dynamic power management strategies*", IEEE Trans. VLSI Syst. 13 (2005), 1349–1361.

[36] Jain, T.R. *Microeconomics and Basic Mathematics*. New Delhi: VK Publications. pp. 24, 2006–07.

[37] Karlin, A.R., Manasse, M.S., McGeoch, L.A, Owicki, S.S., "*Competitive randomized algorithms for nonuniform problems*", Algorithmica 11, 542–571, 1994.

[38] Kaxiras, S. and Martonosi, M., *Computer Architecture Techniques for Power-Efficiency*, Morgan and Claypool, 2008.

[39] Khogali, R. and Das, O., *"Cost Minimization for Scheduling Parallel, Single-threaded, Heterogeneous, Speed-scalable Processors*", The 19th IEEE International Conference on Parallel and Distributed Systems (ICPADS "13), Seoul, Korea, Pg. 265-274, Dec 18, 2013.

[40] Khogali, R. and Das, O., "*Extending Battery Life of a Multi-buffered, Single-threaded Processor in a Mobile Computing Device*", The Ninth IEEE Xplore International Workshop on Scheduling and Resource Management for Parallel and Distributed Systems (SRMPDS '13) in conjunction with the 42nd IEEE International Conference on Parallel Processing (ICPP '13), Lyon, France, Oct 1, 2013, (In press).

[41] Khogali, R., Das, O., and Raahemifar, K., "*Mobile Parallel Computing Algorithms for Single-Buffered, Speed-Scalable Processors*", 12th IEEE International Conference on Trust, Security and Privacy in Computing and Communications (TRUSTCOM), Melbourne, Australia, Pg.1832 - 1839, 16-18 July 2013.

[42] Koufaty et al., "*Bias scheduling in heterogeneous multi-core architectures*", EuroSys 2010.

[43] Kumar, K. and Lu,Y. "*Cloud Computing For Mobile Users: Can Offloading Computation Save Energy*?", in IEEE Xplore, pp.52, 2010.

[44] Lam, T.W., Lee, L.-K., To, I.K.-K., Wong, P.W.H., "*Competitive non-migratory scheduling for flow time and energy*", In: Proc. of the 20th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA'08), 256–264, 2008.

[45] Lam, T.-W., Lee, L.-K., To, I.K.-K., Wong, P.W.H., "*Energy efficient deadline scheduling in two processor systems*", In Proceedings of the 18th International Symposium on Algorithms and Computation, Springer LNCS 4835, 476–487, 2007.

[46] Li, M., Yao, A.C., Yao, F.F., "*Discrete and continuous min-energy schedules for variable voltage processors*", In Proceedings of the National Academy of Sciences USA 103 3983–3987, 2006.

[47] Li, M., Yao, F.F., "An efficient algorithm for computing optimal discrete voltage schedules", *SIAM J. Comput. 35*, 658–671, 2005.

[48] Lloyd, S., "*Least squares quantization in PCM*", unpublished memo., Bell Lab., 1957, Information Theory, IEEE Transactions on (Volume:28 , Issue: 2), pp. 129-137, March, 1982.

[49] Merchant, A., et al., "*Analysis of a Control Mechanism for a Variable Speed Processor*" in IEEE Transactions. Comput. , pp.793-801, 1996.

[50] Microsoft. (2013) "Desktop PC Energy Savings for Enterprises" [Online]. Available: http://www.microsoft.com/environment/our-commitment/our-research.aspx.

[51] Microsoft. (2013) "Power Management and ACPI - Architecture and Driver Support" [Online]. Available: http://msdn.microsoft.com/en-us/windows/hardware/gg463220.aspx.

[52] Min, R., et al ., "*Energy-centric enabling technologies for wireless sensor networks*", IEEE Trans. Wireless Commun., vol. 9, no. 4, pp. 28–39, Aug. 2002.

[53] Morad, T.Y., Weiser U.C., Kolodny,A., Valero, M., Ayguadé., E.,"*Performance, power efficiency and scalability of asymmetric cluster chip multiprocessors*", IEEE Comput. Archit, Jan 2006.

[54] Ontario Ministry of Labour. (2013, May), "Minimum Wage" [Online]. Available:http://www.labour.gov.on.ca/english/es/pubs/guide/minwage.php. Access on 2013, June 19.

[55] Parkkila, J. and Porras, J., "*Improving Battery Life and Performance of Mobile Devices with Cyber Foraging*", in IEEE, pp.91-95, 2011.

[56] Pruhs, K., Uthaisombut, P. and Woeginger, G. "*Getting the best response for your erg*" Proc. 9th Scandinavian Workshop on Algorithm Theory (SWAT), Springer LNCS 3111, pp.15–25, 2004.

[57] Pruhs, K., Sgall, J. and Torng, E., "*Online scheduling*", In J. Leung, editor, Handbook of Scheduling: Algorithms, Models and Performance Analysis, pp. 15-1–15-41. CRC Press, 2004.

[58] Pruhs, K., Uthaisombut, P., Woeginger, G.J., "*Getting the best response for your erg*", ACM Trans. Algorithms 4, 2008.

[59] Pruhs, K., van Stee, R. and Uthaisombut, P., "*Speed scaling of tasks with precedence constraints*", Theory Comput. Syst. 43 (1), pp. 67–80, 2008.

[60] Sleator, D.D., Tarjan, R.E., "*Amortized efficiency of list update and paging rules*", Comm. ACM 28, 202–208, 1985.

[61] Sniedovich, M., *Dynamic Programming Foundations and Principles*, Second Edition, CRC Press, 2010.

[62] Vaknin,S. (2009, June 18). "Nokia powering up self-charging cell phone", CNET [Online]. Available: http://news.cnet.com/8301-17938_105-10267006-1.html.

[63] Wierman, A., Andrew, L. L. H., and Tang, A., "*Power-aware speed scaling in processor sharing systems: Optimality and robustness*" Performance Evaluation, 69 (12), pg. 601-622, 2012.

[64] Wikipidea. (2013) "Sleep mode" [Online]. Available: http://en.wikipedia.org/wiki/Sleep_mode.

[65] Wikipidea. (2013, Feb 1). “PlayStation Vita” [Online]. Available: http://en.wikipedia.org/wiki/PlayStation_Vita. Access on 2013, Mar 10.

[66] Williams, M. (2009, Feb 12), LG, “Samsung Develop Solar-powered Cell Phones PCWorld” [Online]. Available: http://www.pcworld.com/article/159507/article.html.

[67] XTG Technology. (2013, Feb 1). “xtgtechnology Products” [Online]. Available: http://www.xtgtechnology.com/Products_c_11-2-0.html.

[68] Yao, F., Demers, A. and Shenker, S., “*A scheduling model for reduced CPU energy*”, Proc. 36th Annual Symposium on Foundations of Computer Science, pp.374–382, 1995.

[69] Yuan, L., and Qu, G.,“*Analysis of energy reduction on dynamic voltage scaling-enabled systems*”, IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst. 24 (12), pp. 1827–1837, 2005.

[70] Bansal, N., Chan, H.-L., Pruhs, K. “*Speed scaling with an arbitrary power function*" In Proceedings of the 20th ACM-SIAM Symposium on Discrete Algorithm, 2009.

[71] Chan, H., et al., “*Optimizing Throughput and Energy in Online Deadline Scheduling*”, ACM Transactions on Algorithms, Vol. 6, No. 1, Article 10, Dec 2009.

Appendices

Appendix I is relevant to Chapter 3 (Section 3.6) of the thesis.

Appendices II and III are relevant to Chapter 6 (Sections 6.6 and 6.7) of the thesis.

Appendix I: Initial Modeling of A Task's Energy & Power Consumption

Initially, we were not formally aware of dynamic speed scaling, but we were still able to reasonably model the energy and power consumption of a task. This was done using a statement from a paper in the Cloud Computing literature. We were able to deduce $\alpha = 3$ (α being the exponent of a CMOS processor's power function). This deduction is consistent with the assumptions made in current literature pertaining to the Dynamic Speed Scaling field. We show the deduction as follows.

For a task: $T_k \in T$, let ε_k be the task's expected energy consumption in Joules. According to Kumar and Lu [43], we are to: "*Execute programs slowly. When a processor's clock speed doubles, the power consumption nearly octuples. If the clock speed is reduced by half, the execution time doubles, but only one quarter of the energy is consumed.*"

We assert that the energy consumption of a task is directly proportional to the processing rate to a non-negative degree and is directly proportional to the execution time. Let $\propto$ denote the relationship of direct proportionality.

$\varepsilon_k \propto (P_k)^\alpha$ where $\alpha \in \Re^+$, $\varepsilon_k \propto t_k$ and $t_k \propto \frac{1}{P_k}$

Using the abovementioned statement of [43] we deduce $\alpha = 3$ and derive the following equations.

$$\varepsilon_k = \lambda_j (P_k)^3 t_k \tag{A.1}$$

$$t_k = \frac{B_k}{P_k} \tag{A.2}$$

B_k relates t_k to P_k, and is actually the task's remaining computation volume in base instructions (n).

We define λ_j, measured in $(J.S^2 / n^3)$, to be the processor energy inefficiency coefficient. We know that power consumption is the rate of energy consumption. Let us define the expected power consumption of a task as Pow_k.

$$Pow_k = \varepsilon_k / t_k = \lambda_j (P_k)^3 \ (Watts) \qquad \text{(A.3)}$$

It is straight forward to verify the assertions of (A.1), (A.2) and (A.3) using the above mentioned statement of [43]. Using (A.1) and (A.2), we further deduce:

$$\varepsilon_k = \lambda_j B_k (P_k)^2 \quad \textit{(Joules)} \qquad \text{(A.4)}$$

After further studying dynamic speed scaling, we generalized $\alpha \in (1,3]$. Furthermore we classified it as a heterogenic parameter of a given jth processor (α_j),where $\alpha_j \in (1,3]$.

Appendix II: Calibrating the Ratio of Time and Energy Prices under EPARBEP Mode

Let us calibrate the ratio of unit prices ($u_{t,k} / u_\varepsilon$) that happen to correlate with processing rate and power consumption of a given task T_k . Recall in Chapter 4, for a given $U_k = (u_\varepsilon, u_{t,k})$, associated with the task T_k, we want a one to one correspondence with $P*_k$ or $P_{s,j}$ which introduces the issue of calibration.

$$P_{Max,j} \geq P_{s,j} = P*_k = \left(\frac{\varepsilon_{\%,j} u_{t,k}}{(\alpha_j - 1) u_\varepsilon \lambda_j} \right)^{\frac{1}{\alpha_j}} \geq p_{\mu,k}$$

$$\Rightarrow \left\{ \left(P_{Max,j}\right)^{\alpha_j} \geq \frac{\varepsilon_{\%,j} u_{t,k}}{(\alpha_j - 1) u_\varepsilon \lambda_j} \geq \left(p_{\mu,k}\right)^{\alpha_j} \right\}$$

$$\Rightarrow \left\{ \frac{(\alpha_j - 1)}{\varepsilon_{\%,j}} \lambda_j \left(P_{Max,j}\right)^{\alpha_j} \geq \frac{u_{t,k}}{u_\varepsilon} \geq \frac{(\alpha_j - 1)}{\varepsilon_{\%,j}} \lambda_j \left(p_{\mu,k}\right)^{\alpha_j} \right\} \qquad \text{(A.5)}$$

Relation (A.5) is consistent with minimum and maximum processing constraints.

As battery[49] energy depletes (decreasing $\varepsilon_{\%,j}$), the calibration region in Fig. A1 uniformly shifts to the right increasing the economy region (or decreasing the economy region), and if the battery recharges (increasing $\varepsilon_{\%,j}$), the calibration region uniformly shifts to the left decreasing the economy region (or increasing the performance region).

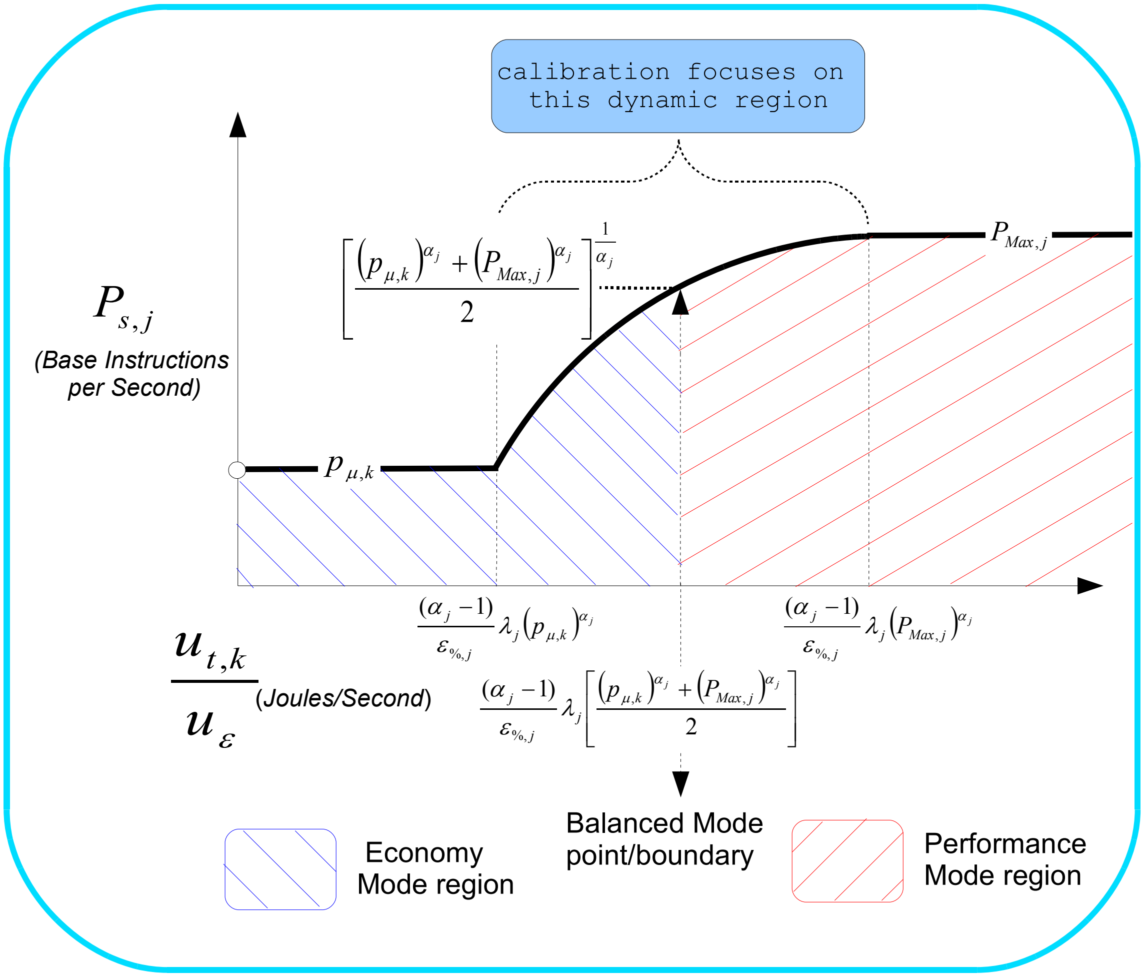

calibration focuses on this dynamic region
$\left[\frac{(p_{\mu,k})^{\alpha_j}+(P_{Max,j})^{\alpha_j}}{2}\right]^{\frac{1}{\alpha_j}}$
$P_{Max,j}$
$P_{s,j}$
(Base Instructions per Second)
$p_{\mu,k}$
$\frac{(\alpha_j-1)}{\varepsilon_{\%,j}}\lambda_j(p_{\mu,k})^{\alpha_j}$
$\frac{(\alpha_j-1)}{\varepsilon_{\%,j}}\lambda_j(P_{Max,j})^{\alpha_j}$
$\frac{u_{t,k}}{u_{\varepsilon}}$
(Joules/Second)
$\frac{(\alpha_j-1)}{\varepsilon_{\%,j}}\lambda_j\left[\frac{(p_{\mu,k})^{\alpha_j}+(P_{Max,j})^{\alpha_j}}{2}\right]$
Economy Mode region
Balanced Mode point/boundary
Performance Mode region

Fig. A1: A task's operating mode and optimum processing rate as a function of user-defined (time/energy) unit prices under EPARBEP mode

[49] jth processing stream's battery.

Fig. A.1 illustrates the robust processing rate of a task as a function of the ratio of time and energy prices. For a given task, if a user wants the task's mode of operation to escape the economy region, he/she should do any or all of the following.

- Recharge battery/batteries (thereby increasing the remaining energy percentage/s).
- Be willing to spend more on time (increase $u_{t,k}$).
- Be willing to spend less on energy[50] (decrease u_{ε}).
- Accept a higher time cost relative to energy (increase $u_{t,k}/u_{\varepsilon}$).

Likewise, if a user wants the task's mode of operation to escape the performance region, he/she should use more depleted batteries, be willing to spend less on time (decrease $u_{t,k}$) or spend more on energy (increase u_{ε}) or rather accept a lower time cost relative to energy (decrease $u_{t,k}/u_{\varepsilon}$). If an advanced user has a deep understanding of $u_{t,k}$ or u_{ε}, he or she would specify it, and allow the SBDPP algorithm to operate on the appropriate mode. Alternatively, a user may want to know the actual extent of a task's mode of operation, and may want to make a decision based on that rather than just the actual values of $u_{t,k}$ or u_{ε}. To do so in a consistent fashion, we need to use a metric that is a linear function of $(u_{t,k}/u_{\varepsilon})$. Referring to Fig. A.1, in order to achieve a linear calibration of the task's processing rate as a function of $(u_{t,k}/u_{\varepsilon})$, we first identify each constant range (flat line portions of the economy and performance mode regions) in the graph and map each of these regions to a point value. We also need to *linearize* the curved portion of the figure (calibration region) via a non-linear transformation.

[50] If the price of energy is determined by the OS based on time of day, a decrease in energy price can result from a transition between peak hours and off-peak hours.

Appendix III: Determining a Task's Mode of Operation with EPARBEP mode

As mentioned in Chapter 4, in order to consistently determine a task's mode of operation we linearly calibrate the ratio of the user defined prices $\left(u_{t,k}/u_{\varepsilon}\right)$ by non-linearly transforming the task's processing rate. We achieve this by using the task's power consumption instead of the task's processing rate.

$\left(u_{t,k}/u_{\varepsilon}\right)$ is defined as the ratio of unit time price *($/Second*) and unit energy price *($/Joule*). It is convenient that the resulting dimension of $\left(u_{t,k}/u_{\varepsilon}\right)$ is indeed *Joule/Second* or *Watt*. According to equation (3.2), we see that $\left(u_{t,k}/u_{\varepsilon}\right)$ is the power consumption of a task multiplied by a factor of $\frac{(\alpha_j - 1)}{\varepsilon_{\%,j}}$.

The modified definition of the *(user specified) power sensitivity factor* (S_j) under EPABEP mode is as follows.

$$\text{Let } \frac{u_{t,k}}{u_{\varepsilon}} = \frac{(\alpha_j - 1)}{\varepsilon_{\%,j}} \lambda_j \left[\left(p_{\mu,k}\right)^{\alpha_j} + \left(\left(P_{Max,j}\right)^{\alpha_j} - \left(p_{\mu,k}\right)^{\alpha_j} \right) S_j \right] \qquad \text{(A.6)}$$

where $S_j \in [0,1]$.

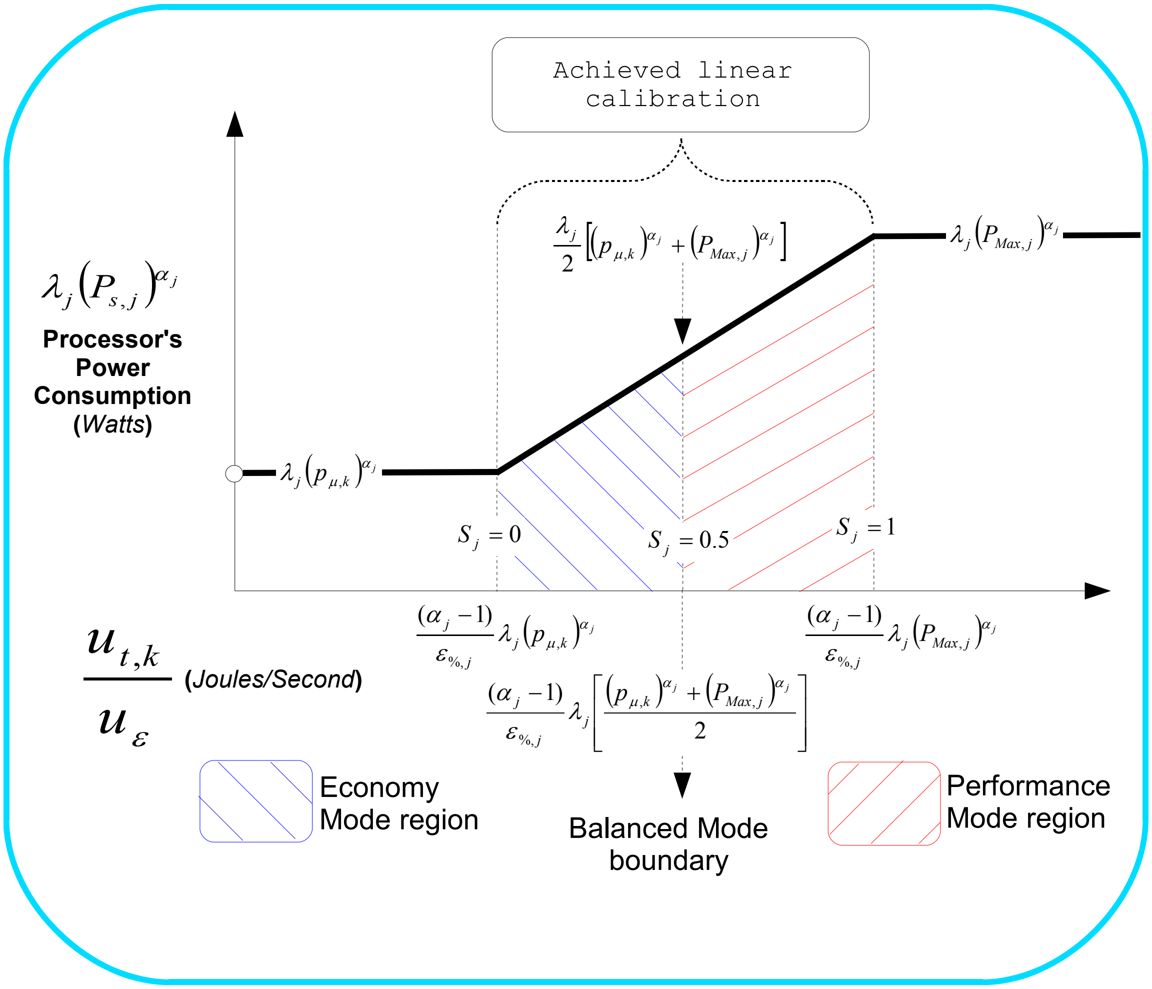

Achieved linear calibration
$\frac{\lambda_j}{2}\left[(p_{\mu,k})^{\alpha_j}+(P_{Max,j})^{\alpha_j}\right]$
$\lambda_j(P_{Max,j})^{\alpha_j}$
$\lambda_j(P_{s,j})^{\alpha_j}$
Processor's Power Consumption (Watts)
$\lambda_j(p_{\mu,k})^{\alpha_j}$
$S_j=0$
$S_j=0.5$
$S_j=1$
$\frac{u_{t,k}}{u_\varepsilon}$ (Joules/Second)
$\frac{(\alpha_j-1)}{\varepsilon_{\%,j}}\lambda_j(p_{\mu,k})^{\alpha_j}$
$\frac{(\alpha_j-1)}{\varepsilon_{\%,j}}\lambda_j(P_{Max,j})^{\alpha_j}$
$\frac{(\alpha_j-1)}{\varepsilon_{\%,j}}\lambda_j\left[\frac{(p_{\mu,k})^{\alpha_j}+(P_{Max,j})^{\alpha_j}}{2}\right]$
Economy Mode region
Balanced Mode boundary
Performance Mode region

Fig. A.2: Illustrating the linear calibration of a task's operation mode by utilizing the processor's power consumption during execution under EPARBEP mode

In Fig. A.1, we see that a task's robust processing rate as a function of $(u_{t,k}/u_\varepsilon)$ does not linearly determine the operation mode of a task. In Fig. A.2, a task's power consumption as a function of $(u_{t,k}/u_\varepsilon)$ does indeed linearly determine the operation mode of a task.

This works because a task's power consumption is a non-linear transformation of its processing rate. In extension, observe that in Figs. A.1 and A.2, the balanced mode of a task's execution is identified by average of its minimum and maximum power consumption and not the mean of its minimum and maximum processing rate.

As mentioned in Chapter 4, S_j is used to linearly parameterize a task's power consumption over the calibration region[51] (spanned by $(u_{t,k}/u_\varepsilon)$). S_j informs us on the actual extent of power consumption while executing a task under software and hardware processing constraints, and it also linearly determines a task's mode of operation.

Using (A.2), it is quite convenient that the robust[52] processing rate that factors processing constraints reduces elegantly to:

$$P_{s,j} = \left(\frac{\varepsilon_{\%,j} u_{t,k}}{(\alpha_j - 1) u_\varepsilon \lambda_j} \right)^{\frac{1}{\alpha_j}} = \left[\left(p_{\mu,k}\right)^{\alpha_j} + \left(\left(P_{Max,j}\right)^{\alpha_j} - \left(p_{\mu,k}\right)^{\alpha_j} \right) S_j \right]^{\frac{1}{\alpha_j}}, \text{ for } S_j \in [0,1].$$

When $S_j \in [0,1]$, we get $P_{Max,j} \geq P_{s,j} = \left(\frac{\varepsilon_{\%,j} u_{t,k}}{(\alpha_j - 1) u_\varepsilon \lambda_j} \right)^{\frac{1}{\alpha_j}} \geq p_{\mu,k}$ (as desired).

[51] In equation (A.6) and Fig. A.1 and A.2, we **redefined** S_j, the *user specified power sensitivity factor* under EPAREP mode.

[52] The processing rate becomes optimum when the remaining energy percentage is equal to one, i.e. $\varepsilon_{\%,j} = 1$.