Imperial College London
Department of Computing

Improving Trade-Offs between Multiple Metrics in Parallel Queueing Systems

Tommi Topi Kalevi Pesu

Submitted in part fulfilment of the requirements for the degree of
Doctor of Philosophy in Computing of Imperial College London
and the Diploma of Imperial College London

Abstract

Parallel Queueing Networks can be used to model and optimise systems in many different environments, such as distributed storage facilities, multi-core processors, RAID systems, supply chains and public services such as hospitals. The various stakeholders involved with the systems will often measure the performance of such systems using a wide range of metrics that often conflict with each other. Metrics of interest include task response time, subtask dispersion and energy consumption. Subtask dispersion is a recent metric, which is the difference in time of the first and last subtask to complete.

The trade-offs between metrics can be controlled in various ways. Within this context, this thesis makes four primary contributions, the first of which of is to compare various delay-padding techniques in split–merge and fork–join parallel queueing models, with respect to task response time and subtask dispersion. We compare seven techniques from the literature, including some of our own, against each other across multiple case studies, in order to determine their strengths and weaknesses. Our results indicate that dynamic delay padding in a fork–join setting is currently the most promising technique for improving the trade-off between subtask dispersion and task response time.

Our second contribution is to extend existing delay-padding techniques to work in a class of multi-layered parallel queueing environments, specifically Hidden Stochastic PERT Networks. We develop a technique which uses a state-of-the-art genetic algorithm to improve the trade-off between task service time and subtask dispersion. The method is able to robustly control subtask dispersion and task response time in a case study network.

The third contribution is a systematic survey, which investigates alternative approaches for improving the performance of parallel queueing systems. Promising candidates include service restart and service replication.

The final contribution is to combine multiple techniques: delay padding, state-dependent service strategies, service restart and service replication to provide greatly improved performance in terms of a three-way optimisation involving task response time, subtask dispersion and energy consumption. In the best case we managed to reduce the cost function by over 90% compared to an unoptimised system.

Copyright

© The copyright of this thesis rests with the author and is made available under a Creative Commons Attribution Non-Commercial No Derivatives licence. Researchers are free to copy, distribute or transmit the thesis on the condition that they attribute it, that they do not use it for commercial purposes and that they do not alter, transform or build upon it. For any reuse or redistribution, researchers must make clear to others the licence terms of this work

Acknowledgements

I would like to express my gratitude to following people:

- Will
- Aad, Jani, Katinka
- Mom and Dad
- Family and friends
- NHS
- Gambit Research

Contents

Abstract **3**

Copyright **5**

Acknowledgements **7**

1 Introduction **24**

1.1 Motivation 24

1.1.1 Preventing Front-Running on the Stock Market 26

1.1.2 Fairness in Online Gaming 27

1.1.3 Delivery Processing at Online Retailers 28

1.2 Aims and Objectives 29

1.3 Contributions 31

1.3.1 Reducing Subtask Dispersion in Hidden Stochastic PERT Networks . . . 31

1.3.2 Investigating Restart Mechanisms to Improve System Efficiency 32

1.3.3 Computing Probability Density Function of Dynamically Padded Split–Merge System 32

1.3.4 Trade-Off of Multiple Metrics in Split–Merge Queueing Systems 33

1.4 Statement of Originality 33

1.5 Publications 33

1.6 Thesis Outline 35

2 Background **37**

2.1 Queueing Models 38

2.1.1 M/M/1 Queue 39

2.1.2 M/G/1 Queue 39

2.2 Parallel Queueing Models 40

2.2.1 Split–Merge System 40

2.2.2 Fork–Join System 41

2.2.3 Stochastic PERT Networks 42

2.3 Performance Metrics 44

2.3.1 Subtask Dispersion 44

2.3.2 Task Response Time 46

2.3.3 Trade-Off Metric 47

2.4 Optimisation Algorithms 48

2.4.1 Newton's Method 48

2.4.2 Nelder–Mead Method 49

2.4.3 CMA-ES 50

2.4.4 Bayesian Optimisation 50

2.5 Summary of Related Research 51

2.5.1 Task Response Time 51

2.5.2 Subtask Dispersion 52

2.5.3 Energy Consumption Optimisation 53

2.5.4 State-dependent Service in Queueing Systems 54

2.6 Trends in Restart Related Research in the Past Decade (2007 - 2017) 55

2.6.1 Survey of Research of the Past Decade 56

2.6.2 Main Points of Review 60

3 Comparison of Existing Techniques 61

3.1 Minimising Subtask Dispersion in Split–Merge Queueing Systems with Static Delays 62

3.1.1 Introduction 62

3.1.2 Calculation of Subtask Delay Vector 62

3.2 Minimising Trade-Off in Split–Merge Systems 63

3.2.1 Introduction 63

3.2.2 Calculation of Subtask Delay Vector 64

3.3 Minimising Subtask Dispersion in Fork–Join Queueing Systems with Dynamic Delays 65

3.3.1 Introduction 65

3.3.2 Computation of Subtask Delay Vectors 66

3.4 Dynamic Subtask Dispersion Reduction in Split–Merge Queueing Systems . . . 68

3.4.1 Introduction 68

3.4.2 Calculating Subtask Dispersion in Dynamic Split–Merge Systems 68

3.4.3 Fork–Join Extension for Exponentially-Distributed Service Times 70

3.5 Numerical Comparison of Techniques 71

3.5.1 Analysed Techniques 71

3.5.2 Case Study 1 72

3.5.3 Case Study 2 72

3.5.4 Discussion 73

4 Optimising Hidden Stochastic PERT Networks 75

4.1 Introduction 75

4.2 Preliminaries 77

4.2.1 Task Response Time 77

4.2.2 Subtask Dispersion 78

4.2.3 Trade-off Metric 78

4.3 Method to Optimise PERT Networks 78

4.3.1 Optimisation Procedure 79

4.3.2 Validation of Optimisation Procedure 80

4.4 Results 81

4.5 Conclusion 85

5 Models for Restart and Replication 90

5.1 Introduction 90

5.2 Metrics to Assess the Benefits of Restarts 92

5.2.1 Hazard Rate 93

5.2.2 Moments of Service Time 93

5.2.3 Probability of Meeting Deadlines 94

5.2.4 Example Distributions 94

5.3 Expressions for Completion Time under Restarts 97

5.3.1 Moments of Completion Time: Unbounded Number of Restarts 97

5.3.2 Moments of Completion Time: Finite Number of Restarts, Restart with Non-Identical Intervals 101

5.3.3 Expressions for Probability of Meeting Deadlines 103

5.4 Optimization of Restart Strategies 103

5.4.1 Optimal Restart Times for Expected Service Time 104

5.4.2 Optimal Restart Times for Higher Moments 105

5.4.3 Using Restart to Optimise Deadline Probability 107

5.5 Replication Strategies 109

5.5.1 n-fold Replication 110

5.5.2 Hedged Requests 111

5.5.3 Tied Requests 111

5.6 Conclusion 111

6 Three-way Optimisation in Split–Merge Systems 113

6.1 Introduction 113

6.2 Trade–offs Using Dynamic Subtask Dispersion 115

6.2.1 Dynamic Subtask Dispersion 117

6.2.2 Using Dynamic delays to Optimise Task Response Time and Subtask Dispersion 118

6.3 State-Dependent Delay Vectors 120

6.3.1 Background 121

6.3.2 The Algorithm 121

6.4 Energy Metric and Service Time Manipulations 123

6.4.1 Server Replication 123

6.4.2 Service Restart 124

6.4.3 Energy Metric 126

6.4.4 Trade-off Metric 126

6.4.5 Problems defining the Task Response Time, Subtask Dispersion and Energy Consumption Trade-Off Metric 127

6.5 Experimental Setup 129

6.5.1 Exploring Response Time and Subtask Dispersion 129

6.5.2 Computational Resources 129

6.5.3 Exploring Response Time, Subtask Dispersion and Energy 130

6.6 Results 132

6.6.1 Subtask Dispersion vs. Response Time trade-off 132

6.6.2 Response Time vs. Subtask Dispersion vs. Energy Trade-Off 135

6.7 Conclusion 136

7 Conclusion **138**

7.1 Key Highlights 138

7.2 Future Work 139

7.2.1 Optimising Trade-Offs in Fork-Join Systems 140

7.2.2 Trade-Offs in Stochastic PERT Networks 140

7.2.3 Making Service Restart More Tolerant to Modelling Error 140

7.3 Concluding Remarks 141

Bibliography **141**

Appendices **158**

A Probability Distributions **159**

A.1 Exponential Distribution 159

A.2 Hyper-Exponential Distribution 160

A.3 Erlang Distribution 160

A.4 Hypo-Exponential Distribution 161

A.5 Pareto Distribution 161

A.6 Normal distribution 162

A.7 Folded Normal Distribution 163

A.8 Log-normal distribution 163

A.9 Uniform Distribution 164

A.10 Power Distribution 165

List of Tables

2.1 Relevant literature since 2007, categorized. 55

2.2 Further categorization of papers in 'Modelling & Analysis' category of Table 2.1 56

5.1 Optimal restart intervals for finite and unbounded number of restarts. 105

5.2 Equihazard restart intervals and associated probability of meeting the deadline ($d = 0.7, \mu = -2.3, \sigma = 0.97$). 109

List of Figures

1.1 Microwave pathways between southern England and Germany that are used by various high frequency trading companies [6] 27

1.2 Example of Amazon delivery preference option 29

2.1 A split–merge system Queueing System, credit: [125], (CC BY-SA 3.0). 41

2.2 A fork–join system credit: [125], (CC BY-SA 3.0). 42

2.3 An example PERT Network with two subtasks, which complete 9 and 10 43

2.4 Estimating the optimisation space by Bayesian Optimisation [120], (CC BSD 3). 50

3.1 Dynamic subtask dispersion in a two server example with service time distributions being: $\text{Exp}(\lambda = 1), \text{Exp}(\lambda = 2)$ 70

3.2 Squashing subtasks in a two server example with service distributions being: $\text{Exp}(\lambda = 1), \text{Exp}(\lambda = 2)$ 70

4.1 PERT network used to analyse results 83

4.2 Subtask dispersion for delays $(0, x, y)$ 86

4.3 Subtask dispersion for delays $(x, 0, y)$ 86

4.4 Subtask dispersion for delays $(x, y, 0)$ 86

4.5 Task response time for delays $(0, x, y)$ 87

4.6 Task response time for delays $(x, 0, y)$ 87

4.7 Task response time for delays $(x, y, 0)$ 87

4.8 Trade-off ($\alpha = 0.5$) for delays $(0, x, y)$ 88

4.9 Trade-off ($\alpha = 0.5$) for delays $(x, 0, y)$ 88

4.10 Trade-off ($\alpha = 0.5$) for delays $(x, y, 0)$ 88

4.11 Influence of α on optimisation metrics 89

5.1 The probability density function of the mixed hyper/hypo-exponential distribution. For a single restart the optimal restart interval is 0.25 and for unbounded number of restarts the optimal restart interval is 0.19. 98

5.2 The probability density function $f_\tau(x)$ task service completion time with unbounded number of restarts (based on hyper/hypo-exponentially distributed completion time without restarts, with restart time $\tau = 0.1$ and cost $c = 0.02$). . 99

5.3 Restart time versus the normalised difference between unbounded restarts and no restarts, for the first three moments. 100

5.4 Illustration of the restart process. 102

5.5 Expected service time for different amount of restarts 105

5.6 Second moment of service time for different amount of restarts. 106

5.7 Optimal restart times, for the moments $\mathrm{E}[T_{15}]$, $\mathrm{E}[T_{15}^2]$ and $\mathrm{E}[T_{15}^3]$. 107

5.8 The probability of service completion before the deadline when using one restart ($d = 0.7, \mu = -2.3, \sigma = 0.97$). 108

6.1 Split–merge system, credit: [125], (CC BY-SA 3.0). 115

6.2 An example of processing a task in a three server split–merge system 116

6.3 A two server $\mathrm{Exp}(\lambda = 1), \mathrm{Exp}(\lambda = 2)$ example demonstrating difference between Dynamic and Static delays to reduce subtask dispersion 116

6.4 Showing the parametric curve of delays of the form (0,x,y). Formed by application of Tsimashenka's trade-off technique [128] for varying range of utilisation, on a three server split–merge system $\mathrm{Exp}(\lambda = 1), \mathrm{Exp}(\lambda = 5), \mathrm{Exp}(\lambda = 10)$ . . . 122

6.5 Changes in cumulative probability distribution of subtask service time when the number of servers is varied from 1 to 6. 124

6.6 Comparison of a hyper-exponential distribution with/without service restart. $\tau = 0.2$, $c = 0.05$, $f(x) = \frac{1}{2}\exp(\lambda = 1.0) + \frac{1}{2}\exp(\lambda = 0.2)$ 125

6.7 Trade-Off Results of **Methods 3 and 4** 133

6.8 Task Response time of **Methods 3 and 4** 134

6.9 Subtask Dispersion of **Methods 3 and 4** 134

Chapter 1

Introduction

In real world applications there are quite often multiple criteria one might like to be as good as possible. There is also a catchphrase saying "choose two out of three: fast, good and cheap", indicating that it is very difficult to satisfy all the potentially conflicting criteria the performance of a system is being judged by. This dissertation examines techniques in parallel queueing systems to provide insights in how to optimally trade off a set of metrics against each other so that the service level objectives of particular applications can be met.

1.1 Motivation

Due to an ever increasing demand for performance and speed in the modern world and the eventual exhaustion of possible optimisations to single-process systems, more and more of the world is turning towards parallel and distributed systems for its various processing needs. This trend is especially apparent in the world of IT, where companies are building distributed storage facilities, multi-core processors, RAID (redundant array of independent disks) systems [81, 80] and huge distributed cloud computing platforms. However, computing is not the only area where such demand is present. In finance, equities, options and futures are nowadays traded at lightning speeds on a vast number of exchanges. High dispersion and response time in order execution leads to monetary losses [23], while at the same time such systems are very expensive

and consume a large amount of electricity. In addition, manufacturers are making complex products with ever growing supply chains. In "just-in-time" manufacturing, companies wish to have their supply chains to be as lean as possible, as it helps them eliminate waste and reduce costs. This is accomplished by having the parts needed in production to arrive just moments before assembly of the product begins. Even in hospitals patient care is being studied and improved with the help of queueing models [13].

Parallel queueing network models are a mathematical tool to describe task flow in a system. In parallel queueing networks the service of one big task is split into a number of subtasks, where each subtask must be completed for the whole task to be completed. This dissertation investigates how to reduce subtask dispersion, task response time and energy consumption in a selected set of parallel queueing systems.

The three metrics used to evaluate the performance of parallel queueing networks are the following:

- **Subtask Dispersion** is the difference in completion time of the first and last subtask to finish.
- **Task response time** on the other hand is the time it takes from the point when a task enters the queue to be processed to the point it has been fully serviced and exits the process.
- **Energy usage** of a system is the amount of energy needed to operate it.

A simple example to conceptualize the benefit of reducing subtask dispersion is to think of a game, where a user sends a message over a TCP connection to multiple participants. The winner of the game is the player who receives the message first. The player who is closest to the originating server is most likely to win, as the message sent to him has to travel the least distance. Therefore, to make the game fair a time penalty should be added on players who are near the central server and this penalty should be bigger the closer they are to the central server.

The next subsections introduce real-world examples where an improvement with respect to the metrics mentioned above leads to an overall improvement in performance of the system.

1.1.1 Preventing Front-Running on the Stock Market

Subtask dispersion has recently become a topic that receives a lot of attention in the automated world of high frequency trading taking place on the stock markets around the world [23, 5]. The modern financial markets are distributed over a wide geographical area and multiple exchanges. In them equities and bonds and a variety of products derived from them are traded between market participants.

Whenever an investor wants to make a big buy order they send orders to multiple exchanges so that they are able to take advantage of a larger pool of liquidity. However, this can create a problem for the buyer: if the buy order arrival times on the various exchanges differ from each other too much, a high frequency trader using microwave towers to communicate can learn about the trade in one exchange and send a signal to another exchange. The high frequency trader is then able to buy the stock available on that exchange before the incoming order arrives via fibre optic cable. The high frequency trader can then resell the shares they just bought at a slightly increased price to the incoming buy order, hence making an immediate profit at the expense of the original investor.

The high frequency traders have connected their systems between the stock exchanges with microwave connections, which can transfer data faster than fibre-optics. An example of these microwave links between exchanges that have been setup by various high frequency companies between London and Frankfurt can be seen in Fig 1.1. As a result traders who wish to avoid paying a tariff each time they execute a big trade must be smart when they execute a buy order and make sure that the subtask dispersion of multi-exchange orders is minimised.

Subtask dispersion is only one of the many issues that trading boutiques need to deal with. Attractive buying opportunities in the market tend to be fleeting as it is often a race between multiple parties trying to snatch up value. As a result of the arms race between high frequency

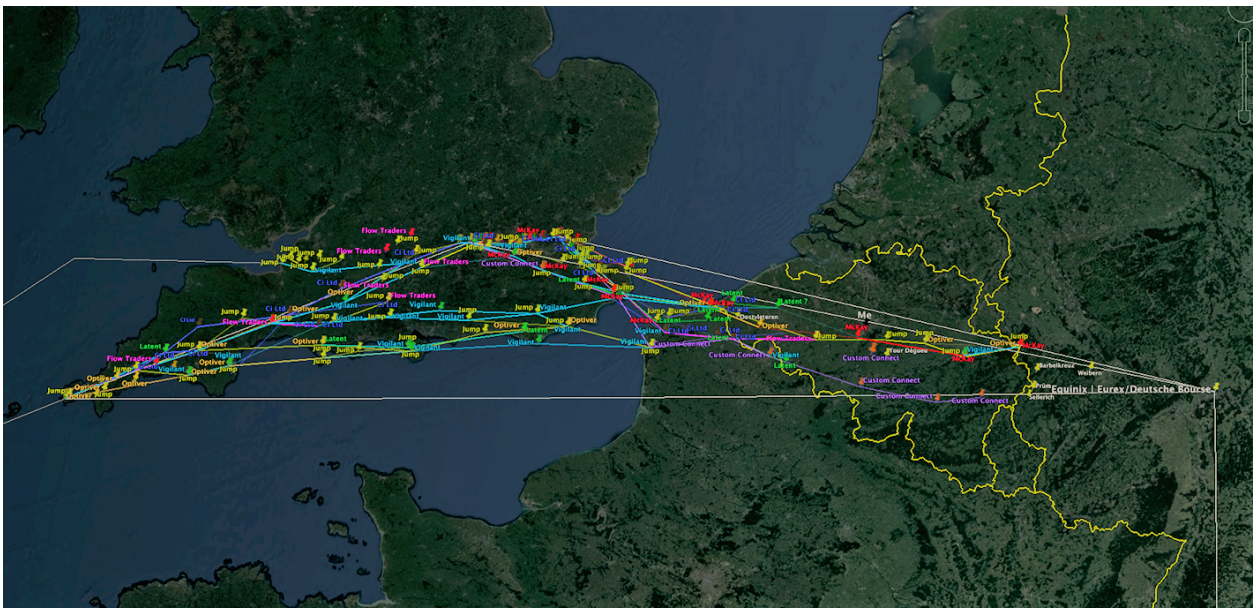

Figure 1.1: Microwave pathways between southern England and Germany that are used by various high frequency trading companies [6]

trading companies exchanges have multi-tiered pricing based on how much latency there is between the exchange servers and the end user [105].

The problem of simultaneously optimising three metrics is quite a complicated ordeal in some cases. Especially when subtask dispersion, task response time and energy usage/cost metrics are conflict with each other.

1.1.2 Fairness in Online Gaming

Online games are often played between players located in different corners of the world. For example, one player can be physically located in Europe and another in Asia. This introduces lag into games as information describing the actions of a players will have to be transmitted over large distances. It has been found that lag is detrimental to the performance of a player in First Person Shooter games [144] as well as frustrating to players [124]. Also it should be noted that the distribution of lag between players is not uniform. Players who are physically located near to the hosting server experience less lag and players far away experience more. Therefore, a subset of players will experience a larger lag throughout the whole game, while others do not.

The issue of lag in online games is becoming especially important, as electronic gaming has

grown from a small niche form of entertainment in the 1970s to one of the largest forms of entertainment along with movies and music. In addition, competitive gaming is booming and becoming a multi billion dollar industry [51], with prize pools for the biggest competitions being tens of millions of dollars [73].

Therefore, techniques to minimise subtask dispersion and task response time are useful in the context of online gaming. This is because it will help make it possible to deliver a more pleasant and fair gaming experience to players, which will support the financial growth of the industry in the future and enable people to enjoy their free time more.

The delay strategy should not be calibrated in a way that all players get the same amount of lag as the slowest player, but so that the few fastest players get a slightly increased lag to bring them more in line with the average lag. This could be done by using a metric, which optimises the product of subtask dispersion and task response time, with a higher weight on task response time.

1.1.3 Delivery Processing at Online Retailers

Ideally, online retailers would like to first collect everything the customer has ordered and then ship the items in one package. However, sometimes this is not possible due to time and space constraints in shipping warehouses or customer demands. This then leads to multiple packages being sent to the customer. Sending multiple packages instead of one increases the cost to the online retailer as they are charged on a per package basis by the package delivery companies.

An example of such behaviour can be seen when Amazon asks the customer whether they wish to receive all the items in a single package to save on shipping costs or if they should be sent separately so that some of them can arrive faster. An example of this can be seen in Fig. 1.2. The customer has a choice between two delivery techniques. If the customer chooses to receive their items sent separately it means that the customer prefers to optimise the task response time metric of the arriving goods. If the customer prefers to receive all their items at the same time they prefer to optimise the subtask dispersion of their order.

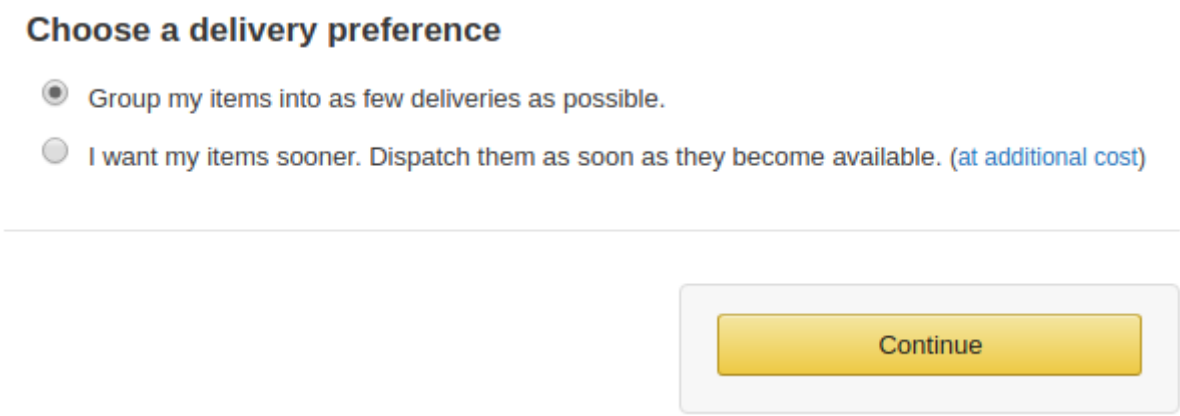

Figure 1.2: Example of Amazon delivery preference option

The retailer in the above scenario has to choose an optimal delivery method based on multiple criteria. The retailer itself wishes to optimise the amount of profit it makes from the transaction. The profit of the company depends on multiple factors. The company profits by being able to send multiple items in a single package. However, it will have to balance this saving with the prospect of an angry customer who does not want to wait too long for their items, which could cause them to either order less items from the store in the future or even worse write a complaint about the company on a feedback website. This in turn could cause multiple lost sales from multiple customers in the future.

At the heart of this optimisation problem is the fact that the performance of package delivery is being assessed in accordance to multiple metrics: subtask dispersion, task response time and cost. In the example above, Amazon has decided to solve the problem by gauging the interests of the customer, by placing a financial incentive for the customer to choose the cheaper delivery option. If the customer strongly prefers to receive their goods faster, Amazon can then provide a fast but expensive service. Otherwise it is able to provide the cheaper but slower service.

1.2 Aims and Objectives

The primary hypothesis of this thesis is that it is possible to control parallel queueing systems in a way that improves the performance of the system with respect to an objective function that reflects multiple conflicting performance metrics. The first project we undertook to complete

the goal was to investigate existing methods of controlling subtask dispersion and task response time in split–merge and fork–join queueing networks. Secondly, we analysed a more complicated structure a multi-layered queueing structure, the hidden stochastic PERT system, where the system needs good performance for both task response time and subtask dispersion. Thirdly, we performed a survey of the field of using restart and replication mechanisms to improve system performance. Finally, we combined existing techniques to reduce subtask dispersion with what we discovered during our investigation on restart, replication as well as state-dependent service.

To achieve this the following steps need to be taken:

- Survey literature for existing research into optimising system performance and identify key techniques that can be used to improve performance of parallel queueing systems with regard to multiple metrics.

- Apply and modify accordingly the key techniques that were identified during the literature survey phase in a way that they can be used to optimise performance of parallel queueing systems with regards to multiple metrics.

- Build simulations to accurately model parallel queueing systems, which can be modified according to techniques discovered during the literature survey phase. The simulations should be flexible, as we want to optimise over multiple performance criteria and therefore need the model to output a wide range of information about the system.

- It is also important that the developed techniques are tractable, as the goal of the thesis is not only to develop theory of parallel queueing systems, but also optimise said systems applicable to real-world problems.

- Construct suitable procedures for optimising the performance of parallel queueing system, whose performance is being assessed with respect to multiple performance criteria.

1.3 Contributions

This dissertation improves upon existing research in multiple ways. Firstly, it presents an investigation on restart mechanisms to improve system efficiency in literature. Secondly, the dissertation examines multiple existing techniques and studies how they can be combined together to improve queueing systems that have multiple conflicting performance metrics. Finally, the dissertation expands upon existing techniques to work in more general situations.

1.3.1 Reducing Subtask Dispersion in Hidden Stochastic PERT Networks

The first contribution of the thesis is a numerical technique to both compute and optimise subtask dispersion in Hidden Stochastic PERT networks. The task flow in the network can be controlled through a series of delays, which are inserted to slow down the processing of activities in the system.

The technique performs a black-box optimisation of the system. The black-box takes as input a set of delays and returns as output information on how the system performed on various optimisation metrics for a single task. The technique works by using a genetic optimisation algorithm CMA-ES to find a globally-optimal set of delays, which optimise the system in terms of subtask dispersion and task response time.

The work is relevant because existing subtask dispersion minimisation techniques are only applicable to single-layered queueing networks such as the split–merge and fork–join systems, meaning that work on all subtasks can be begun straight away. In contrast the PERT networks define a partial order in which the activities it consists of need to be completed in.

The partial ordering of tasks, that PERT provides is important in many real world applications, as there are often restrictions on the order tasks need to be completed in. For example, the process of manufacturing a car you can't put on the tires before the chassis has been built.

1.3.2 Investigating Restart Mechanisms to Improve System Efficiency

The second contribution of the thesis is an in-depth analysis and a literature survey into the use of service restart to improve system performance.

The main three conclusions that we drew from the survey were the following:

- High coefficient of variation is needed for system restart.
- Many use cases in real world problems benefit from system restart
- Research is also being done on estimating degradation levels of systems, so that errors can be detected before occur

1.3.3 Computing Probability Density Function of Dynamically Padded Split–Merge System

Thirdly, this dissertation contains a derivation of the probability distribution of task service times in split–merge systems, given that delays have been inserted pre-service and the delays are cut short if a sibling subtask completes service. This derivation then enables us to analytically calculate a task response time of a split–merge system, which use dynamic padding.

This is useful as previous work [103] only contains derivations for analytical computation of subtask dispersion and not task response time. Being able to compute task response time of systems mentioned above allows us to optimise such systems in terms of other. An example of such a metric is the trade-off between task response time and subtask dispersion introduced in paper [128].

1.3.4 Trade-Off of Multiple Metrics in Split–Merge Queueing Systems

Fourthly, the work in this dissertation contains an investigation as to how the product of task response time, subtask dispersion and energy consumption can be improved with the help of multiple existing techniques from literature. The four techniques include: dynamic subtask dispersion, state dependent delays, subtask service restart and subtask service duplication.

Key insights made during the work include:

- Product of service metrics can be improved by varying service standards depending on the state of the system. For example, when utilisation of the system is very high it might pay off to decrease service standard of subtask dispersion somewhat in order to provide faster task response time.
- Duplication and service restart of bottleneck tasks, which have the tendency to hang can have a huge impact on the overall performance of the system. However, research in the thesis indicated that users should choose between restarting and duplication, as the combination of both techniques was not able to improve performance further from just using one technique.

1.4 Statement of Originality

I declare that this thesis was composed by me and that the work described in it is my own, expect where stated otherwise.

1.5 Publications

The following publications are related to my PhD and they form the foundation of this dissertation:

- **Optimising Hidden Stochastic PERT Networks. Proc. 10th International Conference on Performance Evaluation Methodologies and Tools (VALUETOOLS 2016). Taormina, Italy October 2016** This paper introduces a technique for minimising subtask dispersion in hidden stochastic PERT networks. The technique improves on existing research in two ways. Firstly, it enables subtask dispersion reduction in DAG structures, whereas previous techniques have only been applicable to single-layer split–merge or fork–join systems. Secondly, the exact distributions of subtask processing times do not need to be known, so long as there is some means of generating samples. The technique is further extended to use a metric which trades off subtask dispersion and task response time.

  The work in this publication is solely my own work.

- **Three-way Optimisation of Response Time, Subtask Dispersion and Energy Consumption in Split–Merge Systems. Proc. 11th International Conference on Performance Evaluation Methodologies and Tools (VALUETOOLS 2017). Venice, Italy December 2017** This paper investigates various ways in which the triple trade-off metrics between task response time, subtask dispersion and energy can be improved in split–merge queueing systems. Four ideas, namely dynamic subtask dispersion reduction, state-dependent service times, multiple redundant subtask service servers and restarting subtask service, are examined in the paper. It transpires that all four techniques can be used to improve the triple trade-off, while combinations of the techniques are not necessarily beneficial.

  This publication is joint work with Jani Kettunen and Katinka Wolter. I performed most of the work in the paper. Kettunen implemented the state-dependent split–merge optimisation using Bayesian Optimisation. Wolter suggested to use system restart to improve system performance and also provided help with editing and proofreading the paper.

  A poster adaptation of this paper won first place at the Imperial Google Poster Competition, in the 3rd and 4th year PhD student category in 2018.

- **Book Chapter: Model-Based Assessment and Optimisation of Restart and Rejuvenation Strategies in Models for Restart and Rejuvenation. Handbook of Software Aging and Rejuvenation World Scientific. 2018** This chapter considers model-based assessment and optimisation of restart and rejuvenation strategies. It provides an up-to-date survey of the research literature. Within the chapter, we first discuss how to determine the success of restart strategies by introducing appropriate metrics for comparison. We then provide expressions of completion time for general completion time distributions and provide examples of distributions for which restarts are particularly effective (such as heavy-tail distributions). We discuss model-based approaches for the optimisation of restart timing strategies, aiming to minimise moments of completion time and probability to meet a deadline, respectively. We conclude the chapter with open challenges in model-based assessment and optimisation of restart strategies.

  This publication is joint work with Aad van Moorsel from Newcastle University and Katinka Wolter from Freie Universität Berlin. I carried out the literature survey portion of the paper.

1.6 Thesis Outline

The remainder of the thesis is organised as follows:

- **Chapter 2** contains background information and related research as well as introduction of terms and techniques referred to in the thesis. This chapter includes definitions for multiple types of parallel queueing networks as well as multiple metrics, which are often used to analyse the performance of said queueing networks. The chapter continues with a review of existing related research. The chapter conludes with a literature survey of papers, which investigate restart phenomena.

- **Chapter 3** Takes an in-depth look at existing research that is relevant to this PhD. The chapter first explains related techniques in detail and then compares the various

techniques via three case studies. The metrics used in the comparisons are task response time and subtask dispersion.

- **Chapter 4** introduces a technique to optimise the trade-off between subtask dispersion and task response time in hidden stochastic PERT networks. The technique works numerically by using CMA-ES optimisation algorithm to optimise a cost function, which is approximated via simulation of the system.

- **Chapter 5** introduces the concepts of service restart and service replication. The chapter investigates how both techniques can be used to improve service times of tasks. In addition, we investigate how the restart interval of a task should be chosen.

- **Chapter 6** brings together multiple existing techniques from literature to improve the triple trade-off product of subtask dispersion, task response time and energy consumption. The techniques we study include:

  - Dynamically-adjusted delays inserted in front of subtasks
  - State-dependent delays inserted in front of subtasks
  - Use of service restart to improve service completion times of subtasks with heavy-tailed service times.
  - Use of service replication to speed up service of subtasks

- **Chapter 7** concludes the thesis by summarising the research achievements of the dissertation, as well as discussing potential applications. Finally the chapter concludes with a discussion of future research.

Chapter 2

Background

This chapter covers the most relevant aspects of queueing theory, which are needed to understand this thesis. Queueing theory is the mathematical study of waiting lines and their associated phenomena. In these processes, information/products/people travel between service centres of the system and the capacity of the system to transfer objects flowing from one state to another is rate-limited. Aspects of queueing theory can be observed in many aspects of our daily lives, such as when you queue to get to the counter in the supermarket or when you are transferring data over the internet or when you are commuting back home after work.

The first part of the chapter discusses Kendall's notation, which is a generalised marking technique to label queueing nodes. Along with the describing some examples of queueing models that are used in this thesis are discussed. The examples are the M/M/1 and M/G/1 queueing models.

The second part of this chapter covers parallel queueing models that are relevant to the work performed later on. We first cover simpler single-layer systems: split–merge and fork–join queueing systems and then proceed to introduce stochastic PERT networks, which is a more general multi-layered queueing network model.

The third part of this chapter discusses multiple metrics that are often used to analyse the performance of queueing networks. In many occasions the performance of the queueing system

is analysed with respect to multiple criteria, and the different performance metrics often conflict with each other, such as wanting to service each task to a high standard and wanting to deliver service quickly.

The chapter also covers the performance metrics used to evaluate the performance of the queueing systems. The first metric we discuss is the task response time metric. This metric measures how quickly incoming tasks are processed by the system. The second metric is the subtask dispersion of the system. This metric measures the difference in completion times between the first and last subtasks. The final metric of discussion is the energy consumption/cost metric. This metric represents the extra cost that is incurred when additional servers are added in an attempt to improve the performance measured in terms of other metrics such as subtask dispersion and task response time.

The chapter then continues to discuss numerical optimisation techniques such as Newton's method, CMA-ES and Bayesian Optimisation, which are used for tuning free parameters of the queueing system optimisation techniques in later chapters.

The chapter concludes with a literature survey on task response time, subtask dispersion, energy consumption and state-dependent service in queueing system, and a survey into trends in restart relalated research in the past decade.

2.1 Queueing Models

In 1953 D.G Kendall proposed the $A/S/c$ notation to classify queueing models. A defines the interarrival time distribution of incoming tasks. S defines the distribution of the service time of tasks and c is the number of parallel servers servicing tasks. Typical values for A and S are M for Markovian, D for Deterministic and G General. c is a non-zero integer.

Sometimes a more descriptive $A/S/c/K/N/D$ model is used, to define extra parameters of the queueing system. In this model, the extra letters $K/N/D$ define the capacity of the system, size of the queueing population, and the scheduling discipline. In the context of this dissertation

the extra parameters are not necessary to determine the type of queueing system. The systems studied here always have an infinite capacity and queueing population, and the queueing systems always use the First in First out as the service discipline.

2.1.1 M/M/1 Queue

M/M/1 is one of the simplest queueing systems, and as a result is an frequently-used model in many applications. In M/M/1 queues, incoming tasks have an interarrival rate that is exponentially distributed with a parameter λ, which determines the rate of arrivals [12]. The service time of the tasks is also exponentially distributed with a parameter μ, which determines service rate of the tasks in the system. The utilisation (or load factor) of a M/M/1 queue ρ is equal to λ/μ. The system is stable when $\rho < 1$ [1]. By stability it is meant that the average queue size of the system is finite.

The M/M/1 queue can be modelled as a continuous time Markov chain where the state of the system is defined by queue length and whether a task is currently in service. The memoryless property can often be exploited when analysing M/M/1 queues.

2.1.2 M/G/1 Queue

The M/G/1 queue is another important and well understood queueing system. The incoming tasks have an interarrival rate that is exponentially distributed with a parameter λ to determine the rate of arrivals. The service time of tasks in a M/G/1 queue is generally distributed [20]. Meaning that an arbitrary probability distribution can be used to describe the service time. The support for general service time of tasks makes the model more applicable to real life problems, but at the same time makes it harder to derive closed-form solutions for various metrics of the M/G/1 system.

When analysing the M/G/1 queueing system, the PASTA (Poisson Arrivals See Time Averages) property can be used. The PASTA property states that probability of the system being in a

given state is the same when seen from the point of view of outside observer and an arriving task [138]. The M/M/1 queue is a special case of the M/G/1 queue and therefore all results for M/G/1 queue are also valid for M/M/1 queues.

An important formula in the study of the M/G/1 queue is the Pollaczek–Khinchine formula. The formula states the task response time of a M/G/1 queue. λ is the parameter to the exponential function, which determines tha task interarrival rate. μ is the service rate. $\rho = \lambda/\mu$ is the utilisation (or load factor) of the system. G is the service time distribution.

$$\mathrm{E}[\mathrm{Resp}(\lambda, \mu, \mathrm{G})] = \frac{\rho + \mu\lambda \mathrm{Var}[\mathrm{G}]}{2(\mu - \lambda)} + \mu^{-1} \tag{2.1}$$

2.2 Parallel Queueing Models

Parallel queueing networks are a branch of queueing networks. They describe the service of a task whose service is split into subtasks that can be serviced in parallel. The subtasks can either be totally disjoint as in the case of split–merge and fork–join systems, or partially depend on each other as is the case in PERT networks.

This section contains a brief introduction to the parallel queueing models that are used throughout the dissertation. It begins by discussing the split–merge model. Next it proceeds to discuss the fork–join model, which is a less restricted version of the split–merge model. Finally, we also introduce the Stochastic PERT network, which can be used to model more complex multi-layered processing of tasks. In PERT networks tasks are first split into subtasks, and subtasks can be further split-up into activities, which can depend on each other for completion as well as being shared by different subtasks.

2.2.1 Split–Merge System

Split–merge systems are single-layer systems. By single-layer systems, we mean that work can begin straight away on all parts of the task. Split–merge systems have task arrivals with an

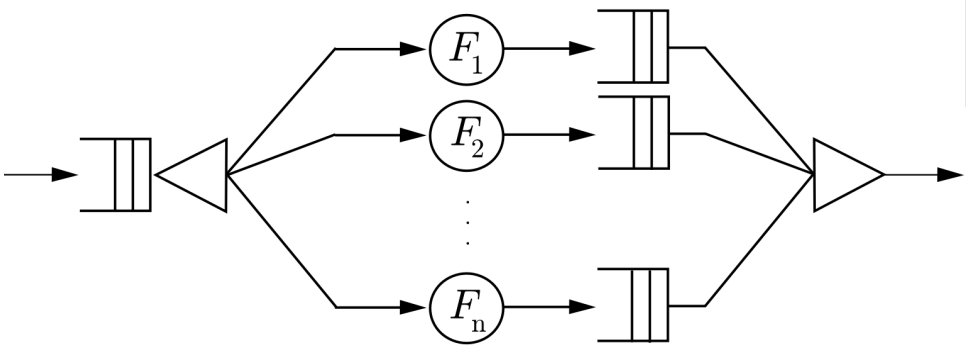

Figure 2.1: A split–merge system Queueing System, credit: [125], (CC BY-SA 3.0).

interarrival rate that is exponentially distributed with a rate of λ. The system structure is shown in Fig. 2.1.

Arriving tasks will either enter service directly if the system is idle or join the queue to wait for their turn. When the service of a task is completed it exits the system and service on the next task in the queue begins. If the queue is empty the system will be idle until a new task enters the system.

When a task enters service it gets split up into N subtasks. The system has a dedicated server for each subtask of the task. Each subtask has a specific service time probability distribution associated with it, where f_i and F_i are the respective probability density function and cumulative distribution function of service time of the ith subtask. A task is considered complete once all of its subtasks have completed service.

2.2.2 Fork–Join System

The fork–join system is very similar to the split–merge system. Fork–join system is also a single layer system, like the split–merge system. The main difference is that incoming tasks in fork–join systems are split into subtasks on task arrival. In split–merge systems the splitting is done when the service of a task begins. The structure of a fork–join system is shown in the Fig. 2.2

New tasks enter the system with a exponential interarrival rate that is distributed exponentially

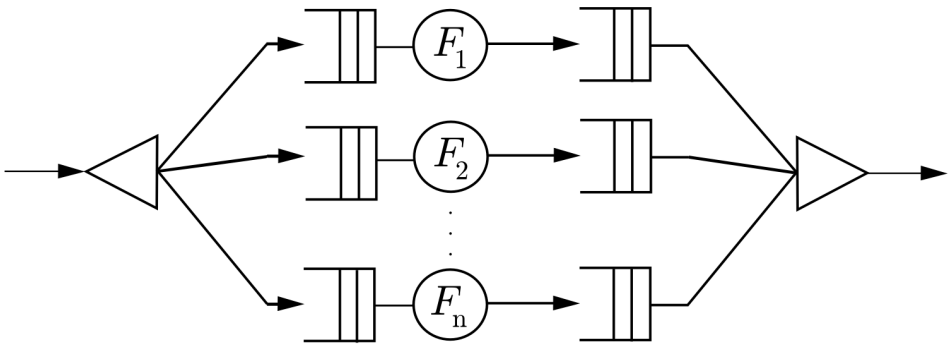

Figure 2.2: A fork–join system credit: [125], (CC BY-SA 3.0).

with a rate of λ. Arriving tasks into the fork-join system are split into N subtasks on arrival, before queueing occurs. Each server responsible for servicing a specific type of a subtask has its own queue. The subtasks service servers pull tasks out of their own queue independently of each other.

A task is considered complete once all the N subtasks it was split into have completed service. As before f_i and F_i are respectively the probability density function and cumulative distribution function of service time of the server servicing the ith subtask.

2.2.3 Stochastic PERT Networks

The project evaluation and review technique, abbreviated as PERT, is used in project management to analyse statistical information related to the completion of a single one off project. The PERT DAG (Directed Acyclic Graph) also provides a helpful visualisation of the individual activities and milestones that make up the project [26]. Fig. 2.3 shows an example of a Stochastic PERT DAG.

The PERT system is a multi-layered system. This is because, there is a hierarchy which defines a partial order, in which the activities need to be serviced in. The task service portion of split–merge and fork–join systems can be thought of as a simple PERT system, with no restrictions and one customer.

A PERT system only deals with individual tasks and has no queueing in the system. You could

for example build a PERT DAG to model the construction of a building. Each node in the DAG is related to a milestone of completing a set of activities in the system. An activity is a directed edge connecting two nodes. The milestone where the directed edge begins must be reached before the activity can begin service.

Service on activities that begin from a node with no incoming activities can begin straight away. Each sink node in the PERT defines a subtask (9 and 10). A subtask is considered complete once all activities that can be used to reach it from the source node are completed (e.g. f_3, f_7 and f_9 for subtask defined by node 10). The task is completed when all activities in the PERT DAG are completed (once all f_i are completed in our example case).

A stochastic PERT DAG differs from a regular PERT DAG by having completion times of activities be probability distributions instead of constants.

The service time of an activity in a stochastic PERT DAG is denoted by the probability density function and cumulative distribution function as f_n and F_n. Service of an activity cannot begin before all activities pointing to the node where the activity starts from have been serviced.

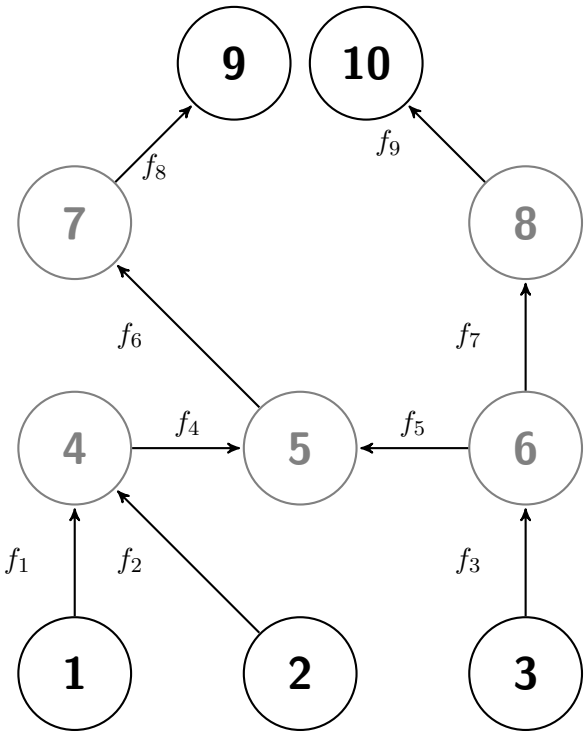

Figure 2.3: An example PERT Network with two subtasks, which complete 9 and 10

2.3 Performance Metrics

This section discusses three performance metrics that are used in the thesis, namely task response time, subtask dispersion and energy consumption.

2.3.1 Subtask Dispersion

Subtask dispersion is the difference in completion times between the first and last subtask belonging to the same job. This subsection covers two existing techniques that can be used to reduce subtask dispersion. The first technique was analysed in [127, 128, 126, 125] and a second technique was presented in [103]. Both techniques add a delay d_i in-front of the service of each subtask. The delays are used to align the subtask service completion times to minimise subtask dispersion of the task.

Static Subtask Dispersion

Here, we present a technique using static delays to minimise subtask dispersion in split–merge systems. Static in our context means that subtasks delays are not modified after a task enters service.

With N subtasks the expected time of first and last subtask to finish can be calculated with the theory of heterogeneous order statistics [34]. The cumulative distribution functions of first and last subtask to finish can be seen in Eqn. (2.2) and (2.3). Both formulas use the Bayes Theorem for independent events.

The cumulative distribution function for the completion time of the last subtask to finish can be computed by multiplying all the individual cumulative distribution functions of subtasks together. The completion time of the first subtask can be obtained with the use of similar logic on the inverse probability of a subtask having not completed.

$$X_{(1)}(t) = Pr\{X_{(1)} < t\} = 1 - \prod_{i=1}^{N}[1 - F_i(t)] \tag{2.2}$$

$$X_{(N)}(t) = Pr\{X_{(N)} < t\} = \prod_{i=1}^{N}[F_i(t)] \tag{2.3}$$

Static subtask dispersion can be derived with the help of $X_{(1)}(t)$ and $X_{(N)}(t)$. The probability at time t that at least one and not all subtasks have completed service is $X_{(N)}(t) - X_{(1)}(t)$.

$$\mathrm{E[Disp]} = \int_0^\infty X_{(N)}(t) - X_{(1)}(t)\mathrm{d}t \tag{2.4}$$

$$\mathrm{E[Disp]} = \int_0^\infty \Big(\prod_{i=1}^{N}[F_i(t)]\Big) - \Big(1 - \prod_{i=1}^{N}[1 - F_i(t)]\Big)\mathrm{d}t \tag{2.5}$$

Dynamic Subtask Dispersion

Here, we present a technique using dynamic delays to minimise subtask dispersion in split–merge systems. Dynamic in our context means that subtask delays might be modified after a task has entered service. More specifically, delays added at the start of task service are removed later on when a sibling subtask completes service.

Subtask dispersion can be improved by adding dynamic delays into the system. In our formulation we have a delay vector $\mathbf{d}$, which is used to delay the ith subtask by d_i amount.

When using dynamic delays any remaining delays on sibling subtasks are removed, once the first subtask of the task completes. Dynamic delays improve subtask dispersion in cases where a particular subtask has finished service quicker than expected, as it removes unnecessary delays on sibling subtasks. Below we provide some insight into how dynamic subtask dispersion can be measured.

Let $T(i, t, \mathbf{d})$ be the probability that subtask i is the first to finish at time t. Then $E_r(i, t', \mathbf{d})$

is the expected completion time of the remaining subtasks, given that subtask i finished at time t'. $G_j(t, t', d_j)$ is the probability distribution of a subtask, given that a sibling subtask has finished already. $F_i(t)$ and $f_i(t)$ are the cumulative distribution function and probability density function of service time of ith subtask.

$$\mathrm{E}[\mathrm{Disp}(\mathbf{d})] = \sum_{i=1}^{N} \int_0^\infty T(i, t, \mathbf{d}) E_r(i, t, \mathbf{d}) dt \tag{2.6}$$

where

$$T(i, t, \mathbf{d}) = f_i(t - d_i) \prod_{j \neq i} [1 - F_j(t - d_j)] \tag{2.7}$$

and

$$E_r(i, t', \mathbf{d}) = \int_0^\infty [1 - \prod_{j \neq i} G_j(t, t', d_j)] dt \tag{2.8}$$

with

$$G_j(t, t', d_j) = \begin{cases} F_j(t) & \text{if } t' < d_j \\ F_j(t + (t' - d_j) | t > 0) & \text{otherwise} \end{cases} \tag{2.9}$$

subject to conditions

$$\prod_{i=1}^{N} d_i = 0 \tag{2.10}$$

and

$$\forall i \quad d_i \geq 0 \tag{2.11}$$

2.3.2 Task Response Time

Task response time is equal to the duration between a task entering the system and finishing service. Much effort has been spent on researching task response time in various settings. Examples of this can be seen in [15, 58, 61].

Task response time can be computed analytically for split–merge systems with the Pollaczek–Khinchine formula that is defined for M/G/1 queues below:

$$\mathrm{E}[\mathrm{Resp}(\lambda, \mu, \mathrm{X}_{(\mathrm{N})})] = \frac{\rho + \mu\lambda \mathrm{Var}[\mathrm{X}_{(\mathrm{N})}]}{2(\mu - \lambda)} + \mu^{-1} \tag{2.12}$$

Where μ is the service rate, λ is the arrival rate and $\rho = \lambda/\mu$ is the utilization of the server. $Var[X_N]$ is the variance of service time of the last subtask to finish, as well as the variance of service time of the task. The split–merge system is equivalent to a M/G/1 queue. The task service time of the M/G/1 queue is equal to the completion time of the last subtask to finish in a split–merge system as discussed in Sec. 2.3.1.

Fork–join systems currently have no general analytical formula to calculate response time. Some simple cases with only two subtasks have been solved analytically [41, 40]. There also exists work on approximating task response time in fork–join queues [95, 82, 113].

2.3.3 Trade-Off Metric

Sometimes performance is assessed from the perspective of multiple metrics simultaneously. In such a case the individual performance metrics should be combined into a single metric. This subsection discusses an existing metric by Tsimashenka [128]. The metric was originally inspired by the energy–response time product analysis of power policies for server farms [48, 49]. The original paper combines the two metrics: task response time and subtask dispersion in the following way:

$$\mathrm{T}(\lambda, \mu, \mathrm{X}_{(\mathrm{N})}) = \mathrm{E}[\mathrm{Disp}] \times \mathrm{E}[\mathrm{Resp}(\lambda, \mu, \mathrm{X}_{(\mathrm{N})})] \tag{2.13}$$

In the later chapters, this dissertation extends on this metric in multiple ways. The extensions include: (a) adding a energy metric as a third metric into the product, (b) using weights to determine the importance of each metric, and (c) performing the necessary derivations to make it possible to apply dynamic subtask dispersion from Sec. 2.3.1.

For split–merge systems a trade-off equation between subtask dispersion and task response time

can be expressed in the following way:

$$\mathrm{T}(\lambda, \mu, \mathrm{X}_{(\mathrm{N})}) = \left[\int_0^\infty 1 - \prod_{i=1}^{N} F_i(t) - \prod_{i=1}^{N} (1 - F_i(t)) \mathrm{d}t \right] \left[\frac{\rho + \mu\lambda \mathrm{Var}[\mathrm{X}_{(\mathrm{N})}]}{2(\mu - \lambda)} + \mu^{-1} \right] \tag{2.14}$$

where

$$\mathrm{Var}[\mathrm{X}_{(\mathrm{N})}] = 2 \int_0^\infty t(1 - \prod_{i=1}^{N} F_i(t)) \mathrm{d}t - \left(\int_0^\infty 1 - \prod_{i=1}^{n} F_i(t) \mathrm{d}t \right)^2 \tag{2.15}$$

For fork–join systems the trade-off metric has to be quantitatively measured through simulations, since – to the best of our knowledge – there are no closed form solutions for either subtask dispersion or task response time for such systems.

2.4 Optimisation Algorithms

This section describes the various optimisation algorithms used throughout the thesis. Optimisation algorithms are used to find the best element from the solution space. The best element is determined by the objective function, which can be used to determine the value of each element in the solution space. The section begins by introducing some classical optimisation algorithms, namely Newton's method and the Nelder–Mead algorithm. It then proceeds to discuss algorithms more suited for noisy non-convex objective functions, which include CMA-ES [56] and Bayesian Optimisation [101].

2.4.1 Newton's Method

The Newton Method also known as Newton-Raphson method is an iterative technique to find local extrema of function $f(x)$. The starting point x_0 can be picked arbitrarily. The technique requires the the second derivative of the objective function. The advantage of the Newton

Method is that it is quick to converge and therefore not many iterations of the algorithm are needed.

One iteration of the multidimensional Newton method is performed in the following way:

$$\mathbf{x}_{k+1} = \mathbf{x}_k - \mathrm{J}(\mathbf{x}_k)^{-1}\mathbf{f}(\mathbf{x}_k) \tag{2.16}$$

where the multidimensional function is defined as follows:

$$\mathbf{f}(\mathbf{x}) = (f_m(x_1, \ldots, x_n)), \quad \text{for} \quad m = 1, \ldots, N \tag{2.17}$$

and the Jacobian matrix is defined as:

$$\mathrm{J}_{m,n}(\mathbf{x}) = \frac{\partial f_m}{\partial x_n} \tag{2.18}$$

For the jacobian matrix to exist at a given point the, function $f(x)$ needs to be differentiable at x.

2.4.2 Nelder–Mead Method

The Nelder–Mead algorithm was developed in 1965 by John Nelder and Roger Mead. The Nelder–Mead method is a numerical technique to find a local extreme of a function $f(x)$ in a multidimensional optimisation space. The Nelder–Mead method does not require the use of derivatives. The algorithm keeps track of $n + 1$ points that form a simplex, where n is the dimensionality of the problem.

During each iteration of the algorithm, the point with highest cost as measured by the objective function is removed. A new point is then computed to form a new set of $n + 1$ points. This is done by constructing a line between the point that was removed and the central point

of the remaining set of points. The new point is either chosen via reflection, expansion or contraction [14].

2.4.3 CMA-ES

CMA-ES is short form for Covariance Matrix Adaptation Evolution Strategy [54]. CMA-ES is an evolutionary algorithm, which can be used to find local extreme of a non-convex continuous function. CMA-ES does not use derivatives of the function that is being optimised.

The CMA-ES works iteratively. at the beginning of each iteration the algorithm starts by having a solution candidate pool. CMA-ES then generates new candidates by 'mutating' existing solution candidates. In the final step of a single iteration the algorithm removes the least fit candidates from the pool and then begins a new round of iteration.

2.4.4 Bayesian Optimisation

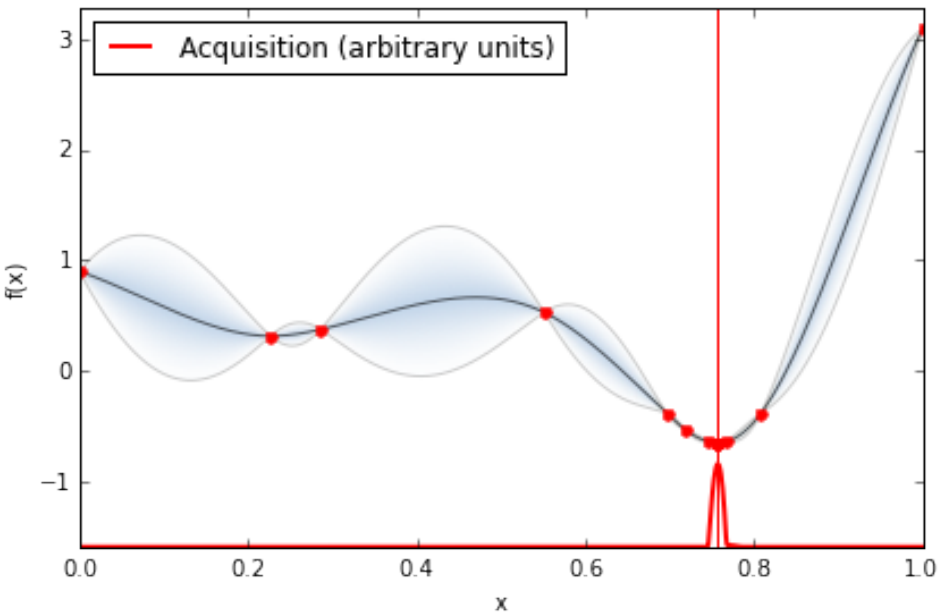

Figure 2.4: Estimating the optimisation space by Bayesian Optimisation [120], (CC BSD 3).

Bayesian Optimisation is a technique to find local extrema of an function $f(x)$, with the aim of using as little function calls as possible on the function that is being optimised. As a result,

it is suited for optimising functions, which are expensive to call [101].

The technique constructs an approximation of the optimisation space. The point, which is queried next by the optimisation function is the point the model currently says is the most promising candidate according to internal acquisition model of the optimiser. Next the algorithm queries the point decided by the internal acquisition model and updates the internal model. It then iteratively repeats the process. An Example of Bayesian optimisation can be seen in Fig. 2.4,

2.5 Summary of Related Research

This section contains literature surveys on topics which are relevant to the research that is presented in the later chapters of this dissertation. The literature survey includes a subsections on task response time, subtask dispersion, energy consumption optimisation and state-dependent service in queueing systems.

2.5.1 Task Response Time

We've first included a few important historical developments that were necessary for the study of task response time in parallel queueing networks. The whole field and many important results are based on the work done by Markov in 1906 [90] on Markov Chains. The first results in the field of queueing theory were derived by Erlang [39, 24]. The results included analysis of basic queueing models such as the $M/D/1$ and $M/D/k$ queueing models. Another early breakthrough was the work by Pollaczek and Khinchine which resulted in the closed form solution to the average waiting time of $M/G/1$ queueing system [104, 77].

The first results in the field of parallel queueing systems was by Heidelberger and Trivedi, who came up with a very accurate approximation techniques for a mean queue length and mean response time of an $M/M/1$-based queueing system where tasks are split into two or more subtasks [61] in 1983. Flatto, Leopold and Hahn derived a stationary distribution for

the number of queueing subtasks in a fork–join system with two subtasks, with the assumption that interarrival task arrival times are exponentially distributed. [41, 40].

Over the years, a multitude of approximations and bounding results regarding fork–join systems have been proposed by various researchers [95, 74, 112, 82, 132, 58]. However, no analytical results for the stationary distribution for general fork–join queues are not known and it remains an open problem.

2.5.2 Subtask Dispersion

The study of subtask dispersion has only recently garnered attention. The term subtask dispersion has first been introduced by Tsimashenka and Knottenbelt in papers [126, 128, 127] and in Tsimashenka's PhD thesis [125]. The first paper introduced a scheme for split–merge systems, where static delays of varying length are selectively inserted in front of subtask service to reduce subtask dispersion of the system [126]..

Their second paper [128] introduced a technique to improve the trade–off between subtask dispersion and task response time of the system. As before, delays are inserted in front of subtasks to control the subtask dispersion of the system. However, when the duration of the pre-emptive delays is computed, they take into account the effect of the delays into both task response time and subtask dispersion and minimise the 'Resp $\times$ Disp' product.

Their final paper [127] investigated how delays inserted in front of subtask service could be used to reduce subtask dispersion in fork–join systems. Fork–join models are interesting as the queueing in the system occurs at the subtask level, instead of the task level. The subtask level queueing system allows fork–join systems more freedom in how they schedule subtasks, which makes their response time better when compared to split–merge systems. Due to some constraints with the tractability of the model, it only works with subtasks which are exponentially distributed.

In 2014 Pesu and Knottenbelt [103] demonstrated that subtask dispersion can be further reduced with dynamic delays. In their technique any remaining delays are cancelled when a

sibling subtask completes service.

Zander, Leeder and Armitage investigated how unevenly distributed lag in online gaming affects player performance. They discovered that higher lag has negative effect on player performance and that online games can be made mode fair by inserting delays in front of the transmission of data to players having comparatively small amount of lag [144].

2.5.3 Energy Consumption Optimisation

In 1996 in [71] Jennings, Mandelbaum, Massey and Whitt computer the number of servers needed for service by taking into account expected future demand. They modelled the system as a Markovian $M_t/M/s_t$ model. The model includes a constraint to keep the probability of all servers being busy under a given threshold. Work done in 1998 by Adan and van der Wal [3] studied two models which combine *make everything to order* and *make everything to stock* inventory control models. They suggest a combination of the two techniques, which incorporates the good features of both models. In [21] Borst, Mandelbaum and Reiman studied the operator staffing problem in large call centers. Such systems are modelled as a $M/M/N$ queueing system, where N is very large. The paper analyses how to minimise the weighted sum of staffing costs and customer queueing time.

In [143] Yao, Demers and Shenker propose a scheduling model for jobs, which incorporates key aspects of energy minimization. They present an off-line algorithm that computes the minimum energy usage for a group of jobs, while assuming that energy usage per unit time is a convex function. In [17] Bansal, Kimbrel and Pruhs represent a speed scaling model to manage device temperatures, with the constraint that tasks meet their service deadlines. In [106] Pruhs, Uthaisombut and Woeginger minimise average response time of a collection of dynamically released jobs, given a constraint on the amount of energy used.

In [65] Irani, Shukla and Gupta presented two ways to save battery in embedded devices. The first technique is to place the device to sleep. The second way is to use speed scaling. They prove that their algorithm for the control of the system is within a factor of two of the optimal

algorithm. In [48] Gandhi, Gupta, Harchol-Balter and Kozuch use the energy–response time product to study the energy-performance trade-off. They studied the optimality of server farm management policies and present first theoretical results. In [49] Gandhi, Harchol-Balter and Adan consider server farms, which have setup costs in the form of power cost or time delay. The paper derives closed form solutions for mean response time and mean power cost of server farms with setup costs, which enables optimisation of the system.

In 2017 Marin and Rossi showed that power consumption can be reduced in saturated fork–join systems by slowing down subtasks whose service is not time critical [88]. In 2018 Marin, Rossi and Sottana studied how processing power should be split among the service of K tasks in Fork–Join systems. Their overall aim was to minimise the join-queue lengths, which also reduces the expected job service time [89].

2.5.4 State-dependent Service in Queueing Systems

The study of queueing systems that feature state-dependent arrival and services rates was begun by Harris in 1967 [57]. Harris provides the steady state distribution for the number of tasks in the system for a $M/M_n/1$ queueing system, given that the rate of service is $\mu_n = n\mu_1$. The paper also analysed a two state $M/M/1$ queue where the service rate depends on whether the system is empty at the start of service. Shanthikumar derived the Laplace transform of the steady-state waiting time distributions in a state-dependent two-state $M/G/1$ queueing system [117].

Gupta and Rao studied a queueing system, where both rate of arrival and service are determined by the number of tasks in the system. The queue has a finite buffer and they assume that service times are only adjusted when a task begins service. They managed to obtain a distribution for the number of the tasks in the system [53]. Kerner derived a closed form solution of the probability distribution for the number tasks in the system, which takes the amount of time the system is idle as input. The results apply for state-dependent $M_n/G/1$ queues [72].

Hossein and Opher generalised the results of Gupta, Rao and Kerner. In their results for the

$M_n/G_n/1$ queueing system, both arrival and service rate of the system depend on the amount of tasks in the queue. The paper derived both the steady-state and average waiting time for a task in the system [2].

In this section, we conduct a systematic review of the literature of the past ten years. The Habilitation thesis of Wolter, on which this chapter is partly based, only considered the literature until 2007. We therefore review recent trends in model-based analysis of restart, reboot and rejuvenation that have taken place since Wolter's review.

The collection of reference papers for the literature survey was obtained as follows. We searched using the following keywords on Google Scholar: 'restart', 'software rejuvenation', 'checkpointing', and 'reboot'. We went through the search results for each keyword we searched for web page after web page. For each paper, we decided whether they were on-topic, and we continued until relevant papers stopped appearing. We then did a similar exercise, searching for all papers that cited the key literature relevant to this chapter, particularly the author's papers on this topic. In this manner we aimed to have captured the main literature relevant to this chapter over the past decade.

2.6 Trends in Restart Related Research in the Past Decade (2007 - 2017)

Applications	[19, 83, 45, 50, 38, 18, 122, 145, 87, 75, 76, 100]
Implementation	[141, 33]
Modeling & Analysis	[109, 70, 110, 60, 137, 108, 85, 99, 115, 135, 69, 68, 66, 86, 93, 134, 133, 52, 37, 97, 9, 47, 43, 84, 63, 44, 98, 96, 119, 10, 136, 140, 16, 142, 94, 67, 62, 111, 102]
Real Data	[31, 123, 78, 118, 27, 28, 92, 11, 22]
Survey	[7, 29]

Table 2.1: Relevant literature since 2007, categorized.

Table 2.1 list all 63 papers we eventually considered, after a sifting process to be described in the following paragraphs. We have classified the papers according to five broad categories depending on the content of the paper. The categories we used for labeling are: 'Applications',

'Implementation','Modeling & Analysis', 'Real Data' and 'Survey'. We chose these categories because we felt that research papers can almost always be split into one of the categories, and that there is little overlap between the categories.

Adaptivity Metric	[108]
Black-box Restart	[137, 99, 135, 134, 47, 44, 136, 133, 140, 102]
DNA Computing	[60]
Markov Model	[70, 69, 66, 68, 86, 52, 97, 63, 67]
Machine Learning	[119, 10]
Petri Net	[115, 16]
Queueing Model	[109, 85, 43]
Semi–Markov Model	[110, 98, 96, 94, 62, 111, 37]
Stochastic Reward Net	[142, 84]
Supervisory Control Theory	[9]

Table 2.2: Further categorization of papers in 'Modelling & Analysis' category of Table 2.1

The 38 'Modelling & Analysis' papers from Table 2.1 were further divided to sub-categories based on the type of model used in the paper, as shown in Table 2.2. For this table, we constructed the set of labels through an iterative approach, until we felt that the categories were useful in understanding the survey outcomes. First, we categorized based on the type of model used in the paper. Then we included other themes, such as machine learning or the application area, if these dominated the papers. We also removed a number of papers, especially in checkpointing, in which the model actually was a system model, not a model for evaluation or optimisation.

2.6.1 Survey of Research of the Past Decade

We subdivide between modeling and non-modeling papers, first reviewing the papers from Table 2.1 that are in the category 'Modeling & Analysis', we have divided them in groups as shown in Table 2.2. After this we review the non-modelling papers. We note that since Wolter's previous literature survey [139], we have come across two other related interesting literature surveys [7, 30], both focused on rejuvenation. [7] provides a comprehensive system perspective on rejuvenation techniques, while [30] provides a broad discussion on rejuvenation and software aging. Our survey is complementary to the rejuvenation surveys since we include

restart techniques and particularly emphasize modeling aspects.

Survey of Modeling & Analysis Papers

In this section, we consider the modeling and analysis papers from Table. 2.2.

Black box Models. Most directly relevant to this chapter are the 'Black box Restart' papers, in which there is no representation of system structure in the used model. A number of interesting ways of expanding on the results in this chapter have emerged in the past decade. The following papers and a PhD used black box restart to improving mobile offloading [134, 136, 135, 137, 133]. The papers describe a strategy, based only on time elapsed, where several attempts are made to offload a intensive computation to a computing platform better adept at computing the problem. Local computation of the problem is only done if the offloading attempts fail. Okamura et al. optimise the restart interval with the help of empirical data [99]. Compared to [130], it relaxes the assumption of equidistant restart intervals and derives a solution with the help of Laplace Transforms. Gagliolo et al. study the k-armed bandit problem to decide how much information to gain about the service time distribution in order to determine an optimal policy for restarting [47]. In other work, the metrics of interest are more complex than in this chapter. [44, 140] investigate how system performance is affected when multiple users in the system use restart to improve their own performance. [102] has a case study on how the performance of a split–merge system can be improved with respect to task response time, subtask dispersion and energy consumption.

Markov Models. Markov models have been used by a number of authors. As we have seen, Markov models are limited in that exponential delays cannot represent aging, however, states can be used to represent levels of degradation. For instance, [70] derives various performance metrics for the machine repair problem with reboot and imperfect coverage under the care of a single unreliable server, while [69, 68, 66] derive results for the availability characteristics of a repairable system where time-to-failure, time-to-repair and time-to-delay are all exponentially distributed. [63] computes steady state probabilities of a system with switching failure, reboot delay and repair pressure. [67] presents a model for restart policies in cluster systems. The

paper derives various performance indicators such as the availability, mean time to failure and down time cost. [86] presents a rejuvenation strategy where both the virtual machine and the virtual machine monitor are rejuvenated at the same time to minimise downtime. A different use of states is seen in [52], which obtains an analytical model for the communication between two machines with a finite buffer. When the buffer becomes full the machine is forced to remain idle until the buffer becomes empty. [97] introduces a Markov Decision Process for rejuvenation, which rewards long run-average of the running process.

Semi-Markov Models. The research in this category all expand on the semi–Markov model to generate a software rejuvenation schedule first presented by [36]. [37] uses a semi–Markov decision process and reinforcement learning to derive an optimal software rejuvenation policy to maximise the steady-state system availability, whereas [110] develops a fast estimation algorithm for the optimal periodic rejuvenation schedule. [98, 96] present an opportunity-based software rejuvenation technique, which can only rejuvenate during given periods in time and [96] investigates optimal rejuvenation policies when system failures are correlated. [62] expands the semi–Markov decision process to take into account the unreliablility of the rejuvenation process and [111] presents a statistically non-parametric adaptive algorithm to estimate the optimal preventive rejuvenation schedule.

Queueing Models. A few papers studied the effects of restart using queuing models, to analyze delays or blocking probabilities. [109] is a study of the effect of client-side restart and server-side rejuvenation policies on system and service availability. [85] provides analysis of the optimal stopping problem for software rejuvenation in a deteriorating job processing system and [43] contains a theoretical investigation into a network of queues with multiples classes of customers and restart signals.

Petri and Stochastic Reward Nets. To represent more complex system structure and behaviour, Petri Nets and its variants are shown to be particularly intuitive in a number of recent papers. [16] describes a model for a phased-mission system with software rejuvenation. It analyses the impact of software rejuvenation on the success probability and completion time distribution of the mission. An investigation of effects of the frequency of rejuvenation on

system utilization and service availability can be founds in [115]. [142] investigates software rejuvenation policies in cluster computing systems where dependency exists between nodes, with the use of stochastic reward nets. [84] also uses Stochastic Reward Nets in a study on three different types of rejuvenation policies in virtual machines.

Machine Learning and Other Developments. Machine learning approaches related have also been pioneered in recent years. To inform proactive restarts, [119] uses machine learning to detect anomalies, which cause crashes, while [10] uses machine learning to determine software degradation levels of a system. Three other papers have a strong modeling flavor and do not fit any of the above categories. [108] uses adaptivity metrics to determine the optimal restart rate. [60] studies restarts in the context of DNA computing. [9] presents a method for restarting manufacturing systems with the help of safe states to make sure production goals and specifications are fulfilled during restarts, the paper uses supervisory control theory to model the system.

Survey of Non-modeling Papers

Regarding the non-modeling papers we discuss three categories, namely applications, implementations and the use of real-life system data.

Applications. The most common application of system restart among the papers we reviewed is to improve the performance of various search problem algorithms. Examples of such problems include SAT solvers [19, 18], optimisation algorithms [83, 100, 50], statistics regarding random walks [45, 122, 75, 76] and flow shop scheduling [38]. Shylo et al. performed a study on optimising restarts in Las Vegas algorithms [118], also using real data.

Implementation. Yamakita et al. implemented a phased-reboot of Xen 3.4.1 running para-virtualized Linux 2.6.18, where the aim is to reduce downtime of recovery-based rebooting [141]. Their experiments showed that downtime was 34.3 to 93.6% shorter compared to a normal reboot. Danilkina et al. have created a project called Sfera, which provides a simulation framework for restart algorithms in Service-Oriented Systems [33].

Real Data. Multiple papers describe experiments performed on real systems, where they analyse whether system suffers from software ageing: Android [27], operating systems [28], Java virtual machine [31, 92], cloud computing systems [11, 78, 123]. The papers identified relevant parameters to software ageing such as workload, device configuration, and resource utilization. In addition they confirm that software ageing is a real phenomenon affecting modern software.

2.6.2 Main Points of Review

Plenty of real world use cases can be found for system restart in the real world as witnessed by the examples in the black box model, implementation and application papers. In Sec. 5.2 we mentioned that good use cases for system restart have a high coefficient of variation. This observation holds for the examples in our literature survey.

In addition to restart being helpful in a large amount of use cases, there are many ways to decide when is the optimal time to restart. Many of the Markov model, semi-Markov model, queueing model, Petri and stochastic reward net model papers slightly vary the underlying modelling environment and cover different metrics, which can be used to assess the efficiency of the system. They then introduce restart policies, which provide good results for the respective metrics and models.

The third category of papers that stood out in the literature review were papers that try to estimate the amount degradation of the system, as in try to estimate future errors in computation before the occur. Examples of such papers are found among the machine learning and real data papers.

Chapter 3

Comparison of Existing Techniques to Reduce Subtask Dispersion and Task Response Time in Split–Merge and Fork–Join Queueing Systems

This chapter provides an in-depth analysis of earlier work on minimising subtask dispersion and task response time in split–merge and fork–join queueing systems. More specifically, it provides an overview of the techniques described in the following papers: [126, 128, 127, 103]. Chapter 6 expands on some of the techniques discussed in this chapter.

In addition to introducing existing research to the reader, this chapter also contains three case studies, which are used to evaluate the effectiveness of the techniques we introduce. Parts of this chapter are based on the research performed by the author in [103].

3.1 Minimising Subtask Dispersion in Split–Merge Queueing Systems with Static Delays

This section summarises previous research on minimising subtask dispersion in split–merge systems. The technique was originally presented by Iryna Tsimashenka and William Knottenbelt in [126].

3.1.1 Introduction

This section uses the definition of a split–merge system from Sec. 2.2.1. When tasks begin service each task is split into N subtasks. Under the static delay policy considered here constant precomputed delay is inserted in front of the service of each subtask. The delays are used to minimise subtask dispersion by having longer delays in front of subtasks with shorter expected service time, shorter delays in front of subtasks with longer expected service time, and no delays in front of bottleneck subtasks.

The delays inserted before subtasks are defined by a delay vector $\mathbf{d} = [d_1, \ldots, d_N]$ with two constraints: All delays are non-negative, that is $d_i \geq 0$, and at least one delay is equal to zero, that is $\prod d_i = 0$. The second constraint is used to prevent the system from being unnecessarily delayed.

3.1.2 Calculation of Subtask Delay Vector

The technique uses the subtask dispersion formula from Sec. 2.3.1 and adjusts the subtask service time distributions by inserting delays. The delays are represented mathematically by replacing t with $t - d_i$. The expectation of subtask dispersion for a given delay vector $\mathbf{d}$ can then be calculated with the following formula:

$$\mathrm{E}[\mathrm{Disp}(\mathbf{d})] = \int_0^\infty \Big[1 - \prod_{i=1}^{N}(1 - F_i(t - d_i))\Big] - \Big[\prod_{i=1}^{N} F_i(t - d_i)\Big]\mathrm{d}t \tag{3.1}$$

The first part of the equation is the probability that at least one subtask has completed service, and the second part is the probability that all subtasks have completed service.

The next step is to find a optimal delay vector $\mathbf{d}$, which minimises $\mathrm{E}[\mathrm{Disp}(\mathbf{d})]$. It has been proven in [125] that the subtask dispersion function given in Eqn. (3.1) is convex. The convexity of the underlying cost function guarantees that the local minimum produced by a minimisation routine is also the globally optimal solution. The optimisation problem is defined as follows:

$$\mathbf{d}_{\min} = \arg\min_{\mathbf{d}\geq 0} \mathrm{E}[\mathrm{Disp}(\mathbf{d})] \qquad (3.2)$$
$$s.t. \quad \prod_{i=1}^{N} d_i = 0$$

The optimal solution can be calculated with various classical optimisation methods such as Nelder–Mead and Newton's method [126].

3.2 Minimising Trade-Off between Subtask Dispersion and Task Response Time in Split–Merge Queueing Systems

This section summarises previous research on optimising split–merge systems in terms of both task response time and subtask dispersion. The technique was originally introduced in the paper [128] by Iryna Tsimashenka and William Knottenbelt.

3.2.1 Introduction

The technique for minimising the product of subtask dispersion and task response time builds on the previous work in Sec. 3.1, as it uses the dispersion component introduced in it. The task response time component of the cost function uses the theory laid out in Sec. 2.3.3. We assume the definition of a split–merge system given in Sec. 2.2.1.

While the optimisation part in what follows is similar to the technique in Sec. 3.1, the logic for determining subtask delays differs. The way to optimise task response time alone is to use no delays, but subtask dispersion alone is minimised with non-zero delays. As a result, there is contention between the two metrics and the optimal solution minimises the product of the two metrics represents a compromise between the two extremes.

As before, the delays inserted in front of subtasks are defined by a delay vector $\mathbf{d} = [d_1, ..., d_N]$ with two constraints: All delays are non-negative $d_i \geq 0$ and at least one delay is equal to zero $\prod d_i = 0$ to prevent unnecessary idle time.

3.2.2 Calculation of Subtask Delay Vector

The technique combines the ideas presented in Sec. 2.3.1, 2.3.2 and 2.3.3, adjusting the subtask service time distributions by inserting the predefined delays into the system. The expectation of the trade–off product between subtask dispersion and task response time for a given delay vector $\mathbf{d}$ can be computed as:

$$\mathrm{T}(\lambda, \mu, \mathrm{X}_{(\mathrm{N})}, \mathbf{d}) = \mathrm{E}[\mathrm{Disp}(\mathbf{d})] \times \mathrm{E}[\mathrm{Resp}(\lambda, \mu, \mathrm{X}_{(\mathrm{N})})] \tag{3.3}$$

which, when expanded, becomes:

$$\mathrm{T}(\lambda, \mu, \mathrm{X}_{(\mathrm{N})}, \mathbf{d}) = \left[\int_0^\infty 1 - \prod_{i=1}^{N} F_i(t-d_i) - \prod_{i=1}^{N} (1 - F_i(t-d_i)) \mathrm{d}t \right] \left[\frac{\rho + \mu\lambda \mathrm{Var}[\mathrm{X}_{(\mathrm{N})}]}{2(\mu - \lambda)} + \mu^{-1} \right] \tag{3.4}$$

where

$$\mathrm{Var}[\mathrm{X}_{(\mathrm{N})}] = 2\left(\int_0^\infty t(1 - \prod_{i=1}^{N} F_i(t-d_i)) \mathrm{d}t \right) - \left(\int_0^\infty 1 - \prod_{i=1}^{n} F_i(t-d_i) \mathrm{d}t \right)^2 \tag{3.5}$$

Task response time of the system is computed with the Pollaczek-Khintchine formula. $\mathrm{Var}[\mathrm{X}_{(\mathrm{N})}]$ is the variance of task completion time, which is also equal to the variance of completion time of the last subtask.

The next step is to find a delay vector $\mathbf{d}$ which finds a good local minimum for the cost function shown in Eqn. (3.4). Unfortunately, the trade–off product between subtask dispersion and task response time in Eqn. (3.4) is not necessarily convex and therefore it is not always possible to find the global minimum (at least not in a straightforward manner). The minimisation problem can be expressed as an optimisation problem in the following way:

$$\mathbf{d}_{\min} = \arg\min_{\mathbf{d}\geq 0} \mathrm{T}(\lambda, \mu, \mathrm{X}_{(\mathrm{N})}, \mathbf{d}) \qquad (3.6)$$
$$s.t. \quad \prod_{i=1}^{N} d_i = 0$$

We remark that this technique can be further optimised by varying the set of delays added before service of subtasks based on the current queue length. Further details can be found in Chapter 6, where a queue-dependent delay optimisation of the split–merge system is presented.

3.3 Minimising Subtask Dispersion in Fork–Join Queueing Systems with Dynamic Delays

This section summarises previous research into minimising dispersion in fork–join systems with dynamically-controlled delays. The method was originally introduced in the paper [127] by Iryna Tsimashenka and William Knottenbelt.

3.3.1 Introduction

This section describes a technique for minimising subtask dispersion and task response time in fork–join queueing systems. More information on fork–join systems can be found in Sec. 2.2.2. The technique uses the subtask dispersion formula that was also used in the previous two methods. However, it is applied in a slightly different way. This is, because queueing in a fork–join system happens at the subtask level. As opposed to at the task level as is the case in split–merge systems.

There are two reasons as to why this technique is able to produce better performance. In the fork–join system subtasks of the next task can begin service before the previous task has fully completed service, hence speeding up processing. Secondly, the dynamic nature of the algorithm allows subtasks to begin service immediately after one of their sibling subtasks has finished service, which also speeds up processing and improves subtask dispersion. Subtask dispersion is improved, because delaying is harmful to subtask dispersion once a sibling-subtask has finished service.

The performance of fork-join systems are harder to optimise mathematically. There are no known formulas for computing subtask dispersion and task response time of fork–join systems in the general case. As a result the work assumes exponentially distributed subtask service times.

3.3.2 Computation of Subtask Delay Vectors

The fork–join system consists of N parallel heterogeneous servers. We assume that the interarrival time distribution of incoming tasks and subtask service times are exponentially distributed. The cumulative probability distribution of subtask service times are as follows:

$$F_i(t) = 1 - e^{-\lambda_i t} \tag{3.7}$$

The completion time of $q + 1$ exponentially distributed subtasks is given by the Erlang distribution $\mathrm{Erl}_{\mathrm{X}}(q + 1, \lambda)$, if the exponential tasks all have the same λ. The Erlang distribution is the result of convolving the $\exp_{\mathrm{X}}(\lambda_i)$ distribution $q_i + 1$ times.

Each time a subtask completes service the following procedure is used to reassign delays of subtasks that have not begun service:

1. If one or more sibling subtasks has completed service, the subtask begins service straightaway.

2. The delay for ith subtask is calculated as follows:

(a) Construct a vector $\mathbf{q}$, where elements are equal to the queueing positions of sibling subtasks (0: in service, 1: first in the queue and so on)

(b) Fill in Eqn. (3.9), where F_i is the cumulative probability distribution function of $\mathrm{Erl}(\lambda_i, q_i)$.

(c) Finally, to compute the optimal delay vector $\mathbf{d}$ solve the minimisation problem presented in Eqn. (3.8)

$$\mathbf{d}_{\min} = \arg\min_{\mathbf{d}' \geq 0} \mathrm{E}[\mathrm{Disp}(\mathbf{d}')] \qquad s.t. \quad \prod_{i=1}^{N} d_i = 0 \tag{3.8}$$

where

$$\mathrm{E}[\mathrm{Disp}(\mathbf{d})] = \int_0^\infty 1 - \prod_{i=1}^{N}(1 - F_i(t - d_i)) - \prod_{i=1}^{N} F_i(t - d_i)\mathrm{d}t \tag{3.9}$$

The memoryless property of the exponential distribution can be used to save on computation time. As a result of the memoryless property, the exponential distribution does not change when first n time units of the distribution are removed and the remaining parts are scaled to have the probability to be equal to one.

The memoryless property in our case means that, the optimal delays are the same irrespective of if some subtasks have begun service or not and the only variable that matters is the queue position of each subtask. Therefore it is possible to have a cache in the form $\mathbf{q} \rightarrow \mathbf{d}'$. We remark that this delay scheme could be improved in optimising subtask dispersion and task response time in fork–join queues. The delay scheme utilises the knowledge of subtasks finishing, by cutting delays of sibling subtasks to zero. However, this knowledge is not used when computing the optimal delays. This is taken into account in the technique presented below in Sec. 3.4. The technique also fails to fully utilise the flexibility fork–join systems, which allows service of subtasks from different tasks in parallel. This is done better in the technique presented below in Sec. 3.4.3.

3.4 Dynamic Subtask Dispersion Reduction in Split–Merge Queueing Systems

This section summarises previous research in minimising subtask dispersion in split–merge systems with dynamically adjusted delays. The method was first presented in the MSc thesis of the author and in the paper [103] by Tommi Pesu and William Knottenbelt.

3.4.1 Introduction

Here we introduce a improved technique for reducing subtask dispersion in split–merge systems. To improve performance with regard to subtask dispersion and task response time a *start work* signal is sent to the sibling subtasks when a subtask completes service. By dynamically adjusted delays we mean that subtask delays can be changed until the subtask begins service. Once servicing of a subtask has started it will proceed uninterrupted until completion. The *start work* signal is able to reduce both task response time and subtask dispersion when compared against algorithms described in earlier sections.

3.4.2 Calculating Subtask Dispersion in Dynamic Split–Merge Systems

The technique described here readjusts sibling subtask delays to zero when a subtask completes service. As a result, it is not possible to use equation from Sec. 2.3.1 to accurately compute subtask dispersion of the system. A new formula was derived in [103] is shown below:

$$\mathrm{E}[\mathrm{Disp}(\mathbf{d})] = \sum_{i=1}^{N} \int_0^\infty T(i,t,\mathbf{d}) E_r(i,t,\mathbf{d}) dt \tag{3.10}$$

where

$$T(i,t,\mathbf{d}) = f_i(t-d_i) \prod_{j \neq i} [1 - F_j(t-d_j)] \tag{3.11}$$

and

$$E_r(i, t', \mathbf{d}) = \int_0^\infty [1 - \prod_{j \neq i} G_j(t, t', d_j)]dt \tag{3.12}$$

with

$$G_j(t, t', d_j) = \begin{cases} F_j(t) & \text{if } t' < d_j \\ F_j(t - (t' - d_j)|t > 0) & \text{otherwise} \end{cases} \tag{3.13}$$

The rest of the optimisation proceeds in the same way as the previously described techniques. The final step of the procedure is to perform a search in the optimisation space to discover a suitable delay vector $\mathbf{d}$, which minimises subtask dispersion.

The delay vector $\mathbf{d}$ indicates how long the service of subtasks are delayed at each server. Eqn. (3.10) is used to compute the subtask dispersion of the system for the delay vector $\mathbf{d}$. The formula is composed of two main terms: $T(i, t, \mathbf{d})$ and $E_r(i, t, \mathbf{d})$.

The function $T(i, t, \mathbf{d})$ returns the probability that server i is the first server to finish service at time t when using delay vector $\mathbf{d}$. The definition of the function $T(i, t, \mathbf{d})$ can be seen in Equation (3.11). The second function $E_r(i, t, \mathbf{d})$ computes the expected time required to service all the remaining subtasks, given that the first task i completed service at time t and the system is using the delay vector $\mathbf{d}$.

$G_j(t, t', d_j)$ is the probability distribution of completion time of server j. If $t' < d_j$ the first subtask finished before the delay d_j ran out and service is begun immediately as a result of the *start work* signal. Otherwise, the distribution is readjusted to take into account that service did not complete in the first $t - d_j$ (time units).

The optimal delay vector $\mathbf{d}_{\min}$ can be found by solving the following equation:

$$\mathbf{d}_{\min} = \arg\min_{\mathbf{d}' \geq 0} \mathrm{E}[\mathrm{Disp}(\mathbf{d}')] \tag{3.14}$$
$$s.t. \quad \prod_i d'_{\mathrm{i}} = 0$$

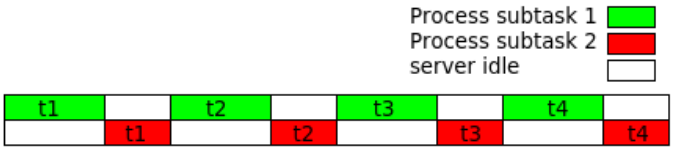

Figure 3.1: Dynamic subtask dispersion in a two server example with service time distributions being: $\mathrm{Exp}(\lambda = 1), \mathrm{Exp}(\lambda = 2)$

3.4.3 Fork–Join Extension for Exponentially-Distributed Service Times

The technique presented right above in Sec. 3.4.2 is in general very effective in reducing subtask dispersion. However, an optimisation can be spotted when service times are exponentially distributed. In our test cases the algorithm begins service on a subset of subtasks straight away and the rest of the subtasks will wait until one subtask completes. This can be observed in Fig. 3.1, and as a result the technique could be improved in terms of task response time. The task response time of the system can be improved significantly if the processing of subtasks is "squashed" to be performed in a fork–join fashion. In practice this happens by starting the bottleneck subtask(s) and simultaneously processing all the unfinished subtasks of the previous task. An example of "squashing" is shown in Fig: 3.2.

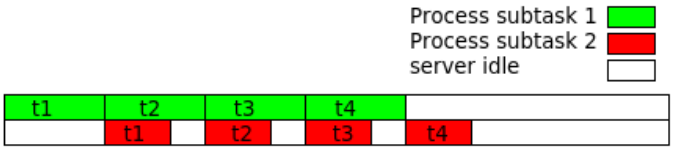

Figure 3.2: Squashing subtasks in a two server example with service distributions being: $\mathrm{Exp}(\lambda = 1), \mathrm{Exp}(\lambda = 2)$

We remark that the technique described above does not cover all the possible cases. For example, if the expected service time of the remaining subtasks is larger than the expected service time of the first subtask to complete, the process can add a large penalty to subtask dispersion. A better strategy might be to delay the service of the bottleneck subtask(s) of the next task until the expected task completion time of the current task has dropped below a specific threshold.

3.5 Numerical Comparison of Techniques

This section presents three case studies, which compare the efficiency of the different algorithms to reduce subtask dispersion and task response times of split–merge and fork–join systems. The optimisation algorithms were implemented by the author; some of the results have been presented before in [103], which is a paper by the author written before the PhD thesis.

The metrics for split–merge based techniques 1, 2, 3 and 6 are evaluated analytically. The metrics for fork–join based techniques 4, 5 and 7 are simulated with 5 replicas of 5 million tasks each. The performance of each system was measured in terms of task response time, subtask dispersion and the trade-off metric which is a product of the two.

3.5.1 Analysed Techniques

Method 1 represents a vanilla split–merge system where no subtask delays are applied. The tasks are processed one at a time, meaning the service of the next task does not begin before all the previous task's subtasks have finished.

Method 2 represents a split–merge system where subtask dispersion is minimised according to the technique described in Sec. 3.1.

Method 3 represents a split–merge system where trade-off is minimised according to the technique described in Sec. 3.2.

Method 4 represents a vanilla fork–join system with no delays applied in front of subtasks. Each task is split into subtasks and the subtasks queue individually for their respective server.

Method 5 represents a fork–join system with dynamically controlled delays algorithm [127]. This algorithm uses interruptions to start processing of sibling subtasks once a subtask finishes. The system uses the formula for subtask dispersion given in Sec. 3.3.

Method 6 represents a split–merge system that uses dynamic subtask dispersion technique from Sec. 3.4.2 to calculate initial delays, which are then removed once a sibling subtask fin-

ishes service. Once the first subtask has completed the two remaining subtasks begin service immediately.

Method 7 modifies the split–merge system found in method 6 to derive a fork–join system in which idling time is squeezed according to the principles of Sec. 3.4.3.

3.5.2 Case Study 1

The methods used in this case study are the same as the methods used above. The interarrival time of new tasks entering the system is exponentially distributed with $\lambda = 0.4$ tasks per time unit. The service time densities of the parallel servers are independent of each other and are given below:

$$\begin{aligned} X_1 &\sim \text{Exp}(\lambda = 1) \\ X_2 &\sim \text{Exp}(\lambda = 2) \\ X_3 &\sim \text{Exp}(\lambda = 2) \end{aligned}$$

Method	Task response time (time units)	Subtask dispersion (time units)	Trade-off (time units)^2
1	2.315	1.083	2.508
2	2.777	1.038	2.882
3	2.315	1.083	2.508
4	1.913	1.627	3.114
5	2.227	1.099	2.447
6	4.667	0.750	3.500
7	2.586	0.921	2.381

3.5.3 Case Study 2

Here the interarrival time of new tasks entering the system is exponentially distributed with $\lambda = 0.75$ tasks per time unit. The service time densities of the parallel servers are independent

of each other and are given below:

$$
\begin{aligned}
X_2 &\sim \text{Erl}(n = 5, \lambda = 5) \\
X_3 &\sim \text{Uni}(a = 0.2, b = 0.5) \\
X_1 &\sim \text{Exp}(\lambda = 3)
\end{aligned}
$$

Methods 5 and 7 used in the first two case studies only support exponentially distributed service times and are therefore left out of this case study.

Method	Task response time (time units)	Subtask dispersion (time units)	Trade-off (time units)^2
1	3.144	0.823	2.587
2	11.154	0.511	5.698
3	3.645	0.654	2.385
4	2.922	2.702	7.896
5	N/A	N/A	N/A
6	∞	0.460	∞
7	N/A	N/A	N/A

3.5.4 Discussion

Method 6, which utilises the *start work* signal is the best method for minimising subtask dispersion in the two case studies. The use of the *start work* signal is especially efficient at minimising subtask dispersion when there is one large bottleneck subtask and the other subtasks have a short service time, As can be seen in the Case Study 1.

In Case Study 2 (Sec. 3.5.3) the task interarrival rate λ is bigger than the task service rate μ of **Method 6** and therefore the task response time and trade-off metrics receive infinite values. To be able to use the technique with a finite service time interarrival rate λ needs to be either lowered or techniques further discussed in Chapte. 6 could be used.

When the end user wants to minimise the trade-off between subtask dispersion and task response time the best method according to our first case study is **Method 7**. The technique is able to leverage both the *start work* signal to improve subtask dispersion and the squeezing of service to improve task response time of the system. However, the technique does not work with non-exponential service times. The best technique to use in case study 2, which includes non-exponential service times **Method 3**.

Method 7 points at an interesting research avenue into developing a technique to minimise the trade-off product in general fork–join systems. This is because the method shows that the fork–join system can be combined with the *start work* signal to produce superior results. Especially as **Method 7** is just a rough concept to demonstrate that this can be done in fork–join systems with exponentially distributed service times, Instead of a fully fledged optimal algorithm.

It can be sen from the case studies, that task response time is increased when delays are added to task service and sometimes the systems become unstable as a result. The system is stable when $\lambda < \mu$. More ways to further improve the performance of split–merge systems are introduced in Chapter 6. In context of system stability the work on state-dependent delays are especially useful. As they provide a smarter way to add delays so that the stability of the system is preserved.

Chapter 4

Optimising Hidden Stochastic PERT Networks

This chapter introduces a technique for minimising subtask dispersion in hidden stochastic PERT networks. The technique improves on existing research in two ways. Firstly, it enables subtask dispersion reduction in DAG structures, whereas previous techniques have only been applicable to single-layer split–merge and/or fork–join systems. Secondly, the exact distributions of subtask processing times do not need to be known, so long as there is some means of generating samples. The technique is further extended to use a metric, which trades off subtask dispersion and task response time.

4.1 Introduction

Project **E**valuation and **R**eview **T**echnique (PERT) is a widely used scheduling technique in industry [4, 116, 46]. PERT networks are DAG (Directed Acyclic Graph) structures. The DAG defines restrictions on the order which activities must be serviced in. In *stochastic* PERT, the service times of activities are represented by probability distributions instead of numerical constants. *Hidden* in our context indicates that the user does not know the graph of the PERT network. A more thorough introduction to PERT Networks can be found earlier in the

dissertation in Sec. 2.2.3.

In this chapter, effectiveness of PERT networks are analysed with respect to two criteria: subtask dispersion [126, 103] (difference in time between the subtask that completes first and the subtask that completes last) and task response time [58, 131] (time for all the subtasks of the task to complete). In the case where an end user considers both subtask dispersion and task response time to be important, a trade-off metric can be used [128]. Task response time and subtask dispersion were chosen as metrics to analyse the performance of our system, because the combination of the two metrics match quite closely with what the end user finds important in the examples below.

Minimising subtask dispersion in hidden stochastic PERT networks is useful in scenarios where information is distributed to multiple competing parties and receiving the information before others gives an advantage to one party. An example of this happens in online games that require fast reflexes. Players who have a lower than average lag have an advantage, as they are able to react first to in-game events [144].

In the financial markets it is advantageous to buy from multiple exchanges when executing large orders, as this allows one to tap into more liquidity. However, if the order execution is done in a way that causes high subtask dispersion, then high frequency traders are able to transmit the order details between exchanges, buy up the stock and then moments later resell it at a slightly higher price to the original buyer [23].

Network protocols TCP and UDP, which are the underpinnings of the modern internet do not provide time guarantees for packet delivery. For example, UDP packets might never arrive. On the other hand the arrival of TCP packets might be delayed due to error correction and congestion control [42].

As a result, it makes sense to treat the routing structure between computers in a network as a hidden stochastic PERT network due to the inability of TCP and UDP to offer time guarantees. In this case, the user is typically not aware of its structure or the performance characteristics of individual activities (in this case sending a message between two routers). Here the server's *task*

is to broadcast a message to the clients. This comprises in turn several *subtasks*, each of which involves the delivery of the message to one of the clients. The server wishes that all the clients receive the information with a low subtask dispersion and a low task response time. The server can control subtask dispersion and task response time by adding delays to the transmission of messages. However, as the server does not have control of the whole network, it can only add delays to a limited set of routers.

This chapter presents a new technique for minimising subtask dispersion – or a trade-off product metric involving subtask dispersion and task response time – in hidden stochastic PERT networks. The technique approximates subtask dispersion and task response time by simulation for a given set of added delays – a decision made because analytical techniques such as [59] cannot be applied when specifics of the underlying PERT network are unknown. The Covariance Matrix Adaptation Evolution Strategy (CMA-ES) algorithm [56] further discussed in Sec. 2.4.3, which is noted for its ability to cope with noise, is used to minimise the approximation function.

4.2 Preliminaries

This section contains a brief explanation of how the performance of PERT networks is measured.

4.2.1 Task Response Time

Task response time measures the time it takes to service all activities. As the user often does not know exact details of the topology or service time distributions of activities in the PERT network, the user is not able to construct a analytical solution to the problem. Therefore, in this chapter, task response time is calculated via Monte Carlo simulation. As there is no queueing in PERT networks, the task response time of a system is the same as the task service time.

4.2.2 Subtask Dispersion

For a stochastic PERT network, subtask dispersion is defined as the difference in time between the subtask that was last to complete and the subtask that was first to complete. It is possible to calculate subtask dispersion analytically, if the exact details of the PERT network are known, as shown in [103, 59].

However, we consider settings where the exact topology and service time distributions are not known. Therefore, calculation of subtask dispersion is simulated via Monte Carlo method, as was the case for task response time.

4.2.3 Trade-off Metric

It is possible to measure the overall performance of the system with a trade-off metric defined as the product of subtask dispersion and task response time [128], i.e.

$$T(\mathbf{d}) = \text{Disp}(\mathbf{d})\text{Resp}(\mathbf{d}) \tag{4.1}$$

Here, we extend this with a weight $0 \leq \alpha \leq 1$, which indicates the relative importance of the two metrics:

$$\text{T}(\mathbf{d}, \alpha) = \text{Disp}(\mathbf{d})^{1-\alpha}\text{Resp}(\mathbf{d})^{\alpha}, 0 \leq \alpha \leq 1 \tag{4.2}$$

4.3 Method to Optimise PERT Networks

The approach to use simulations to optimise the trade-off between subtask dispersion and task response time differs from past research. Past research has focused on single-level parallel processing systems and have assumed full knowledge of system parameters to construct analytical functions, which are minimised [126, 128, 127, 103]. However, these techniques are manifestly unsuitable for application in the context of hidden PERT networks.

Our new technique not only works on more general tasks, which have a graph structure, whereas the existing techniques only work on simple structures such as split–merge and for–join systems. Also the technique does not require knowledge of the underlying probability distributions of activity service times. Instead, this technique only needs to be able to randomly sample the probability density function. It is assumed that the user has control over a subset of the activities in the system S_c and the user is able to apply non-negative delays before the processing of these activities.

4.3.1 Optimisation Procedure

The optimisation procedure has two main phases. Firstly, subtask dispersion – for a given set of n subtasks and delays for activities in S_c – is calculated via repeated simulation of t tasks. When t is increased, accuracy increases at the expense of computation effort and *vice versa* when it is reduced. Secondly, CMA-ES is applied on the approximation function to find optimal delays. CMA-ES was chosen due to it being considered something of a standard in black box optimisation [55], which performs better in the context of non-convex optimisation than many classical methods. This is an important feature as the estimates generated via random sampling contain noise.

Where it is desired to trade off subtask dispersion and task response time, we can similarly apply CMA-ES to minimise the penalty function $\mathrm{T}(\mathbf{d}, \alpha)$, i.e.

$$\begin{aligned} \mathbf{d}_{\min} &= \arg\min_{\mathbf{d}\geq 0} \mathrm{T}(\mathbf{d}, \alpha) \\ s.t. \quad & \prod_{i=1}^{N} d_{\mathrm{i}} = 0 \end{aligned} \tag{4.3}$$

in which the two main components of $T(\mathbf{d})$ are generated as averages of Monte Carlo simulations of t tasks.

The function is used to generate subtask dispersion and task response time estimates for the CMA-ES. The PERT network can be seen in Fig. 4.1. In practice, this function would be

generated by repeatedly trying to send a messages between a server and clients and then measuring subtask dispersion and task response time. The penalty function is then minimised.

4.3.2 Validation of Optimisation Procedure

The method described above was validated by simulating delays for various split merge-systems, which can be solved analytically [126]. The CMA-ES was supplied with a black-box function that estimated the task response time and subtask dispersion of the split merge-systems for a given set of delays. The CMA-ES was then able to replicate the analytical results with a good degree of accuracy.

Only subtask dispersion is validated as split–merge systems potentially have to queue to enter service. The task response time of a PERT network would be equal to the task service time of a split–merge system.

Case Study 1

Consider a system where subtask service times are distributed in the following way:

$$\begin{aligned} X_1 &\sim \text{Exp}(\lambda = 5) \\ X_2 &\sim \text{Erl}(n = 2, \lambda = 3) \\ X_3 &\sim \text{Uni}(a = 0.2, b = 0.5) \end{aligned}$$

given $t = 10^3$ and 10 repeats results in a subtask dispersion of 0.451 with a variance of 3.6705×10^{-6}, while the optimal subtask dispersion is 0.448. A smaller $t = 100$ resulted in a subtask dispersion of 0.452 and a variance of $8.443 \times 10^{-}6$.

Case Study 2

Consider an another system where subtask service times are distributed in the following way:

$$\begin{aligned} X_1 &\sim \text{Exp}(\lambda = 1) \\ X_2 &\sim \text{Exp}(\lambda = 5) \\ X_3 &\sim \text{Exp}((\lambda = 10) \end{aligned}$$

given $t = 10^3$ and 10 repeats results in a subtask dispersion of 0.788 with a variance of 9.195×10^{-6}, while the optimal subtask dispersion is 0.783. A smaller $t = 100$ resulted in a subtask dispersion of 0.795 and a variance of $7.922 \times 10^{-}5$.

Validity of Results

The optimisation routine we presented above achieves similar results as the original analytical technique [125, 128, 127]. The results also become more accurate as t is increased, as seen when comparing $t = 100$ and $t = 10^3$.

4.4 Results

In this subsection we apply the technique introduced above in the context of the hidden stochastic PERT network of Figure 4.1. It has three sources $(1, 2, 3)$ and two sinks $(9, 10)$. The user-controlled activities have distributions f_1, f_2 and f_3. Information regarding the various probability distributions can be found in Sec. A. The service time of activities f_i are distributed

as follows:

$$f_i = \begin{cases} \text{Exp}(\lambda = 0.2) & i = 1 \\ \text{Exp}(\lambda = 0.5) & i = 2 \\ \mathcal{N}_F(\mu = 1, \sigma^2 = 0.5)| & i = 3 \\ \text{Uni}(a = 0.2, b = 0.7) & i = 4 \\ \text{Pow}(a = 3) & i = 5 \\ \mathcal{N}_F(\mu = 0.5, \sigma^2 = 1) & i = 6 \\ \text{Pow}(a = 2) & i = 7 \\ \text{Uni}(a = 0.75, b = 0.8) & i = 8 \\ \mathcal{N}_F(\mu = 5, \sigma^2 = 1) & i = 9 \end{cases}$$

The service time duration of the two subtasks T_9 and T_{10} can be derived from the component activities as follows:

$$T_9 = \max(\max(X_1, X_2) + X_4, X_3 + X_5) + X_6 + X_8 \tag{4.4}$$

$$T_{10} = X_3 + X_7 + X_9 \tag{4.5}$$

Where X_i is a random variable generated from the function f_i.

Method 1 Optimised task response time

The results for no delays were calculated using a zero delay vector for 10^7 tasks. Metrics were averaged over 10 runs, which resulted in the following:

Subtask dispersion:	2.841 time units
Task response time:	6.689 time units
Trade-off, $\alpha = 0.5$:	4.358 time units

Method 2 Optimised subtask dispersion.

Here 1 000 samples per measurement point were taken. We ran the CMA-ES algorithm 10 times and the subtask dispersion and task response time for each set of delays was calculated via simulation of 10^7 tasks.

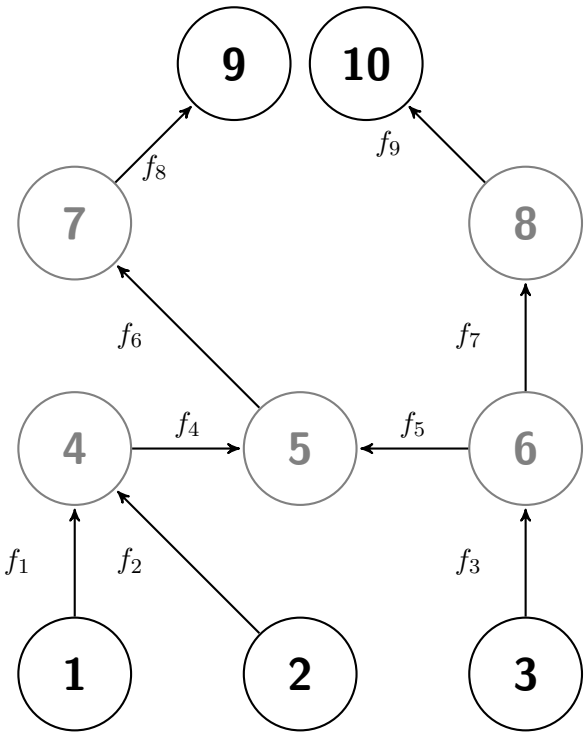

Figure 4.1: PERT network used to analyse results

This resulted in the following:

Subtask dispersion:	1.096 time units
Task response time:	7.833 time units
Trade-off, $\alpha = 0.5$:	2.921 time units

Method 3 Optimised trade-off with $\alpha = 0.5$.

Again 1 000 samples per measurement point were taken. The we ran the CMA-ES algorithm 10 times and the subtask dispersion and task response time for each set of delays was calculated via simulation of 10^7 tasks.

This resulted in the following:

Subtask dispersion:	1.105 time units
Task response time:	7.166 time units
Trade-off, $\alpha = 0.5$:	2.814 time units

Figures 4.2, 4.5 and 4.8 display how task response time, subtask dispersion and trade-off ($\alpha =$

0.5) behave as the set of delay vectors is varied. In each picture one of the three delays of the controlling set S_c is set to 0 and the two others are allowed to vary between 0 and 15 with a stepsize of 0.1. The colouring of the image represents the resulting subtask dispersion (averaged over 1 000 samples) given the corresponding delay vector.

The black dots in the figures show the solutions produced by the CMA-ES technique. In each figure, we show the value of the cost function for the configuration that two of the delays are varied and one is set to zero. Hundred simulations were ran for each two variable space and the results show that the CMA-ES is consistently able to find good candidate solutions for the optimum. This is shown by the fact that the black dots are consistently located in the middle of the red band.

Figure 4.11 shows how metrics vary as α is varied. The trade-off metric approaches near-optimal value when α is between 0.2 and 0.8. When α approaches 1 the lowest possible task response time is found; however, this comes at the expense of subtask dispersion. Similarly, when α approaches 0 the lowest possible subtask dispersion is found; however, this comes at the expense of task response time. This is how you'd expect the three optimisation metrics to behave.

In our example, node 5 is the critical node. It acts as a cork that prevents the completion of T_9 until both sides of the PERT network have started service. Therefore, if the task f_3 is completed much earlier than the tasks f_1, f_2 and f_4, a high subtask dispersion occurs as a result. This is caused by subtask T_{10} completing earlier than subtask T_9.

The top two figures in Fig. 4.2 on the other hand show that a fairly good result of subtask dispersion of around three can be expected if the group of tasks f_1, f_2 and f_4 is serviced before the task f_3.

The graphs suggest that either f_1 and/or f_2 should be delayed for approximately 4 (time units) longer than f_3 to achieve optimal subtask dispersion. The Fig. 4.5 suggests that the bottleneck subtask for task response time is T_{10} as all three figures seem to suggest that dropping delays below four in tasks f_1 and f_2 has no noticeable improved effect on task response time.

Therefore to find a optimal solution under the trade-off metric either f_1 or f_2 should have roughly a four second delay and task f_3 should have no delay.

4.5 Conclusion

In this chapter, we have developed techniques to analyse and optimise parallel queueing systems under real world conditions where it is often difficult to gather precise data about network topology, routing probabilities and service time distributions. The particular formalism we have adopted is the hidden stochastic PERT network. In this context, we have shown how beneficial it can be to apply optimisation algorithms that can cope with non-convex noise data, such as the CMA-ES algorithm.

Our method to optimise hidden stochastic PERT networks appear to find robustly near-optimal solutions for our test cases when optimising for subtask dispersion alone or for a trade-off metric, which balances subtask dispersion and task response time. Figures 4.2, 4.3, 4.4, 4.8, 4.9, 4.10 in total show 600 trial runs, 100 for each figure. Out of the 600 runs there seems to be just one case – in Fig. 4.3 – where the simulation appears to have failed and produced the solution (0,0,0), which is not in the middle of the red band of best solutions. We suspect that this is an issue in the Python implementation of CMA-ES.

There is still lots of research to be performed on the topic. For example, the technique could be improved to obtain confidence intervals for the cost function. Another improvement could be to investigate procedures that use minimal amount of sampling of the underlying probability distribution, as it can be expensive to query it.

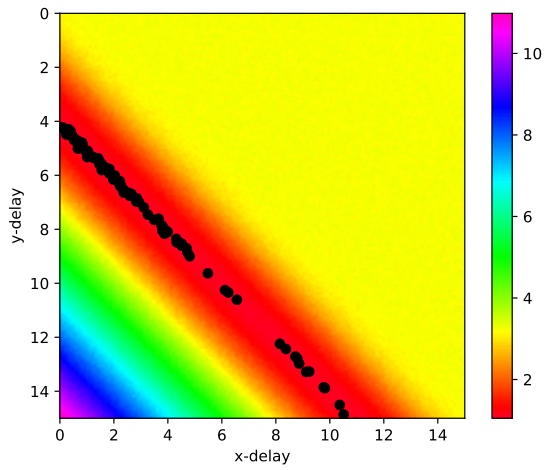

Figure 4.2: Subtask dispersion for delays $(0, x, y)$

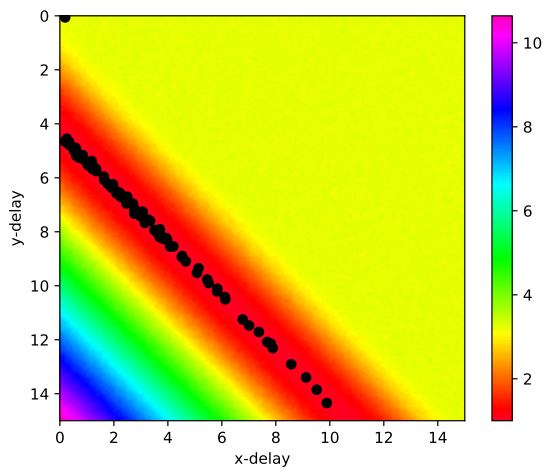

Figure 4.3: Subtask dispersion for delays $(x, 0, y)$

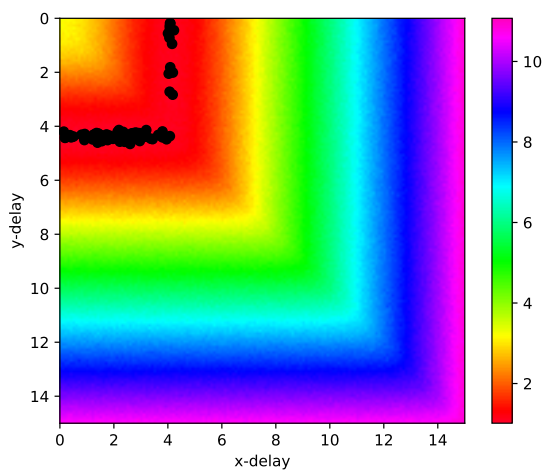

Figure 4.4: Subtask dispersion for delays $(x, y, 0)$

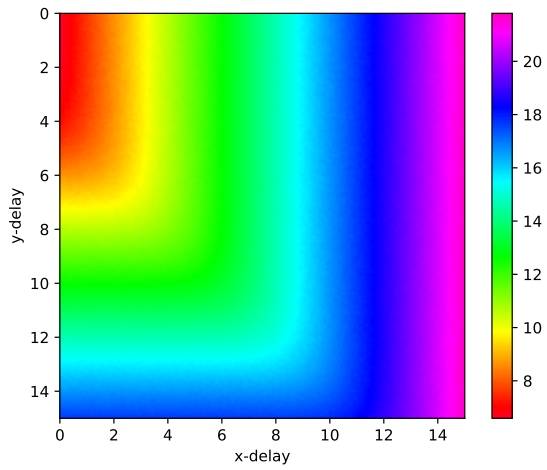

Figure 4.5: Task response time for delays $(0, x, y)$

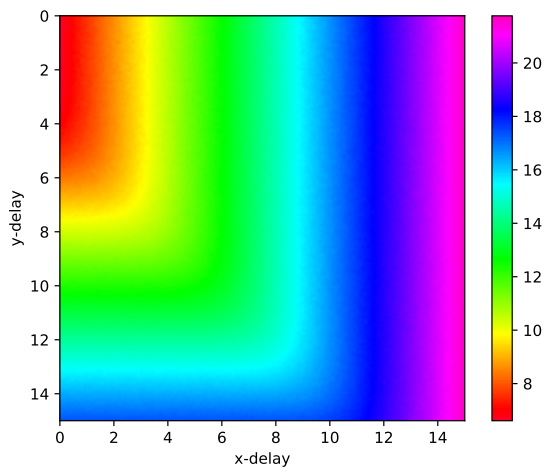

Figure 4.6: Task response time for delays $(x, 0, y)$

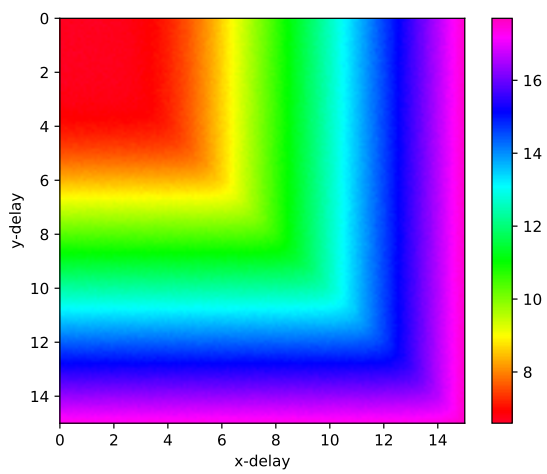

Figure 4.7: Task response time for delays $(x, y, 0)$

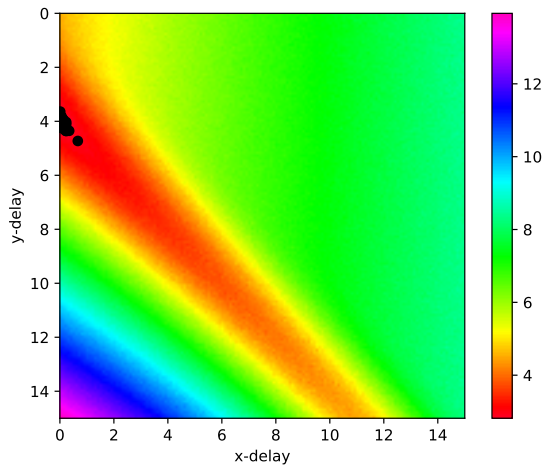

Figure 4.8: Trade-off ($\alpha = 0.5$) for delays $(0, x, y)$

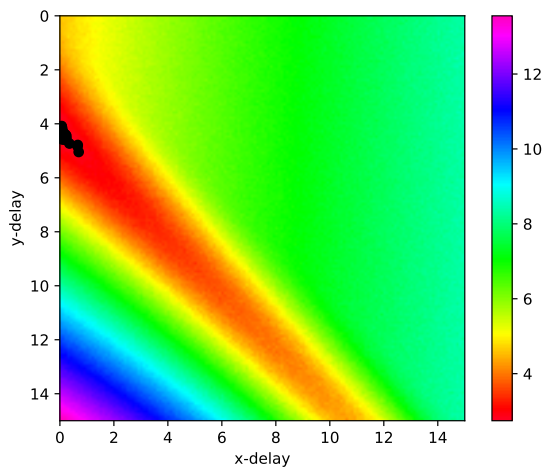

Figure 4.9: Trade-off ($\alpha = 0.5$) for delays $(x, 0, y)$

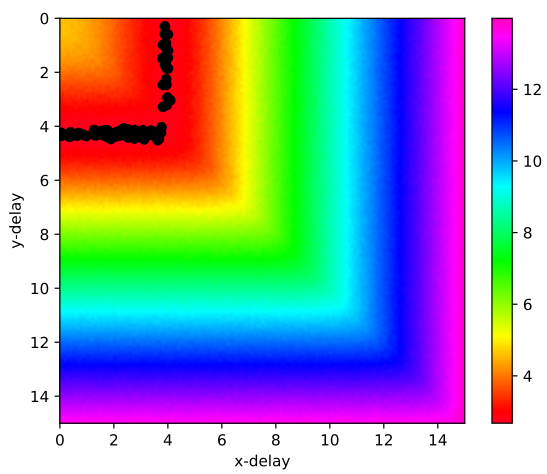

Figure 4.10: Trade-off ($\alpha = 0.5$) for delays $(x, y, 0)$

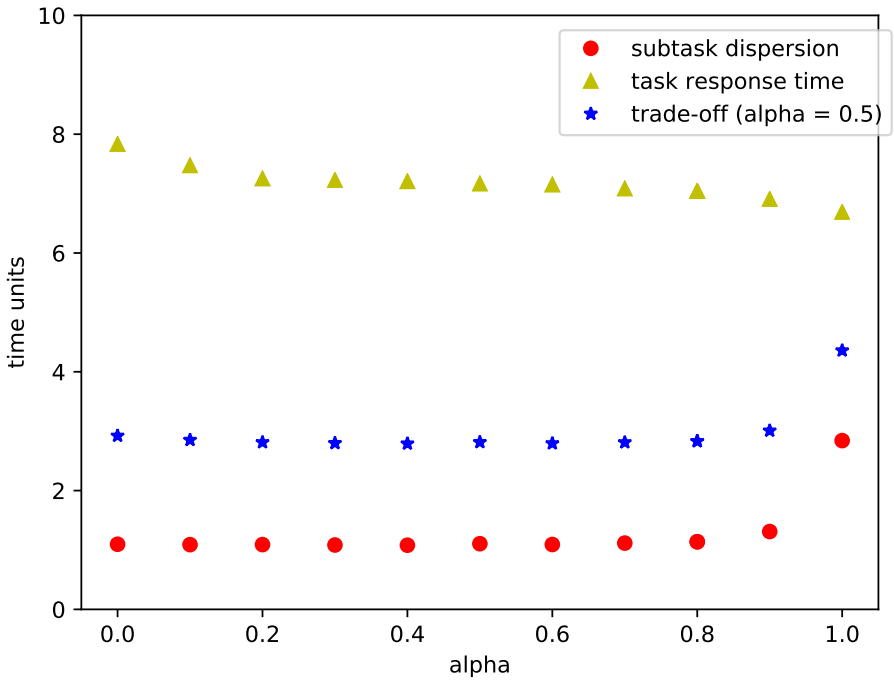

Figure 4.11: Influence of α on optimisation metrics

Chapter 5

Models for Restart and Replication

Thus far the only technique we have considered to improve the overall performance of parallel queueing systems has been delay padding, which is the addition of delays in front of subtasks to improve metrics involving subtask dispersion. In this chapter, we investigate alternative techniques that could be used in conjunction with delay padding. Specifically, the chapter provides an overview of analysis and optimisation of 'restart' and 'replication' strategies, as well as providing an up-to-date survey of the research literature of restart models in general. This chapter is an extension of the book chapter written by Katinka Wolter, Tommi Pesu, Aad Van Moorsel and William Knottenbelt for the Handbook of Software Aging and Rejuvenation (To appear).

The work in this chapter is a prelude to the next chapter. In this chapter we explore restart and replication and how to perform them optimally. In the next chapter results are applied in the context of split–merge queueing systems.

5.1 Introduction

Service restart is the act of abandoning the current service attempt and starting again. Service restart can help reduce the overall service time of a task. For example, when it is believed

that something has gone wrong with the original service attempt. A machine crashing or being overloaded could be considered as an example of a failure.

Service replication is the act of performing the service multiple times in parallel and then picking the fastest service. Service replication can help reduce the overall service time of a task. For example, when a random element is involved with the service of task you are more likely get lucky faster if you are doing multiple attempts simultaneously.

Service replication and restart are not the only possible ways to improve task service time. For example, work has been done on studying speed-scaling as a way to improve task service time.

The first few sections of this chapter, which discuss models, metrics and related quantitative methods for using restart to improve service time, are a much-abridged version of [139].

The chapter is organised as follows. First in Sec. 5.2 we discuss the Hazard Rate, which defines the intensity of completion of a task at time t. Secondly, we discuss the following metrics to assess effectiveness of restart: moments of completion time and the probability of meeting a deadline. Finally we discuss restart through the lens of some example probability distributions, which include exponential, hyper- and hypo-exponential and heavy-tailed distributions.

Sec. 5.3 derives expressions for the moments of completion time under restarts, as well as deriving expressions for the probability of making a deadline.

Sec. 5.4 studies the optimal restart time for both these metrics, again subdivided in optimising moments and deadline probabilities, respectively.

Sec. 5.5 discusses replication strategies, which can be used to reduce service duration of lagging tasks. The three strategies covered include: n-fold replication, hedged requests and tied requests.

Finally Sec. 5.6 concludes the chapter by summarising key aspects of it.

5.2 Metrics to Assess the Benefits of Restarts

It might not seem very obvious that restart could be a way to improve performance. However, many readers have likely used restart to improve performance when browsing the internet. When a web page takes a long time to load, users tend to hit the refresh button to reload the page, because if they do not click refresh the page might never appear. The paper [79] discusses the technical reasons behind the phenomenon and explains that refreshing the page 'overrules' the TCP retransmission timer, which often improves the overall download time.

Tasks that restart is applied to usually complete after a certain time, but the completion time is not fixed. The restart tasks can be formed in several ways. For example, the old instance might or might not be aborted. Similarly, sometimes the task perform after a restart is identical to first attempt, but sometimes parameters or even requirements might change.

The models discussed in this chapter make two assumptions. Firstly, a restart of a task terminates the previous attempt. Secondly, the service time of successive service attempts are statistically independent and identically distributed. The second assumption has been found to match well with reality in the case of downloading web pages [91, 107].

The restart problem is formulated as follows. Let the random variable T define the task service time, with probability distribution function $F(t), t \in [0, \infty)$, and probability density function $f(t), t \in [0, \infty)$. For convenience, but without loss of generality[1] we assume that $F(t)$ is continuous and defined for $t \in [0, \infty)$ and $F(t) > 0$ if $t > 0$. τ defines the restart time[2], and the random variable T_τ denotes the task completion time when an unbounded number of retries is allowed, which means a retry takes place every τ time units until a deadline (if set) has passed or the job has completed. $f_\tau(t)$ and $F_\tau(t)$ are the density and distribution of T_τ.

[1] With more notation and a in-depth investigation of special cases the results of this section can be applied to distributions in finite domains, defective distributions and distributions with jumps.

[2] τ can also refer to restart interval.

5.2.1 Hazard Rate

The hazard rate defines the intensity of completion at time t. To complete a task quickly the hazard rate should be high. A decaying hazard rate suggests restart is beneficial and a increasing hazard rate suggests restart is harmful. The Hazard rate is defined as follows:

$$h(t) = \frac{f(t)}{1 - F(t)} \tag{5.1}$$

5.2.2 Moments of Service Time

The first metrics we discuss are the moments of the completion time of the task $\mathrm{E}[T^n]$, for $n \in \mathcal{N}$. We assume independent identically-distributed service time for each successive retry. When considering the first moment, it is beneficial to restart at time τ when:

$$\mathrm{E}[T] < \mathrm{E}[T - \tau \mid T > \tau]. \tag{5.2}$$

In practice, this means expected completion time of the task $\mathrm{E}[T]$ is less than $\mathrm{E}[T - \tau \mid T > \tau]$, which is the remaining completion time without restart. Eqn (5.2) is only concerned with one restart. In Sec. 5.3 it is shown that Eqn (5.2) is a necessary and sufficient condition for *any* number of restarts to improve the first moment (mean) of service time.

A similar expression can also be defined for higher moments of task completion time $\mathrm{E}[T^n]$:

$$\mathrm{E}[T^n] < \mathrm{E}[(T - \tau)^n \mid T > \tau]. \tag{5.3}$$

In Sec. 5.3 it is shown that for higher moments, the implication that 'a single restart is better than no restart then an unbounded amount of restarts is even better' is not valid. Therefore it is not possible to use condition shown in Eqn. (5.3). For higher moments, repeated restarts

are beneficial if the condition below holds:

$$\mathrm{E}\left[T^n\right] < \mathrm{E}\left[T_\tau^n\right]. \tag{5.4}$$

5.2.3 Probability of Meeting Deadlines

The second interesting metric is the probability to meet a deadline. The completion time distribution with restarts can be treated as a repeated Bernoulli trial. Between successive restarts there is a probability $F(\tau)$ that service is completed. If the deadline d is a multiple of the restart time τ the probability of missing the deadline with restart is given below:

$$1 - F_\tau(d) = (1 - F(\tau))^{\frac{d}{\tau}}. \tag{5.5}$$

Then, the probability of meeting the deadline under restart is

$$F_\tau(d) = 1 - (1 - F(\tau))^{\frac{d}{\tau}} \tag{5.6}$$

The system should only be restarted if the probability of meeting the deadline is higher with restart than without. That is, if

$$F_\tau(d) > F(d). \tag{5.7}$$

5.2.4 Example Distributions

In this subsection, we study various probability distributions and analyse whether they benefit from service restart. We investigate the exponential distribution and two variants of it, namely the hypo- and hyper-exponential distribution, all of which have an exponential tail. In addition we investigate the Pareto distribution, which has a heavy tail.

Exponential Completion Time Distribution

For exponentially-distributed task service times the expected completion times are the following:

$$\begin{aligned} \mathrm{E}\left[T\right] &= \int_0^\infty t\lambda \cdot e^{-\lambda t}\, dt = \frac{1}{\lambda} \\ \mathrm{E}\left[T - \tau \mid T > \tau\right] &= \int_\tau^\infty (t-\tau)\lambda \cdot e^{-\lambda(t-\tau)}\, dt = \frac{1}{\lambda} \end{aligned}$$

As a result the exponential distribution is neutral in terms of restart. In Section 5.2.1 we discussed the hazard rate. The hazard rate can be used to tell if a restart is helpful. The hazard rate function $h(t) = \lambda$ for exponential distributions indicates that restart has no impact on the exponential distribution, as the function is just constant. For exponentially-distributed service completion time the Eqn. (5.7) is equal to:

$$F_\tau(d) = F(d)$$

Hyper-Exponential Completion Time Distribution

A hyper-exponential distribution is a mixture of exponential distributions. Further information can be found in Sec. A.2. The hyper-exponential distribution has a higher coefficient of variation than the exponential distribution and as a result benefits from service restart [114].

Intuitively, distributions which pick values from distinct random variables with a given probability are possibly good candidates for restarting. In other words distribution X_i is picked with probability p_i, where $\sum p_i = 1$. In such distributions as time passes it becomes more likely that you were *unusually unlucky* and were assigned a distribution with a larger than average mean. Restarting service then allows for a new chance to be more in line with the average of the distribution.

Hypo-exponential Completion Time Distribution

A hypo-exponential distribution is the sum of multiple exponentially-distributed functions where the parameter λ of each individual exponential distributions can vary. More details can be found in Sec. A.4. Hypo-exponential distributions (such as the Erlang distribution) have a lower coefficient of variation than the exponential distribution and never benefit from restarts since the condition Eqn. (5.7) never holds.

This is because, multiple subtasks need to be completed sequentially to finish service, where none of the individual subtasks benefit from restart. If you were to restart your amount of completed subtasks would go back to 0, which is equal or worse to the current state. In addition, you would not be in a better position in terms of servicing a specific subtask faster. This is, because they are distributed exponentially and abide by the memoryless property.

Heavy Tails: Pareto Completion Time Distribution

The Pareto distribution is an example of a heavy-tailed distribution. More details on the Pareto distribution can be found in Sec. A.5.

For $a \neq 1$, the expected completion times are the following:

$$\begin{aligned} \mathrm{E}[T] &= \frac{a \cdot b}{a-1} \\ \mathrm{E}[T - \tau \mid T > \tau] &= \int_{\tau}^{\infty} (t-\tau) \frac{b}{(t-\tau)^2}\, dt \\ &= [b \ln(t-\tau)]_{\tau}^{\infty} = \infty. \end{aligned}$$

The expected completion time after waiting for time τ does not converge. Therefore it is always useful to restart a task which has a Pareto-distributed task service time. Indeed unlimited number of restarts is an especially attractive, as it makes the expected service time be finite instead of infinite.

Conclusion

To summarise, distributions with heavy tails often benefit from restart. In such distributions, the tail decreases polynomially in such a way as to leave a large probability mass for the high values of T. Heavy-tailed and related distributions commonly appear when studying applications communicating over the internet [79]. Distributions with exponentially decaying tails in some cases can also exhibit behaviour that benefits from restart, for example hyper-exponential distributions. By contrast, it is never beneficial to restart in the context of hypo-exponential distributions.

5.3 Expressions for Completion Time under Restarts

This section covers the mathematical formulation for moments of task service time and the probability of missing a deadline under restarts. Firstly, for moments of task completion time we study the case of an unbounded number of restarts in Sec. 5.3.1, where restarts occur after a fixed interval. Secondly, we consider the case where restart intervals can vary and be finite in number in Sec. 5.3.2. Finally, we take a look at how the probability of missing a deadline is expressed mathematically in Sec. 5.3.3.

5.3.1 Moments of Completion Time: Unbounded Number of Restarts

We first study the task completion time distribution with an unbounded number of restarts. Meaning that the system will be restarted once every τ time units until the task has completed. We use notation from Sec. 5.2 and introduce an overhead cost c, which is the duration of the restart process. $F_\tau(t)$, the service time distribution with restarts can be presented in terms of $F(t)$ (the service time distribution without restarts) and c as follows:

$$1 - F_\tau(t) = \begin{cases} (1 - F(\tau))^k (1 - F(t - k(\tau + c))) \\ \quad \text{if } \; k(\tau + c) \leq t < k(\tau + c) + \tau \\ (1 - F(\tau))^{k+1} \\ \quad \text{if } \; k(\tau + c) + \tau \leq t < (k+1)(\tau + c) \end{cases} \tag{5.8}$$

for $k = 0, 1, 2, \ldots$. The lower branch of Eqn: (5.8) refers to the idle period c, caused by the cost of restart. The probability density function is given below:

$$f_\tau(t) = \begin{cases} (1 - F(\tau))^k f(t - k(\tau + c)) \\ \quad \text{if } \; k(\tau + c) \leq t < k(\tau + c) + \tau \\ 0 \\ \quad \text{if } \; k(\tau + c) + \tau \leq t < (k+1)(\tau + c) \end{cases} \tag{5.9}$$

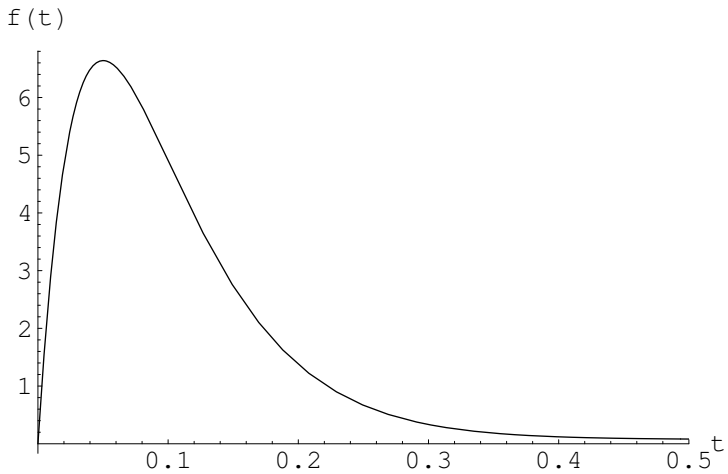

Figure 5.1: The probability density function of the mixed hyper/hypo-exponential distribution. For a single restart the optimal restart interval is 0.25 and for unbounded number of restarts the optimal restart interval is 0.19.

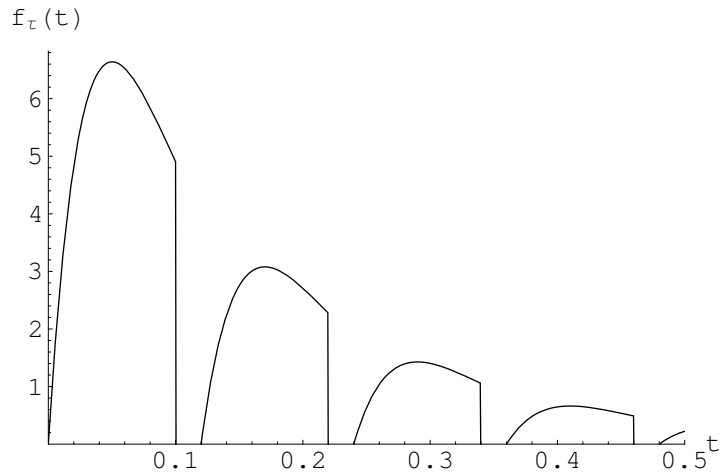

Figure 5.2: The probability density function $f_\tau(x)$ task service completion time with unbounded number of restarts (based on hyper/hypo-exponentially distributed completion time without restarts, with restart time $\tau = 0.1$ and cost $c = 0.02$).

For visualisation of a probability density function T_τ, see Fig. 5.1 and 5.2, for a mixed hyper/hypo-exponentially distributed T. The hyper/hypo-exponential is defined as follows:

$$f(x) = \begin{cases} \text{Erl}(\text{n} = 2, \lambda = 10) & p_1 = 0.9 \\ \text{Erl}(\text{n} = 2, \lambda = 1) & p_2 = 0.1 \end{cases}$$

Fig. 5.1 represents the no restart probability density function of the hyper/hypo-exponential distribution and Fig. 5.2 represents the restart probability density function f_τ of the hyper/hypo-exponential.

Later on we show that for a single restart the optimal restart time is $\tau = 0.25$ and for unbounded restarts the optimal restart interval is $\tau = 0.19$. The expectation of service time with no restarts is 0.190, with a single restart 0.136 and with unbounded number of restarts 0.127.

The moments of service time are defined as follows:

$$M_n(\tau) = \int_0^\tau t^n f(t)dt = \int_0^\tau t^n f_\tau(t)dt. \tag{5.10}$$

The partial moments of T and T_τ are identical between 0 and τ, because the probability density

functions are identical before a restart has taken place.

Theorem 5.1. *The moments* $E[T_\tau^n] = \int_0^\infty t^n f_\tau(t)dt$, $n = 1, 2, \ldots,$ *of the task service time with unbounded number of restarts, restart interval* $\tau > 0$, *and restart penalty time* c, *is shown below:*

$$E[T_\tau^n] = \frac{M_n(\tau)}{F(\tau)} + \frac{1 - F(\tau)}{F(\tau)} \sum_{l=0}^{n-1} \binom{n}{l} (\tau + c)^{n-l} E\left[T_\tau^l\right], \tag{5.11}$$

where $E[T_\tau^0] = 1$.

Proof. For details of the proof see [130]. □

The expected task service time is given by:

$$\mathrm{E}[T_\tau] = \frac{M_1(\tau)}{F(\tau)} + \frac{1 - F(\tau)}{F(\tau)}(\tau + c). \tag{5.12}$$

The first term of Eqn. (5.12) is the expected service time on the interval where service completes. The second term sums up the intervals where service failed to complete. The expression for the other moments can be computed with Eqn. (5.11). For example, $n = 2$ is variance of service time.

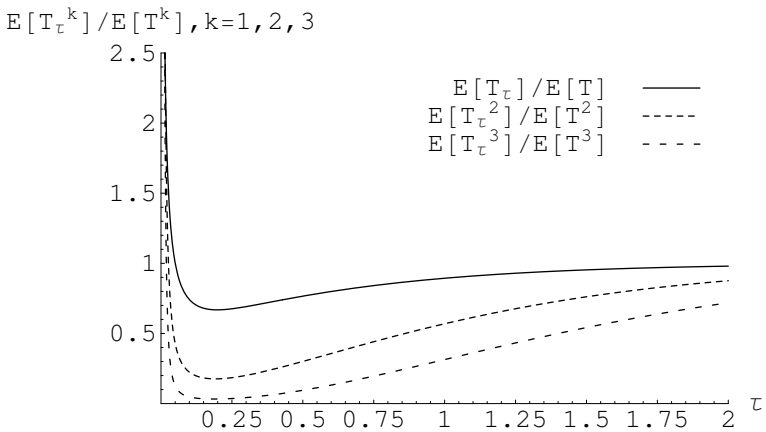

Figure 5.3: Restart time versus the normalised difference between unbounded restarts and no restarts, for the first three moments.

Fig. 5.3 displays the relative improvement when the restart interval is varied in terms of the first three moments of service time. When the number of the measured moments is increased the relative improvement achieved by restart also increases. In addition, it can be seen that if the restart interval is too short it can be costly, but having a longer than optimal restart interval only causes a modest increase in service time.

As a corollary of Theorem 5.1 we present an important result, which was observed in [25] in the context of failure detectors.

Corollary 5.1.1. *When using an unbounded amount of restarts, expectations and higher moment of service time T_τ with restart interval $\tau > 0$ (and $F(\tau) > 0$), are always finite regardless of the original distribution.*

This is especially useful when there is a non-zero probability of the task failing service (infinite service time).

5.3.2 Moments of Completion Time: Finite Number of Restarts, Restart with Non-Identical Intervals

Sometimes it is useful to have a finite number of restarts, or a non-constant restart interval. In Sec. 5.3.1, the hyper/hypo-exponential example shows a low restart interval can increase the service time greatly if there is no bound on the number of restarts. However, for many distributions it can be beneficial to limit the number of restarts, or increase the restart interval, because there is a large penalty for having a restart interval that is too short. For an extreme example, assume a distribution where there is 99% chance to complete after 1 second and otherwise completion happens after 10 seconds. Now assume that we have a slight measurement error and we decide to restart after 0.99 seconds, which results in terrible performance.

In Fig 5.4 we assume the number of restarts is K, the restart intervals are $\tau_1, \tau_2, \ldots, \tau_K$ and i-th interval start time is $s_i = \sum_{k=1}^{K}(\tau_k + c)$. The service time given K restarts is the random variable $T_{\tau_1,\ldots,\tau_K}$.

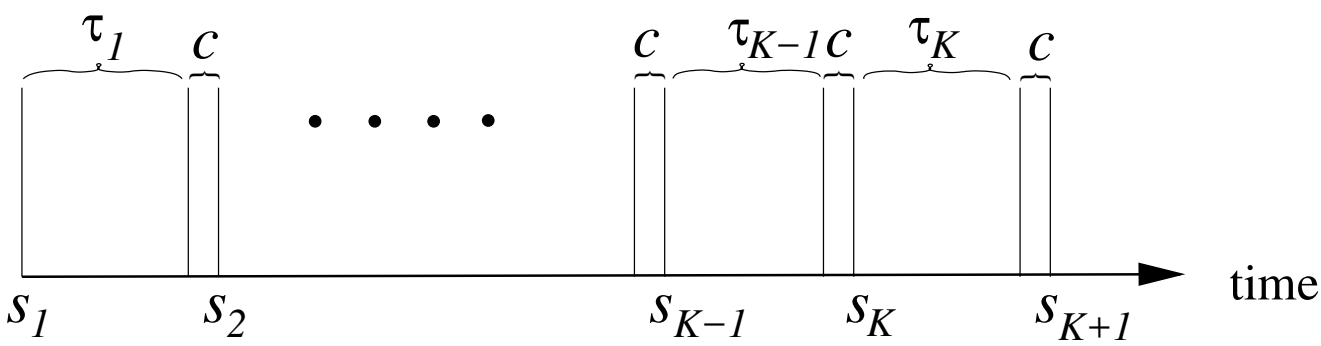

Figure 5.4: Illustration of the restart process.

The cumulative distribution function of service time $F_{\tau_1,\ldots,\tau_K}$ and probability density function $f_{\tau_1,\ldots,\tau_K}$ are derived in a similar way as the unbounded case, shown in Eqn. (5.8) and (5.9). When defining $\tau_{K+1} = \infty$ we can define the density and distribution function in a piecewise manner as follows:

$$1 - F_{\tau_1,\ldots,\tau_K}(t) = \begin{cases} \prod_{i=1}^{k-1}(1 - F(\tau_i))(1 - F(t - s_k)) \\ \quad \text{if} \;\; s_k \leq t < s_k + \tau, k = 1, \ldots, K+1 \\ \prod_{i=1}^{k}(1 - F(\tau_i)) \\ \quad \text{if} \;\; s_k + \tau_k \leq t < s_{k+1}, k = 1, 2, \ldots, K \end{cases}$$

$$f_{\tau_1,\ldots,\tau_K}(t) = \begin{cases} \prod_{i=1}^{k-1}(1 - F(\tau_i)) f(t - s_k) \\ \quad \text{if} \;\; s_k \leq t < s_k + \tau, k = 1, \ldots, K+1 \\ 0 \quad \text{if} \;\; s_k + \tau_k \leq t < s_{k-1}, k = 1, 2, \ldots, K \end{cases} \tag{5.13}$$

The expression for moments in the case of finite restart with varying restart intervals are given in the following theorem.

Theorem 5.2. *The moments* $E\left[T^n_{\tau_1,\ldots,\tau_K}\right] = \int_0^\infty t^n f_{\tau_K,\ldots,\tau_1}(t)dt$, $n = 1, 2, \ldots,$ *of the completion time with* K *restarts, restart interval lengths* $\tau_1, \tau_2, \ldots, \tau_K$, *and time* c *consumed by each restart, can be expressed as:*

$$E\left[T^n_{\tau_1,\ldots,\tau_K}\right] = M_n(\tau_1) + (1 - F(\tau_1)) \sum_{l=0}^{n} \binom{n}{l} (\tau_1 + c)^{n-l} E\left[T^l_{\tau_2,\ldots,\tau_K}\right], \tag{5.14}$$

where $E\left[T^0_{\tau_2,\ldots,\tau_K}\right] = 1$.

The proof of the Theorem 5.2 can be found in [130]. The theorem also allows for the efficient numerical computation of the moments. The first moment is given below as an example:

$$\mathrm{E}\left[T_{\tau_1,\ldots,\tau_K}\right] = M_1(\tau_1) + (1 - F(\tau_1))(\tau_1 + c + \mathrm{E}\left[T_{\tau_2,\ldots,\tau_K}\right]). \tag{5.15}$$

5.3.3 Expressions for Probability of Meeting Deadlines

If deadline d is a whole number multiple of the restart interval τ, the probability of missing a deadline is computed as follows:

$$1 - F_\tau(d) = (1 - F(\tau))^{\frac{d}{\tau}}. \tag{5.16}$$

The general case of using non-constant restart intervals and deadline d, where d is not a whole number sum of τ is more complicated:

$$1 - F_\tau(d) = \begin{cases} \prod\limits_{i=1}^{k}(1 - F(\tau_i)) \cdot (1 - F(d - \sum\limits_{i=1}^{k}(\tau_i + c))) \\ \qquad \text{if} \quad \sum\limits_{i=1}^{k}(\tau_i + c) \leq d < \sum\limits_{i=1}^{k+1} \tau_i + kc \\ \prod\limits_{i=1}^{k+1}(1 - F(\tau_i)) \\ \qquad \text{if} \quad \sum\limits_{i=1}^{k+1} \tau_i + kc \leq t < \sum\limits_{i=1}^{k+1}(\tau_i + c). \end{cases} \tag{5.17}$$

5.4 Optimization of Restart Strategies

This section investigates how to compute optimal restart intervals. It covers the first moment of completion time in Sec. 5.4.1, higher moments in Sec. 5.4.2 and deadline probabilities in Sec. 5.4.3. In each category the optimal restart interval depends on number of restarts and whether intervals are of equal length.

5.4.1 Optimal Restart Times for Expected Service Time

Theorem 5.3. *The optimal restart time $\tau^* > 0$ that minimises the expected completion time $E[T_\tau]$ is such that:*

$$\frac{1 - F(\tau^*)}{f(\tau^*)} = E[T_{\tau^*}] + c. \tag{5.18}$$

That is, if $c = 0$, the inverse of the hazard rate at τ^* equals the expected completion time under unbounded restarts.

Proof. To obtain this result, we equate to zero the derivative with respect to τ of $\mathrm{E}[T_\tau] = \frac{M_1(\tau)}{F(\tau)} + \frac{1-F(\tau)}{F(\tau)}(\tau + c)$ (the base relation (5.12)). After some manipulation we find:

$$h^{-1}(t) = \frac{d}{d\tau}\mathrm{E}[T_\tau] = 0 \iff \frac{1 - F(\tau)}{f(\tau)} = \mathrm{E}[T_\tau] + c.$$

□

Eqn. (5.18) may hold for more than one restart value, including $\tau \to \infty$, and therefore can hold for any local minima and maxima and not just the global optimum.

Paper [130] states that any number of restarts is beneficial to completion time if and only if a single restart improves completion time. If a single restart improves the mean service time, then increasing the amount of restarts will improve it further all the way to an unbounded number of restarts, as illustrated in Fig. 5.5. Any amount of restarts improves the expected service time once a threshold (approx $t > 0.05$) is crossed.

The theorem presented in this section applies for unbounded restarts, where restart is repeated every τ time units. In the paper [130], an iterative algorithm is provided for computing optimal finite number of varying restart intervals.

Table 5.1 displays the optimal restart interval given that a number of restarts can still be performed after the current restart. The service time distribution is the mixed hyper/hypo-exponential distribution that has been discussed earlier, with the same parameters as were used in Fig. 5.2.

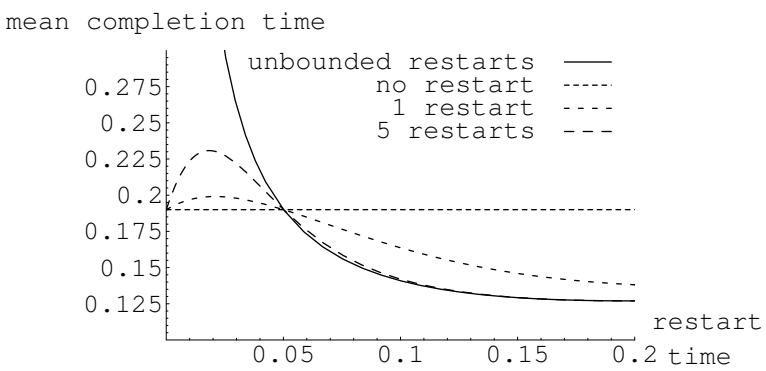

Figure 5.5: Expected service time for different amount of restarts

amount of restarts left	optimal length of next restart interval
unbounded	0.198254
10	0.198254
9	0.198254
8	0.198254
7	0.198256
6	0.198265
5	0.1983
4	0.199
3	0.200
2	0.209
1	0.249

Table 5.1: Optimal restart intervals for finite and unbounded number of restarts.

From table 5.1 it can be observed that the optimal restart interval grows, when the amount of future restarts decreases. When there are still many restarts to come afterwards the restart interval is very close to the unbounded value of 0.198254 time units. In contrast, last two restart intervals, which are 0.209 and 0.249 time units.

5.4.2 Optimal Restart Times for Higher Moments

Optimising higher moments is a more complicated, as each moment has its own optimal restart interval. Also there can be situations such as $t < 0.05$ in Figure 5.6 when two restarts improve the second moment, but an unbounded number of restarts makes it worse. This means that

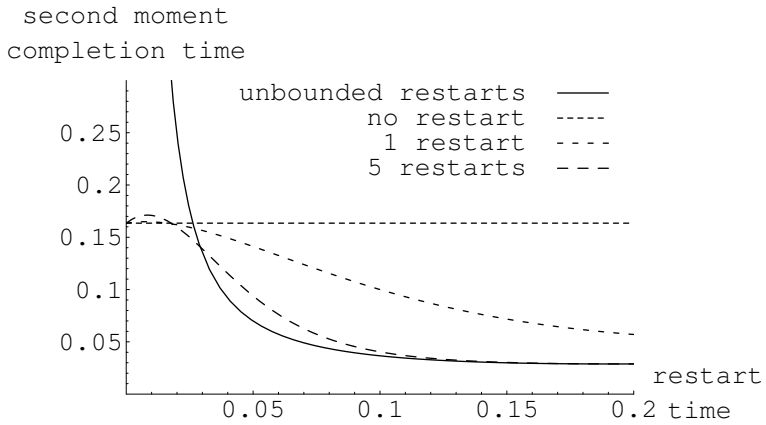

Figure 5.6: Second moment of service time for different amount of restarts.

increasing the number of restarts is not always beneficial when optimising higher moments. Also it is normally not optimal to apply a constant restart interval to optimise a higher moment, even if you allow an unbounded amount of restarts.

Therefore optimal restart interval computation needs to consider the following: the specific moment being optimised, number of restarts and whether intervals should be identical. The paper [130] has more information on optimising under such conditions.

In this section we limit the discussion to optimising the restart intervals to the case where service time T is distributed log-normally, with parameters $\mu = -2.31$ and $\sigma = 0.97$. The log-normal distribution is a heavy-tailed and therefore benefits from restarts. The parameter values were fitted from experimental data of HTTP GET service times in [107].

T_K denotes service time with K restarts. We will calculate for $K = 15$ the restart intervals that minimise the first, second and third moments of service time. The restart times (with an interpolating curve) are shown in Fig. 5.7. The figure also displays the optimal restart time for τ_∞ of the first moment.

Fig. 5.7 shows that when the first moment is minimised restart time $\tau_{k|K}$ converges monotonically when k approaches the unique optimum τ_∞, given that K is large enough. This behaviour can also be observed in Table. 5.1. The convergence for higher moments is not as simple, but it can be noted that using the restart times that minimise the first moment are an approximate

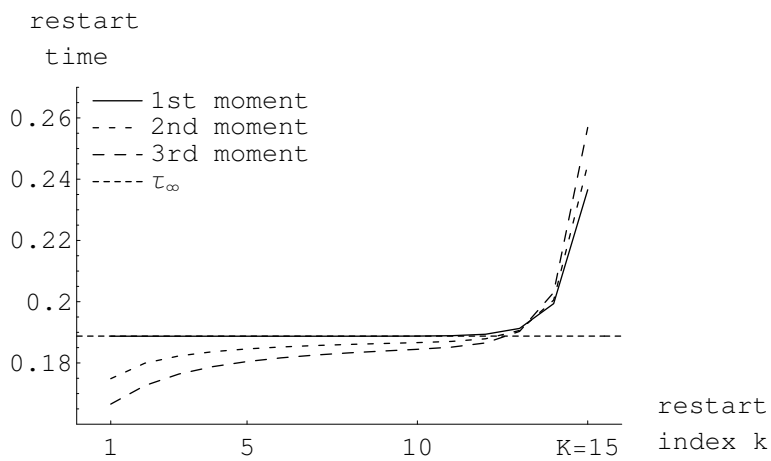

Figure 5.7: Optimal restart times, for the moments $\mathrm{E}[T_{15}]$, $\mathrm{E}[T_{15}^2]$ and $\mathrm{E}[T_{15}^3]$.

indicator of optimal restart intervals for higher moments when K is large enough.

5.4.3 Using Restart to Optimise Deadline Probability

This section discusses the equihazard and equidistant strategies for maximising the probability of meeting a deadline. The equihazard strategy strategy picks restart intervals which correspond to the local extrema of the hazard rate. The equidistant strategy uses the same restart interval for restarts.

Using a single retry with a finite interval $[0, d)$ with a retry time of $\tau < d$ the probability of completion before the deadline d is:

$$F_\tau(d) = 1 - (1 - F(\tau))(1 - F(d - \tau)). \tag{5.19}$$

The local extrema of the Eqn. 5.19 can be found with the help of derivatives and is the following:

$$\frac{f(\tau)}{1 - F(\tau)} = \frac{f(d - \tau)}{1 - F(d - \tau)},$$

which is equivalent to:

$$h(t) = h(d - t). \tag{5.20}$$

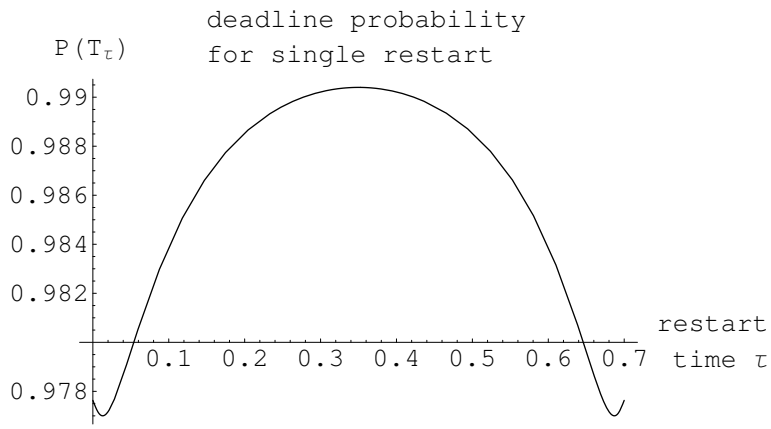

Figure 5.8: The probability of service completion before the deadline when using one restart ($d = 0.7, \mu = -2.3, \sigma = 0.97$).

The above can be interpreted to mean that minimum and maximum probability of meeting the deadline is achieved with *equihazard* restart intervals. The *equidistant* restart intervals $\tau = \frac{d}{2}$ are a special case of equihazard intervals, and therefore form a local extremum

With multiple retries, similar logic can be applied. Given restarts at $\tau_1, \tau_1 + \tau_2, \ldots, \sum_{n=1}^{N} \tau_n$ and assuming without a loss of generality that $\sum_{n=1}^{N} \tau_n = d$, the optimum with respect to all retry intervals $\tau_1, \ldots, \tau_N$ happens when:

$$h(\tau_1) = h(\tau_2) = \ldots = h(\tau_N). \tag{5.21}$$

As before, the extrema occur at equihazard intervals, with as special case the equidistant restart intervals $\tau_n = \frac{d}{N}$.

Fig. 5.8 demonstrates typical behaviour when using one restart. The results are for a log-normal distribution with the same parameters as in Fig. 5.7 and a deadline of $d = 0.7$. The equidistant restart (at $\tau = 0.35$) is the maximum, and other equihazard points are minima ($\tau = 0.013$ and $\tau = 0.687$). When using the deadline $d = 0.7$ the probability of making the deadline improves from 0.977 to 0.990. Table 5.2 displays all sets extremal equihazard intervals for different restart periods. It can be seen that for the example equidistant hazard rates always outperform other equihazard points.

# restarts	equihazard intervals	$P(T_{\{\tau\}} < d)$
0	—	0.978
1	0.35, 0.35	0.990
1	0.013, 0.687	0.977
2	0.23, 0.23, 0.23	0.993
2	0.019, 0.34, 0.34	0.990
2	0.013, 0.013, 0.674	0.976
3	0.175, 0.175, 0.175, 0.175	0.99374
3	0.024, 0.225, 0.225, 0.225	0.993
3	0.019, 0.019, 0.331, 0.331	0.989
3	0.013, 0.013, 0.013, 0.660	0.976
4	0.14, 0.14, 0.14, 0.14, 0.14	0.99366
⋮	⋮	⋮

Table 5.2: Equihazard restart intervals and associated probability of meeting the deadline ($d = 0.7, \mu = -2.3, \sigma = 0.97$).

Equidistant restarts are optimal for all experiments with log-normal distributions. It is possible to construct an example where two non-equidistant points outperform equidistant points; however, it only seems possible if not restarting is even better non-equidistant points. However, this remains as a conjecture, as we don't have a proof for this.

5.5 Replication Strategies

CPU cores, processor caches, memory bandwidth and network bandwidth are often shared when multiple programs share resources. In addition, background daemons can cause delays in job processing if they are scheduled in between processes [35]. These issues are especially problematic, if a task consists of hundreds of small subtasks. Since even low probability events become likely to disrupt the service of at least one subtask, which then delays the service of the whole job.

One promising technique to avoid excessive wait times is to replicate subtasks. Replication is frequently both possible and feasible, because utilisation is often low. For example, at Facebook the median utilisation of resources is low: time slots 21%, CPU 19% and memory 17% [8]. Secondly a large share of the energy costs are caused by short periods of peak consumption and

data centres have not found it worth their time to have processes to shut down idling machines. As a result lots of idle computing capacity exists in data centres [8].

This section will explain why service replication is a good strategy for improving task response time and discuss reasons as to why consuming extra resources can be justified in certain circumstances. In addition, we also discuss three different replication strategies.

The first strategy is n-fold replication, which is just running n copies of the program. Second strategy is hedged requests, which spawns a restart strategy only after the first run under normal conditions should have finished. The final strategy is tied requests, which sends n replicas to be serviced. However, once one tasks begins service the replicas are canceled.

5.5.1 n-fold Replication

The simplest replication strategy is to clone a job n times and pick the job that completes first. n-fold replication will be effective if the service time of jobs has a high variance. However, if there is little variance, replication will offer little benefit. The downside of n-fold replication is the n-fold increase in resource usage.

The cumulative distribution function of the minimum service time of n independent subtasks can be calculated in the following way:

$$F_n(t) = 1 - (1 - F(t))^n \tag{5.22}$$

The corresponding probability density function is:

$$f_n(t) = n(1 - F(t))^{n-1} f(t). \tag{5.23}$$

5.5.2 Hedged Requests

The idea behind hedged requests is to gain some of the speed advantages of replication, while at the same time minimising the amount of extra resources [35]. In hedged requests, service replication happens only after the task should have completed service under normal conditions. For example, in Google data centres tasks are replicated after 95% of similar tasks have completed [35].

It should be noted that hedging requests is very similar to using service restart to improve performance and indeed the formulas from Sec. 5.4 can be directly applied and used to optimise the hedging interval. Eqn. (5.8) and (5.9) define the cumulative distribution function and probability density function when the previous job is cancelled upon hedging.

5.5.3 Tied Requests

Another technique to further reduce the amount of extra computational resources needed due to replication is to send replicated jobs and then cancel the duplicates once a sibling job begins service [35]. This is beneficial, because majority of the variation of service time is often caused by the time spent in the queue and the service of a job has little variation. The advantage of tied requests compared to hedged requests, is that sending a cancel message is less resource intensive than servicing the job. Another benefit is derived from the fact that the multiple copies begin queueing straight away. Instead of waiting for a while before extra copies are dispatched.

5.6 Conclusion

This chapter covered how service restart and service replication can be used to improve task service time.

The chapter began by discussing the effectiveness of restart and analysed it with various exam-

ples such as the exponential, hypo- and hyper-exponential and heavy-tailed distributions. Next we derived expressions for the moments of completion time with restarts and the probability of service completion before a deadline. The next part of the chapter focused on how the service restart interval should be chosen to optimise the performance of task service.

The chapter finishes by presenting three replication strategies namely: n-fold replication, hedged requests and tied requests.

Chapter 6

Three-way Optimisation of Response Time, Subtask Dispersion and Energy Consumption in Split–Merge Systems

This chapter investigates various ways in which the triple trade-off metrics between task response time, subtask dispersion and energy can be improved in split–merge queueing systems. Four ideas, namely dynamic subtask dispersion reduction, state-dependent service times, multiple redundant subtask service servers and restarting subtask service, are examined in the chapter. It transpires that all four techniques can be used to improve the triple trade-off, while combinations of the techniques are not necessarily beneficial.

6.1 Introduction

In real world applications quite often there are multiple criteria by which one may wish to be as good as possible. There is also a catchphrase saying "choose two out of three: fast, good and cheap", indicating that it is very difficult to satisfy all criteria. In this chapter, we investigate split–merge systems, a particular variety of parallel queueing systems that have three desired qualities. First we wish the system has a low subtask dispersion [88, 103, 128, 126]

(difference in completion time between first and last subtasks to complete), low task response time (difference between task arrival and completion of service) and low energy usage. This chapter also discusses four techniques, which are compared with the method from [128].

Sec. 6.2 and 6.3 focus on two techniques, which are both based on dynamically adding delays so as to reduce the variation between the fastest and the slowest subtasks. The first technique inserts delays before the service of fast subtasks to make them take longer to complete with the ideal outcome that all subtasks complete at the same time. We will first show the mathematical formulation for dynamic subtask dispersion as introduced in [103]. The second technique is to select the delays inserted in front of subtasks based on the queue length of the system. Delays are determined with the help of Bayesian optimisation.

Sec. 6.4 introduces two techniques that use replication of subtask servers and service restart of subtasks. These techniques work best when the underlying subtask service time distribution has high coefficient of variation, more precisely subtask service restart is beneficial if the service time distribution has large variance compared to the mean. In addition, such tasks must be idempotent, i.e. replication, abortion and restart of the task must not have undesired side effects, such as a duplicated trade or transaction. Both methods are applicable only if tasks can be shifted, duplicated and repeated with no harm.

It is possible to use dynamic subtask dispersion if the time to transmit information that a sibling subtask has finished is small compared to the service time of subtasks. State-dependent delays have very little prerequisites and can be used almost always as they only require knowledge of the queue length.

We find that restarting subtasks and replicating service of subtasks is very helpful in cases where the subtask service time has high coefficient of variation. However, formulating guidelines as to which of the methods to use best is not straightforward as it will depend on the system-specific parameters. Our results indicate that combining the two redundancy techniques is not necessarily able to further improve results. We attribute this to the fact that both techniques work by reducing the variance of the subtask service time distribution.

6.2 Trade–offs Using Dynamic Subtask Dispersion

In this section, we first introduce the split–merge queueing model and then investigate how the trade-off delay scheme introduced in [128] can be improved by substituting the subtask-dispersion technique with the improved delay scheme introduced in [103].

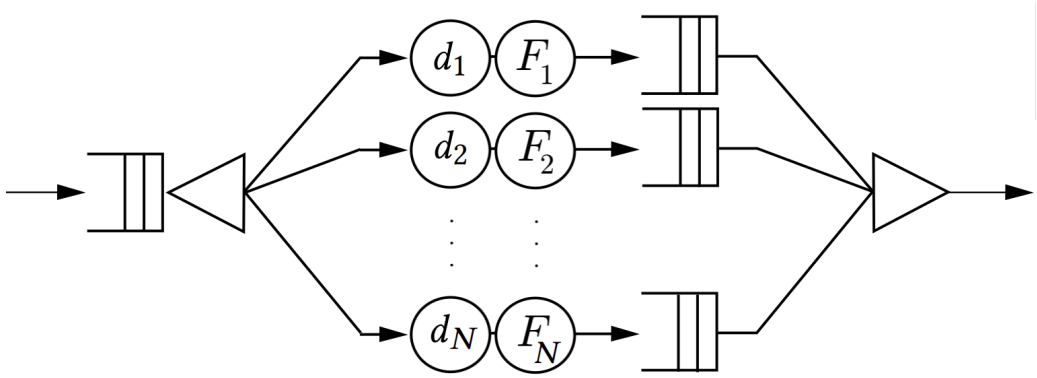

Figure 6.1: Split–merge system, credit: [125], (CC BY-SA 3.0).

In a split-merge system each time a new task enters service it is split into N subtasks. Those subtasks are served by N different parallel servers. Upon completion, the subtasks are re-assembled into one task for further processing. This situation is illustrated in Fig. 6.1, more information on the split–merge system can be found in Sec. 2.2.1. Some of the parallel servers will finish processing their subtask much earlier than others which leads to undesired dispersion in the completion time of the subtasks. To avoid this, different measures can be taken. The most straightforward solution is to insert a delay before each subtask. These delays are specified in a delay vector $\mathbf{d}$.

The dynamic version of the algorithm removes any leftover delay, once a sibling subtask has completed service. In the static algorithm delays are kept constant once they are defined. Sec. 6.2.1 and 6.2.2 discuss the dynamic algorithm. More information on dynamic subtask dispersion can be found in Sec. 2.3.1. For the dynamic algorithm intuition dictates that subtask dispersion should be reduced if delays are cut after a sibling subtask has completed. Similarly it dictates that task response time should decreases when delays are decreased.

An example of a potential operation by the algorithm is shown in Fig. 6.2. The advantage of

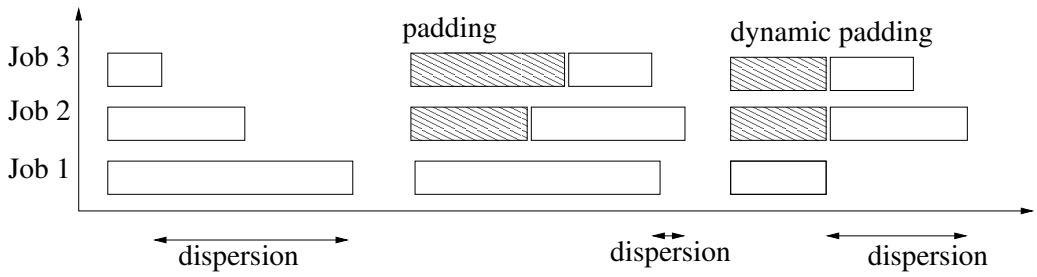

Figure 6.2: An example of processing a task in a three server split–merge system

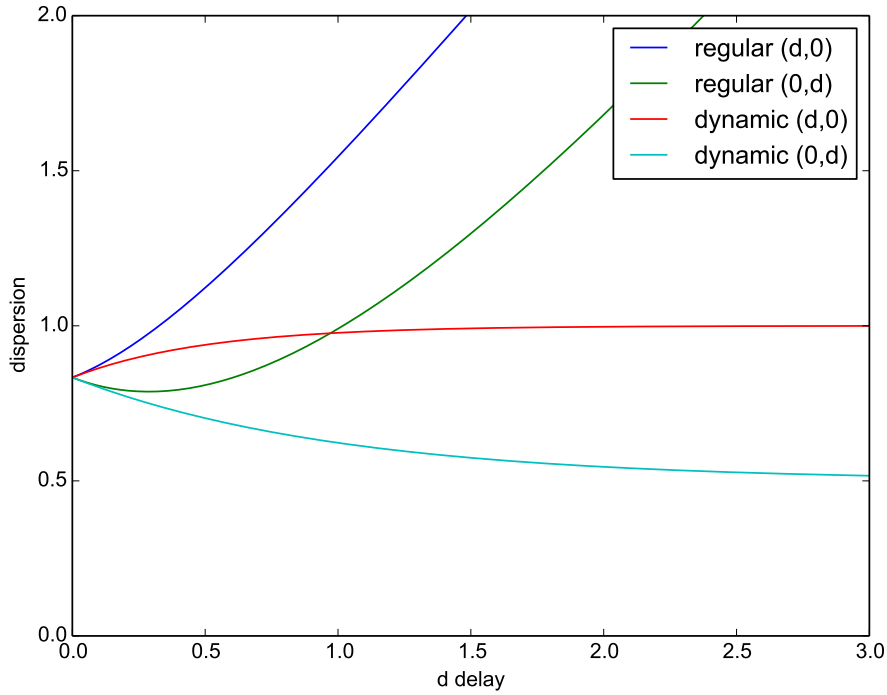

Figure 6.3: A two server $\mathrm{Exp}(\lambda = 1), \mathrm{Exp}(\lambda = 2)$ example demonstrating difference between Dynamic and Static delays to reduce subtask dispersion

dynamic subtask padding is shown by the results for an example in Fig. 6.3. We observe that removing delays once a sibling subtask finishes can dramatically reduce subtask dispersion.

The next subsection defines the optimisation problem that must be solved to determine suitable delays that minimise subtask dispersion. In Sec. 6.2.2 we formulate the corresponding optimisation problem for minimising the task response time. Finally, at the end of Sec. 6.2.2 we combine both into the formulation of the trade-off between task response time and subtask dispersion. The trade-off is specified in Eqn. (6.14).

6.2.1 Dynamic Subtask Dispersion

A technique to minimise dynamic subtask dispersion has been introduced in the paper [103]. The paper is summarised in Sec. 3.4, more information on subtask dispersion can be found in Sec. 2.3.1. We briefly revisit the result here, as we extend upon in Sec. 6.2.2. This section introduces how subtask dispersion of a split-merge system can be computed for a given set of delays $\mathbf{d}$.

Let $T(i, t, \mathbf{d})$ be the probability that subtask i is the first to finish at time t. Then $E_r(i, t', \mathbf{d})$ is the expected completion time of the remaining subtasks, given that subtask i finished at time t'. $G_j(t, t', \mathbf{d_j})$ is the probability distribution of a subtask, given that a sibling subtask has finished already. $F_i(t)$ and $f_i(t)$ are the cumulative distribution function and probability density function of service time of ith subtask.

Those terms can be used to compute subtask dispersion for a given set of delays $\mathbf{d}$. To minimise dispersion a set of delays $\mathbf{d}$ must be found that minimise the Disp($\mathbf{d}$) function.

$$\mathrm{Disp}(\mathbf{d}) = \sum_{i=1}^{N} \int_0^{\infty} T(i, t, \mathbf{d}) E_r(i, t, \mathbf{d}) dt \tag{6.1}$$

where

$$T(i, t, \mathbf{d}) = f_i(t - d_i) \prod_{j \neq i} [1 - F_j(t - d_j)] \tag{6.2}$$

and

$$E_r(i, t', \mathbf{d}) = \int_0^\infty [1 - \prod_{j \neq i} G_j(t, t', d_j)]dt \tag{6.3}$$

with

$$G_j(t, t', d_j) = \begin{cases} F_j(t) & \text{if } t' < d_j \\ F_j(t + (t' - d_j)|t > 0) & \text{otherwise} \end{cases} \tag{6.4}$$

subject to conditions

$$\prod_{i=1}^{N} d_i = 0 \tag{6.5}$$

and

$$\forall i \quad d_i \geq 0 \tag{6.6}$$

The Disp($\mathbf{d}$) function is used in the next section in the trade-off definition in Eqn. (6.14).

6.2.2 Using Dynamic delays to Optimise Task Response Time and Subtask Dispersion

In this section, we first derive an expression for the task response time in a split-merge system when using dynamic delay padding. Then we combine this result with the expression for subtask dispersion derived in the previous section to formulate the trade-off between task response time and subtask dispersion, which is one of the main results introduced in this chapter.

The task response time of a split–merge system can be computed using the Pollaczek–Khinchine (PK) formula known for M/G/1 queues. The split–merge queue is very similar to a M/G/1 queue, because the time to complete all N subtasks can be seen as the service time of a task in the M/G/1 queue as shown below:

$$\text{Resp}(\lambda, \mathbf{d}) = \frac{\rho + \mu\lambda \text{Var}[X_{(N)}]}{2(\mu - \lambda)} + \mu^{-1} \tag{6.7}$$

Here, the arrival rate of incoming tasks is λ when the time between arrival of tasks is exponentially distributed, the task service rate is μ, ρ is the utilisation of the system given by λ/μ,

and $\mathrm{Var}[X_{(N)}]$ is the variance of the service time of last subtask to complete, equivalently the variance of the service time of the task.

We will now derive the mean and variance of the task service time. The first step is to derive a probability distribution for the service time of a task, which is shown below:

$$f(t, \mathbf{d}) = \sum_{i=1}^{N} \int_0^t T(i, t', \mathbf{d}) E(i, t', t - t', \mathbf{d}) dt' \tag{6.8}$$

Function $T(i, t, \mathbf{d})$ calculates the probability that subtask i is the first subtask to finish at time t given the subtask delay vector $\mathbf{d}$.

$$T(i, t, \mathbf{d}) = f_i(t - d_i) \prod_{j \neq i} [1 - F_j(t - d_j)] \tag{6.9}$$

Function $E(i, t', t, \mathbf{d})$ describes the probability that the remaining subtasks finish in time t. It takes as priori i, which is the number of the subtask that finished first and the time t' when it finished.

$$E(i, t', t, \mathbf{d}) = \frac{d}{dt} \prod_{j \neq i} G_j(t, t', d_j) = \sum_{j \neq i} g_j(t, t', d_j) \prod_{k \neq i|j} G_k(t, t', d_j) \tag{6.10}$$

Function $G_j(t, t', d_j)$ renormalises the jth subtask service distribution to take into account that a sibling subtask has finished at time t'. If no other subtask has started service it immediately starts service. If the subtask has begun service the service time probability distribution is renormalised to take into account that it did not finish before time t'.

$$G_j(t, t', d_j) = \begin{cases} F_j(t) & \text{if } t' < d_j \\ F_j(t + (t' - d_j) \mid t > 0) & \text{otherwise} \end{cases} \tag{6.11}$$

Once the probability distribution of the service time of a task has been defined its mean and variance can be derived in a standard way. The standard expressions for mean and variance are given below:

$$\mathrm{E}(X_{(N)}(\mathbf{d})) = \int_0^\infty t f(t, \mathbf{d}) dt \tag{6.12}$$

$$\mathrm{Var}[X_{(N)}](\mathbf{d}) = \int_0^\infty (t - \mathrm{E}[X_{(N)}](\mathbf{d}))^2 f(t, \mathbf{d}) dt \tag{6.13}$$

Service time is given by the following relation $\mu^{-1} = \mathrm{E}(X_{(N)}(\mathbf{d}))$

After deriving the service time and variance of the response time the algorithm from paper [128] and Sec. 2.3.3 can be applied. The optimal delay vector $\mathbf{d}_{min}$ can be found by solving the following optimisation problem:

$$\mathbf{d}_{\min} = \arg\min_{\mathbf{d}' \geq 0} \mathrm{Resp}(\lambda, \mathbf{d}')\mathrm{Disp}(\mathbf{d}') \tag{6.14}$$
$$s.t. \quad \prod_{i=1}^{N} d_i = 0$$

When solving the equation in practice, the integrals are solved using numerical integration techniques and the minimisation of Eqn. (6.14) can be done using an optimisation algorithm such as Nelder–Mead.

Details on how to numerically solve Eqn (6.14) are discussed in Sec. 6.5.1. This trade-off is evaluated in Sec. 6.6.1.

6.3 State-Dependent Delay Vectors

While Sec. 6.2 has focused on how the trade-off between subtask dispersion and task response time can be optimised using a technique which ignores system load, this section introduces the concept of state-dependent delay vectors, i.e. where the delay vector is determined based on how many tasks are currently in the queue waiting to be served.

6.3.1 Background

There is a large body of research on state-dependent M/G/1 and M/M/1 queues [2, 57, 117], as well as analytical solutions. However, the analytical solution of the response time distribution is defined in terms of Laplace transforms, infinite sums and recursive equations, which make computation of accurate results quickly unfeasible. We were unable to produce a stable and fast computations to perform a optimality search over the function in sufficiently high dimensional space. We therefore adopted numerical techniques.

Our intuition is that delays should be small when the queue length is large and delays should be larger for a short queue. Reducing delays of subtasks may increase subtask dispersion for the corresponding task, but it benefits the response time of all subsequent tasks in the queue. In addition this will make the system more tolerant of fitting errors (such as inaccuracies in the interarrival rate or service time functions) and a change of step in the task interarrival rate λ. Change in λ will reflect on the average queue length of the system. The system will pick up on and be able to automatically adjust itself.

Next we present a two-step technique to find optimal delays for different queue lengths. A naïve search would optimise on an $O(N \times m)$ dimensional search space, where N is number of subtasks and m is the number of tasks in the queue for which a unique delay vector is specified. Our technique reduces this to a $O(\max(n, m))$ dimensional search space. The search space reduction is especially necessary as our objective function evaluations are expensive.

6.3.2 The Algorithm

With the first step of the algorithm we wish to construct a function that maps from a one dimensional parameter onto a delay vector, which optimises the product of expected subtask dispersion and expected task response time:

$$f(\lambda, \mathbf{d}) = \text{Resp}(\lambda, \mathbf{d})\text{Disp}(\mathbf{d}) \tag{6.15}$$

Tsimashenka's trade-off technique [128] can be used to determine an optimal delay vector for a given λ. Varying λ from 0 until utilisation of the system is 1 gives us a parametric curve of delays, as shown in Fig. 6.4. We can see that at low utilisations relatively large delays are added in order to reduce subtask dispersion, while at high utilisations, the system concentrates on the protection of response time by minimising delays. If the technique does not produce a clear line, regression analysis can be applied, as it is in the case of dynamic subtask dispersion.

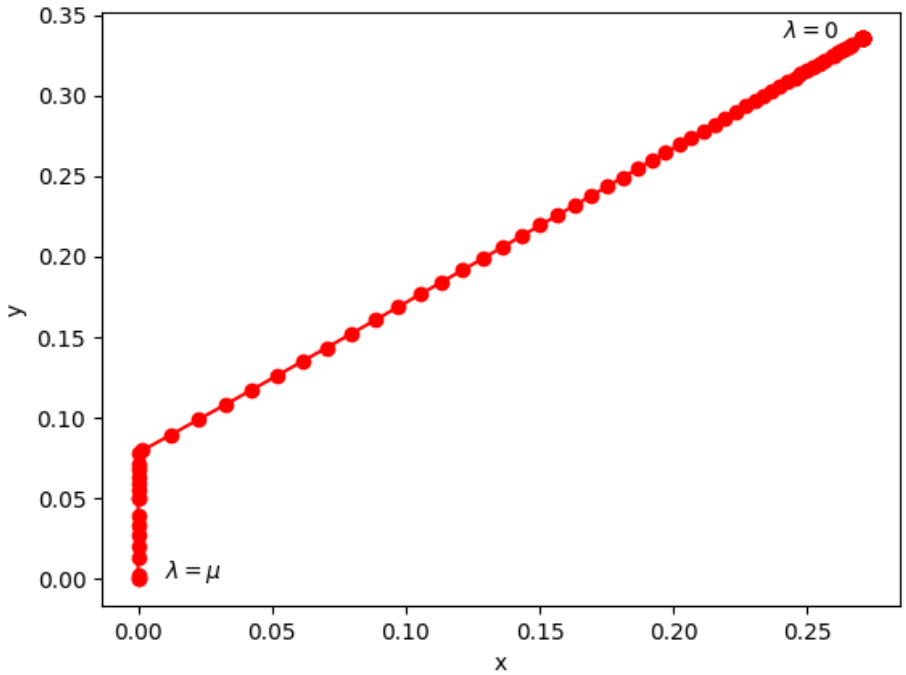

Figure 6.4: Showing the parametric curve of delays of the form (0,x,y). Formed by application of Tsimashenka's trade-off technique [128] for varying range of utilisation, on a three server split–merge system $\mathrm{Exp}(\lambda = 1), \mathrm{Exp}(\lambda = 5), \mathrm{Exp}(\lambda = 10)$

The second step of our algorithm makes application of delay vectors state-dependent by mapping queue length thresholds onto points on the parametric curve of delays. The objective function of our optimisation takes as input an m element vector. The ith element of the vector is a real number in the same range as λ in the previous step. The ith element maps to a delay vector via the parametric curve of delays, using interpolation between evaluated points where necessary. This delay vector is applied when there are i tasks waiting in the queue. If there are more than m tasks in the queue, the last element of the vector is used.

In the objective function, the split–merge system is simulated with a large number of tasks to

obtain an estimate of the product of mean subtask dispersion and mean task response time when applying state-dependent subtask delays using the m element vector.

Due to the noisy nature of the optimisation function we chose Bayesian optimization, a well-established technique for optimizing noisy, non-convex functions. GpyOpt [120] was used for implementing the Bayesian optimisation.

Experimental setup and results for this technique are presented in Sec. 6.5.1 and 6.6.1, where it used in Methods 3 and 4.

6.4 Energy Metric and Service Time Manipulations

This section introduces subtask service restart and subtask service replication as potential techniques for improving trade-offs between performance metrics in parallel queueing systems. It also introduces the energy metric as a necessary addition to the previously-used metrics for two reasons. Firstly, energy consumption considerations are typically a critical consideration in large scale service delivery. Secondly, if there is no cost associated with service restart and replication, it is often possible to arbitrarily improve performance in terms of subtask dispersion and task response time by increasing the number of server replications without a limit.

6.4.1 Server Replication

Given a non-deterministic service time distribution with high variance and assuming no correlation between consecutive service attempts, it is possible to improve the service time of a subtask using multiple servers. Several copies of the subtask are served in parallel by these servers and the result of the fastest server is used. An example of how server replication improves service time can be see in Fig. 6.5.

The cumulative distribution function of the minimum service time of n subtasks can be calculated in the following way:

$$F_n(t) = 1 - (1 - F(t))^n \tag{6.16}$$

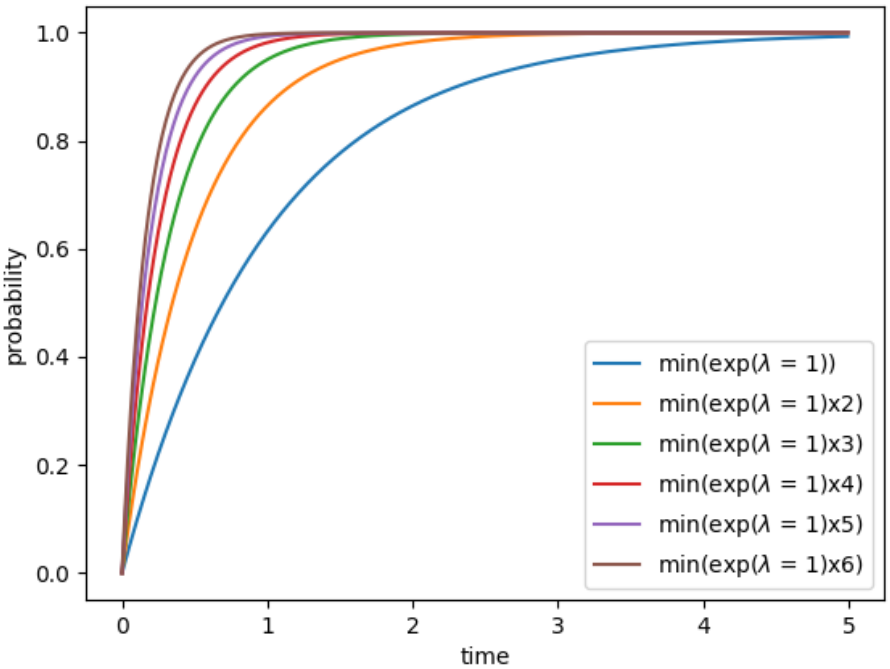

Figure 6.5: Changes in cumulative probability distribution of subtask service time when the number of servers is varied from 1 to 6.

The corresponding probability density function is the following:

$$f_n(t) = n(1 - F(t))^{n-1} f(t). \tag{6.17}$$

Details on how server replication is used are discussed in Sec. 6.5.3. This technique is evaluated in Sec. 6.6.2.

6.4.2 Service Restart

An alternative to serving redundant copies of subtasks is to use service restart if service has not completed by a given target time τ. Restart comes with a time cost c. Restart will not always provide an improvement in terms of service time. However, there exists a class of service time distributions where the conditional remaining service time of a job increases as time progresses, more on this can be read in Sec. 5.2.4. It has been shown that the average service time of such tasks can be decreased by periodically restarting service [129]. More information on using

unbounded number of restarts to improve task service time can be found in Sec 5.3.1.

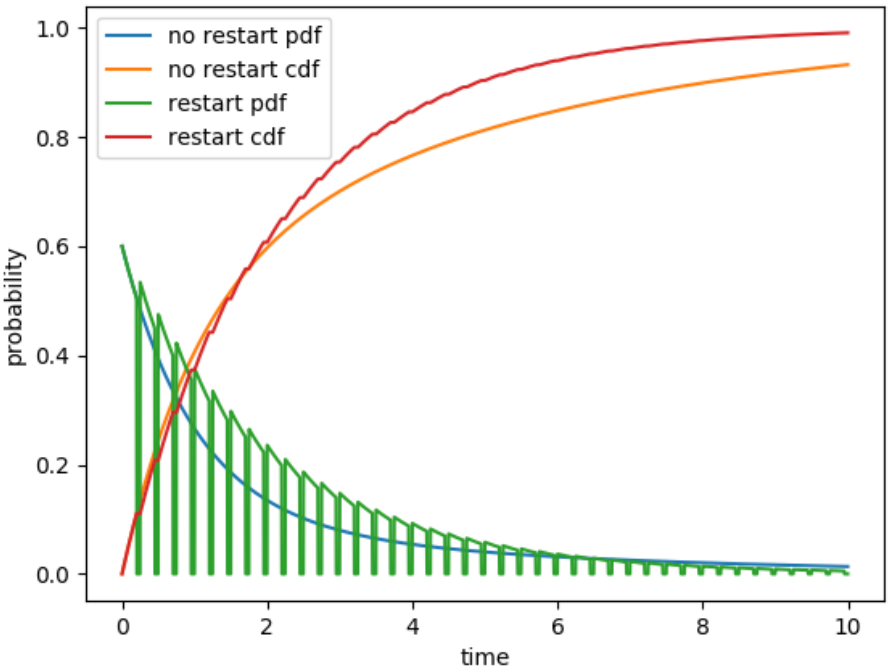

Figure 6.6: Comparison of a hyper-exponential distribution with/without service restart. $\tau = 0.2$, $c = 0.05$, $f(x) = \frac{1}{2}\exp(\lambda = 1.0) + \frac{1}{2}\exp(\lambda = 0.2)$

The new probability density function of service time is:

$$f_\tau(t) = \begin{cases} (1 - F(\tau))^k f(t - k(\tau + c)) & \text{if } k(\tau + c) \leq t < k(\tau + c) + \tau \\ 0 & \text{otherwise} \end{cases} \tag{6.18}$$

for $k = 0, 1, 2, \ldots$.

The corresponding cumulative distribution function is:

$$F_\tau(t) = \begin{cases} 1 - (1 - F(\tau))^k (1 - F(t - k(\tau + c))) & \text{if } k(\tau + c) \leq t \\ & < k(\tau + c) + \tau \quad , \\ 1 - (1 - F(\tau))^{k+1} & \text{otherwise} \end{cases} \tag{6.19}$$

for $k = 0, 1, 2, \ldots$.

The higher moments of a probability distribution with heavy tail can be decreased when service

restart is incorporated. The new distribution will then be similar to a geometric distribution. This improves both task response time of the system and subtask dispersion. An example of the effect of service restart on a distribution can be seen in Fig. 6.6.

Details on how server restart is used are also discussed in Sec. 6.5.3. This technique is evaluated in Sec. 6.6.2.

6.4.3 Energy Metric

This section introduces the energy metric, which measures how much energy is consumed by a split–merge system. Split–merge systems in practice come at the energy cost of running several servers. Improving performance may likely increase the energy cost. This chapter uses the following cost function:

$$\text{Energy}(\mathbf{n}) = \sum_{i=1}^{N} n_i(\lambda/\mu_i C_H + (1 - \lambda/\mu_i)C_L) \tag{6.20}$$

The constants defined in the equation are as follows. The rate of incoming tasks is λ, the service rate of subtask i is μ_i. The idle operational cost of a server is C_L, while the cost for service, restart and cool down is C_H. The number of servers serving the ith subtask is n_i, the number of subtasks each task is split into is N.

6.4.4 Trade-off Metric

Energy trade-off metrics have been investigated in the past [48, 32]. For computing the triple trade-off between subtask dispersion, task response time and energy the third power of the product, the RDE^3 metric is used. This is because we found that the RDE and RDE^2 metrics sometimes indicate that an infinite replication would be optimal. Further details can be found in Sec. 6.4.5. Additionally, having a superlinear energy usage often happens in physical systems as well. For example, power usage and voltage in CPU's are related in the following way due to Ohm's law $P \propto V^2$[64].

The trade-off is computed by first applying the multiple servers and/or restart transforms to the subtask service time distributions. Then the subtask dispersion, task response time and energy cost are derived. The optimal delay vector $\mathbf{d}$, server replication factors $\mathbf{n}$ and server restart intervals τ can be found by solving the following optimisation problem:

$$\arg\min_{\mathbf{d}\geq 0,\mathbf{n},\tau} \mathrm{Resp}(\lambda, \mathbf{d})\mathrm{Disp}(\mathbf{d})\mathrm{Energy}^3(\mathbf{n}), \tag{6.21}$$

In our example we allow each subtask to have up to 10 replicated servers (including the original). For each possible server configuration (i.e. for every permutation of $\mathbf{n}$) we ran the Nelder–Mead algorithm 50 times, using random start vectors for τ and $\mathbf{d}$. Finally, we selected the $\mathbf{n}$,τ and $\mathbf{d}$ that minimised Eqn. (6.21).

Details on how the triple trade-off is used in our experiments are presented in Sec. 6.5.3. The trade-off is evaluated in Sec. 6.6.2.

6.4.5 Problems defining the Task Response Time, Subtask Dispersion and Energy Consumption Trade-Off Metric

This section shows that RE and DE metrics sometimes have trivial solutions indicating an infinite number of servers. As a consequence RDE^2 will also suffer from the same problem, as it is RE and DE multiplied together. Therefore, our experiments use the product RDE^3. To demonstrate these problems we analyse a two server split–merge system with exponentially distributed service times.

Task Response Time and Energy Consumption Trade-Off as n approaches ∞

The minimum of two exponentially distributed random variables is exponentially distributed with the μ_s given below:

$$\mu_{\min} = \mu_1 + \mu_2 \tag{6.22}$$

Duplicating service server n times results in a service rate of $n\mu$. Given that the variance of an exponential distribution is μ^{-2}, the variance with n duplicated servers is $(n\mu)^{-2}$.

Substituting $n\mu$ for μ into the Pollaczek–Khinchine formula results in the following Task Response Time and Energy trade-off:

$$\text{Resp}(\lambda) = \frac{\lambda/(n\mu) + n\mu\lambda(n\mu)^{-2}}{2(n\mu - \lambda)} + (n\mu)^{-1} \tag{6.23}$$

$$\text{Energy}(n) = kn \tag{6.24}$$

Multiplying the two metrics and simplifying we obtain:

$$\text{Resp}(\lambda)\text{Energy}(n) = \frac{k\lambda\mu^{-1}}{(n\mu - \lambda)} + \frac{k}{\mu} \tag{6.25}$$

From this it is apparent that when the number of servers $n \to \infty$ the Trade–Off approaches k/μ, which is the optimal.

Subtask Dispersion and Energy Consumption Trade-Off as n approaches ∞

When using dispersion technique from [103] the optimal strategy is to have the slower subtask server finish service first and only then begin work on the second subtask. Assuming we duplicate the faster server n times results in the following subtask dispersion and energy usage:

$$\text{Disp}(n) = \frac{1}{n\mu} \tag{6.26}$$

$$\text{Energy}(n) = 1 + n \tag{6.27}$$

Multiplying the two metrics and simplifying we obtain:

$$\text{Disp}(n)\text{Energy}(n) = \frac{1+n}{n\mu} = \frac{1}{n\mu} + \frac{1}{\mu} \tag{6.28}$$

From the above it can be seen, that when the number of servers $n \to \infty$ the Trade–Off approaches μ^{-1}, which is the optimal.

6.5 Experimental Setup

This section introduces the experimental setup for our two case studies. The first case study explores how dynamic subtask dispersion and queue dependent delays affect subtask dispersion and task response time. The second case study explores how service duplication and restart affect subtask dispersion, task response time and energy consumption.

6.5.1 Exploring Response Time and Subtask Dispersion

This section has a short note on computation resources used and also explains the methods used in the results of Sec. 6.6.1.

6.5.2 Computational Resources

It took approximately one day on one core of an Amazon Web Services C4 computer to compute the final results for **Method 3 and 4** for each data point. The total computation time was roughly two days of computation on the 8 core C4 machine. A 8 core C4 computer costs \$0.464 per hour to operate resulting in a total cost of \$23.

Method 1

For this method, analytical formulas (6.29) and (6.7) were used to find the delay vector which optimises the product of subtask dispersion and task response time, according to the method presented in [128].

Method 2

This method uses the concept of dynamic subtask dispersion from Sec. 6.2.1 and dynamic delay padding from Sec. 6.2.2 to find the delay vector which optimises the trade-off between task response time and subtask dispersion. For computation of subtask dispersion and task response time we use Eqn. (6.1) and (6.7).

Method 3

This method uses the state-dependent delay vectors introduced in Sec. 6.3 to optimise the trade-off between subtask dispersion and task response time. No dynamic delay padding is used.

Method 4

This method uses the state-dependent delay vectors introduced in Sec. 6.3 to optimise the trade-off between subtask dispersion and task response time. In this method dynamic delay padding is applied.

6.5.3 Exploring Response Time, Subtask Dispersion and Energy

This section explains the methods used in the results of Sec. 6.6.2.

All methods the results were computed analytically. For computing subtask dispersion we used the formula from [128]:

$$\text{Disp}(\mathbf{d}) = \int_0^\infty 1 - \prod_{i=1}^{N} F_i(x - d_i) - \prod_{i=1}^{N}(1 - F_i(x - d_i))\text{dx} \tag{6.29}$$

Task response time is computed with the PK formula of Eqn (6.7).

The optimisation is done by applying the formulas from [128] for subtask dispersion and task response time. For the computation of energy metric we used the results from Sec. 6.4.3. If service restart was used we transformed the distributions with the formula in Sec. 6.4.2. If server replication was used we transformed the distributions with the formula in Sec. 6.4.1.

Method 1

In this method with no restart or server replication the optimisation was done by only optimising with respect to the delay vector $\mathbf{d}$.

Method 2

In this method with server replication the delay was optimised individually for each server configuration, where $0 < n_i \leq 10$. After this we picked the server configuration which minimised the cost function.

Method 3

In this method with server restart the optimisation the search space was increased into a four dimensional space. Three dimensions for the delay vector $\mathbf{d}$ and an extra dimension for server restart interval in subtask 3 τ_3. Which can potentially benefit from service restart, as it is a heavy tail distribution.

Method 4

In this method with both server restart and server replication the four dimensional delay + server restart interval for subtask 3 was computed individually for each server configuration, where $0 < n_i \leq 10$. After this we picked the server configuration which minimised the cost function.

6.6 Results

We performed two experiments, the first using the Subtask Dispersion vs. Response Time trade-off, and the second using the Response Time vs. Subtask Dispersion vs. Energy trade-off.

6.6.1 Subtask Dispersion vs. Response Time trade-off

We use a three-server split–merge system with exponentially distributed incoming tasks that have an arrival rate of $\lambda = 0.78$ tasks per time unit. The subtask service times are distributed as:

$$\begin{aligned} X_1 &\sim \text{Exp}(\lambda = 1) \\ X_2 &\sim \text{Exp}(\lambda = 5) \\ X_3 &\sim \text{Exp}(\lambda = 10) \end{aligned}$$

Method 1 The resulting delays for the subtasks are: $\mathbf{d} = (0, 0, 0.068)$. Corresponding performance metrics are:

Task response time:	5.286 time units
Subtask dispersion:	0.946 time units
Trade-off:	4.999 (time units)^2

Method 2 The resulting delays for subtasks are: $\mathbf{d} = (0, 0.019, 1.879)$. Corresponding performance metrics are:

Task response time:	5.437 time units
Subtask dispersion:	0.867 time units
Trade-off:	4.718 (time units)^2

Method 3 and 4 The results can be seen in Fig. 6.7, 6.8 and 6.9. The number of delays in the figures indicate how many delays there are to choose from when picking the delay based on a queue length.

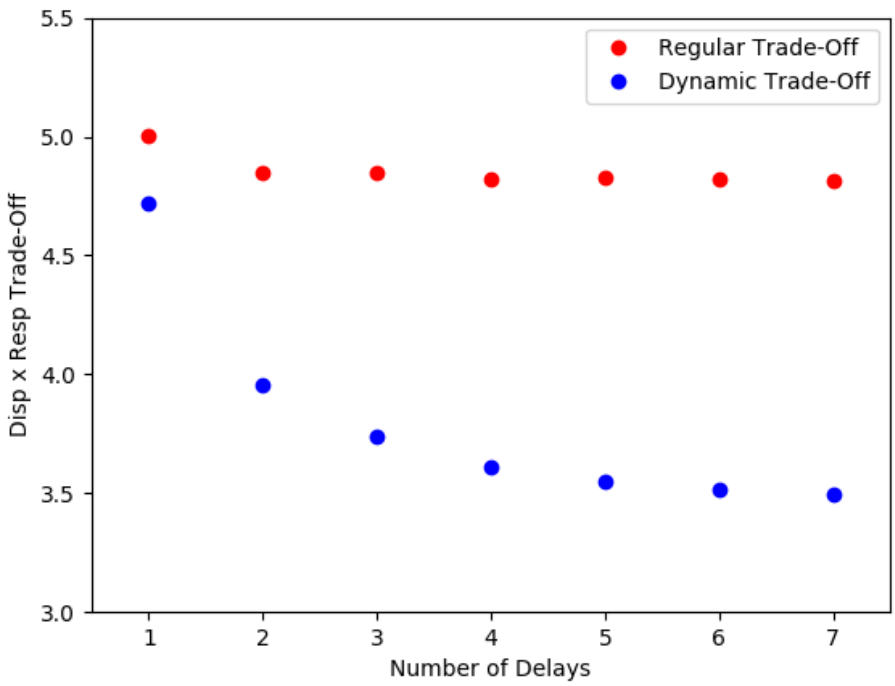

Figure 6.7: Trade-Off Results of **Methods 3 and 4**

The results indicate that the dynamic subtask dispersion technique introduced in Sec. 6.2 is better than the technique introduced by Tsimashenka [128]. **Method 2** is able to produce a 6% decrease in the trade-off cost function when compared against **Method 1**. When **Methods 3 and 4** were compared against their single delay vector counterparts **Methods 1 and 2**, the state-dependent delay vectors introduced in Sec. 6.3 reduced the trade-off cost functions by 4% and 25% respectively.

Fig. 6.8 shows that task response time of dynamic subtask dispersion technique increases as the number of delays is increased. This is caused by the algorithm finding it more worthwhile

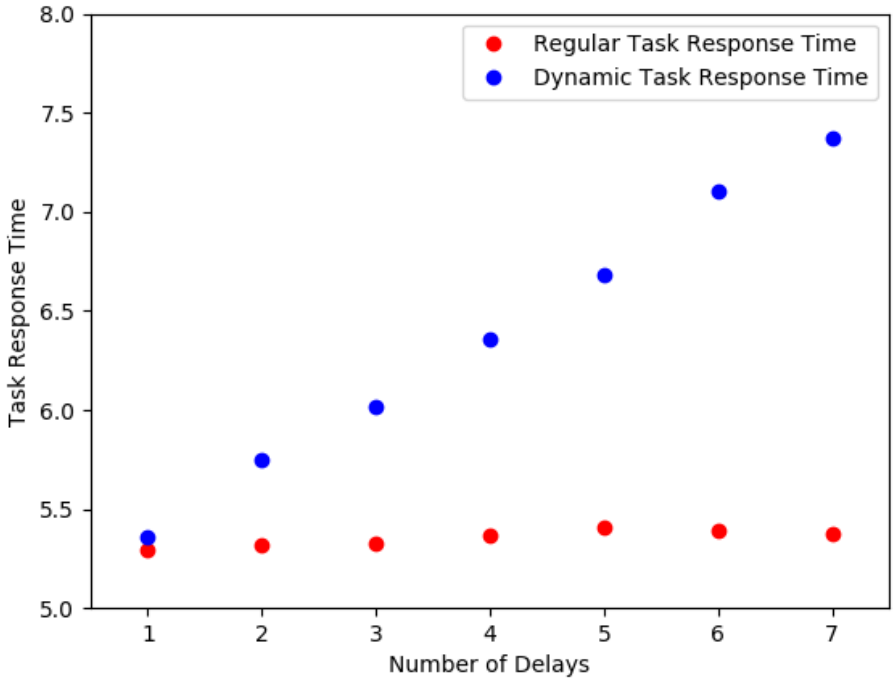

Figure 6.8: Task Response time of **Methods 3 and 4**

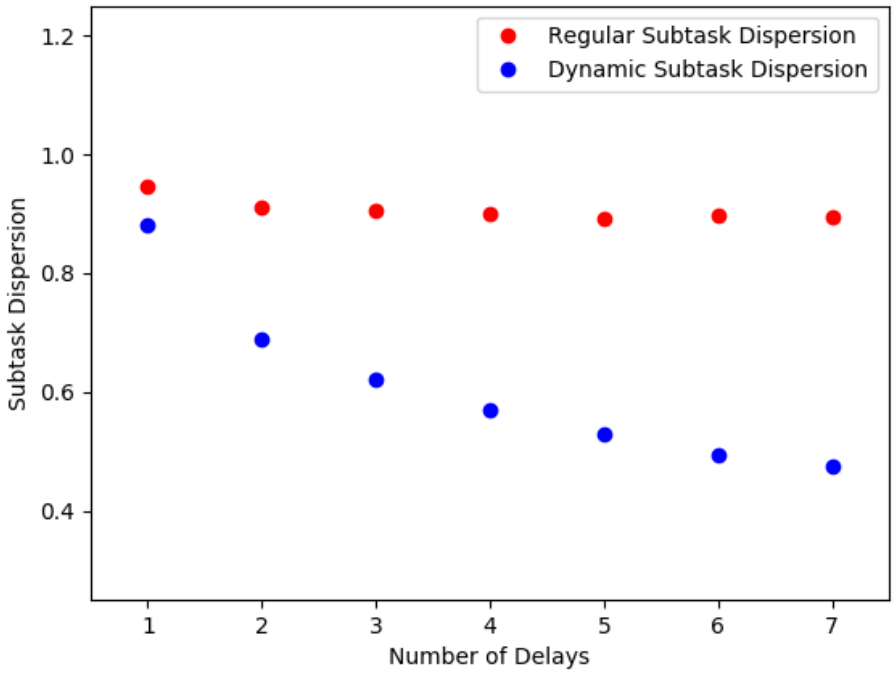

Figure 6.9: Subtask Dispersion of **Methods 3 and 4**

to concentrate on improving subtask dispersion at the expense of task response time, as the percentage wise improvement of subtask dispersion is greater. This then leads to the overall best result in terms of the trade of metric shown Fig. 6.7.

6.6.2 Response Time vs. Subtask Dispersion vs. Energy Trade-Off

In our example case we have a split–merge system with Poisson task arrivals at rate $\lambda = 1.05$ tasks per time unit. The energy metric cost for high utilisation is 1.0 energy units and low utilisation cost is 0.15 energy units. The cost of a restart is 0.1 time units. Subtask service times are distributed as:

$$\begin{aligned} X_1 &\sim \text{Erl}(\text{k} = 5, \lambda = 10) \\ X_2 &\sim \text{Uni}(\text{a} = 0.25, \text{b} = 0.3) \\ X_3 &\sim \frac{1}{2}\text{Exp}(\lambda = 1) + \frac{1}{2}\text{Exp}(\lambda = 5) \end{aligned}$$

Method 1 No service restart or multiple servers were used. The resulting delays for subtasks are: $\mathbf{d} = (0.0, 002, 0)$. Corresponding performance metrics are:

Task response time:	5.729 time units
Subtask dispersion:	0.637 time units
Energy Cost:	1.677 energy units
RDE^3 Trade-off:	17.226 units

Method 2 Subtask server replication factors $\mathbf{n} = (2, 1, 4)$ and no subtask restart was used. The resulting delays for subtasks are: $\mathbf{d} = (0, 0.028, 0.233)$. Corresponding performance metrics are:

Task response time:	0.633 time units
Subtask dispersion:	0.174 time units
Energy Cost:	2.314 energy units
RDE^3 Trade-off:	1.362 units

Method 3 No multiple subtask service servers were used. Optimal restart subtask service

for subtask 3 was ($\tau_3 = 0.363$). The resulting delays for subtasks are: $\mathbf{d} = (0, 0.046, 0)$. Corresponding performance metrics are:

Task response time:	2.241 time units
Subtask dispersion:	0.511 time units
Energy Cost:	1.577 energy units
RDE^3 Trade-off:	4.495 units

Method 4 Subtask server replication factors $\mathbf{n} = (2, 1, 4)$. The optimal restart subtask service for subtask 3 was ($\tau_3 = \infty$). The resulting delays for subtasks are: $\mathbf{d} = (0, 0.028, 0.233)$. The corresponding performance metrics are:

Task response time:	0.633 time units
Subtask dispersion:	0.174 time units
Energy Cost:	2.314 energy units
RDE^3 Trade-off:	1.362 units

The results indicate that multiple servers can be used to improves subtask dispersion, task response time, and energy consumption. Our results show a huge 92% drop in cost associated with the triple trade-off metric as shown by comparison of **Method 1** and **Method 2**. It can also be seen, that the subtask service restart improves the trade-off by 73% when comparing **Method 1** and **Method 3**. However no further improvement was to be gained with the combination of restart and replication. This is shown by **Method 2** and **Method 4** having the same trade-off result, as **Method 4** did not utilise the restart option.

6.7 Conclusion

This chapter used three metrics for measuring the performance of a split–merge system and four techniques that can be used to improve the trade-off of those systems.

Our first set of results demonstrate the superiority of dynamic delays over static delays. Also from first set of results it can be observed that the state-dependent delays introduced in Sec. 6.3

decrease the trade-off cost when compared with a single delay model. The state-dependent delays result in particular is promising as it can be applied in many contexts when optimising between two or more metrics.

From the second set of results, it can be seen that using multiple servers and subtask service reduction are both able to reduce the cost of the trade-off metric defined in Sec. 6.4.4. However, their combination is not able to further reduce the cost, as the combination of the techniques failed to deliver an improvement when compared to subtask service replication alone. This is likely due to both techniques depending on high coefficient of correlation.

An interesting avenue of further research will be to investigate extending our techniques from split–merge queueing systems to fork–join queueing systems. In fork–join systems tasks are queued at the subtask level, which improves task response time, as subtasks from the next task can begin service before all the sibling subtasks of the current task have finished. However, their analysis is generally acknowledged to be considerably harder, due to a lack of a synchronisation point at the start of every task.

Chapter 7

Conclusion

The primary hypothesis of the thesis was that it is possible to control parallel queueing system in a way that improves the performance of the system with respect to an objective function that reflects multiple conflicting performance metrics. In the thesis we took small bite sized chunks of this giant task, and did case analysis on these tractable problems.

We investigated a number of techniques to improve performance and metrics to measure performance and the richness of different approaches has been surprising to us. We believe there is enough evidence to justify the hypothesis that performance of modern systems can and should be measured with respect to multiple metrics, and that performance when compared to existing optimisation routines, which often only optimise with respect to one metric can be greatly improved.

7.1 Key Highlights

We showed that, dynamic delays provide a great performance improvement to systems compared to static delays. The improvement is both in terms of subtask dispersion and task response time. If you have a system with static delays, you can instantly switch it to work with dynamic

delays to gain an improvement[1]. The system can also be re-analysed to take into account the dynamic delay structure, which will result in further performance gains.

We presented a variety of techniques that can essentially talked as the same thing. If you have a task that has a very high coefficient of variance or service potentially never completes, it is better to use restart or replication. This is, because having a fixed server restart interval τ transforms the probability distribution to behave like repeated Bernoulli trial. Meaning, that potentially an ∞ average waiting time becomes finite.

We presented a state-dependent delays as a solution to improve overall performance, a concept which can be easily generalised to be state-dependent service. State-dependent service incorporates the simple idea that during periods of time with heavy load, you should focus on meeting speed of service requirements, as improving speed of service of a single task will improve the speed of service of all tasks currently in the queue (and some that haven't even joined). While doing a good job on the service of the current task will not help you with the service of the incoming tasks.

State-dependent service also has another great advantage, as it makes the system more stable. With non-varying service your service structure will collapse if service arrivals outnumber the amount of tasks you can service. However, if you have a number of varying service levels, where you choose on based on the amount of temporary load the system will automatically self balance itself. In times of high load the system will perform lower quality service, but serve more customers. In times of low load the system will perform service with high quality and speed.

7.2 Future Work

The research carried out during the duration of this thesis has lead to multiple avenues of future research. Below we will list out some of the more promising avenues.

[1]In some special cases (such as very heavy load and deterministic service times), there are no performance gains. As, the dynamic system will perform exactly like the static system.

7.2.1 Optimising Trade-Offs in Fork-Join Systems

In Chapter 3 we compared existing techniques to reduce subtask dispersion in split–merge and fork–join systems. During this comparison two trends stood out. First of all that subtask dispersion is best reduced by dynamic subtask dispersion and secondly task response time is best reduced by utilising a fork–join structure. There was even a small demonstration towards such a technique briefly discussed in Sec. 3.4.3, which demonstrated it can be quite effective.

In addition, a combination of the individual techniques presented in Chapter 6 could be applied on fork–join systems, which would then greatly improve the performance of parallel task processing.

7.2.2 Trade-Offs in Stochastic PERT Networks

In Chapter 4 we investigated Hidden Stochastic PERT networks and did not utilise the knowledge about the inner workings of the PERT network. The optimisation of PERT networks could be of great use in many fields such as supply chain management and high frequency trading. Therefore, it would be of great importance to investigate how the inner workings of Stochastic PERT networks could be used to further optimise systems in the real world.

7.2.3 Making Service Restart More Tolerant to Modelling Error

In the first few sections of Chapter 5 we discuss the current state of the art in determining the service restart interval. However, the model does not take into account potential measurement inaccuracies when determining the underlying distribution, nor does it consider that the underlying distribution might change during operation.

It would be beneficial to introduce a sliding scale for the restart interval, similar to how delays are adjusted depending on the queue length of the system in Sec. 6.3. The technique would compare the expected service time distribution and actual task completions and adjust it if necessary.

This could be done by collecting statistics of service completion for the restart interval and then comparing it against the results the model expects. Then the algorithm could adjust the restart interval if necessary.

7.3 Concluding Remarks

The thesis explored only a small subset of the wast amount of possibilities out there in parallel queueing systems and there is still loads of more research to be done in the industry, for others to carry on with.

Improving Parallel Queueing Networks is becoming an interesting area of research, as the processes of the large technology companies of our day such as Google, Amazon, Facebook... perform the the service of an individual customer in parallel. When a user requests data, each company will needs to perform some of the following tasks: analyse the customer, decide what to display to them, which ads do we display, and so on. There is also an additional requirement that all these jobs need to be performed with a sub second latency to give the customer a fluent experience.

In addition to improving parallel queueing networks being important to optimise, it is also important to develop analysis of associated metrics, which performance is measured by. In the modern world it is becoming more and more important to take into account environmental and other ethical impacts. As well as the varied needs of the customer.

Bibliography

[1] Abate, J., and Whitt, W. Transient behavior of the m/m/l queue: Starting at the origin. *Queueing Systems 2*, 1 (Mar 1987), 41–65.

[2] Abouee-Mehrizi, H., and Baron, O. State-dependent M/G/1 queueing systems. *Queueing Systems 82*, 1-2 (2016), 121–148.

[3] Adan, I. J., and Van der Wal, J. Combining make to order and make to stock. *Operations-Research-Spektrum 20*, 2 (1998), 73–81.

[4] Adlakha, V., and Kulkarni, V. A classified bibliography of research on stochastic PERT networks: 1966–1987. *Information Systems and Operational Research 27*, 3 (1989), 272–296.

[5] Aisen, D., Katsuyama, B., Park, R., Schwall, J., Steiner, R., Zhang, A., and Popejoy, T. Synchronized processing of data by networked computing resources, July 16 2013. US Patent 8,489,747.

[6] Alexandre Laumonier. HFT in my Backyard. `https://sniperinmahwah.wordpress.com/2016/01/26/hft-in-the-banana-land`, September 2014.

[7] Alonso, J., Matias, R., Vicente, E., Maria, A., and Trivedi, K. S. A comparative experimental study of software rejuvenation overhead. *Performance Evaluation 70*, 3 (2013), 231–250.

[8] Ananthanarayanan, G., Ghodsi, A., Shenker, S., and Stoica, I. Why let resources idle? Aggressive cloning of jobs with Dolly. *Memory 40*, 60 (2012), 80.

[9] Andersson, K., Lennartson, B., Falkman, P., and Fabian, M. Generation of restart states for manufacturing cell controllers. *Control Engineering Practice 19*, 9 (2011), 1014–1022.

[10] Andrzejak, A., and Silva, L. Using machine learning for non-intrusive modeling and prediction of software aging. In *Network Operations and Management Symposium, NOMS* (2008), IEEE, pp. 25–32.

[11] Araujo, J., Matos, R., Maciel, P., Vieira, F., Matias, R., and Trivedi, K. S. Software rejuvenation in eucalyptus cloud computing infrastructure: A method based on time series forecasting and multiple thresholds. In *Third International Workshop on Software Aging and Rejuvenation WoSAR* (2011), IEEE, pp. 38–43.

[12] Asmussen, S. *Applied probability and queues*, vol. 51. Springer Science & Business Media, 2008.

[13] Au-Yeung, S. W. M., Harrison, P. G., and Knottenbelt, W. J. Approximate queueing network analysis of patient treatment times. In *Proceedings of the 2nd International Conference on Performance Evaluation Methodologies and Tools* (2007), ValueTools '07, ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), pp. 45:1–45:12.

[14] Avriel, M. *Nonlinear programming: Analysis and methods.* Courier Corporation, 2003.

[15] Baccelli, F., Makowski, A. M., and Shwartz, A. The fork-join queue and related systems with synchronization constraints: Stochastic ordering and computable bounds. *Advances in Applied Probability 21*, 3 (1989), pp. 629–660.

[16] Ballerini, S., Carnevali, L., Paolieri, M., Tadano, K., and Machida, F. Software rejuvenation impacts on a phased-mission system for mars exploration. In *Software Reliability Engineering Workshops ISSREW, 2013 IEEE International Symposium on* (2013), IEEE, pp. 275–280.

[17] Bansal, N., Kimbrel, T., and Pruhs, K. Speed scaling to manage energy and temperature. *Journal of the ACM (JACM) 54*, 1 (2007), 3.

[18] Biere, A. Adaptive restart strategies for conflict driven sat solvers. In *International Conference on Theory and Applications of Satisfiability Testing* (2008), Springer, pp. 28–33.

[19] Blanchette, J. C., Fleury, M., and Weidenbach, C. A verified sat solver framework with learn, forget, restart, and incrementality. In *International Joint Conference on Automated Reasoning* (2016), Springer, pp. 25–44.

[20] Bolch, G., Greiner, S., de Meer, H., and Trivedi, K. S. *Queueing networks and Markov chains: Modeling and performance evaluation with computer science applications.* John Wiley & Sons, 2006.

[21] Borst, S., Mandelbaum, A., and Reiman, M. I. Dimensioning large call centers. *Operations research 52*, 1 (2004), 17–34.

[22] Bovenzi, A., Alonso, J., Yamada, H., Russo, S., and Trivedi, K. S. Towards fast os rejuvenation: An experimental evaluation of fast os reboot techniques. In *24th International Symposium on Software Reliability Engineering ISSRE* (2013), IEEE, pp. 61–70.

[23] Brockes, E. Michael Lewis: 'Wall Street has gone insane'. *The Guardian* (April 2014).

[24] Brockmeyer, E., Halstrm, H., Jensen, A., and Erlang, A. K. The life and works of AK erlang. *Transactions of the Danish Academy of Technical Sciences* (1948).

[25] Chen, W., Toueg, S., and Aguilera, M. K. On the quality of service of failure detectors. In *Proceedings of the International Conference on Dependable Systems and Networks DSN* (2000), pp. 191–200.

[26] Cook, D. L. *Program Evaluation and review technique–Applications in education.* Office of Education, 1966.

[27] Cotroneo, D., Fucci, F., Iannillo, A. K., Natella, R., and Pietrantuono, R. Software aging analysis of the android mobile os. In *27th International Symposium on Software Reliability Engineering ISSRE* (2016), IEEE, pp. 478–489.

[28] Cotroneo, D., Natella, R., Pietrantuono, R., and Russo, S. Software aging analysis of the linux operating system. In *21st International Symposium on Software Reliability Engineering ISSRE* (2010), IEEE, pp. 71–80.

[29] Cotroneo, D., Natella, R., Pietrantuono, R., and Russo, S. Software aging and rejuvenation: Where we are and where we are going. In *Third International Workshop on Software Aging and Rejuvenation WoSAR* (2011), IEEE, pp. 1–6.

[30] Cotroneo, D., Natella, R., Pietrantuono, R., and Russo, S. A survey of software aging and rejuvernation studies. *ACM Journal on Emerging Technologies in Computing Systems 10* (2014).

[31] Cotroneo, D., Orlando, S., Pietrantuono, R., and Russo, S. A measurement-based ageing analysis of the JVM. *Software Testing, Verification and Reliability 23*, 3 (2013), 199–239.

[32] Da Silva, A. P. C., Meo, M., and Marsan, M. A. Energy-performance trade-off in dense wlans: A queuing study. *Computer Networks 56*, 10 (2012), 2522–2537.

[33] Danilkina, A., Reinecke, P., and Wolter, K. Sfera: a simulation framework for the performance evaluation of restart algorithms in service-oriented systems. *Electronic Notes in Theoretical Computer Science 291* (2013), 3–14.

[34] David, H. A., and Nagaraja, H. N. *Wiley Series in Probability and Statistics*. John Wiley & Sons, 1980.

[35] Dean, J., and Barroso, L. A. The tail at scale. *Communications of the ACM 56*, 2 (2013), 74–80.

[36] Dohi, T., Goševa-Popstojanova, K., and Trivedi, K. Estimating software rejuvenation schedules in high-assurance systems. *The Computer Journal 44*, 6 (2001), 473–485.

[37] Dohi, T., and Okamura, H. Dynamic software availability model with rejuvenation. *Journal of the Operations Research Society of Japan 59*, 4 (2016), 270–290.

[38] Dong, X., Chen, P., Huang, H., and Nowak, M. A multi-restart iterated local search algorithm for the permutation flow shop problem minimizing total flow time. *Computers & Operations Research 40*, 2 (2013), 627–632.

[39] Erlang, A. K. The theory of probabilities and telephone conversations. *Nyt Tidsskrift for Matematik B 20*, 6 (1909), 87–98.

[40] Flatto, L. Two parallel queues created by arrivals with two demands II. *SIAM Journal on Applied Mathematics 45*, 5 (1985), pp. 861–878.

[41] Flatto, L., and Hahn, S. Erratum: Two parallel queues created by arrivals with two demands I. *SIAM Journal on Applied Mathematics 45*, 1 (1985), 168–168.

[42] Forouzan, B. A. *TCP/IP protocol suite.* McGraw-Hill, Inc., 2002.

[43] Fourneau, J.-M., and Wolter, K. Mixed networks with multiple classes of customers and restart. In *International Conference on Analytical and Stochastic Modeling Techniques and Applications* (2015), Springer, pp. 73–86.

[44] Fourneau, J.-M., Wolter, K., Reinecke, P., Krauss, T., and Danilkina, A. Multiple class G-networks with restart. In *Proceedings of the 4th ACM/SPEC International Conference on Performance Engineering* (2013), ACM, pp. 39–50.

[45] Fujiwara, Y., Nakatsuji, M., Onizuka, M., and Kitsuregawa, M. Fast and exact top-k search for random walk with restart. *Proceedings of the VLDB Endowment 5*, 5 (2012), 442–453.

[46] Fulkerson, D. R. Expected critical path lengths in PERT networks. *Operations Research 10*, 6 (1962), 808–817.

[47] Gagliolo, M., and Schmidhuber, J. Learning restart strategies. In *IJCAI* (2007), pp. 792–797.

[48] Gandhi, A., Gupta, V., Harchol-Balter, M., and Kozuch, M. A. Optimality analysis of energy-performance trade-off for server farm management. *Performance Evaluation 67*, 11 (2010), 1155 – 1171.

[49] Gandhi, A., Harchol-Balter, M., and Adan, I. Server farms with setup costs. *Performance Evaluation 67*, 11 (2010), 1123 – 1138.

[50] García-Nieto, J., and Alba, E. Restart particle swarm optimization with velocity modulation: a scalability test. *Soft Computing 15*, 11 (2011), 2221–2232.

[51] Gaudiosi, J. Global esports revenues to surpass $1.9 billion by 2018. `http://fortune.com/2015/10/28/global-esports-revenues-nearing-2-billion`, October 2015. Accessed: 2017-10-27.

[52] Gebennini, E., Grassi, A., Fantuzzi, C., Gershwin, S. B., and Schick, I. C. Discrete time model for two-machine one-buffer transfer lines with restart policy. *Annals of Operations Research 209*, 1 (2013), 41–65.

[53] Gupta, U., and Rao, T. S. On the analysis of single server finite queue with state dependent arrival and service processes: M(n)/G(n)/1/K. *Operations-Research-Spektrum 20*, 2 (1998), 83–89.

[54] Hansen, N. The CMA evolution strategy: A tutorial, 2016.

[55] Hansen, N., Auger, A., Ros, R., Finck, S., and Pošík, P. Comparing results of 31 algorithms from the black-box optimization benchmarking BBOB-2009. In *Proceedings of 12th Annual Conference Companion on Genetic and Evolutionary Computation* (2010), pp. 1689–1696.

[56] Hansen, N., Müller, S. D., and Koumoutsakos, P. Reducing the time complexity of the derandomized evolution strategy with covariance matrix adaptation (CMA-ES). *Evolutionary Computation 11*, 1 (2003), 1–18.

[57] Harris, C. M. Queues with state-dependent stochastic service rates. *Operations Research 15*, 1 (1967), 117–130.

[58] Harrison, P., and Zertal, S. Queueing models of RAID systems with maxima of waiting times. *Performance Evaluation 64*, 78 (2007), 664–689.

[59] Harrison, P. G., and Knottenbelt, W. J. Passage time distributions in large Markov chains. In *ACM SIGMETRICS Performance Evaluation Review* (2002), vol. 30, ACM, pp. 77–85.

[60] HE, Y.-z., XIONG, J.-j., and Zhan, W. A research on software rejuvenation policy based on DNA computing. *DEStech Transactions on Computer Science and Engineering*, aics (2016).

[61] Heidelberger, P., and Trivedi, K. S. Analytic queueing models for programs with internal concurrency. *IEEE Trans. Comput. 32*, 1 (Jan. 1983), 73–82.

[62] Hsu, Y.-L., Ke, J.-C., and Liu, T.-H. Standby system with general repair, reboot delay, switching failure and unreliable repair facilitya statistical standpoint. *Mathematics and Computers in Simulation 81*, 11 (2011), 2400–2413.

[63] Hsu, Y.-L., Ke, J.-C., Liu, T.-H., and Wu, C. H. Modeling of multi-server repair problem with switching failure and reboot delay and related profit analysis. *Computers & Industrial Engineering 69* (2014), 21–28.

[64] Intel. Enhanced intel speedstep technology for the intel pentium m processor. webpage, 2004.

[65] Irani, S., Shukla, S., and Gupta, R. Algorithms for power savings. *ACM Transactions on Algorithms (TALG) 3*, 4 (2007), 41.

[66] Jain, M. Availability prediction of imperfect fault coverage system with reboot and common cause failure. *International Journal of Operational Research 17*, 3 (2013), 374–397.

[67] Jain, M., Manjula, T., and Gulati, T. Software rejuvenation policies for cluster system. *Proceedings of the National Academy of Sciences, India Section A: Physical Sciences 86*, 3 (2016), 339–346.

[68] Jain, M., and Preeti. Availability analysis of software rejuvenation in active/standby cluster system. *International Journal of Industrial and Systems Engineering 19*, 1 (2014), 75–93.

[69] Jain, M., and Rani, S. Availability analysis for repairable system with warm standby, switching failure and reboot delay. *International Journal of Mathematics in Operational Research 5*, 1 (2013), 19–39.

[70] Jain, M., Shekhar, C., and Rani, V. N-policy for a multi-component machining system with imperfect coverage, reboot and unreliable server. *Production & Manufacturing Research 2*, 1 (2014), 457–476.

[71] Jennings, O. B., Mandelbaum, A., Massey, W. A., and Whitt, W. Server staffing to meet time-varying demand. *Management Science 42*, 10 (1996), 1383–1394.

[72] Kerner, Y. The conditional distribution of the residual service time in the $M_n/G/1$ queue. *Stochastic Models 24*, 3 (2008), 364–375.

[73] Khan, I. Dota 2's The International 7 breaks esports prize pool record. `www.espn.co.uk/esports/story/_/id/19861533/dota-2-international-7-breaks-esports-prize-pool-record`, August 2017. Accessed: 2017-11-22.

[74] Kim, C., and Agrawala, A. K. Analysis of the fork-join queue. *IEEE Transactions on computers 38*, 2 (1989), 250–255.

[75] Kim, T. H., Lee, K. M., and Lee, S. U. Generative image segmentation using random walks with restart. In *European conference on computer vision* (2008), Springer, pp. 264–275.

[76] Kim, T. H., Lee, K. M., and Lee, S. U. Edge-preserving colorization using data-driven random walks with restart. In *16th International Conference on Image Processing ICIP* (2009), IEEE, pp. 1661–1664.

[77] Kingman, J. F. C. The first Erlang century—and the next. *Queueing Systems 63*, 1 (Nov 2009), 3.

[78] Kourai, K., and Chiba, S. Fast software rejuvenation of virtual machine monitors. *IEEE Transactions on Dependable and Secure Computing 8*, 6 (2011), 839–851.

[79] Krishnamurthy, B., and Rexford, J. *Web Protocols and Practice*. Addison Wesley, 2001.

[80] Lebrecht, A., Dingle, N. J., and Knottenbelt, W. J. Modelling Zoned RAID Systems using Fork-Join Queueing Simulation. In *6th European Performance Engineering Workshop EPEW* (2009), vol. 5652 of *Lecture Notes in Computer Science*, pp. 16–29.

[81] Lebrecht, A. S., Dingle, N. J., and Knottenbelt, W. J. Analytical and simulation modelling of zoned RAID systems. *The Computer Journal 54*, 5 (May 2011), 691–707.

[82] Lebrecht, A. S., and Knottenbelt, W. J. Response time approximations in fork-join queues. In *23rd UK Performance Engineering Workshop UKPEW* (2007).

[83] Loshchilov, I., Schoenauer, M., and Sebag, M. Alternative restart strategies for CMA-ES. In *International Conference on Parallel Problem Solving from Nature* (2012), Springer, pp. 296–305.

[84] Machida, F., Kim, D. S., and Trivedi, K. S. Modeling and analysis of software rejuvenation in a server virtualized system. In *Second International Workshop on Software Aging and Rejuvenation WoSAR* (2010), IEEE, pp. 1–6.

[85] Machida, F., and Miyoshi, N. Analysis of an optimal stopping problem for software rejuvenation in a deteriorating job processing system. *Reliability Engineering & System Safety* (2017).

[86] Machida, F., Xiang, J., Tadano, K., and Maeno, Y. Combined server rejuvenation in a virtualized data center. In *9th International Conference on Ubiquitous*

Intelligence & Computing and 9th International Conference on Autonomic & Trusted Computing UIC/ATC (2012), IEEE, pp. 486–493.

[87] Maisonnave, P., Sanchez, I., Moine, V., Dequin, S., and Galeote, V. Stuck fermentation: Development of a synthetic stuck wine and study of a restart procedure. *International journal of food microbiology 163*, 2 (2013), 239–247.

[88] Marin, A., and Rossi, S. Power control in saturated fork-join queueing systems. *Performance Evaluation* (2017).

[89] Marin, A., Rossi, S., and Sottana, M. Biased processor sharing in fork-join queues. In *Quantitative Evaluation of Systems* (2018), A. McIver and A. Horvath, Eds., Springer International Publishing, pp. 273–288.

[90] Markov, A. Extension of the law of large numbers to quantities, depending on each other (1906). reprint. *Journal lectronique d'Histoire des Probabilits et de la Statistique [electronic only] 2*, 1b (2006), Article 10, 12 p., electronic only–Article 10, 12 p., electronic only.

[91] Maurer, S. M., and Huberman, B. A. Restart strategies and Internet congestion. *Journal of Economic Dynamics and Control 25* (2001), 641–654.

[92] Meng, H., Hei, X., Zhang, J., Liu, J., and Sui, L. Software aging and rejuvenation in a J2EE application server. *Quality and Reliability Engineering International 32*, 1 (2016), 89–97.

[93] Moody, A., Bronevetsky, G., Mohror, K., and Supinski, B. R. d. Design, modeling, and evaluation of a scalable multi-level checkpointing system. In *Proceedings of the 2010 ACM/IEEE international conference for high performance computing, networking, storage and analysis* (2010), IEEE Computer Society, pp. 1–11.

[94] Nagaraju, V., Basavaraj, V. V., and Fiondella, L. Software rejuvenation of a fault-tolerant server subject to correlated failure. In *Reliability and Maintainability Symposium (RAMS), 2016 Annual* (2016), IEEE, pp. 1–6.

[95] Nelson, R., and Tantawi, A. N. Approximate analysis of fork/join synchronization in parallel queues. *IEEE transactions on computers 37*, 6 (1988), 739–743.

[96] Okamura, H., and Dohi, T. Optimization of opportunity-based software rejuvenation policy. In *23rd International Symposium on Software Reliability Engineering Workshops ISSREW* (2012), IEEE, pp. 283–286.

[97] Okamura, H., and Dohi, T. Dynamic software rejuvenation policies in a transaction-based system under markovian arrival processes. *Performance Evaluation 70*, 3 (2013), 197–211.

[98] Okamura, H., and Dohi, T. Optimal trigger time of software rejuvenation under probabilistic opportunities. *IEICE TRANSACTIONS on Information and Systems 96*, 9 (2013), 1933–1940.

[99] Okamura, H., and Dohi, T. Analysis of optimal restart policies for software systems. *Journal of Japan Industrial Technology Association (In Japanese) 66*, 4E (2016), 416–425.

[100] Odonoghue, B., and Candes, E. Adaptive restart for accelerated gradient schemes. *Foundations of computational mathematics 15*, 3 (2015), 715–732.

[101] Pelikan, M., Goldberg, D. E., and Cantú-Paz, E. BOA: The Bayesian optimization algorithm. In *Proceedings of the 1st Annual Conference on Genetic and Evolutionary Computation-Volume 1* (1999), Morgan Kaufmann Publishers Inc., pp. 525–532.

[102] Pesu, T., Kettunen, J., Wolter, K., and Knottenbelt, W. J. Three-way optimisation of response time, subtask dispersion and energy consumption in split–merge systems. In *Proceedings of the 11th EAI International Conference on Performance Evaluation Methodologies and Tools* (2017), EAI.

[103] Pesu, T., and Knottenbelt, W. J. Dynamic subtask dispersion reduction in heterogeneous parallel queueing systems. *Electronic Notes in Theoretical Computer Science 318* (2015), 129 – 142.

[104] Pollaczek, F. Über eine aufgabe der wahrscheinlichkeitstheorie. i. *Mathematische Zeitschrift 32*, 1 (1930), 64–100.

[105] Popper, N. Stock exchange prices grow so convoluted even traders are confused, study finds. `https://www.nytimes.com/2016/03/02/business/dealbook/stock-exchange-prices-grow-so-convoluted-even-traders-are-confused-study-finds.html`, march 2016. Accessed: 2017-09-14.

[106] Pruhs, K., Uthaisombut, P., and Woeginger, G. Getting the best response for your erg. *ACM Transactions on Algorithms (TALG) 4*, 3 (2008), 38.

[107] Reinecke, P., van Moorsel, A., and Wolter, K. A measurement study of the interplay between application level restart and transport protocol. In *Proceedings of International Service Availability Symposium ISAS* (May 2004), no. 3335 in Lecture Notes in Computer Science, Springer.

[108] Reinecke, P., and Wolter, K. Adaptivity metric and performance for restart strategies in web services reliable messaging. In *Proceedings of the 7th international workshop on Software and performance* (2008), ACM, pp. 201–212.

[109] Reinecke, P., and Wolter, K. A simulation study on the effectiveness of restart and rejuvenation to mitigate the effects of software ageing. In *Second International Workshop on Software Aging and Rejuvenation WoSAR* (2010), IEEE, pp. 1–6.

[110] Rinsaka, K., and Dohi, T. A faster estimation algorithm for periodic preventive rejuvenation schedule maximizing system availability. In *International Service Availability Symposium* (2007), Springer, pp. 94–109.

[111] Rinsaka, K., and Dohi, T. Toward high assurance software systems with adaptive fault management. *Software Quality Journal 24*, 1 (2016), 65–85.

[112] Rizk, A., Poloczek, F., and Ciucu, F. Computable bounds in fork-join queueing systems. In *ACM SIGMETRICS Performance Evaluation Review* (2015), vol. 43, ACM, pp. 335–346.

[113] Rizk, A., Poloczek, F., and Ciucu, F. Stochastic bounds in fork–join queueing systems under full and partial mapping. *Queueing Systems 83*, 3-4 (2016), 261–291.

[114] Ruan, Y., Horvitz, E., and Kautz, H. Restart Policies with Dependence among Runs: A Dynamic Programming Approach. In *Proceedings of the Eight International Conference on Principles and Practice of Constraint Programming* (Ithaca, NY, USA, Sept. 2002).

[115] Salfner, F., and Wolter, K. Analysis of service availability for time-triggered rejuvenation policies. *Journal of Systems and Software 83*, 9 (2010), 1579–1590.

[116] Sculli, D. The completion time of PERT networks. *Journal of the Operational Research Society 34*, 2 (1983), 155–158.

[117] Shanthikumar, . J. On a single-server queue with state-dependent service. *Naval Research Logistics (NRL) 26*, 2 (1979), 305–309.

[118] Shylo, O. V., Middelkoop, T., and Pardalos, P. M. Restart strategies in optimization: parallel and serial cases. *Parallel Computing 37*, 1 (2011), 60–68.

[119] Simeonov, D., and Avresky, D. Proactive software rejuvenation based on machine learning techniques. In *International Conference on Cloud Computing* (2009), Springer, pp. 186–200.

[120] The GPyOpt authors. GPyOpt: A bayesian optimization framework in Python. `http://github.com/SheffieldML/GPyOpt`, 2016. Accessed: 2018-09-20.

[121] The Scipy community. Numpy.random.power. `https://docs.scipy.org/doc/numpy-1.13.0/reference/generated/numpy.random.power.html`. Accessed: 2018-08-06.

[122] Tong, H., Faloutsos, C., and Pan, J.-Y. Random walk with restart: Fast solutions and applications. *Knowledge and Information Systems 14*, 3 (Mar 2008), 327–346.

[123] Torquato, M., Maciel, P., Araujo, J., and Umesh, I. An approach to investigate aging symptoms and rejuvenation effectiveness on software systems. In *12th Iberian Conference on Information Systems and Technologies CISTI* (2017), IEEE, pp. 1–6.

[124] Tseng, P.-H., Wang, N.-C., Lin, R.-M., and Chen, K.-T. On the battle between lag and online gamers. In *IEEE International Workshop Technical Committee on Communications Quality and Reliability CQR* (2011), IEEE, pp. 1–6.

[125] Tsimashenka, I. *Reducing Subtask Dispersion in Parallel Queueing Systems.* PhD thesis, Imperial College London, 2014.

[126] Tsimashenka, I., Knottenbelt, W., and Harrison, P. Controlling variability in split-merge systems. In *Analytical and Stochastic Modeling Techniques and Applications*, vol. 7314 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2012, pp. 165–177.

[127] Tsimashenka, I., and Knottenbelt, W. J. Reduction of subtask dispersion in fork-join systems. In *Computer Performance Engineering*, vol. 8168 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2013, pp. 325–336.

[128] Tsimashenka, I., and Knottenbelt, W. J. Trading off subtask dispersion and response time in split-merge systems. In *Analytical and Stochastic Modeling Techniques and Applications*, vol. 7984 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2013, pp. 431–442.

[129] Van Moorsel, A. P., and Wolter, K. Analysis of restart mechanisms in software systems. *IEEE Transactions on Software Engineering 32*, 8 (2006), 547–558.

[130] van Moorsel, A. P. A., and Wolter, K. Analysis of restart mechanisms in software systems. *IEEE Transactions on Software Engineering 32*, 8 (August 2006), 547–558.

[131] Varki, E. Response time analysis of parallel computer and storage systems. *IEEE Transactions on Parallel and Distributed Systems 12*, 11 (2001), 1146–1161.

[132] Varki, E., Merchant, A., and Chen, H. The M/M/1 fork-join queue with variable sub-tasks. *unpublished, available online* (2008).

[133] Wang, Q. *Restart in Mobile Offloading.* PhD thesis, Freie Universität Berlin, 2015.

[134] Wang, Q., and Wolter, K. Detection and analysis of performance deterioration in mobile offloading system. In *Software Reliability Engineering Workshops ISSREW* (2014), IEEE, pp. 420–425.

[135] Wang, Q., and Wolter, K. Automated adaptive restart for accelerating task completion in cloud offloading systems. In *International Conference on Autonomic Computing ICAC* (2015), IEEE, pp. 157–158.

[136] Wang, Q., and Wolter, K. Reducing task completion time in mobile offloading systems through online adaptive local restart. In *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering* (2015), ACM, pp. 3–13.

[137] Wang, Q., and Wolter, K. Accelerating task completion in mobile offloading systems through adaptive restart. *Software & Systems Modeling* (2016), 1–17.

[138] Wolff, R. W. Poisson arrivals see time averages. *Operations Research 30*, 2 (1982), 223–231.

[139] Wolter, K. *Stochastic models for fault tolerance: Restart, rejuvenation and checkpointing.* Springer Science & Business Media, 2010.

[140] Wolter, K., and Reinecke, P. Restart in competitive environments. In *24th UK Performance Engineering Workshop 3–4 UKPEW* (July 2008), Citeseer.

[141] Yamakita, K., Yamada, H., and Kono, K. Phase-based reboot: Reusing operating system execution phases for cheap reboot-based recovery. In *41st International Conference On Dependable Systems & Networks DSN* (2011), IEEE, pp. 169–180.

[142] Yang, M., Min, G., Yang, W., and Li, Z. Software rejuvenation in cluster computing systems with dependency between nodes. *Computing 96*, 6 (2014), 503–526.

[143] Yao, F., Demers, A., and Shenker, S. A scheduling model for reduced CPU energy. In *Proceedings of 36th Annual Symposium on Foundations of Computer Science* (1995), IEEE, pp. 374–382.

[144] Zander, S., Leeder, I., and Armitage, G. Achieving fairness in multiplayer network games through automated latency balancing. In *Proceedings of ACM SIGCHI ACE* (2005), pp. 117–124.

[145] Zerpa, L. E., Sloan, E. D., Koh, C., Sum, A., et al. Hydrate risk assessment and restart-procedure optimization of an offshore well using a transient hydrate prediction model. *Oil and Gas Facilities 1*, 05 (2012), 49–56.

Appendices

Appendix A

Probability Distributions

This section contains the formal definitions of all probability distributions used in this thesis.

A.1 Exponential Distribution

The exponential distribution is a continuous distribution with an interesting property called the memoryless property. That is, when the distribution is renormalised to take into account that time has passed without completion, the distribution looks identical to the original distribution. The exponential distribution has one parameter $\lambda > 0$, which determines the distribution. The average value of an exponentially distributed random variable is $\frac{1}{\lambda}$.

The probability density function is seen below:

$$\mathrm{Exp}(\lambda) = \begin{cases} \lambda e^{-\lambda x} & x \geq 0, \\ 0 & x < 0 \end{cases}$$

The cumulative distribution function is:

$$\int_{-\infty}^{x} \mathrm{Exp}(\lambda)\mathrm{d}x = \begin{cases} 1 - e^{-\lambda x} & x \geq 0 \\ 0 & x < 0 \end{cases}$$

A.2 Hyper-Exponential Distribution

The hyper-exponential distribution is a continuous distribution. The distribution consists of n ($n \subset \mathbb{N}$) exponential distributions, each of which has a p_i and a λ_i associated with it. p_i is the probability that the exponential function i is picked ($\sum p_i = 1$), and $\lambda_i > 0$ is the corresponding λ of the ith exponential function. When sampling a hyper-exponential random variable you first pick one of the exponential distributions and then use the distribution to generate a random variable. The average of a hyper-exponential distribution is $\sum \frac{p_i}{\lambda_i}$.

The probability density function is:

$$\text{HyperExp}(n, \lambda) = \sum_{i=1}^{n} p_i \text{Exp}(\lambda_i)$$

The cumulative distribution function is:

$$\int_{-\infty}^{x} \text{HyperExp}(n, \lambda) \mathrm{d}x = \int_{-\infty}^{x} \sum_{i=1}^{n} p_i \text{Exp}(\lambda_i) \mathrm{d}x$$

A.3 Erlang Distribution

The Erlang distribution is a continuous distribution, which is equal to the sum of $n \subset \mathbb{N}$ sequentially-completed exponentially-distributed tasks, where each task has a service rate of λ, where $\lambda > 0$. The distribution is the result of convolving the distribution $\text{Exp}(\lambda)$ n-times. The Erlang distribution has two parameters: $n \subset \mathbb{N}$ and $\lambda > 0$. The average value of a Erlang distributed random variable is $\frac{n}{\lambda}$.

The probability density function is:

$$\text{Erl}(n, \lambda) = \begin{cases} \frac{1-e^{-\lambda x}}{(n-1)!} & x \geq 0 \\ 0 & x < 0 \end{cases}$$

The cumulative distribution function is:

$$\int_{-\infty}^{x} \mathrm{Erl}(n,\lambda)\mathrm{d}x = \begin{cases} 1 - \sum_{k=0}^{n-1} \frac{e^{-\lambda x}(\lambda x)^n}{n!} & x \geq 0 \\ 0 & x < 0 \end{cases}$$

A.4 Hypo-Exponential Distribution

The hypo-exponential function is a continuous distribution and also happens to be a generalisation of the Erlang distribution. In the Erlang distribution the $n \subset \mathbb{N}$ sequentially completed exponential tasks all have the same λ. In the Hypo-exponential distribution, each task has a unique λ_i corresponding to the ith task, where $\lambda_i > 0$. The Hypo-exponential probability density function is computed by convolving all the individual exponential density functions together. The average of a hypo-exponential distribution is $\sum \frac{1}{\lambda_i}$.

The probability density function is:

$$\mathrm{HypoExp}(n,\lambda) = \bigstar_{i=1}^{n} p_i \mathrm{Exp}(\lambda_i)$$

The cumulative distribution function is:

$$\int_{-\infty}^{x} \mathrm{HypoExp}(n,\lambda)\mathrm{d}x = \int_{-\infty}^{x} \bigstar_{i=1}^{n} p_i \mathrm{Exp}(\lambda_i)\mathrm{d}x$$

The $\bigstar_{i=1}^{n}$ symbol denotes convolving together n functions.

A.5 Pareto Distribution

The Pareto distribution is a continuous distribution, which takes as input $a > 0$ and $b > 0$. a is referred to as the shape and b as the rate of the distribution. The Pareto distribution is a power-law distribution that describes many types of observable phenomena such as wealth

distribution. The Pareto distribution is often referred to by the 80–20 rule. For example, it is said that 20% of people control 80% of the wealth.

The probability density function is:

$$\mathrm{Par}(a,b) = \begin{cases} \frac{a \cdot b^a}{x^{a+1}}, & \text{for} \quad x \geq b \\ 0 & \text{otherwise} \end{cases}$$

The cumulative distribution function is:

$$\int_{-\infty}^{x} \mathrm{Par}(a,b)\mathrm{d}x = \begin{cases} 1 - \left(\frac{b}{x}\right)^a, & \text{for} \quad x \geq b \\ 0 & \text{otherwise} \end{cases}$$

A.6 Normal distribution

The Normal distribution takes $\mu \subset \mathbb{R}$ and $\sigma^2 > 0$ as parameters. The normal distribution is an important function in statistics. The central limit theorem states that the distribution of the sum of n independent identically distributed random variables converges towards the normal distribution (given finite mean and variance). Given a distribution with mean m and variance v and n trials, the corresponding normal distribution is $\mathcal{N}(\mu = mn, \sigma^2 = nv)$. A normally distributed random variable has a mean of μ and variance of σ^2.

The probability density function is:

$$\mathcal{N}(\mu, \sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right)$$

The cumulative distribution function of the Normal distribution is not representable in terms of elementary functions, and needs to be numerically approximated. The cumulative distribution function is usually represented as:

$$\int_{-\infty}^{x} \mathcal{N}(\mu, \sigma^2)\mathrm{d}x = \Phi\left(\frac{x-\mu}{\sigma}\right)$$

where $\Phi(x)$ is the cumulative distribution function of the normal distribution with parameters $\mu = 0$ and $\sigma = 1$

A.7 Folded Normal Distribution

The folded Normal distribution takes $\mu \subset \mathbb{R}$ and $\sigma^2 > 0$ as parameters. The folded Normal distribution corresponds to the absolute value of Normal distribution. The average value of a Folded Normal distributed random value is $\mu_Y = \sigma\sqrt{\frac{2}{\pi}}\exp\left(-\frac{\mu^2}{2\sigma^2}\right) + \mu(1 - 2\Phi(\frac{-\mu}{\sigma}))$ and variance is $\sigma_Y^2 = \mu^2 + \sigma^2 + \mu_Y$.

The probability density function is:

$$\mathcal{N}_F(\mu, \sigma^2) = \begin{cases} \frac{1}{\sqrt{2\pi\sigma^2}}\exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right) + \frac{1}{\sqrt{2\pi\sigma^2}}\exp\left(-\frac{(x+\mu)^2}{2\sigma^2}\right) & x \geq 0 \\ 0 & x < 0 \end{cases}$$

The cumulative distribution function of the folded Normal distribution is not re-presentable in terms of elementary functions, and needs to be numerically approximated. The cumulative distribution function is usually represented as:

$$\int_{-\infty}^{x} \mathcal{N}_F(\mu, \sigma^2)\mathrm{d}x = \begin{cases} \Phi(\frac{x-\mu}{\sigma}) + \Phi(\frac{x+\mu}{\sigma}) - 1 & x \geq 0 \\ 0 & x < 0 \end{cases}$$

where $\Phi(x)$ is the cumulative distribution function of the normal distribution with parameters $\mu = 0$ and $\sigma = 1$

A.8 Log-normal distribution

The log-normal distribution takes $\mu \subset \mathbb{R}$ and $\sigma^2 > 0$ as parameters. The log-normal distribution is defined on a range of $x \in (0, \infty)$. The logarithm of the random variable is normally

distributed. The log-normal distribution is the product of n identically distributed random variables (given finite mean and variance). A log-normally distributed random variable has a average of $\exp(\mu + \frac{\sigma^2}{2})$ and variance of $(\exp(\sigma^2) - 1)\exp(2\mu + \sigma^2)$.

The probability density function is:

$$\mathcal{N}_{\mathcal{L}}(\mu, \sigma^2) = \frac{1}{x\sigma\sqrt{2\pi}} \exp\Big(- \frac{(\ln(x) - \mu)^2}{2\sigma^2}\Big)$$

The cumulative distribution function of the log-normal distribution is not representable in terms of elementary functions, and needs to be numerically approximated. The cumulative distribution function is usually represented as:

$$\int_0^x \mathcal{N}_{\mathcal{L}}(\mu, \sigma^2)\mathrm{d}x = \Phi\Big(\frac{\ln(x) - \mu}{\sigma}\Big)$$

A.9 Uniform Distribution

The uniform distribution is continuous distribution, which takes a and b as parameters, with $a < b$. A uniformly distributed random variable has a constant probability to be any value between a and b and zero probability to be any other value. The average value of a uniformly distributed random variable is $\frac{a+b}{2}$.

The probability density function is:

$$\mathrm{Uni}(a, b) = \begin{cases} (b-a)^{-1} & a \leq x \leq b, \\ 0 & \text{otherwise} \end{cases}$$

The cumulative distribution function is:

$$\int_{-\infty}^x \mathrm{Uni}(a, b)\mathrm{d}x = \begin{cases} 0 & x < a \\ \frac{x-a}{b-a} & a \leq x \leq b \\ 1 & x > b \end{cases}$$

A.10 Power Distribution

The power distribution is a continuous distribution, which takes $a > 0$ as a parameter. a defines the exponent of decay for the function. The power distribution is the inverse of the Pareto distribution [121], with the x and y switched. The average value of a power distributed random variable is $a/(a+1)$.

The probability density function is:

$$\mathrm{Pow}(a) = \begin{cases} ax^{a-1} & 0 \leq x \leq 1 \\ 0 & \text{otherwise} \end{cases}$$

The cumulative distribution function is:

$$\int_{-\infty}^{x} \mathrm{Pow}(a)\mathrm{d}x = \begin{cases} 0 & x < 0 \\ x^a & 0 \leq x \leq 1 \\ 1 & x > 1 \end{cases}$$