On Dynamic Resource Allocation in Systems with Bursty Sources

Thesis by

Joris Slegers

In Partial Fulfillment of the Requirements

for the Degree of

Doctor of Philosophy

Newcastle University

Newcastle upon Tyne, UK

2009

(Submitted February 12, 2009)

To my parents

&

To Louise

Acknowledgements

First and foremost I'd like to thank my joint supervisors: Dr. Nigel Thomas and Prof. Isi Mitrani. You have given me invaluable advice and guidance even though I was sometimes too stubborn to follow it! This thesis owes much to the three-and-a-half years of patient supervision you have both given me. But perhaps more importantly, I, as an academic and a human, owe much to that supervision too.

Secondly I'd like to thank my examiners: Will Knottenbelt of Imperial College London and Aad van Moorsel for a very enjoyable viva and interesting comments.

Many thanks also go to Chris Smith for our discussions about this and related work. As well as the necessary occasional drink and laugh.

I'd also like to thank everyone at Warwick University involved in the 'Dynamic Operating Policies for Commercial Hosting Environments' project. In alphabetical order: David Bacigalupo, Adam Chester, Stephen Jarvis, Dan Spooner and James Wen Jun Xue.

Also many thanks to my friends and family for the help and, especially, the distractions they provided.

And finally I'd like to acknowledge the valuable contributions the anonymous reviewers of my papers, as well as the participants of the workshops and conferences I attended. They have given important perspective and often interesting criticism on my work, thus improving it tremendously.

Abstract

There is a trend to use computing resources in a way that is more removed from the technical constraints. Users buy compute time on machines that they do not control or necessarily know the specifics of. Conversely this means the providers of such resources have more freedom in allocating them amongst different tasks. They can use this freedom to provide more, or better, service by reallocating resources as demand for them changes. However deciding when to reallocate resources is not trivial.

In order to make good reallocation decisions, this thesis constructs a series of models. Each of the models concerns a resource allocation problem in the presence of bursty sources. The focus of the modelling, however, varies.

In its most basic form it considers several different job types competing over the allocation of a limited number of servers. The goal there is to minimize the (weighted) mean time jobs spend in the system. The weighting can reflect the relative importance of the different job types. Reallocation of servers between job types is in general considered to be neither free nor instantaneous. We then show how to find the optimal static allocation of servers over job types. Finding the optimal dynamic allocation of servers is formulated as solving a Markov decision process. We show that this is practically unfeasible for all but the most simple systems.

Instead a number of heuristics are introduced. Some are fluid-approximation based and some are parameterless, i.e. do not require the a priori knowledge of parameters of the system. The performance of these heuristic policies is then explored in a series of simulations.

A slightly different model is formulated next. Its goal is not to optimize allocation of servers over several job types, but rather between powered up and powered down states. In the powered up state servers can provide service for incoming jobs. In the powered down state servers cannot service incoming jobs but incur a profit due to power savings. Balancing power and performance is again formulated as a Markov decision process. This is not explicitly solved but instead some of the heuristics considered earlier are adapted to give dynamic policies for powering servers up and

down. Their performance is again tested in a number of simulations, including some where the arrival process is not only bursty but also non-Markovian.

The third and final model considers allocation of servers over different job types again. This time the servers experience breakdowns and subsequent repairs. During a repair period the servers cannot process any incoming jobs. To reduce the complexity of this model, it is assumed that switches of servers between job types are instantaneous, albeit not necessarily free. This is modeled as a Markov decision process and we show how to find the optimal static allocation of servers. For the dynamic allocation previously considered heuristics are adapted again. Simulations then show the performance of these heuristics and the optimal static allocation in a number of scenarios.

Contents

Acknowledgements **iii**

Abstract **iv**

1 Introduction **1**

1.1 Background 1

1.2 Structure of Thesis 3

1.3 Publication History 4

2 Literature Review **5**

2.1 Introduction 5

2.2 Technical Background 5

2.3 Directly Related Work 6

2.3.1 Dynamic Server Allocation 6

2.3.2 Power management 7

2.3.3 Breakdowns and Repairs 9

2.4 Summary 9

3 Model and Preliminary Results **10**

3.1 Introduction 10

3.2 Model Description 10

3.2.1 Introduction 10

3.2.2 Markov Decision Processes 11

3.2.3 Formal Model 12

3.3 Optimal Static Solution 17

3.4 Stability 23

3.5 Extensions and Limitations 25

3.5.1 Non-server Systems 25
3.5.2 Multiple Jobs per Server 25
3.5.3 Heterogeneous Servers 26
3.5.4 Batch Arrivals 26
3.5.5 Correlated Arrivals 26
3.5.6 Heavy-tailed Arrivals or Service Times 27
3.5.7 Different Optimization Goal 27
3.6 Summary 28

4 Optimal Dynamic Allocation **29**
4.1 Introduction 29
4.2 Optimization Goal 29
4.3 Policy Improvement 30
4.4 Value Iteration 32
4.5 Comparison of Solutions 34
4.5.1 Example 1: Lightly Loaded System 34
4.5.2 Example 2: Medium Loaded System 37
4.5.3 Heavily Loaded Systems 39
4.6 Possible Speedup Methods 39
4.6.1 Censoring Markov Chains 40
4.6.2 Distributed Implementation 41
4.7 Summary 43

5 Heuristic Policies **44**
5.1 Introduction 44
5.2 Fluid-approximation Based Heuristics 44
5.2.1 Average Flow Heuristic 45
5.2.2 On/Off Heuristic 46
5.3 Parameterless Heuristics 47
5.3.1 Window Heuristic 47
5.3.2 Queue Length Heuristic 48
5.4 Comparison to the Optimal Solution 48
5.4.1 Increased Arrival Rates 48

5.4.2 More Expensive Bursts 52

5.4.3 Less Intensive Bursts 53

5.4.4 Increased Switching Times 54

5.5 Performance 56

5.6 Summary 59

6 Power Management 60

6.1 Introduction 60

6.2 Model 60

6.3 Policies 63

6.3.1 Introduction 63

6.3.2 Idle Heuristic 63

6.3.3 Threshold Heuristic 64

6.3.4 Semi-static Heuristic 64

6.3.5 High/Low Heuristic 65

6.3.6 Average Flow Heuristic 66

6.4 Performance 67

6.4.1 Increased Bursts 67

6.4.2 Increasing Cost Differential 68

6.4.3 Asymmetrical Switching Times 70

6.4.4 The Threshold Policy 72

6.5 Non-Poisson arrival processes 72

6.5.1 Hyper-exponential Busy Times 73

6.5.2 Batch Arrivals 75

6.6 Summary 76

7 Breakdowns and Repairs 77

7.1 Introduction 77

7.2 Model 77

7.3 Optimal Static Solution 80

7.4 Heuristics 84

7.4.1 Average Flow 85

7.4.2 High/Low 85

7.5 Experiments . . . 85
7.5.1 Introduction . . . 85
7.5.2 Increasing Arrival Rate . . . 85
7.5.3 Increasing Breakdown Rate . . . 86
7.5.4 Increasing Repair Time . . . 87
7.5.5 Increased Switching Cost . . . 88
7.6 Summary . . . 90

8 Conclusions and Future Work **91**
8.1 Conclusions . . . 91
8.2 Future Work . . . 92

Bibliography **94**

A Notation **98**

Chapter 1

Introduction

1.1 Background

Computing science has changed a great deal since its inception. The need to optimize performance has, however, remained. In this thesis we will formulate and examine a series of these optimization problems. The inspiration for the problems we look at comes from a series of fairly recent technological developments.

The first of these developments is the ever increasing popularity of what we will call 'grid computing'. Although this term can be used in a more precise, technical, fashion here we mean using large sets of computers to complete a task or tasks, without the user knowing, or caring, which specific machine runs his task, cf. cluster computing. These systems can be used to either compute large tasks by cutting them up in smaller tasks or run many smaller tasks. At some level of abstraction the two are indistinguishable since the smaller sub tasks in the former case can be seen as tasks themselves. Likewise the exact hierarchy of the system does not matter, again at some level of abstraction. For example a highly centralized Condor system and the peer-to-peer based Seti@home system have a notional central controller that receives, distributes and collects tasks.

The second is the emergence of utility computing. In that paradigm users can buy *access* to computing resources, both storage and computation, on demand from some third party. This means they consume these resources without owning the infrastructure or hardware required to run them. A parallel can be drawn with traditional utilities like electricity where consumers consume a resource, power, without owning a power plant. And just like with traditional utilities, consumers are metered for their services and billed accordingly. It is this latter aspect that adds a sense of cost and optimization to more traditional grid systems. Clearly a best effort policy is no longer acceptable here: users pay for resources and expect some level of performance.

The third development that has motivated this work lies in the technological layer below the two concepts referred to above. It is the increasing popularity of virtualization. Here we see the emergence of technology that allows us to abstract away the characteristics of specific computing resources, such as which operating system it is running, the exact processor, the file space it has. Instead we can create a virtual machine that runs almost any given piece of software in a, known, standard environment. This gives us, at least notionally, a standard machine, even when the underlying physical machines are very different. This type of technology is developing rapidly. Currently running a virtual machine has a significantly negative impact on the performance but we can imagine that this will greatly improve in the future. And a small negative impact would probably still be acceptable given the computing flexibility it gives to the user. Such reconfiguring of machines is not in general instantaneous, however.

Apart from these technological developments, another observation inspired this work: the jobs or tasks we are discussing here typically arrive in a (very) bursty fashion. By this we mean that they do not tend to come in as a steady trickle, but rather occasionally in large quantities and occasionally in small quantities. This behaviour is seen on many timescales, from say the behaviour of packets in a TCP stream, to say the dramatically increased usage of certain web sites during major news events.

Drawing on these developments, we envision a future, and indeed it happens in the present, where providers will use large amount of servers to complete different 'jobs' for consumers. Instead of providing one or more servers per type of job, the provider will want to reconfigure servers for each job type on demand. This way the provider can get maximum performance out of his available servers especially since he can use his resources for exactly the task that currently has a large demand. Likewise the consumer will no longer care where exactly his job is run, but will demand a certain performance from the provider.

The question then arises: when does a provider reconfigure his servers? This requires a careful balancing of the loss of service during a reconfiguration period, the loss of performance to the job type the server was formally assigned to and the increased performance to the job type that receives the reassigned server. In this thesis we will consider a number of different, but related, models and develop policies for deciding when to reallocate servers from one task to another.

1.2 Structure of Thesis

This thesis consists of eight chapters, a bibliography and an appendix. To start with the latter: appendix A lists most of the notation used in this thesis for reference. The list is certainly not exhaustive but most of the notation not listed there isvery standard. And in any case, whenever a symbol is used for the first time, its definition is given.

The eight chapters start with this introductory Chapter 1. Next a brief literature survey is presented in Chapter 2. In it the reader is given an overview of some of the related work in this area. The difference with the work in this thesis is also stated.

In Chapter 3 we introduce the model and the optimization goal used in Chapters 4 and 5. It concerns a series of servers that can be dynamically reassigned between several job types. The optimization goal is to minimize the (weighted) average time jobs spend in the system. We also give some preliminary results and also discuss how extensible this model is. Chapter 4 then explores the (im)possibility of finding the optimal solution to the optimization problem. Several solution methods and their problems are discussed, as well as two methods that could speed up the computation of the optimal solution.

Having established the difficulty of finding this solution, we turn to heuristics in Chapter 5. Several heuristic policies are introduced and compared to both the optimal solution, where available, and each other under several conditions.

Chapter 6 introduces a different but related model where our primary concern is balancing power consumption and performance of a system of servers that can be powered down to reduce power consumption but also have to service incoming jobs. We introduce some (heuristic) policies that try to minimize the cost incurred through delayed service to jobs, whilst maximizing the energy savings through powering down servers. The performance of these heuristics is then compared to each other for several scenarios.

The last content chapter, Chapter 7, considers yet another system. Here the servers can be reassigned instantaneously between different job types but are subject to breakdowns and subsequent repairs. Again our optimization goal is to minimize the (weighted) average time jobs spend in the system.

Finally in Chapter 8 we summarize the conclusions of the work presented in this thesis. We also make recommendations for future work.

1.3 Publication History

Much of this thesis has been published in peer-reviewed publications and/or presented at workshops and conferences. In this section we will list this publication history.

An early version of the model and most of the preliminary results described in Chapter 3 were first presented at the 2006 Euro-NGI workshop 'Stochastic Performance Models for Resource Allocation in Communication Systems'. A slightly different version of this work was also submitted to the Workshop on Middleware Performance, WOMP 2006, held in conjunction with the International Symposium on Parallel and Distributed Processing and Applications (ISPA 2006). It was accepted and published in the proceedings in [SMT06].

The Euro-NGI workshop was also the starting point for a special issue of 'Annals of Operations Research' which will appear in 2009. Our article [SMT09] has been accepted for that special issue and contains the extended version of the results that is also present in this thesis, albeit in a slightly different form. The only significant part of Chapter 3 that is not included in that paper is Section 3.5 on the limitations and possible extensions of the model.

Very early versions of the results on the optimal dynamic solution found in Chapter 4 were first presented at the Euro-FGI workshop on "New Trends in Modelling, Quantitative Methods and Measurements" in 2007 and the 2007 UK Performance Engineering Workshop. Subsequently an early version of this work was accepted for the 2007 European Performance Engineering Workshop and published in the proceedings in [SMT07]. We were then invited to submit an extended version for publication in a Performance Evaluation special issue. Most of the results in Chapter 4 can also found in that article which has been submitted, reviewed, revised and is currently being re-reviewed.

The results on heuristics found in Chapter 5 have mostly been published in conjunction with the model and preliminary results found in Chapter 3. Its publication history is therefore very much the same. Specifically it can be noted that the two fluid based approximations described in Section 5.2 are present in both [SMT06] and [SMT09]. The parameterless heuristics found in Section 5.3 can only be found in [SMT09].

The publication history for the work in Chapters 6 and 7 is much shorter. An early version of the work in Chapter 6 was accepted for and presented at the SPEC International Performance Evaluation Workshop 2008. It also appears [STM08] in the proceedings. The results of Chapter 7 have not been published in any form so far.

Chapter 2

Literature Review

2.1 Introduction

The optimization problems considered in this thesis do not appear to have been studied before. Nonetheless there is a body of background and related work available. In this chapter we will briefly outline some of the related research strands and provide references for the background to this work.

To this end we first discuss some of the literature that is of interest to this thesis as background work in Section 2.2. Some of this will be referred to in the individual chapters and summarized there; others contain background work assumed to be known to the reader of this thesis. In the subsequent Section 2.3, we will address some more directly related literature. We end this chapter with Section 2.4 which summarizes these discussions.

2.2 Technical Background

The general area of this thesis is that of dynamic optimization. There is extensive literature in this area. Out of the many available general texts those by Ross [Ros83] and particularly Tijms [Tij94] can be highly recommended. The latter contains much of the background theory used in Chapter 3 and 4. For the latter chapter, the seminal work remains the book by Kleinrock [Kle75]. Several other, more specialized, topics are also considered background knowledge in this thesis.

For a general discussion of parallel and distributed algorithms we would like to point the reader to work by Bertsekas and Tsitsiklis [BT89]. The discussion on convergence in Chapter 4, Section 4.6.2 owes much to that work, even though the specific case in this thesis is not considered there.

The method of spectral expansion used in Chapter 7, Section 7.3 was perhaps most lucidly described by Mitrani and Chakka in [MC95]. We refer the reader to that paper for the, rather

technical, details of the technique.

There is a wealth of publications on fluid-approximations. The approximations used in this thesis are first order but more complicated approximations can e.g. be found in work by Kella and Whitt [KW90] and work by Boxma and Dumas [BD98].

Throughout this thesis we will do mean value analysis since our optimization goals are only formulated in terms of mean sojourn times. This means the queueing discipline, i.e. the order in which incoming jobs are processed, is irrelevant. However there is a large body of work that does consider scheduling discipline. A good overview can be found in the special issue of Performance Evaluation Review edited by Mor Harchol-Balter, [HB07].

2.3 Directly Related Work

2.3.1 Dynamic Server Allocation

The most closely related work to this thesis is probably the work by Palmer and Mitrani [PM05]. There the authors too consider a system of servers that can be reallocated between difference job types. The main difference, from a modeling perspective, to the work outlined in Chapters 3, 4 and 5 lies in the arrival process. The arrival process considered there is a normal Poisson process. This means the arrivals do not experience burstiness. The interrupted Poisson process used in this thesis adds considerable complexity to the model.

The added complexity means that, e.g. the optimal static server allocation is very different. We have also not used the dynamic heuristics from this work. There are two reasons for this. Firstly the difference in arrival process makes the adaptation of the heuristic developed in that paper not readily applicable to this model. Secondly the heuristic contains a 'magical' constant, whose meaning is not clear and cannot be explicitly related to the parameters under consideration.

Despite these differences there is some relation between that work and this thesis. Indeed, the work in [PM05] was the direct inspiration for this thesis as can be readily seen from the modeling choices.

There does not seem to exist any other work that considers the dynamic allocation of multiple servers over multiple job types, either with or without bursty arrivals. There is, however, a substantial body of work concerned with polling systems. See e.g. the paper by Levy and Sidi [LS90] for an early introduction. In a polling system a single server visits several queues in some order. A non-zero switching time between queues can be present. A basic version can be found in a paper by

Hofri and Ross, see [HR87] where a system is considered with two, unbounded, queues and a single server. Arrivals occur according to a Poisson process and job services cannot be interrupted. It is found that the optimal policy is then exhaustive service, i.e. a server is only reassigned once the current queue has been emptied. Once the current queue has been emptied, the server should only be reassigned once the other queue exceeds some threshold. Expressions for these two threshold values are given.

Two more general cases are considered by Duenyas and Van Oyen in [DO95] and [DO96]. In [DO95] a polling system is considered with non-zero switching times, but no switching costs. There are N queues with different holding costs. The service and switching times have a general distribution. A heuristic is developed which performs well compared to threshold and exhaustive policies. The optimal policy is partially characterized but not fully computed.

The paper [DO96] by the same authors considers the case where there are no switching times but there are switching costs. In this case an optimal policy is calculated for a truncated state space. The complexity of this calculation is such that this optimal policy can only be found for very simple systems. Hence the authors also consider a heuristic which is shown to perform well.

Similar work on single server polling systems can be found in two papers by Koole, [Koo97] and [Koo98]. Both papers extend the previous work and discuss the optimal policies at some length. Results are either only partial or very difficult to characterize.

We will end our discussion of polling systems with the reference of a paper by Liu, Nain and Towsley, [LNT92], where the authors characterize the optimal polling system policy in the absence of different priorities for different queues and of switching costs. This corresponds to a model where the optimization goal is to minimize the number of jobs in the system. Even though they do allow non-zero switching times, the absence of different holding costs makes their work inapplicable in systems where these do exist.

There is one more paper that should be mentioned in the context of Chapter 3, specifically Section 3.3. The short paper by Buyukkoc, Varaiya and Walrand, [BVW85] shows that when both switching times and switching costs are zero, the optimal policy for a system with multiple servers is to give priority to the most heavily loaded system (weighted with the holding cost).

2.3.2 Power management

Power management, as addressed in Chapter 6, has become an increasingly popular topic for research. Mostly, however, at a 'lower' level than the server level considered here. The work done

at component, or even chip level, cannot be readily applied to server level. This is because the technological constraints and design considerations that exist there are mostly absent, or entirely different at this level. We will therefore focus on the relatively small amount of literature that does model at the server level.

An overview of early work in this area can be found in a paper by Bianchini and Rajamony, [BR04]. There too it is noted that very little previous work addresses the specific challenges in modeling at the server level. In [PBCH03], Pinheiro et. al. model a cluster of homogeneous servers that can be powered down and up at will, albeit with (asymmetrical) delays. They then used an approach inspired by control theory and load balancing to keep the performance acceptable whilst aggressively powering down servers. No explicit trade off between power and performance is made. Their resulting policy is not characterized but rather presented as a set of controllers and parameters. As a result it is unclear what the influence of various system parameters on their system is. In addition the arrival process they consider is not bursty.

An interesting variation on the work above can be found in a paper by Lefurgy, Wang and Ware, [LWW07] where again a controller approach is used, but the goal this time is to maximize performance whilst maintaining a given level of power consumption. This IBM inspired work does not consider bursty arrivals either and generates a policy that is implicit just like in [PBCH03].

A completely different approach can be found in work by Chase et. al [CAT+01]. There the load on a cluster of servers is taken from both real data and synthetic data. Both are bursty and heavy tailed. The requests are for different services and a virtual economy is created to allocate resources (servers) between these different job types and also an idle state which yields energy saving benefits. These 'services' then compete by bidding for the scarce resources. The resulting system is quite adaptive but again the influence of various system parameters is very hard to infer.

Finally we can mention work by Ranganathan et. al. [RLIC06]. The main aim of this work is to reduce power consumption of servers by assigning it multiple tasks and taking implicit advantage of statistical multiplexing effects, i.e. the unlikeliness of simultaneous peak arrivals for several tasks. It too considers an economic approach. Here a system of server blades is considered in conjunction with a set of tasks. Each of the tasks has a usage profile taken from real traces. The servers themselves have an energy budget assigned to them. By dynamically changing its power consumption (through CPU voltage scaling), each server has to complete its tasks within that budget.

2.3.3 Breakdowns and Repairs

The kind of system considered in Chapter 7, i.e. servers that are subject to breakdowns and repairs, is quite widely considered. Probably the earliest work, considering a single server queue with breakdowns and repairs, using preemptive resume, can be found in a paper by White and Christie [WC58]. This strand of work has been generalized to N servers, see for instance [MC95]. Finding the mean queue length of a system of N servers subject to breakdowns and repairs is given as an example of the spectral expansion technique mentioned earlier.

Despite the relatively common nature of this type of model, to the best of our knowledge the combination of reconfigurable servers, bursty arrivals and breakdowns and repairs has not been considered in the literature before.

The nearest work is a series of publications by Mitrani and various other authors, e.g. with King in [KM81], Chakka in [MC95], Thomas in [TM95], Palmer in [PM06] and finally Martin in [MM08]. The main addition of the work in this thesis over those papers lies in the presence of bursty sources.

2.4 Summary

As we have shown above, there is a wide range of work related to the problems under discussion in this thesis. None, however, address the exact models we discuss here. Closest is [PM05] and other work by those authors that, indeed, formed the inspiration for this work. The match is not exact however, as the systems under consideration in this thesis have substantial additional complexity in the form of bursty arrival processes. This makes direct application of that previous work impossible.

To the best of our knowledge, the application of Markov (decision) processes to a server power management problem, as outlined in Chapter 6 is completely new. Some other work on server power management has been referred to above, but none of these papers formally model the system as extensively as in this thesis.

The modeling of breakdowns and repairs has a much longer tradition. However the combination of dynamic server allocation, breakdowns and repairs and bursty arrivals is, to the best of our knowledge, unique.

Chapter 3

Model and Preliminary Results

3.1 Introduction

In this chapter we will describe the model that we will use for most of this thesis in some way or other. In Chapters 4 and 5 this model will be used directly. In Chapter 6 we will apply a very similar model, which will be discussed there. And finally in Chapter 7 we will develop a related model. Since the model discussed here is so central to this thesis, it will be described in some detail in Section 3.2. This will raise two main questions which we will also discuss in this chapter. The first is deriving the optimal static solution to our model, which will be addressed in Section 3.3. The second concerns the stability conditions for the system we are modeling, which will be discussed in Section 3.4. After discussing these two questions, we will briefly consider possible extensions as well as limitations of the model outlined in this chapter, in section 3.5. We will end with a brief summary of this chapter in Section 3.6. Most of the results here have been published previously in [SMT09].

3.2 Model Description

3.2.1 Introduction

The system we examine is illustrated in Figure 3.1. As an example we can think of the BBC server park. The servers there have different tasks. Some will provide, say, webhosting of the BBC News pages, other provide the BBC iPlayer service and yet others provide the up to date football scores. Demand for these services can be expected to fluctuate, typically in a bursty fashion. We could e.g. expect the amount of webpage requests to increase dramatically when a major news event occurs. Likewise the start of a football match will generate more requests for up to date scores, etc. Many of these request peaks will not be easily foreseen and can thus be considered to arrive at random

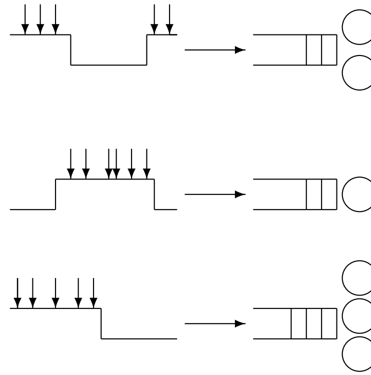

Figure 3.1: Heterogeneous clusters with on/off sources

times. The required resources to deal with each of these requests is, in general, dependent on the type of request. Following our example scenario, it is easy to see that a simple score update request requires much lower service time than an iPlayer request for an entire 1 hour documentary. We also assume that the exact amount of time or service required to complete a request fluctuates. A stream can for instance be paused or, as another example, we can have randomness inherent to the system when hosting a web page due to unpredictable system behavior.

We now consider the servers capable of hosting either of these three services, provided they get a sufficient amount of time to be reassigned and possibly incurring a penalty cost. This will allow us to reassign servers from one type of task, or job type as we will call it throughout this thesis, to another as demand fluctuates. The jobs themselves can also have an associated importance or resilience to delay. A request to update a news web page for example should be handled more promptly than the request for the streaming of a long program. To reflect this, we assign a cost to each request, linearly dependent on the amount of time it spends in the system, and (in general) different for each type of request.

A large part of this thesis will now consider the problem of *when* to reallocate the servers between different job types. The goal here is to minimize the cost incurred by delay to the jobs that pass through the system and switching costs.

3.2.2 Markov Decision Processes

We will model this system using a Markov decision process (MDP). We will give a very brief introduction to this technique here. A more extensive introduction to MDPs can be found in e.g. [Tij94] or [Ros83].

Suppose we have a simple Markov chain, modelling a single queue with a single server. We

will assume there are jobs arriving in the system according to a Poisson process with parameter λ. Likewise job completion is a Poisson process with parameter μ. We can describe the state of the Markov chain by noting the number j of jobs in the system. The transition rates of the Markov chain are

$$r(j, j') = \begin{cases} \lambda & \text{if } j' = j + 1 \\ \mu & \text{if } j' = j - 1 \end{cases}. \tag{3.1}$$

Now suppose we want to consider a slightly different, but related, case where the jobs arriving in the system can be rejected if the queue has grown to a certain size, say K. We do not yet know what K value we choose but rather have an optimization goal in mind. This could be e.g. to have the probability of rejecting a job below a certain value. One way of modelling this is by turning the Markov chain into a Markov decision process. We do this by associating a set of allowable decision $\{a(j)\}$ with each state. Here these decisions are to either allow a job to be accepted, which we will denote by $a(j) = 1$ or to reject any incoming job $a(j) = 0$.

The resulting Markov decision process would be:

$$r_a(j, j') = \begin{cases} a(j)\lambda & \text{if } j' = j + 1 \\ \mu & \text{if } j' = j - 1 \end{cases}. \tag{3.2}$$

The advantage of this model is that we can ask question such as: what decisions should I take so that the probability of rejecting a job is below a certain value? In the next subsection and indeed throughout this thesis we will use these Markov decision process extensively.

3.2.3 Formal Model

A more formal description of the model outlined above is as follows. The system contains N servers, each of which may be allocated to the service of any of M job types. There is a separate unbounded queue for each type. Jobs of type i arrive according to an independent interrupted Poisson process with on-periods distributed exponentially with mean $1/\xi_i$, off-periods distributed exponentially with mean $1/\eta_i$ and arrival rate during on-periods λ_i $(i = 1, 2, ..., M)$. The required service times for type i are distributed exponentially with mean $1/\mu_i$.

Any of queue i's servers may at any time be switched to queue j; the reconfiguration period, during which the server cannot serve jobs, is distributed exponentially with mean $1/\zeta_{i,j}$. If a service is preempted by the switch it is eventually resumed from the point of interruption. This is done to

preserve the Markovian nature of the model since we would otherwise know that a job would take at least a certain amount of time.

We denote the number of jobs of type i present in the system by j_i, the on/off state of job type i by $l_i = 0$ for a job type that is off and $l_i = 1$ for a job type that is on. The number of servers currently assigned to a job type is denoted by k_i and the number of servers currently being switched from type i to type j is denoted m_{ij}.

Using this notation we can describe the state S of the system as:

$$S = (\mathbf{j}, \mathbf{l}, \mathbf{k}, \mathbf{m}) \ , \tag{3.3}$$

where $\mathbf{j}$, $\mathbf{l}$ and $\mathbf{k}$ are vectors of size M, and $\mathbf{m}$ is an $M \times M$ matrix. If no action is taken the instantaneous transition rate $r(S, S')$ from state S to state S' is given by:

$$r(S,S') = \begin{cases} l_i\lambda_i & \text{if } \mathbf{j'} = \mathbf{j} + \mathbf{e_i} \\ \min(k_i, j_i)\mu_i & \text{if } \mathbf{j'} = \mathbf{j} - \mathbf{e_i} \\ m_{ij}\zeta_{ij} & \text{if } \mathbf{k'} = \mathbf{k} + \mathbf{e_i} \\ & \text{and } m'_{i,j} = m_{i,j} - 1 \\ l_i\xi_i & \text{if } l'_i = 0 \\ (1-l_i)\eta_i & \text{if } l'_i = 1 \end{cases} , \tag{3.4}$$

where $\mathbf{e_i}$ is the vector whose i-th element is 1 and all others are 0.

The transition rates in (3.4) correspond to

- Job arrival
- Job completion
- Switch completion
- Arrival stream turning off
- Arrival stream turning on.

The above Markov process becomes a ‘Markov decision process’ by associating with each state, S, a set of actions, $\{a\}$, that may be taken in that state. An allowable action, a, consists of choosing a particular pair of job types, i and j, and switching a number of servers from type i to type j. If

that number is k, then state S changes immediately to state S^a, where

$$k_i^a = k_i - k \ \ ; \ \ m_{ij}^a = m_{ij} + k \ \ ; \ \ k = 0, 1, \ldots, k_i \ . \tag{3.5}$$

The case $k = 0$ corresponds to the action ‘do nothing’.

These immediate state changes are not part of the Markov transition structure. We say that S^a is the ‘resulting’ state of action a in state S. The transition rate of the Markov decision process from state S to state S', given that action a is taken in state S, is denoted $r_a(S, S')$. By definition, it is equal to the transition rate (3.4) from the resulting state S^a to state S':

$$r_a(S, S') = r(S^a, S') \ . \tag{3.6}$$

In order to apply existing Markov decision theory, it is convenient to transform the continuous time process into an equivalent discrete time one. This is done by means of a mechanism called ‘uniformization’ (see e.g. [dSeSG01]), which introduces fictitious transitions from a state to itself, so that the average interval between consecutive transitions no longer depends on the state. A Markov chain is then embedded at these transition instants.

For this we need a uniformization constant, Λ, which is an upper bound for the transition rate out of each state, under all possible actions. Although the tightness of the bound does not matter in principle, the numerical properties of the solution are improved if the bound is tight. The uniformization constant we use is given by

$$\Lambda = \sum_{i=1}^{M} \lambda_i + N \max_i \mu_i + N \max_{i,j} \zeta_{i,j} + \sum_{i=1}^{M} \max(\xi_i, \eta_i) \ . \tag{3.7}$$

The one-step transition probabilities of the embedded Markov chain, in the absence of any actions,

are denoted by $q(S, S')$ and are given by

$$q(S,S') = \begin{cases} l_i\lambda_i/\Lambda & \text{if } \mathbf{j}' = \mathbf{j} + \mathbf{e_i} \\ \min(k_i, j_i)\mu_i/\Lambda & \text{if } \mathbf{j}' = \mathbf{j} - \mathbf{e_i} \\ m_{ij}\zeta_{i,j}/\Lambda & \text{if } \mathbf{k}' = \mathbf{k} + \mathbf{e_i} \\ & \text{and } m'_{ij} = m_{ij} - 1 \\ l_i\xi_i/\Lambda & \text{if } l'_i = 0 \\ (1-l_i)\eta_i/\Lambda & \text{if } l'_i = 1 \\ 1 - \sum\limits_{S' \neq S} q(S,S') & \text{if } S' = S \end{cases} . \quad (3.8)$$

Again, this Markov chain becomes a discrete time Markov decision process by associating actions a with state S. The one-step transition probability of that process from state S to state S', given that action a is taken in state S, is denoted by $q_a(S, S')$. By definition it is equal to the transition probability (3.8) from the resulting state S^a to state S':

$$q_a(S, S') = q(S^a, S') . \quad (3.9)$$

An optimization problem is associated with the Markov decision process. Let c_i be the cost of keeping a type i job in the system per unit time ($i = 1, 2, ..., M$). These 'holding' costs reflect the relative importance, or willingness to wait, of the M job types. In addition, there may be a cost, c^a, associated with carrying out action a (this represents the monetary cost of switching servers from one job type to another). Then the total one-step cost, $c_a(S)$, incurred when the system is in state S and action a is taken, is given by:

$$c_a(S) = c^a + \sum_{i=1}^{M} c_i j_i . \quad (3.10)$$

The special case of $c^a = 0$ represents cost-free, but not necessarily instantaneous, switching.

A mapping, f, from states S to actions a is called a 'policy'. Moreover, f is said to be a 'stationary policy' if the action taken in state S is unique and depends only on S, not on the process history prior to entering that state.

Consider the average long-term cost incurred per step when a stationary policy f is in operation. Denote by Q_f the one-step transition probability matrix of the Markov decision process under policy

f. The elements of Q_f are given by (3.9), with actions specified by f. Then the n-th power of Q_f, Q_f^n, contains the n-step transition probabilities of the process under policy f. By definition, Q_f^0 is the identity matrix.

Suppose that the process starts in state S and proceeds for n steps under policy f. The total average cost incurred over that period, $V_{f,n}(S)$, is equal to

$$V_{f,n}(S) = \sum_{t=0}^{n-1} \sum_{S'} q_f^t(S,S') c_f(S') \ , \tag{3.11}$$

where $q_f^t(S,S')$ is the (S,S') element of Q_f^t, i.e. the t-step transition probability from state S to state S'; $c_f(S')$ is the one-step cost (3.10) incurred in state S' with action specified by f.

The long-term average cost incurred per step under policy f, g_f, is defined as the limit

$$g_f = \lim_{n\to\infty} \frac{1}{n} V_{f,n}(S) \ . \tag{3.12}$$

For an irreducible process (which is our case), the right-hand side of (3.12) does not depend on the starting state S, see e.g. [Tij94].

The optimization problem can now be stated as that of determining the minimum achievable average cost, $g = \min_f\{g_f\}$, together with a stationary policy, f, that achieves it. For this problem to be numerically tractable, the infinite-state Markov decision process must be truncated to a finite-state one. This is done by imposing bounds, $j_{i,\max}$, on all queue sizes. In other words, all one-step transition probabilities $q_a(S,S')$ where S' contains a queue size exceeding its bound, are set to 0. This will be referred to as the 'truncated model'. There are obvious trade-offs in setting the queue size bounds: the larger they are, the more accurate the truncated model, but also the more expensive to solve. We will address this in more detail in Chapter 4.

There is also a second way of looking for an optimal policy. Instead of aiming for the average cost criterion, one could try to minimize the total discounted cost over an infinite horizon. Using a discount factor $0 < \alpha < 1$, the n-step cost (3.11) becomes

$$V_{f,n}(S) = \sum_{t=0}^{n-1} \alpha^t \sum_{S'} q_f^t(S,S') c_f(S') \ , \tag{3.13}$$

and $V_{f,\infty}(S)$ is finite. It then makes sense to look for a policy f that, for each state S, minimizes the total future cost incurred when starting in that state. The advantage of discounted optimization is that the factor α speeds up numerical convergence. The disadvantage is that an optimal policy

under a discounted cost criterion is not necessarily optimal under an average cost one (except in the limit $\alpha \to 1$, where the numerical advantage of α is lost). These methods and their pros and cons will be discussed at some length in Chapter 4. In the next section of this chapter we will focus on a different optimal solution, the optimal *static* solution.

3.3 Optimal Static Solution

In this section we will consider a related problem to that discussed in the previous section. Instead of dynamically reallocating servers between job types, we will consider the problem of pre-assigning the N available servers over the M different job types. So we are interested in finding the vector $\vec{n} = (n_1, n_2, \cdots, n_M)$, where $\sum_{i=1}^{M} n_i = N$, such that allocating n_i servers to the i-th job type will minimize the overall (average) cost of the system. For the case where the arrival process is standard Poisson and the holding costs for each job type are identical, it is well known (this is a property of the Erlang C function) that the so called ρ-rule is optimal. We allocate the available servers roughly proportionate to the loads $\rho_i = \frac{\lambda_i}{\mu_i}$. The 'roughly' here relates to the fact that we have a discrete number of servers and that we have to allocate at least one server to each job type with a non-zero arrival rate. In the case of non-homogenous holding costs, the ρ-rule can easily be adapted to reflect this.

Unfortunately there does not seem to be a similar (simple) result for the system under consideration here. Instead we will find the optimal static allocation in two steps. First we will investigate an isolated queue with n_i servers and an interrupted Poisson arrival process (in the Kendall-notation an $IPP/M/n_i$ queue). We will show how we can calculate the mean queue length for this queue and give a useful approximate expression. Secondly we will show how we can, quite efficiently, search through the possible allocations to find the optimal one, using this approximate expression or the actual mean queue length expression. To simplify notation, the index i will be omitted in this section.

The state of the queue is described by the pair (j, u), where j is the number of jobs present and u is 0 if the arrival process is in an off-period, 1 if it is on. For a graphical depiction see Figure 3.2. Let $p_{j,u}$ be the equilibrium probability of state (j, u). Also denote by μ_j the total service completion rate when there are j jobs present: $\mu_j = \min(j, n)\mu$.

The steady-state probabilities satisfy the following set of balance equations ($j = 0, 1, \dots\ ;\ u = 0, 1$):

$$[\lambda u + \mu_j + \xi u + \eta(1-u)]p_{j,u} = \lambda u p_{j-1,u} + \mu_{j+1} p_{j+1,u}$$

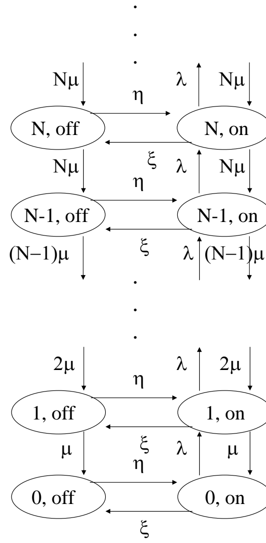

Figure 3.2: Transition diagram for single queue, no switching

$$+ [\xi(1-u) + \eta u] p_{j,1-u} \ , \tag{3.14}$$

where $p_{-1,u} = 0$ by definition.

This model can be solved numerically by treating it as a 'Markov-modulated queue'. The Markovian environment that influences the behaviour of the queue is the phase of its arrival process. Then one can compute performance measures by applying either the spectral expansion (see [MC95, MM91]) or the matrix-geometric solution method (see [Neu81]). However, the present model is sufficiently simple to allow both an explicit exact analysis and an approximate solution in closed-form. This avoids the necessity of computing the matrix R used by the matrix-geometric solution, or the eigenvectors needed by the spectral expansion.

It is convenient to introduce the generating functions of the probabilities corresponding to off- and on-periods, respectively:

$$g_0(z) = \sum_{j=0}^{\infty} p_{j,0} z^j \ \ ; \ \ g_1(z) = \sum_{j=0}^{\infty} p_{j,1} z^j \ . \tag{3.15}$$

Multiplying the balance equations (3.14) by z^j and summing, these can be transformed into a set

of two equations for $g_0(z)$ and $g_1(z)$.

$$[\eta z - n\mu(1-z)]g_0(z) = \xi z g_1(z) - \mu(1-z)P_0(z) \ , \tag{3.16}$$

$$[\lambda z(1-z) - n\mu(1-z) + \xi z]g_1(z)$$

$$= \eta z g_0(z) - \mu(1-z)P_1(z) \ , \tag{3.17}$$

where $P_0(z)$ and $P_1(z)$ are two polynomials involving 'boundary' probabilities (corresponding to states with state-dependent departure rates):

$$P_0(z) = \sum_{j=0}^{n-1}(n-j)p_{j,0}z^j \ \ ; \ \ P_1(z) = \sum_{j=0}^{n-1}(n-j)p_{j,1}z^j \ . \tag{3.18}$$

From equations (3.16) and (3.17), $g_0(z)$ and $g_1(z)$ can be expressed in terms of $P_0(z)$ and $P_1(z)$.

$$g_0(z) = \mu\frac{(1-z)(n\mu - \lambda z)P_0(z) - \xi z[P_0(z) + P_1(z)]}{d(z)} \ , \tag{3.19}$$

$$g_1(z) = \mu\frac{(1-z)n\mu P_1(z) - \eta z[P_0(z) + P_1(z)]}{d(z)} \ , \tag{3.20}$$

where

$$d(z) = \lambda\eta z^2 + n\mu[(1-z)(n\mu - \lambda z) - (\xi + \eta)z] \ . \tag{3.21}$$

Setting $z = 1$ in (3.19) and (3.20), and bearing in mind that $g_0(1) + g_1(1) = 1$ (this is the normalizing equation), we obtain

$$P_0(1) + P_1(1) = n - \frac{\lambda\eta}{\mu(\xi + \eta)} \ . \tag{3.22}$$

This result has a clear intuitive explanation. The left-hand side of (3.22) gives, by the definitions (3.18), the average number of idle servers. The overall arrival rate, averaged over the on and off periods, is equal to $\lambda\eta/(\xi + \eta)$. Therefore, the fraction in the right-hand side of (3.22) is the offered load, i.e. the average amount of work that comes into the system per unit time. Since, in a stable system, the offered load is equal to the average number of busy servers, the right-hand side of (3.22) is also equal to the average number of idle servers.

The necessary and sufficient condition for stability is that the offered load, ρ, is less than the

number of servers:

$$\rho = \frac{\lambda\eta}{\mu(\xi+\eta)} < n \; . \tag{3.23}$$

So far, the generating functions $g_0(z)$ and $g_1(z)$ have been expressed in terms of $2n$ unknown probabilities, the coefficients of $P_0(z)$ and $P_1(z)$. The balance equations (3.14) for $j < n-1$ supply $2n-2$ equations for those unknowns. An additional equation is provided by (3.22). We need one more equation in order to determine the unknowns.

The final equation is obtained by observing that any generating function of a probability distribution must be analytic in the interior of the unit disc. Therefore, if the denominator, $d(z)$, appearing in (3.19) and (3.20), has a zero in that region, then the numerators must also vanish at that point. Now, $d(z)$ is quadratic in z. Moreover, we note that $d(0) > 0$, $d(1) < 0$ and $d(\infty) > 0$ (the second of these inequalities is due to the stability condition). Therefore, $d(z)$ has two real zeros, z_1 and z_2, such that $0 < z_1 < 1$ and $1 < z_2 < \infty$. These zeros are given by

$$z_1 = \frac{b - \sqrt{b^2 - 4ac}}{2a} \;\; ; \;\; z_2 = \frac{b + \sqrt{b^2 - 4ac}}{2a} \; , \tag{3.24}$$

where $a = \lambda(\eta + n\mu)$, $b = n\mu(\lambda + n\mu + \xi + \eta)$ and $c = n^2\mu^2$.

By the analyticity of the generating functions, setting $z = z_1$ in the numerator of, say, (3.20), should make it vanish:

$$(1 - z_1)n\mu P_1(z_1) - \eta z_1[P_0(z_1) + P_1(z_1)] = 0 \; . \tag{3.25}$$

This gives us the additional equation we need. The numerator of (3.19) is also equal to 0 at $z = z_1$, but that is not an independent equation.

All unknown probabilities, and hence the full distribution of the queueing process, are now determined.

The average number of jobs present, L, is given by

$$L = g_0'(1) + g_1'(1) \; . \tag{3.26}$$

We can also derive an approximate expression for L, which is much simpler and easier to evaluate, and is asymptotically exact in heavy traffic.

The quadratic denominator, $d(z)$, given by (3.21) can be written in the form

$$d(z) = \lambda(n\mu + \eta)(z - z_1)(z - z_2) \; , \tag{3.27}$$

where z_1 and z_2 are its two zeros. Since z_1 is also a zero of the numerators in (3.19) and (3.20), those numerators are of the form $(z - z_1)Q_0(z)$ and $(z - z_1)Q_1(z)$, respectively, where $Q_0(z)$ and $Q_0(z)$ are some polynomials involving the boundary probabilities. Adding (3.19) and (3.20), canceling the factor $(z - z_1)$ and dividing the resulting numerator by $(z - z_2)$, we find that the generating function of the number of jobs present, $g_0(z) + g_1(z)$, has the form

$$g_0(z) + g_1(z) = Q(z) + \frac{a}{z_2 - z} \; , \tag{3.28}$$

where $Q(z)$ is some polynomial whose coefficients are linear combinations of the boundary probabilities, and a is a constant. This form implies that the tail of the queue size distribution is geometric with parameter $1/z_2$. Moreover, when the queue is heavily loaded, the boundary probabilities are small and hence the coefficients of $Q(z)$ are small. Indeed, in the limit when the left-hand side of (3.23) approaches the right-hand side, $Q(z)$ vanishes. Neglecting the first term in the right-hand side of (3.28) and treating the queue size as being geometrically distributed, leads to a very simple approximation for the average queue size:

$$L = \frac{1/z_2}{1 - 1/z_2} = \frac{1}{z_2 - 1} \; . \tag{3.29}$$

Using this approximation speeds up the search for the optimal static server allocation considerably.

We now have exact and approximate expressions for evaluating the cost function for any given static policy that allocates servers to queues at time 0 and thereafter leaves them in place. An allocation $(n_1, n_2, \ldots, n_M)$, with $n_1 + n_2 + \ldots + n_M = N$, is feasible if the stability condition (3.23) is satisfied for every queue. For such static policies to exist, the number of queues must not exceed the number of servers, $M \leq N$.

When the number of possible partitions of N into M positive components is not very large (i.e., when either M is small or M is close to N), one can find the optimal static policy by an exhaustive search through all feasible allocations: evaluate the cost C for each allocation and choose the best.

If the exhaustive search is prohibitively expensive (N, M and $N - M$ are all large), then we propose a 'steepest descent' method for minimizing the cost. This can be justified by arguing that

the cost function is convex with respect to its arguments $(n_1, n_2, \ldots, n_M)$. Intuitively, it is clear that queue i benefits as n_i increases, but does so less and less. On the other hand, as n_k decreases, queue k is penalized more and more. Such behaviour is an indication of convexity. One can therefore assume that any local minimum reached is, or is close to, the global minimum.

The algorithm suggested by the above observation works as follows.

1. Start with some allocation, $(n_1, n_2, \ldots, n_M)$; for example, choose n_i roughly proportional to the product of holding cost and offered load, $n_i \propto c_i\rho_i$, if that is feasible.

2. At each iteration, try to reduce the cost by increasing n_i by 1 and decreasing n_k by 1 (where $i, k = 1, 2, \ldots, M\ ;\ i \neq k$). If more than one of these 'swaps' achieves a reduction, choose the best.

3. If no swap reduces the cost, stop the iterations and return the current allocation.

If the approximate expressions (3.29) for the queue sizes are used, then the arguments n_i of the cost function can be treated as *continuous* variables. One can then employ an existing constrained minimization procedure (several are available in Matlab) to find the optimal allocation. The latter is not integer, so its integer neighbours need to be evaluated and the best one chosen. Continuous minimization tends to be very fast.

As an example we can consider a test case where $N = 60$ servers are to be distributed over $M = 6$ job types. The other parameters of the system are chosen randomly to reflect a medium loaded system. Here a brute force search using the approximation formula takes roughly 15 seconds, examining in the order of 10^5 server allocations and ignoring in the order of 10^6 server allocations due to stability conditions. When solving the continuous problem and examining the integer neighbours of that continuous optimum, the calculation takes less than a second, examining just 113 server distributions. If we increase the number of servers to $N = 100$ and the number of job types to $M = 10$, the procedure using the continuous optimum still finds the optimal server allocation in less than a second, examining 153 server allocations. The brute force search does not scale at all in this case and takes over 24 hours to complete. Here the approximate formula was used, but using the exact cost per server distribution instead, adds a linear increase in computational effort per state examined, indicating that solving the continuous problem and then looking at all integer neighbours is still effective.

Since the computation of the exact solution is considerably more expensive than that of the approximation, it may also be advantageous to use a hybrid approach. That is, the optimum obtained

with the continuous minimization of the approximate expressions is used as a starting point for the iterative algorithm with the exact solution. This should reduce the number of iterations required.

As a final remark we should note that there is a certain amount of hand waving involved in this convexity argument since the functions only exist for integer values of the number of servers. Convexity for integer functions tends to be a lot more laborious than for continuous functions. Furthermore the mean queue length function is quite complex and not easily described in closed form. This makes a rigorous proof infeasible within the context of this thesis.

3.4 Stability

In this section we turn our attention to the question of the stability of the Markov decision process. This is, of course, dependent on the policy but here we will try to say something about the stability of the system under the optimal policy, without defining it. Stability of Markov decision processes is a very interesting but also very complex subject. In the context of an implemented system the point is also often moot since the performance of any queueing system will rapidly deteriorate once the load it experiences nears the capacity it can handle. Because of this and the difficulty of the subject, determining the stability of the system is largely out of scope for this thesis, but we will make some initial observations nonetheless.

If we consider the optimal *static* policy, the stability constraint is fairly straightforward. There needs to be a partition $\vec{n}$, with $\sum_{i=1}^{M} n_i = N$, of the servers so that:

$$\forall i \in \{1, 2, \cdots, M\} : \frac{\lambda_i \eta_i}{\xi_i + \eta_i} < n_i \mu_i \ , \tag{3.30}$$

which simply reflects the necessary and sufficient condition that the completion rate should be higher than the overall arrival rate. It is clear that this is also a sufficient condition for the stability of the system under the optimal *dynamic* policy. But we can expect it to be too restrictive and not strictly necessary. On the other hand we have the, somewhat similar, necessary condition:

$$\sum_{i=1}^{M} \frac{\lambda_i \eta_i}{\mu_i (\eta_i + \xi_i)} < N \ . \tag{3.31}$$

Condition (3.31) states that the total capacity of the system should be greater than the total arrival rate of the system. This is clearly a necessary requirement. This would also be the sufficient requirement in the presence of instantaneous switching. We can then model the entire system as

one big queue with jobs of various size being served by multiple servers that do not have to be reconfigured. However since even the best policy will have to contend with, in general, non-zero switching time, we might expect requirement (3.31) to be necessary but not sufficient. We will argue that this is probably not true and that (3.31) is indeed the necessary and *sufficient* condition for stability of the system under the optimal policy.

We can compare the stability conditions for this system to those of a polling system. For that system the stability condition is known, see e.g. [FL96] and other work by these authors. A polling system is a system with *one* server and several queues that are visited (polled) in turn by the server. The exhaustive service in the aforementioned paper denotes that when the server arrives at a non-empty queue it serves this queue until it is empty, including arrivals during that service time. The decision on what queue to poll next is not specified here, but can be state dependent. An example would be *greedy* routing that chooses the largest queue as a next polling queue. The time it takes for the server to move from one queue to another is not assumed to be zero. Finally, the arrival process under consideration is a Poisson process. To be exact: the arrival instances is assumed to be homogeneous Poisson. At each instance, there can be bulk arrivals.

They prove that the system is stable if and only if:

$$\sum_{i=1}^{M} \frac{\lambda_i}{\mu_i} < 1 \; . \tag{3.32}$$

This would be similar to the, certainly necessary, condition (3.31) for our system.

At first glance this polling system seems to closely resemble the system we are interested in with a very specific policy, i.e. polling. Although this would not help us in determining the optimal policy, it will give us a stability bound since the optimal policy would surely be at least as stable as this polling policy. There are, however, some problems. The result holds for *one* server. Furthermore it is claimed that the results hold for more general (independent identically distributed) arrival processes, but no proof is given.

Intuitively we would expect either problem to be manageable. There seems to be little difference (stability wise) in several servers polling as a group versus one of them. Although there remains the finesse that a straight-forward adaptation like that means a group of servers may poll to a queue where there is not enough work in the queue and thus capacity is wasted. But this seems a minor problem. And if the result does hold for more general arrival processes, the relatively well-behaved interrupted Poisson process under consideration here should qualify. Unfortunately the proof for the original polling system (as found in [FL96]) is complicated and very technical. This makes adapting

it to the current case out of scope for this system.

3.5 Extensions and Limitations

In this section we take a look at some possible extensions of the model and also at aspects it cannot model without extensive changes. The aim is to give the reader insight into how general this model is, but also for which situations it is not suitable.

3.5.1 Non-server Systems

Perhaps the first thing to note is that although we will consistently talk about servers, there is nothing in the model preventing us from applying it to, say, cores in a multicore system. The main limitation there is that the optimization goal formulated in this thesis is exclusively interested in minimizing the mean time jobs spend in the system. For a system of processors that is probably not a suitable optimization goal, since there we want e.g. the processors (servers) to be powered down periodically to decrease the heat generated. Furthermore the locality of the cores (servers) is not considered in our model but is probably of interest. Adding this to the optimization goal is somewhat difficult.

In general the optimization goal will be the biggest constraint for our application area. As long as it is possible to focus on the average time 'jobs', whatever they may be, spend in the system the model should be fairly adaptable.

3.5.2 Multiple Jobs per Server

It is sometimes more efficient for servers to service multiple jobs in parallel. This can be due to programming considerations, the presence of multiple processors (cores) or to exploit multiplexing gains. It is straightforward to adapt the model outlined here for that case. We just consider the notion of virtual servers, whereby for each physical server capable of running, say, 4 jobs in parallel our model contains 4 virtual servers. The only difference lies in the decisions that we can make. Instead of being able to switch 1 or more servers between job types, we can now only switch a given numbers of servers, 4 in this example. It does not impact our model significantly and the heuristics we will consider in Chapter 5 can also easily be adapted.

3.5.3 Heterogeneous Servers

The assumption that the servers are identical can also be relaxed, albeit at significant cost to the size of the state space. We can simply group likewise servers into separate job types. Say we have two types of servers and three job types, we consider a system where there are 6 virtual job types. We assign the servers to a suitable starting state, i.e. servers of type 1 are assigned to one of the three virtual job types associated with that server type and likewise for type 2. Switching between un-associated job types is then made prohibitively slow and/or expensive. The service given to incoming jobs is now more complicated as two server pools share the same job queue. But this too can be easily adapted and again the heuristics in the later Chapter 5 can be adapted to handle this situation.

If the servers are very heterogenous, i.e. they are not easily grouped in a small number of types, this solution is probably unfeasible. The amount of job types and switching decisions we must consider grows too rapidly then. This is not surprising, just a reflection of the much-increased complexity of the system under consideration.

3.5.4 Batch Arrivals

A much easier extension is that of batch arrivals. We have defined our model in such a away that at every arrival instance only one job enters the system. Nothing prevents us to change this to batch arrivals, if these are of fixed size. This would have no impact on the complexity of the system. If the size of the batch is again a stochastic variable, the complexity of the system does grow greatly and this can really be considered an extension that is out of scope for the work considered here.

3.5.5 Correlated Arrivals

Some systems experience (highly) correlated arrivals. This can for example be due to the arriving jobs being part of larger streams of requests. The model we use here is not easily adapted to deal with this situation. Even though there is a correlation between arrivals in the sense that they experience ‘on’ and ‘off’ periods, arrivals within such periods are uncorrelated. This is an essential aspect of our system since it preserves the Markovian nature of our arrival process, and cannot really be changed without getting a completely different model. Using this Markovian model to get estimates for correlated arrivals should be done with caution since it has been noted many times that such estimates can be significantly off.

3.5.6 Heavy-tailed Arrivals or Service Times

Similarly to the correlation discussed above, heavy-tailedness is also sometimes considered an important characteristic of arrival processes. This is also sometimes considered a characteristic of service times, corresponding to the occasional presence of a 'problematic' or exceptionally large task. This cannot be easily included in the model presented in this thesis, neither for the arrival nor for the completion process. As before, the Markovian nature would most likely be broken and as a consequence the model changes significantly. A possible generalization that does not break this Markovian assumption is to introduce service times that are distributed according to a phase-type distribution. This allows for much more complicated service times with high second moments. Although not properly heavy-tailed, since the second moment will still be finite, this might be a sufficient description for some arrival processes. The major downside of this is the large increase in state space this will cause.

3.5.7 Different Optimization Goal

As a final possible extension of the model we will look at a the possibility of using a different optimization goal. In formulating this model we have assumed that performance is measured in terms of the average time jobs spend in the system. For some contracts or SLAs this is not appropriate. There the performance is e.g. measured as a percentile of a submitted stream. Say: 95% of the jobs has to be completed and returned within 1 minute. It is also possible to have a per job pricing structure such that the provider gets his full reward if a job is completed within a certain time restriction.

In both cases the model described here cannot easily be adapted. Optimizing for mean value is after all not necessarily the same as optimizing for percentile and also not the same as optimizing to keep the time each job spends in the system below a certain value. So although interesting problems in their own right, results for more complicated optimization goals are not easily derived from the current model. It then, e.g. becomes important what queueing discipline is used since mean value analysis is not sufficient. Perhaps the required modeling techniques would therefore come from queueing theory, rather than straight-forward Markovian theory.

3.6 Summary

In this chapter we discussed a model we will be using for substantial parts of this thesis. It was formally defined in Section 3.2 and we also introduced the related optimization problem. Having defined model and problem, we showed how we can calculate the optimal static solution in Section 3.3. We also gave an explicit formula that is an approximate solution. Although an approximation, it is asymptotically exact under heavy traffic assumptions and furthermore it allows us to do quick calculations. In Section 3.4 we briefly discussed the stability of the Markov decision process under consideration. Here we hypothesized a necessary and sufficient stability condition. Although no formal proof was given, since the matter is rather complex, we gave some convincing arguments why this is probably true. In the final Section 3.5 we considered the generality of the model by looking at some possible extensions. This concludes this chapter, introducing the model. In the next chapter we will explore the possibility and difficulty of finding the optimal dynamic policy.

Chapter 4

Optimal Dynamic Allocation

4.1 Introduction

In this chapter we will discuss how to calculate the optimal solution to the optimization problem outlined in the previous chapter in Section 3.2. This will be described in more detail in Section 4.2. To solve this problem we will introduce two techniques for finding the optimal policy of a Markov decision process, policy improvement, in Section 4.3 and value iteration, in Section 4.4. Both these algorithms have a full cost and a discounted cost 'flavour' which we will both describe. In Section 4.5 we will discuss these two algorithms in both discounted and full cost form and compare their results. Next, in Section 4.6, we will address possible speedups to the methods used and finally end with a summary of this chapter in Section 4.7. Most of the work in this chapter has been previously published in [SMT07].

4.2 Optimization Goal

In Section 3.2 we formulated the long term average cost incurred per step under a policy f as the limit (3.12):

$$g_f = \lim_{n\to\infty} \frac{1}{n} V_{f,n}(S) \ . \tag{4.1}$$

And we also introduced its discounted counter-part (3.13):

$$V_{f,n}(S) = \sum_{t=0}^{n-1} \alpha^t \sum_{S'} q_f^t(S,S') c_f(S') \ , \tag{4.2}$$

and noted that $V_{f,\infty}(S)$ is finite for a discount factor $0 < \alpha < 1$.

So it then makes sense to look for a policy f that, for each state S, minimizes the total future

cost incurred when starting in that state. Here we have to choose between discounted and full cost optimization. The advantage of discounted optimization is that the factor α speeds up numerical convergence. The disadvantage is that an optimal policy under a discounted cost criterion is not necessarily optimal under an average cost one (except in the limit $\alpha \to 1$, where the numerical advantage of α is lost). We will show results for both in this chapter.

We will use a known result in Markov decision theory (see [Tij94]), which states that if there exist a set of numbers, $\{v_S\}$ (one for each state), and a number g, such that for every S,

$$v_S = \min_{a \in A(S)} \{c_a(S) - g + \sum_{S'} q_a(S, S') v_{S'}\} , \tag{4.3}$$

where $A(S)$ is the set of all possible actions in state S, then

1. The actions achieving the minima in the right-hand side of (4.3) constitute an optimal stationary policy.

2. The long-term average cost achieved by that policy is g.

The numbers v_S are not actual incurred costs in various states but may be thought of as 'relative costs'. Note that if a set of relative costs provides a solution to (4.3), then adding any fixed constant to all of them would also produce a solution. Hence, one of the relative costs can be fixed arbitrarily, e.g. $v_S = 0$ for some particular S.

4.3 Policy Improvement

In this section we will look at policy improvement to find the optimal policy. The policy improvement algorithm is due to Howard [How60] and has four steps.

1. Choose an initial policy, f, i.e. allocate to every state S, an action a to be taken in it. For example, one could choose the policy that 'does nothing' in all states. Also, select the state whose relative cost will be 0.

2. For the policy f, calculate the relative costs, v_S, and the average cost, g. This requires the solution of the set of simultaneous linear equations:

$$v_S = c_f(S) - g + \sum_{S'} q_f(S, S') v_{S'} . \tag{4.4}$$

   There are as many equations here as unknowns, since we also set one of the v_S to 0.

3. Find, for every state, the action a that achieves the minimum in

$$\min_{a \in A(S)} \{c_a(S) - g + \sum_{S'} q_a(S, S')v_{S'}\} , \tag{4.5}$$

   using the relative costs and g computed in step 2. This set of actions defines a policy, f', which is at least as good as f and possibly better.

4. If f' and f are identical, terminate the algorithm and return f and g as the optimal policy and the minimal average cost. Otherwise set $f = f'$ and go to step 2.

The computational complexity of this algorithm tends to be heavily dominated by step 2. It is convenient to rewrite this step in matrix and vector form:

$$\mathbf{V} = \mathbf{c}_f + A_f\mathbf{V} , \tag{4.6}$$

where $\mathbf{V} = (\mathbf{v}, g)$ is the vector of relative costs v_S and the average cost g; $A_f = [Q_f, -\mathbf{1}]$ is the one-step transition probability matrix under policy f, extended with a column of (-1)s; the last equation (4.6) is the condition v_S=0, for the chosen S.

This equation can be rewritten in the standard form

$$(I - A_f)\mathbf{V} = \mathbf{c}_f . \tag{4.7}$$

There are many numerical methods for solving this type of equation. We have used the direct solution method (a version of partial pivoting) provided by Matlab (and inherited from LAPACK). The choice is somewhat pragmatic. The other method tried, Gauss-Seidel iterations, suffered from more problems. We have not tried further methods, e.g. conjugate gradient methods, although they could perhaps perform slightly better. This can sometimes suffer from numerical instabilities; it turns out to be better to solve yet another form of equations (4.6), namely:

$$(I - A_f)^*(I - A_f)\mathbf{V} = (I - A_f)^*\mathbf{c_f} , \tag{4.8}$$

where B^* denotes the transpose of matrix B. This equivalent equation is more convenient because we now from linear algebra that for any non-singular matrix B the matrix B^*B is positive definite. This greatly helps the numerical stability of most procedures, including the ones used by Matlab. The form (4.8) was therefore adopted.

This full cost policy improvement algorithm can easily be adapted to solve the discounted cost problem (4.2). We simply replace the equations (4.4) and (4.5) by their discounted forms:

$$v_S = c_f(S) - g + \sum_{S'} \alpha q_f(S, S') v_{S'} \ , \tag{4.9}$$

and

$$\min_{a \in A(S)} \{c_a(S) - g + \sum_{S'} q_a(S, S') v_{S'}\} \ . \tag{4.10}$$

As mentioned before, this discounted problem has better convergence properties. The reason for this can be found in linear algebra. It is know (see e.g. [GL96]) that iterative solutions of the linear equations of the form $Ax = b$ converge at a geometric rate if the spectral radius $\rho(A) < 1$. But since A is a stochastic matrix we know that $\rho(A) = 1$. So if we use a discount factor $0 < \alpha < 1$, it is guaranteed that $\rho(\alpha A) < 1$ and we get geometric convergence. For $\rho(A) = 1$ the convergence is much more complicated in general (see e.g. [Szy98]) but an exception can be made for matrices that are positive definite, where the convergence is geometric once again. This is the reason why we used form (4.8) over (4.7).

The main advantage of policy improvement is that it has a definite stopping criterion. When in step 4 the two policies are identical, they are also guaranteed to be optimal. Furthermore its convergence has some nice properties as well. The only formal statement we can make is that it converges in a finite number of steps. For the proof of this, and similar statements about convergence in the next section, see [Tij94]. Empirically it has been found that the convergence is typically very fast and somewhat independent of the size of the state space.

Conversely the main disadvantage of the policy improvement algorithm lies in the computational intensity of step 2. Here we solve a large set of simultaneous equations. This severely limits the size of the state space we can handle.

4.4 Value Iteration

The value iteration algorithm is due to White [Whi63]. It too has four steps.

1. Initialize the cost V_0 at step 0 of each state S to some value. Here we used the obvious choice of the holding cost as the selected starting cost:

$$V_0(S) = \sum_{i=1}^{M} c_i j_i \ . \tag{4.11}$$

Also initialize some termination accuracy ϵ.

2. Choose a state S^*. Calculate the cost in that state as:

$$g_n = \min_{a \in A}[c_a(S^*) + \sum_{S'} q_a(S^*, S')V_{n-1}(S')] \ . \tag{4.12}$$

We use this as our normalizing cost.

3. Given the $n-1$ step cost V_{n-1} for each state, calculate the n step cost $V_n(S)$ and n step optimal decision $a(S)$ in each state. We do this by finding the decision a that minimizes:

$$V_n(S) = \min_{a \in A}[c_a(S) - g_n + \sum_{S'} q_a(S, S')V_{n-1}(S')] \ . \tag{4.13}$$

and the cost $V_n(S)$ that results from this decision.

4. Calculate the maximum M_n and minimum m_n change in cost as:

$$M_n = \max S[V_n(S) - V_{n-1}(S)] \text{ and } \quad m_n = \min S[V_n(S) - V_{n-1}(S)] \ . \tag{4.14}$$

If the termination criterion:

$$M_n - m_n \leq \epsilon m_n \ , \tag{4.15}$$

is satisfied, we terminate with the decisions $a(S)$ as output. Otherwise we go to step 2.

Again this algorithm can be converted to solve the discounted cost problem (4.2). This can be done by just introducing the discount factor α in the relevant equations. But a more efficient method is to remove step 2 from the algorithm. This step was only introduced to counter the problems of the costs V_n tending to infinity in the full cost case. Having removed step 2, we replace equation (4.13) by:

$$V_n(S) = \min_{a \in A}[c_a(S) + \alpha \sum_{S'} q_a(S, S')V_{n-1}(S')] \ . \tag{4.16}$$

The rest of the algorithm is left unchanged.

The main advantage of the value iteration algorithm is that it does not require the solving of any large simultaneous set of equations. It just has a large set of simple arithmetic operations that have to be executed. This not only relaxes the memory constraints greatly, it also allows possible parallel implementation. We will discuss this further in Section 4.6.

The convergence properties of this method are less appealing. The upper bound M_n and lower bound m_n converge monotonically, but recall that this is the difference in increase between two steps of the algorithm and does not relate directly to the value (or cost) for each state. More encouraging is that, under a technical assumption satisfied here, it holds that there are finite constants $\alpha > 0$ and $0 < \beta < 1$ such that:

$$|M_n - m_n| \leq \alpha\beta^n \ , \quad n \geq 1 \ . \tag{4.17}$$

This means in particular that in the limit M_n and m_n converge. However it does not help us in picking the factor ϵ where we deem the values sufficiently close. Furthermore we can expect the number of iterations required for convergence to depend heavily on the size of the state space, because we in essence 'see' n steps deep into the chain after n steps of the algorithm. For large state spaces this is a rather unappealing property since it implies the algorithm will need more steps to converge, despite the scalability of each individual step.

4.5 Comparison of Solutions

In this section we will compare the two algorithms mentioned above, policy improvement and value iteration. We vary both the truncation level and the discount factor, including setting it to 1, i.e. using the full cost. We then compare the different methods in terms of performance achieved (in terms of achieved cost) and time required to compute.

4.5.1 Example 1: Lightly Loaded System

The first case considers a system with just two job types and two servers, i.e. $N = 2$ and $M = 2$. The system is symmetrical in both job types, lightly loaded and each job type is 'on' half of the time. In terms of the parameters: $\lambda_1 = \lambda_2 = 0.047$, $\mu_1 = \mu_2 = 0.113$ and $\eta_1 = \eta_2 = \xi_1 = \xi_2 = 0.01$. The two job types are *not* symmetrical in cost assigned to them. The holding cost for job type 2 is twice that of job type 1, $c_1 = 1$, $c_2 = 2$. And finally switching is free but takes an average of one completion time to finish, i.e. $C_{sw} = 0$ and $\zeta_{1,2} = \zeta_{2,1} = 0.113$. We examine the effect of the chosen discount factor on this system for the policy improvement algorithm and fix the truncation levels of our system at queue length 20. The effect of the truncation level will be discussed in more detail later on and in other examples. Results generated by the value iteration algorithm will also be discussed in a later example.

Table 4.1: Optimal actions for example 1 with various discount factors: ($\alpha = 0.9$, $\alpha = 0.99$, $\alpha = 0.999$). 1 denotes switching a server from job type 1 to job type 2 and -1 denotes the converse switch. The columns represent different values for queue length 2 and the rows for queue length 1

j1	j2 = 0	1	2	3	4	5	6	7	8	9	10
0			1,0,0	1,1,1	1,1,1	1,1,1	1,1,1	1,1,1	1,1,1	1,1,1	1,1,1
1								1,1,1	1,1,1	1,1,1	1,1,1
2								1,0,0	1,1,1	1,1,1	1,1,1
3								1,0,0	1,1,0	1,1,1	1,1,1
4								1,0,0	1,1,0	1,1,1	1,1,1
5	-1,-1,-1							1,0,0	1,0,0	1,1,1	1,1,1
6	-1,-1,-1							1,0,0	1,0,0	1,1,0	1,1,1
7	-1,-1,-1							1,0,0	1,0,0	1,1,0	1,1,1
8	-1,-1,-1								1,0,0	1,0,0	1,1,1
9	-1,-1,-1								1,0,0	1,0,0	1,1,0
10	-1,-1,-1								1,0,0	1,0,0	1,1,0

Table 4.1 shows some of the actions the optimal policy makes, given various discount factors. An optimal policy is of course defined for *every* state and the table here only shows the actions made when one server is assigned to each of the two job types and both job types are in an ‘on’ period.

In the table the actions for queue lengths up to 10 are displayed. The first number denotes the action made by the optimal policy calculated with a discount factor of 0.9, the second for a discount factor of 0.99 and the third one for a discount factor of 0.999. Here 1 denotes the decision to switch a server from job type 1 to job type 2, -1 denotes the decision to switch a server from job type 2 to job type 1 and 0 denotes the decision not to switch. Where the table is left blank, all three policies made the decision not to make any switch.

Recall that the job types are symmetrical but that job type 2 is twice as expensive. This explains the much higher willingness of all the optimal solution to switch a server to job type 2, rather than the other way round. The optimal solution is also less willing to switch when the discount factor is closer to 1, i.e. when future costs are discounted less. The explanation seems to be that although there is a short term benefit in reducing the current queue length by switching a server to a (relatively) heavily loaded system, there is also a longer-term disadvantage since the system is taken out of a stable state. This means that the other queue length will grow and the server will have to switched back at some point. Systems with a discount factor closer to 1 should penalize this behavior more heavily. Not shown in the table are the actions of the optimal policy when there is no discount, i.e. $\alpha = 1$. For the states in the table, these actions are identical to those generated by the $\alpha = 0.999$ discounted policy. In fact, the action is different in just 48 of the 17640 states considered here.

The obvious next question is how these different decisions affect the system in terms of the

performance of the policy, expressed as a cost. It should be noted that the cost is derived from direct computation, not simulation. The idea is that, given the (optimal) decision $f(S)$ in each state S, we calculate the steady state distribution denoted $\vec{\pi_f}$. The average cost of the system is then calculated as the product $g_f = \vec{\pi_f'} \cdot V(\vec{S})$. There is a slight complication. Due to the policy some states can be unreachable. Those states are removed from the system in order to solve the steady state equations. All this is not necessary in the case of the optimal solution found by the non-discounted policy improvement algorithm, as the optimal cost g^* is actually outputted there.

The cost achieved by the optimal policy generated with a discount factor of $\alpha = 0.9$, $\alpha = 0.99$, $\alpha = 0.999$ and the full cost version, is 1.5095, 1.4250, 1.4249 and again 1.4249 respectively. In this example there is a clear benefit of setting the discount factor to at least 0.99, a very modest benefit to setting it to 0.999 and no noticeable benefit to calculating the full cost policy. The downside of setting a higher discount policy is the increased computation time it requires. There is no straightforward formula for calculating the exact increase since it is not caused by an increase in state space, but by the ease with which the matrix equation (4.8) can be solved. In general this is harder (i.e. requires more iterations) if α is closer to 1, a property relating to the size of the spectral radius. We know that a smaller spectral radius means quicker calculation although the exact nature of this relationship is somewhat obscure. Indeed in this example it took about half an hour to calculate the policy with discount factor 0.9, 1 hour for $\alpha = 0.99$, 5 hours for $\alpha = 0.999$ and 7 hours for the full cost version.

It is difficult to draw any definitive conclusions from this example since the results are completely dependent on the parameters of the system, e.g. although here a discount factor of 0.99 seems to achieve a reasonable balance between performance and computational effort, it could very well be that for different system parameters the 'best' discount factor is different. Indeed in the next few examples different values will be presented. But the trade-off is general: higher discount factors require significantly more computation time but offer better performance.

A second factor of interest is the effect of the truncation level on the policy. Table 4.2 shows the actions of two policies, both generated by the full cost policy improvement algorithm, one for a system truncated at maximum queue lengths of 10 and a second truncated at queue lengths 20. As mentioned before, the second actions are identical to those generated with $\alpha = 0.999$. The most striking differences in decisions between the two policies can be found where both queue lengths are big. Here the policy computed with the maximum queue length set at 10 makes seemingly odd decisions. When, e.g. $j_1 = 10$ and $j_2 = 7$ it still decides to switch the server assigned to queue 1 to

Table 4.2: Example 1 continued: Optimal actions for different truncation levels (10,20)

j1 j2	0	1	2	3	4	5	6	7	8	9	10
0				1,1	1,1	1,1	1,1	1,1	1,1	1,1	1,1
1								1,0	1,1	1,1	1,1
2									1,1	1,1	1,1
3										1,1	1,1
4										1,1	1,1
5	-1,-1									1,0	1,1
6	-1,-1									1,0	1,1
7	-1,-1									1,0	1,1
8	-1,-1								1,0	1,0	1,0
9	-1,-1								1,0	1,0	1,0
10	-1,-1							1,0	1,0	1,0	1,0

Table 4.3: Example 2: cost of policy computed with various queue lengths and discount factors

	QL=10	QL=20	QL=30	QL=40
PI, 0.9	4.1769	6.8937	11.003	16.77
PI, 0.99	4.0411	6.7794	10.745	16.201
PI, 0.999	3.961	5.8398	8.4187	13.644
PI, full	3.961	5.7952	8.2851	12.794
VI, 0.9	4.1769	6.8937	11.003	
VI, 0.99	4.0411	6.7794	10.745	
VI, full	3.961	5.8231		

queue 2. The explanation is that the system cannot get any worse with respect to job type 1 at that point, so it makes sense to try and decrease the more expensive queue 2 and not make any efforts towards reducing queue 1. While this is truly the optimal solution for the truncated system, it is obviously nonsensical if the system under consideration allows larger or even infinite queues. This makes the generalization of results for truncated systems to larger ones problematic.

4.5.2 Example 2: Medium Loaded System

The second example considers a system with very similar parameters to the previous one. The only change is in the load during an on period. Here these are $\lambda_1 = \lambda_2 = 0.1$, leading to a system that could be considered to experience medium load. The focus will be on the effect of truncation and discount factor on the performance of the policy, rather than on its form. There are also some results from the value iteration algorithm.

Table 4.3 shows the cost of the policy computed under various conditions. Horizontally we vary the allowed maximum queue length and vertically we consider several ways of computing the policy. Here 'QL' stands for the maximum allowed queue size of both job types, PI stands for the policy improvement algorithm, VI for the value iteration algorithm, the numbers 0.9, 0.99, 0.999 for the

discount factors used and the addition 'full' indicates that the full cost policy was computed. The cost increase with the maximum allowed queue size because the cost is computed for the system with the specified maximum queue length. So if $QL = 20$ any job request arriving to a queue that already contains 20 jobs (including those being served), is rejected. There is no cost attached to this rejection. This makes the different policies somewhat more difficult to compare, but as noted previously, there is no clear cut way of generalizing a policy to allow for a system with higher queue lengths.

Although some patterns are similar to ones noted earlier, one remarkable difference can be found in the effect of the discount factor. Where in the previous example a discount factor of 0.99 seemed to generate reasonable results, here a big reduction of cost seems to be possible when using the higher 0.999 discount factor or even using a full cost policy. This effect is much more pronounced when the allowed maximum queue lengths are higher. This re-emphasizes the caution one must use when computing a discounted policy.

The results of the value iteration are very similar to those of the equivalent policy improvement algorithm. This is probably caused by the stopping criterion used. Recall that criterion mentioned in Section 4.4 means the algorithm terminates when the biggest and the smallest change in values between two iterations are relatively (with some proportional factor ϵ) close. Here that factor was chosen as $\epsilon = 0.001$, a fairly strict but somewhat arbitrary criterion. To guarantee stopping in a reasonable amount of time, there was a second stopping criterion. If the number of iterations exceeded 10000 the algorithm terminated as well. For a maximum queue length of 10, both the 0.99 discounted and the full cost calculations terminated after less than 10000 iterations, meaning that the relative convergence criterion was achieved. This yields equivalent policies (and hence cost) to the guaranteed optimal solution generated by the policy improvement algorithm. For a maximum allowed queue length of 20, the 0.99 discounted computation also terminated on the ϵ criterion, after 1115 iterations. However the full cost version terminated due to reaching the 10000th iteration. As can be seen, this results in a suboptimal policy.

In a way this is rather surprising. It implies that the changes made in the policy by the value iteration algorithm are significant, even after many iterations. For the policy improvement algorithm this is widely considered (see e.g. [Tij94]) not to be the case. There the changes made in the initial iterations have by far the most impact on the performance of the policy.

Table 4.4 shows some of the compute times and number of iterations required. The compute times are only an indication since they are dependent on the machine running the algorithm. In

Table 4.4: Example 2 continued: compute time and number of iterations required for various policies

	QL=10	iterations	QL=20	iter	QL=30	iter	QL=40	iter
PI, 0.9	2 mins	5	27 mins	5	3 hours	5	7 hours	5
PI, 0.99	3 mins	7	45 mins	8	3.5 hours	8	14 hours	10
PI, 0.999	3 mins	8	45 mins	8	3.5 hours	8	14.5 hours	10
PI, full	3 mins	8	51 mins	9	4.5 hours	10	15 hours	10
VI, 0.9	32 mins	311	5.5 hours	328	23 hours	344		
VI, 0.99	2 hours	1115	1 day	1596	4.5 days	1789		
VI, full	14 hours	10000	5 days	10000				

this case a 2.8 GHz. desktop with 2 GB of RAM was used. The size of the state space is 4841 for a truncation level of 10 (implying 4841^2 possible transitions, although most of them are zero) and 17641, 38441 for truncation levels 20 and 30 respectively. In general, for N servers, M job types and QL maximum queue length the size of the state space is:

$$|S| = (QL+1)^M \cdot 2^M \cdot \binom{N+M^2-1}{M^2},$$

with the terms representing the size of the queue length space, the amount of possible on/off states and the amount of possible server allocations respectively.

It should be noted that the time to compute the policies using the value iteration algorithm, significantly exceeded that of the policy improvement algorithms. For the smaller or more heavily discounted systems, this could perhaps be helped by setting a less stringent convergence criterion. But the case of the full cost value iteration algorithm with a maximum queue length of 20 provides an indication that this is not likely to consistently generate optimal results.

4.5.3 Heavily Loaded Systems

We will not show experiments on heavily loaded systems. In these the mean queuelength is large. And furthermore, the probability of being in a system with large queuelength is non-negligible. This means the states at the edges are important. But we have already noted that the policy derived from a truncated model is unreliable for these states. Consequently a policy derived from truncating a heavily loaded system would be very poor for an untruncated system.

4.6 Possible Speedup Methods

In this section we will discuss two, unrelated, possible speed ups for the calculation of the optimal solution. We will show why one method, using censored Markov chains, does not seem to work and

how another, distributed computation, could be implemented. Both subsections will be rather brief and describe ideas and concepts and not implementation.

4.6.1 Censoring Markov Chains

In this subsection we will outline an idea and why it does not work. The idea was to not use truncation of the Markov chain, but instead use a censoring approach to get a model for the infinite Markov chain. This model can then be used as the input for either a policy improvement or value iteration algorithm to get the solution of the related Markov *decision* process.

The idea of censoring Markov chains dates back to [Sen67]. The idea is this: suppose we have a large, possibly infinite, Markov chain. Getting results for it might be prohibitively expensive or even impossible. So instead we *censor* the Markov chain, i.e. we only look at a smaller, finite, part of it. We then do analysis on this censored Markov chain. This is used as an indication of, and sometimes a bound on, results for the original, uncensored Markov chain.

There is a fundamental difference between this approach and the truncation of the state space we applied earlier. There we simply prohibited transitions out of the truncated state space. This in effect redistributes the probability weight over the state space evenly. In a censored Markov chain the states outside the bound are simply disregarded and their probability weight is lost. Although, as we will see, in order to get bounds and do analysis, we often reassign it to a boundary state.

We are interested in using this approach because it might negate some of the difficulties we had with decisions near the truncation level. Since here there is a notion that things *can* get worse when the queue lengths grow. Hopefully this will allow us to model a system effectively using smaller maximum queue lengths and thus a smaller state space.

We follow here the algorithms outlined by Truffet in [Tru97]. It is defined for a one-dimensional Markov chain, or at least a Markov chain that can be ordered in such a way. We will get back to this point later. The first step is to simply add the missing probability weight to the final state. This makes the final state a lumped 'bad' state. In our terminology: the final state is the worst queue length state we can achieve. Next we want to apply 'Vincent's algorithm' [AAV98]. Details can be found there or, since that paper seems quite hard to find, in [FPY07]. Roughly speaking Vincent's algorithm ensures the rows of the transition matrix are stochastically monotone. This will give us a lower bound for the cost in each state. A lower bound because the cost of the censored states is considered as bad as the worst cost in the observed state space. This is an underestimation.

Unfortunately this does not seem feasible as it turns out that Vincent's algorithm is $\mathcal{O}(|S|^3)$,

which we can reduce to $\mathcal{O}(|S|^2)$ at the cost of increased memory usage. Clearly this is unacceptable for the type of problem we are interested in. We can recall that although most of the work presented so far seemed to be of that same order, the sparsity of the matrices involved makes it $\mathcal{O}(|S|)$.

A next try could be to just use a part of Truffet's approach of adding all the missing probability weight to a lumped, 'bad' state and not using Vincent's algorithm. This would mean our result is not a bound, but it might still be useful. Here an implementation problem occurs. Recall that our state space is $M \cdot 2^M \cdot \binom{N+M^2-1}{M^2}$ dimensional for M job types. We can more or less collapse this, at least notionally, to M dimensional, corresponding to the queue length dimensions, since the 'badness' of a state seems to be dominated by those parameters. But our algorithms all consider a single lumped state and there seems to be no obvious way of generalizing this. The most obvious work-around seems to be to select a single lumped state. This again will mean this model is not necessarily a bound.

However even this does not seem to work. Between each iteration of the policy improvement algorithm, the decisions change and hence the model given by the censoring changes. So we are in essence trying to solve a different model at each iteration with a policy that is applicable for the previous model. Although we could have hoped the impact of this to be minimal, it turns out that, at least for the simple models tried, the consecutive iterations do not converge at all. Even a model with very reduced queue sizes did not converge on a policy after over 500 iterations of the policy improvement loop; this in stark contrast to the relative quick convergence for the truncated model.

This seems to lead to the inevitable conclusion that using censored Markov chains is inapplicable for this problem. There are technical difficulties in how to censor a multi-dimensional problem, numerical problems with the complexity of algorithms that provide bounds and, most problematic of all, the change in model between iterations of the policy improvement algorithm prevents convergence on a single policy.

4.6.2 Distributed Implementation

In this subsection we consider the possibility of a distributed implementation of the optimal policy computation. We did not implement a distributed implementation since it would enable at best a linear increase, with respect to the number of available computers, in the size of the state space we can handle. However model size increases worse than linear in almost all interesting parameters, e.g. for the truncation level this can be linear (if we consider an increase in the truncation level of only one job type), but even then the constant of the linear increase can be expected to be very large. If

we increase the amount of servers in the system, the size of the state space grows much worse than linear. And an even worse growth in state space size can be found when the amount of job types in the system is increased. All in all this makes distributed implementation not very attractive.

Furthermore we can only feasibly implement the value iteration in a distributed fashion. The policy improvement algorithm has as its most computationally intensive step the solving of a large set of related linear equations. Although there has been work on solving these in a distributed fashion, there does not seem to be any efficient way of doing this.

In contrast, the value iteration algorithm is in essence an iterative algorithm of the form:

$$V_i^n = f_i(\vec{V}^{n-1}) \; .$$

We can solve these equations in a parallel fashion, assigning to each available computer a subset S' of the state space. Each computer would then be able to compute the new values for $V_{S'}^n$ based solely on the values of the function from the previous iteration. A downside of this would be that the individual computers would have to wait for all to finish before they can start a new iteration. In addition they have to propagate the values of the previous iteration throughout the system which might be a bottleneck if the state space is large.

More efficient would be a distributed implementation. The difference with an parallel systems is that in a distributed system we assume that each computer works independently on part of the state space for multiple iterations. For our purpose it seems reasonable to assume only partial asynchronism in the system. In essence this means that we can use outdated values for some of the costs per state V_S^{n-1}, but there is a finite bound on how outdated these values can be. For a much more detailed description of partial asynchronism, see [BT89]. There we can also find the proof (based on Lyapunov stability) that this type of partial asynchronous algorithm converges and, if the cost function is linear, the convergence is geometric.

Thus we suggest the following distributed implementation:

1. Set an initial cost for each state on a central database. Also specify a stopping criterion.

2. Partition the state space S in i sets, where i is the number of available servers and an initial number of iterations n_{iter} that each run will consists off. This should all done centrally. The next steps are executed by each server independently.

3. Let each server calculate, independently, the cost for its chunk of the state space S_i for n_{iter} iterations. Here we use the newly calculated, and locally updated, cost for the states in S_i and

the initial, outdated, cost for the other states.

4. Return the newly calculated cost after n_{iter} iterations for S_i and overwrite these (centrally) as the cost for these states.

5. Check the stopping criterion. If this is not satisfied, get the most up to date cost calculations from the central database for each state and go to step 3 to calculate the cost of S_i for the next n_{iter} iterations.

In the algorithm above, the art would lie in picking the partition and the number of iterations in step 2. Clearly we want to pick the size of each partition proportional to the speed with which it can calculate the cost over these states. We also want to minimize the interdependency between each partition, i.e. we want each partition to be strongly connected (in a topological sense) but the partitions to be weakly connected to each other. The number of iterations each computer should run before updating its information should balance the overhead of doing so with the improved convergence properties. This is difficult as there seems to be no way to estimate the latter. The use of theory on graph or hypergraph partitioning would seem appropriate here. All in all the speed up gains from a distributed implementation seem, therefore, not significant enough to outweigh the technical difficulties in implementing it.

4.7 Summary

In this chapter we presented, in Section 4.2, several optimization goals for the model presented in Chapter 3. We then discussed two solution methods: policy improvement, in Section 4.3 and value iteration, in Section 4.4. These were compared in Section 4.5. There it was found that policy improvement seems the preferable algorithm. But it was also shown that it is only possible to get the optimal policy for very small systems and that even then there are several non-trivial modeling decisions that we have to take. This seems to strongly suggest that calculation of the optimal policy is infeasible for interesting systems. Even careful precalculation of the optimal solution would be problematic since the optimal policy depends on all the parameters of the system. In Section 4.6 we discussed two possible speed up methods: using censored Markov chains and a distributed implementation of the calculation. Although the latter seems a promising method, both have serious drawbacks and do not seem to be a panacea for the problems of calculating the optimal solution.

Chapter 5

Heuristic Policies

5.1 Introduction

Since we showed in the previous chapter that computing the optimal policy is almost always infeasible, we will introduce some heuristics in this chapter. These should give a good performance without the need of lengthy computations. We will split our discussion on possible heuristics in two separate parts. In the first part, in Section 5.2, we discuss two fluid-approximation based heuristics. In the second Section, 5.3, we consider two heuristics that do not require knowledge of the parameters of the system.

We then compare these for a few limited cases to the optimal dynamic solution in Section 5.4. Next, in Section 5.5, we compare the performance of the heuristics with each other and two static allocations. We end the chapter with the usual summary in Section 5.6. It should be noted that these policies were recently implemented in a real system by A. Chester et. al. from Warwick University, see [CXHJ08] for more information.

5.2 Fluid-approximation Based Heuristics

In this section we will discuss two heuristics based on fluid approximation of queues. These two heuristics, the average flow heuristic discussed in Subsection 5.2.1, and the on/off heuristic discussed in 5.2.2, are very similar in approach. They both treat the various queues in the system as fluid queues. This is a very common approximation method and will allow us to get simple, explicit expressions for the time it takes to empty each queue, given a decision. This in turn will allow us to pick the best decision. The main difference between both heuristics will lie in the way they model the on/off periods in the arrival process.

The use of a first order fluid approximation, which we use here, is appropriate when we are only interested in mean value analysis. Furthermore, for the approximation to be accurate, the number of jobs passing through the system should be large.

5.2.1 Average Flow Heuristic

This policy ignores the on/off periods and treats queue i as a deterministic fluid which arrives at rate γ_i, given by

$$\gamma_i = \frac{\lambda_i \eta_i}{\xi_i + \eta_i} \ . \tag{5.1}$$

That fluid is consumed at rate $k_i\mu_i$, where k_i is the number of servers currently allocated to queue i.

Suppose that two queues, i and j, have current sizes k_i and k_j, and currently allocated numbers of servers k_i and k_j, respectively. If no further actions are taken and both queues are stable (i.e. $\gamma_i < k_i\mu_i$ and $\gamma_j < k_j\mu_j$), then those fluid queues would decrement at constant rates and would empty in times $j_i/(k_i\mu_i - \gamma_i)$ and $j_j/(k_j\mu_j - \gamma_j)$, respectively. The total holding costs incurred would be proportional to the areas of the triangles defined by the slope of the service rates and the intervals until the queues empty. For a graphical representation of this, see Figure 5.1.

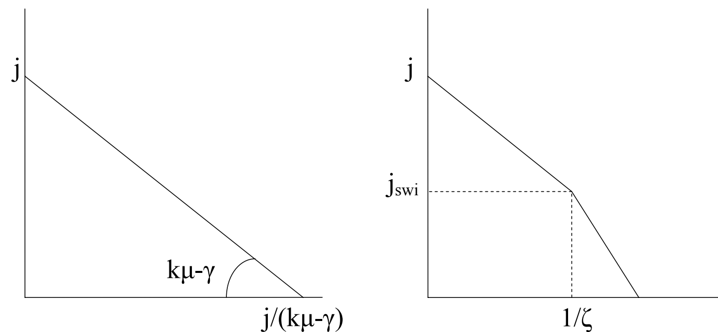

Figure 5.1: The triangle whose area represents the holding cost (left) and the cost when there is a switch present (right).

Hence, the Average Flow heuristic estimates the cost of taking no action with queues i and j as

$$C_0(i,j) = \frac{c_i j_i^2}{2(k_i\mu_i - \gamma_i)} + \frac{c_j j_j^2}{2(k_j\mu_j - \gamma_j)} \,. \tag{5.2}$$

On the other hand, if a decision is made to switch a server from queue j to queue i, and that switch takes time $1/\zeta_{j,i}$ (deterministic), then the service rate at queue j immediately reduces to $(k_j - 1)\mu_j$, while that at queue i remains the same for the duration of the switch and then increases to $(k_i + 1)\mu_i$. Assuming that queue i does not empty during the switch, its size at the point when the switch is completed would be equal to m_i, where

$$m_i = j_i - (k_i\mu_i - \gamma_i)/\zeta_{j,i} \,. \tag{5.3}$$

See also Figure 5.1. The total holding cost incurred in clearing both queues is estimated as

$$C_1(i,j) = \frac{c_i(j_i + m_i)}{2\zeta_{j,i}} + \frac{c_i m_i^2}{2((k_i+1)\mu_i - \gamma_i)} + \frac{c_j j_j^2}{2((k_j-1)\mu_j - \gamma_j)} \,. \tag{5.4}$$

As before, the terms in the right-hand side correspond to the areas of triangles bounded by different service slopes.

At every arrival or departure event, the Average Flow heuristic evaluates C_0 and C_1 for every pair of queues i and j, where i is the queue where the arrival occurred, or j is the queue where the departure occurred. If $C_1 < C_0$, a server is switched from queue j to queue i. If that inequality holds for more than one queue i, the switch is made to the queue for which the difference $C_0 - C_1$ is largest. If a contemplated switch would leave queue j potentially unstable (i.e., $(k_j - 1)\mu_j \leq \gamma_j$), then it is not made. If, at a decision instant, a server is in the process of being switched, it is counted as being already available at the destination queue.

5.2.2 On/Off Heuristic

When making allocation decisions, this heuristic assumes that the current phase of each arrival process, whether it is 'on' or 'off', will last forever. So if queue 1 is in an 'on' arrival state and queue 2 in an off arrival state, this approximation assumes they will stay that way. Again each queue i is treated as a fluid, but the arrival rate is taken to be $\gamma_i = \lambda_i u_i$, where $u_i = 1$ if the queue i arrival process is in an on-period and $u_i = 0$ if it is in an off-period.

Switching decisions are made not only at arrival and departure instants, but also when an arrival

process changes phase from 'on' to 'off' or vice versa. As well as evaluating the estimated costs C_0 and C_1, of doing nothing or switching *one* server from queue j to queue i, the On/Off heuristic evaluates the costs C_s, of switching s servers from queue j to queue i, for $s = 2, 3, \ldots, k_j$. This is necessary because a phase change can make a big difference to the arrival rate at a queue, requiring or releasing more than one server. When calculating C_s for $s > 1$, one could assume that all s servers become available at queue i after a switching interval of length $1/\zeta_{i,j}$. Alternatively, the assumption could be that the s switches complete at different times: the earliest after an interval $1/(s\zeta_{i,j})$, the next after a subsequent interval $1/((s-1)\zeta_{i,j})$, etc. The first alternative would be appropriate if the switching times are nearly constant, the second if they are exponentially distributed. In both cases, the costs are evaluated by adding together areas under linear segments. We will use the second version throughout.

As before, the switching decision that yields the largest cost reduction is taken. If no reduction is possible, or if all estimated costs are infinite (that can happen, for example, if all arrival processes are in an on-period and the corresponding arrival rates are greater than the available service rates), then no action is taken.

5.3 Parameterless Heuristics

In this section we will discuss two other heuristics. They are somewhat similar in that neither of them uses a priori knowledge of the parameters of the system. They do have some basic knowledge about the system, e.g. the existence of on/off periods, the number of job types, relative holding cost of each job type etc. But this knowledge can be assumed for most systems. What they lack is the value of parameters of the system such as the arrival rates, completion rates, length of on/off periods, etc. Also neither of them considers the presence of switching costs. At the given point in time, the new allocation is implemented, regardless of the cost of that switch.

5.3.1 Window Heuristic

This policy uses a sliding window of length w for the purpose of collecting queue size statistics. It ignores the existence of on- and off-periods. Allocation decisions are made at intervals of fixed length, v. If, at a decision point t, the average size of queue i observed during the interval $(t-w, t)$ is L_i $(i = 1, 2, \ldots, M)$, then the number of servers allocated to queue i for the duration of the next

decision interval is roughly proportional to c_iL_i. More precisely, those numbers are given by

$$k_i = \left\lfloor N \frac{c_i L_i}{\sum_{j=1}^{M} c_j L_j} + 0.5 \right\rfloor (i = 1, ..., M-1) \ ; \ k_M = N - \sum_{j=1}^{M-1} k_j \ , \tag{5.5}$$

if all k_i are non-zero. If any k_i is zero, replace k_i with 1 and the largest k_j $(i \neq j)$ by $k_j - 1$. Repeat this process until all k_i are non-zero. If N is sufficiently large, the resulting allocation should be feasible.

5.3.2 Queue Length Heuristic

Again, a sliding window of length w and a decision interval of length v are used and again on- and off-periods are ignored. However, the statistics collected during the window are for the purpose of estimating the current arrival rate of type i, λ_i, and the current service rate of type i, μ_i; the resulting estimate of the current offered load of type i is $\rho_i = \lambda_i/\mu_i$. Based on these observations, the number of servers allocated to queue i is set to be roughly proportional to $c_i\rho_i$:

$$n_i = \left\lfloor N \frac{c_i \rho_i}{\sum_{j=1}^{M} c_j \rho_j} + 0.5 \right\rfloor (i = 1, ..., M-1) \ ; \ n_M = N - \sum_{j=1}^{M-1} n_j \ , \tag{5.6}$$

adjusting, if necessary, the numbers to make them positive. Also, if all loads are observed to be zero during the current window, then servers are allocated proportional to holding costs only.

5.4 Comparison to the Optimal Solution

The cost of each policy is calculated throughout by simulation. For some of the policies it is possible to calculate an analytical cost (although not completely trivial). But for the two heuristics that are responsive, i.e. the window and the Queue Length heuristics, this is not possible. So for consistency we used simulation for all of them. It should also be remembered that for a feasible calculation of the optimal policy, we need to restrict the maximum allowed queue size quite severely. Here all optimal policies were calculated for queue sizes up to 30. This means the simulations are also limited thusly. This could be thought of as corresponding to admission blocking when the queue size is 30.

5.4.1 Increased Arrival Rates

As a first experiment we increase the number of arrivals for a system with two job types who are almost completely symmetrical. Both arrival streams are on for a mean time of $\xi_1^{-1} = \xi_2^{-1} = 200$

and off for a mean time of $\eta_1^{-1} = \eta_2^{-1} = 100$. Jobs can be completed at a rate of 0.1 for either job type by each of the three servers in the system. The only asymmetrical aspect of the system is that job type 2 is considered twice as expensive as job type 1. We increase the arrival rate for this system from 0.015 to 0.15 for both job types.

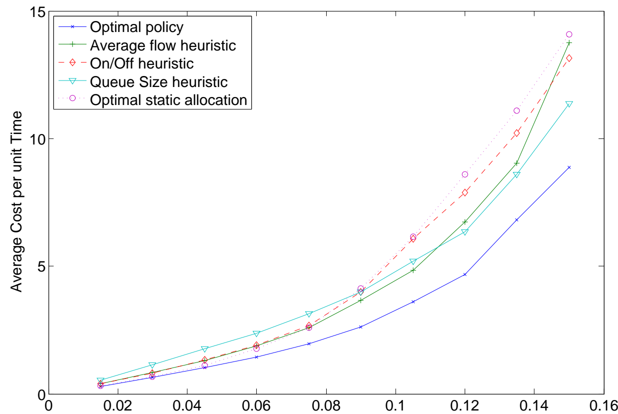

Figure 5.2: Increasing arrival rate for both job types on x-axis, mean cost on y-axis, for optimal policy and several heuristics

In Figure 5.2 we show the resulting cost for the optimal policy and several heuristics. They show a very similar trend, where the heuristics are reasonably close to the performance of the optimal policy, if we keep the amount of effort required to obtain the latter in mind. It is especially noteworthy that the parameter free Queue Length heuristic is performing well.

It is interesting to see the way in which the optimal policy improves on the heuristics. To this end we show the amount of switches made by each policy in Figure 5.3. The remarkable drop in the number of switches by the On/Off heuristic when the arrival rate reaches 0.1 is due to the fact that the arrival rate at 0.1 is equal to the completion rate of both job types. This implies that assigning just one server to a job type that is on will make that queue (temporarily) unstable, both in reality and according to the On/Off heuristic. This will prevent the heuristic from making any switch leaving just one server serving a queue that is on. The other heuristics and the optimal policy do not have these considerations.

If we look at the amount of switches it seems that the optimal policy is not so different; if anything it is slightly more conservative. So the difference in performance seems to lie mainly in the intelligence of the switches. We will examine this in a bit more detail for the case where

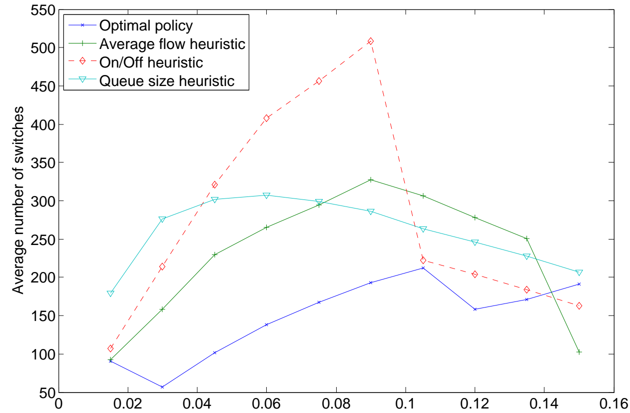

Figure 5.3: Increasing arrival rate for both job types on x-axis, number of switches on y-axis, for optimal policy and several heuristics

Table 5.1: Decisions of (optimal policy, Average Flow heuristic, On/Off heuristic). 1 denotes switching a server from job type 1 to job type 2 and -1 denotes the converse switch. Likewise -2 denotes the switch of two servers from job type 2 to job type 1. The columns represent different values for queue length 2 and the rows for queue length 1.

j1	j2 = 0	1	2	3	4	5	6	7	8
0			0,0,1	1,0,1	1,0,1	1,0,1	1,0,1	1,0,1	1,0,1
1	0,-2,-2								
2	-2,-2,-2	-1,0,0							
3	-2,-2,-2	-1,-1,0							
4	-2,-2,-2	-1,-1,-1	0,-1,-1						
5	-2,-2,-2	-1,-1,-1	0,-1,-1	0,-1,-1					
6	-2,-2,-2	-1,-1,-1	0,-1,-1	0,-1,-1					
7	-2,-2,-2	-1,-1,-1	0,-1,-1	0,-1,-1	0,-1,-1				
8	-2,-2,-2	-1,-1,-1	0,-1,-1	0,-1,-1	0,-1,-1	0,-1,-1			

$\lambda_1 = \lambda_2 = 0.06$.

We can look at the decisions of what is in a sense the most straightforward case. Both job types are on, 1 server is allocated to job type 1 and 2 to job type 2, which is twice as expensive. Here we would expect the policies to do very little. And indeed the optimal policy does little when the queue lengths are small, as we can see in Table 5.1. As before, the decisions are shown as triples, this time signifying the decision made by the optimal policy, the average flow heuristic and the on/off heuristic. Empty spaces mean all three policies made the decision not to switch any server. Both heuristics however, are much more aggressive in switching servers from job type 1 to job type 2, when the latter queue grows.

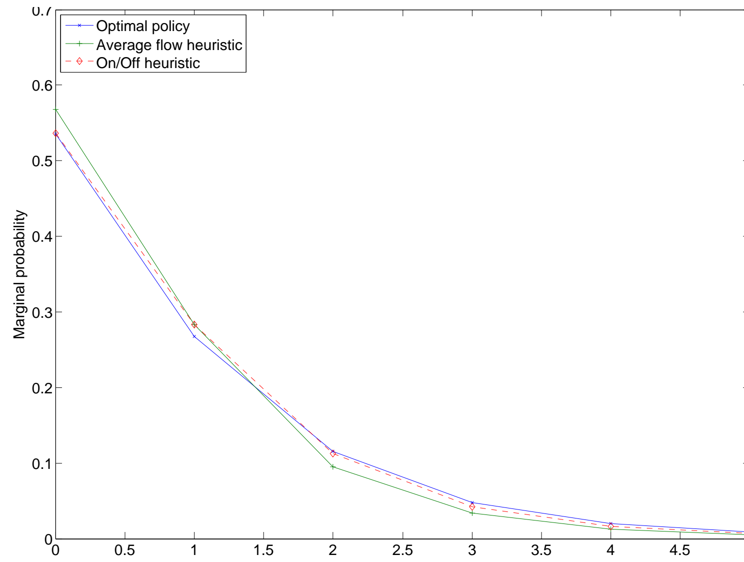

Figure 5.4: Marginal density for low queue lengths, for job type 1

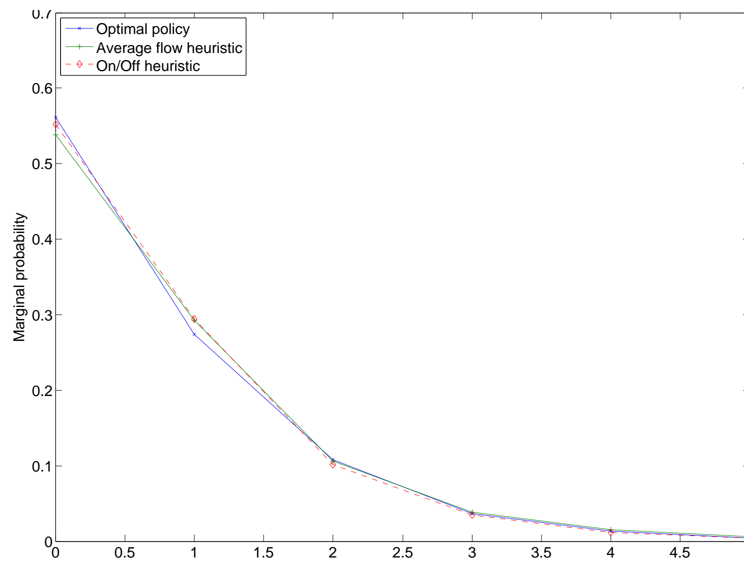

Figure 5.5: Marginal density for low queue lengths, for job type 2

The effect of this on the marginal densities for both job types can be seen in Figures 5.4 and 5.5. Job type 2 has a (slightly) lower marginal probability for low queue lengths under the heuristic policies. But this is more than compensated for by the lower marginal probability for lower queue lengths of job type 1.

5.4.2 More Expensive Bursts

Next we look at a case where job type 2 represents steady 'background' traffic without any on/off periods. The average arrival and completion rate of this type is 1, which is also the mean time it takes to perform any switch. Job type 1 is bursty. It has off periods with a mean of 100 units of time and on periods with a mean of 10 units of time. When it is on, the arrival rate is 10. The objective here is to increase the cost, but not the intensity, of the bursts and see how the system copes with this. To this end we fix the cost of job type 2 at 1, but increase the cost of job type 1 from 0.5 to 10. There are three servers in the system.

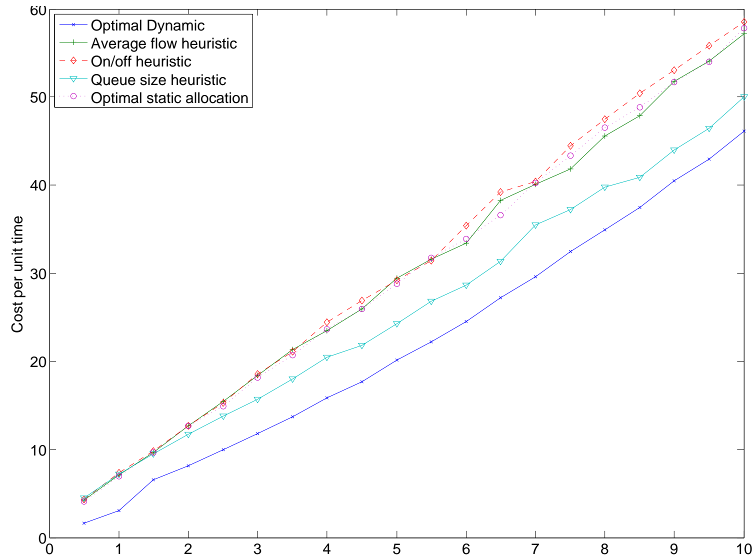

Figure 5.6: Increasingly expensive bursts. The x-axis displays the cost of the bursty job type, while the y-axis has the mean cost per unit time

In Figure 5.6 we can see the effect this has on the overall cost. That in itself is not so interesting, since we could expect the cost of the system to rise more or less linearly. We note that the heuristics perform about as well as the optimal static allocation. The notable exception being the Queue Length heuristic which does very well indeed. The window heuristic is not displayed in this graph since it performs so poorly that it makes the graph less legible. More interesting than these observations with regards to the increased cost, is the impact this increased cost has on the decisions the different policies make. For the optimal policy this is fairly minimal, even when we compare the policy for the lowest cost of job type 1, i.e. 0.5, to that with the highest, 10. The policy is just slightly more willing to move a server away from the cheaper background traffic to the more expensive bursty traffic when both job types have jobs arriving. But for both the on/off and the average flow heuristic there is no difference at all. This is somewhat surprising. When we examine the policies in detail, we see

they more or less behave in the following way. When there is no burst, all the servers are assigned to the (background) job type 2. When a burst starts and the queue for job type 1 increases, one of the three servers is moved from job type 2 to job type 1. When the burst ends and the queue for job type 1 is emptied, that server switches back to job type 2. Although there is clearly room for improvement over this policy, this cannot be enforced by even a very hefty increase in cost for the bursty job type.

5.4.3 Less Intensive Bursts

In this example we examine the effect of increasingly less intensive bursts on the system. Job type 2 will again represent steady background traffic, devoid of on/off periods. The average arrival rate is 0.1. Job type 1 will be increasingly less bursty, whilst keeping the average number of arrivals over time the same. This means that the mean on-time (ξ_1^{-1}) will vary from 50 to 140 units of time, whilst this job type will be off with a mean time of 50. To keep the mean number of arrivals for this job type the same throughout, the arrival rate is fixed at:

$$\lambda_1 = \frac{\xi_1 + \eta_1}{10\eta_1} \ . \tag{5.7}$$

There are three servers in the system that can serve either job type 1 or job type 2 at a rate of 0.1 each. Both job types are equally expensive. Switching is free, but in the mean takes the equivalent of one job completion, i.e. $\mu_1 = \mu_2 = 0.1 = \zeta_{1,2} = \zeta_{2,1}$.

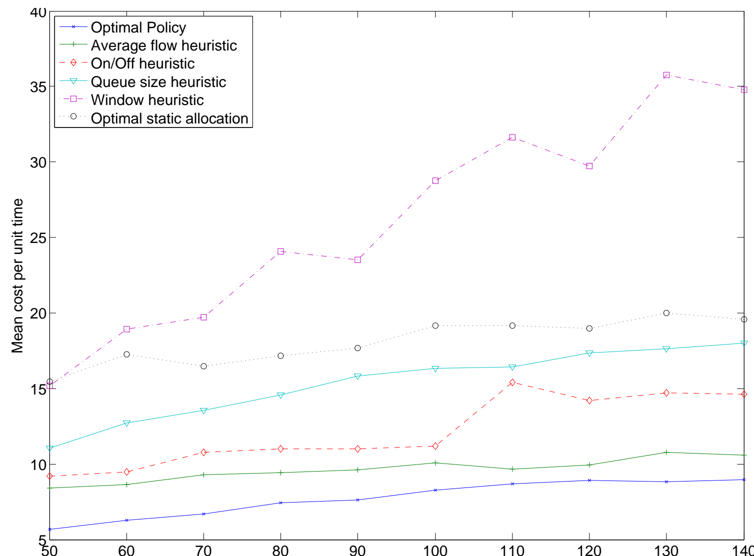

Figure 5.7: Increasingly less intensive bursts, keeping the total number of arrivals the same. The x-axis displays the mean length of the bursts, while the y-axis has the mean cost per unit time

In Figure 5.7 the average cost per unit time is plotted against the length of the bursts. The cost is an average in the sense that these costs are the mean of 50 simulation runs with the policy. The length of each run is about 10000 units time, roughly equivalent to between 100 and 50 cycles of on/off periods, depending on the parameters of the system. The relative 95% percentile error of these averages is typically small, i.e. below 7%, except for the window heuristic, where it can get to be as high as 23 %.

All policies seem to be doing better when the bursts are shorter and hence more intensive. This seems quite logical as that is the scenario in which they can make the biggest gain by switching resources over when the bursts become active. The difference between the performance of the optimal policy and the average flow heuristic is quite small. The other heuristics do not perform quite that well and fall in the band between the optimal dynamic policy and the optimal static allocation, where no switching is carried out. The notable exception here is the window heuristic which does remarkably poorly. The reason for this is not immediately obvious. It can perhaps be attributed to several causes. First of all the heuristic requires a problem specific time between switching decisions T_{swi}. Choosing this can be fidgety and it is hard to make a good choice. The results in the graph are for $T_{swi} = \frac{4}{\xi_1}$, but the results are similar for many other choices. And secondly, the heuristic estimates the parameters of the system based on a past window size, T_{win} which again has to be chosen quite carefully. This makes it hard to strike a good balance between stability of the results and flexibility to reflect changes in the system. By contrast, the policy that allocates servers based on mean queue length does quite well. Here the window size and switching frequency were both chosen at $T_{win} = T_{swi} = \frac{2}{\xi_1}$ and the heuristic seems sufficiently robust to give good results.

5.4.4 Increased Switching Times

In the final experiment shown in this section, the switching times are increased. Whereas previously we had a switching time of roughly one job completion, we now consider switching times from 1 job completion to 2.5 job completions. The other parameters of the system are the same as those in the example considered in Subsection 5.4.1. So there are three servers, two symmetrical job types with a mean on-time of 200 and a mean off-time of 100. The arrival rate for both job types is fixed at 0.06. The two job types differ in the holding cost associated with them, i.e. job type 2 is twice as expensive (important) as job type 1.

In Figure 5.8 we see the effect of this increase in mean switching time. It should be noted that

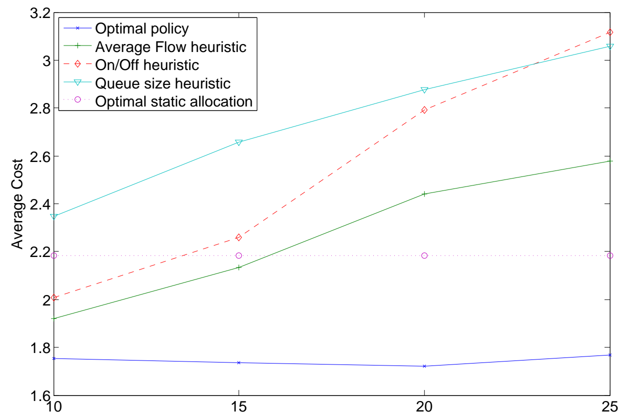

Figure 5.8: Increasingly longer switching times. The x-axis displays the mean length of the switching time, while the y-axis has the mean cost per unit time

the system is fairly lightly loaded, so it is not surprising that the heuristics are outperformed by the static policy when the switching times increase too much. We can expect that the more heavily loaded the system is, the higher the switching times can be to still allow switching to be profitable.

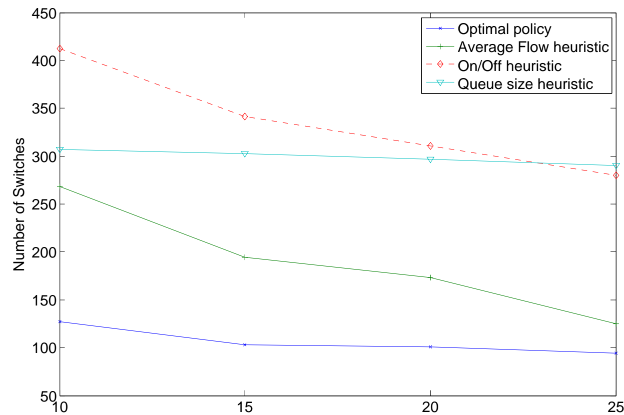

Figure 5.9: Increasingly longer switching times. The x-axis displays the mean length of the switching time, while the y-axis has the number of switches

It is surprising that the optimal policy seems very insensitive to increased switching times. This can partly be explained by Figure 5.9. There we can see the optimal policy switches relatively little and seems fairly insensitive to the mean switching times in the amount of switching it does. We

also see that the average flow and on/off heuristics do decrease the amount of switches they make, more or less linearly, when the switching times go up. The Queue Length heuristic however, does not. This is because there is no notion of switching time. We can therefore expect this heuristic to perform poorly when the switching times are very large. But then that is also the case where switching is a lot less attractive.

5.5 Performance

In this section we will investigate the performance of the heuristics in more detail and for more realistic systems than those considered in the previous Section 5.4. We also show some results using the ρ-rule allocation which was discussed previously in this thesis. Because the systems under consideration in this section contain a lot more servers and experience fairly high loads, they require very large state space descriptions which make calculation of the optimal dynamic allocation unfeasible.

The following parameters were kept fixed throughout.

Number of job types: $M = 2$.

Number of servers: $N = 20$.

Average required service times: $1/\mu_1 = 1/\mu_2 = 1$.

For the Queue Size and Load heuristics, the statistics window and the decision interval were equal: $w = v = 50$.

In the first experiment, the average 'on' and 'off' periods were equal at the two queues, and so were the average switching times: $1/\xi_i = 1/\eta_i = 100$, $i = 1, 2$; $1/\zeta_{i,j} = 1$, $i, j = 1, 2$. The two arrival rates were also equal, and were increased simultaneously. Some asymmetry was introduced by making type 2 jobs more expensive than type 1: the holding costs were $c_1 = 1$, $c_2 = 1.5$. The simulated time for each run was 10000 time units. Since each arrival process is 'on' for about half of that time, if $\lambda_1 = \lambda_2 = 10$, a total of about 100000 jobs go through the system. (Note that the simulations were required only for the dynamic policies. The static ones could have been solved numerically, but since the simulation programs could easily be adapted to different policies, they were used in all cases.) The 95% confidence intervals were computed, but are not included in the figures. Their worst half-widths vary from around 20% of the estimated value, for the heavily loaded

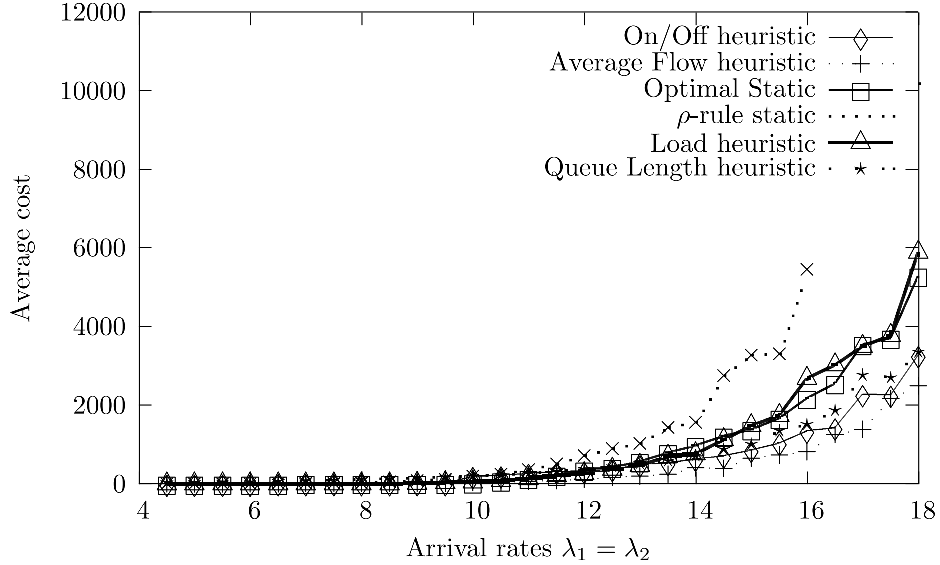

Figure 5.10: Policy comparisons: increasing λ_1, λ_2

ρ-rule results, to around 5% for the heavily loaded average flow heuristic. Generally, the results are more accurate when the load is lower.

Figure 5.10 shows the average costs achieved by the two static and four dynamic policies, as the load increases. As expected, at light loads it does not matter very much which policy is adopted. However, differences start appearing at medium loads and become even larger at heavy loads.

The ρ-rule has the worst performance. The application of Equation (5.6) results in 8 servers being allocated to type 1 and 12 servers to type 2 throughout. Hence, when $\lambda_1 \geq 16$, the system becomes unstable and in the long run incurs infinite cost. The Optimal Static policy allocates 9 servers to type 1 and 11 to type 2 for most of the range, changing to 10 and 10 when the arrival rates become greater than about 17. That policy achieves considerably lower costs than the ρ-rule, and remains stable for $\lambda_i < 20$. The Load heuristic has a similar performance to the optimal static policy.

The other dynamic allocation policies have a significantly better performance, particularly for $\lambda_i > 13$. The costs they achieve are similar, although the Average Flow policy appears to be consistently slightly better than On/Off heuristic, which in turn is slightly better than the Queue Length heuristic.

Note that the Queue Length heuristic does not require any knowledge of parameters for its operation; it relies on its own statistics. The fact that it performs almost as well as policies that need the exact values of all parameters is very encouraging.

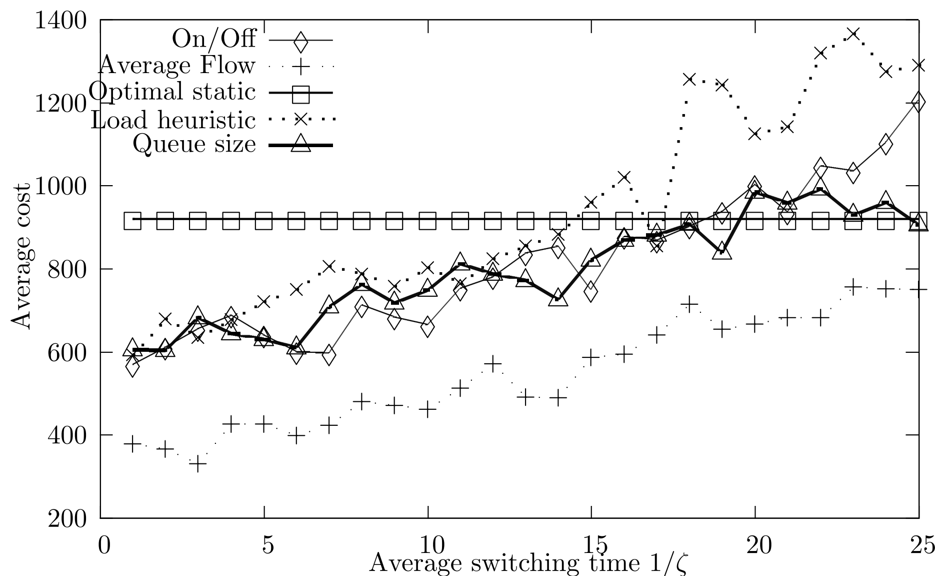

Figure 5.11: Performance of different policies for increasing switching times

The second experiment aims to examine the effect of increasing the average switching time, in a moderately loaded system ($\lambda_1 = \lambda_2 = 14$). The results are shown in Figure 5.11. The half-widths of the 95% confidence intervals of the results are around 10% of the mean for all of the results except the Load heuristic, where they can go up to 15% of the mean. We observe that all dynamic heuristics outperform the Optimal Static policy (whose allocation does not change) for most of the range, and the Average Flow heuristic does so for the entire range. When the switching times are large, the Queue Length heuristic is the next best one, with a similar performance to the Optimal Static; the Load and On/Off heuristics appear to become worse.

The static allocation policy based on the ρ-rule is not included in these and subsequent comparisons because its performance can only be worse than that of the optimal static policy.

It is again notable that the Queue Length heuristic performs well over the entire range of switching times, despite having no information about those parameters.

The last experiment involves a system with asymmetric traffic characteristics. Type 1 jobs arrive in a stream which is 'on' most of the time: $\xi_1 = 1/100000$, $\eta_1 = 1$, $\lambda_1 = 10$. That stream would need at least 10 servers in order to remain stable. Jobs of type 2 arrive in short bursts, with long intervals in between: $\xi_2 = 1/25$, $\eta_2 = 1/500$ (i.e., the arrival stream of type 2 is 'on' for less than 5% of the time). The arrival rate of type 2 during 'on' periods, λ_2, is increased from 20 to 120 in steps of 5, thus increasing the overall load of job type 2. The switching costs are very small.

Figure 5.12 illustrates very clearly the benefits of using an appropriate dynamic allocation policy.

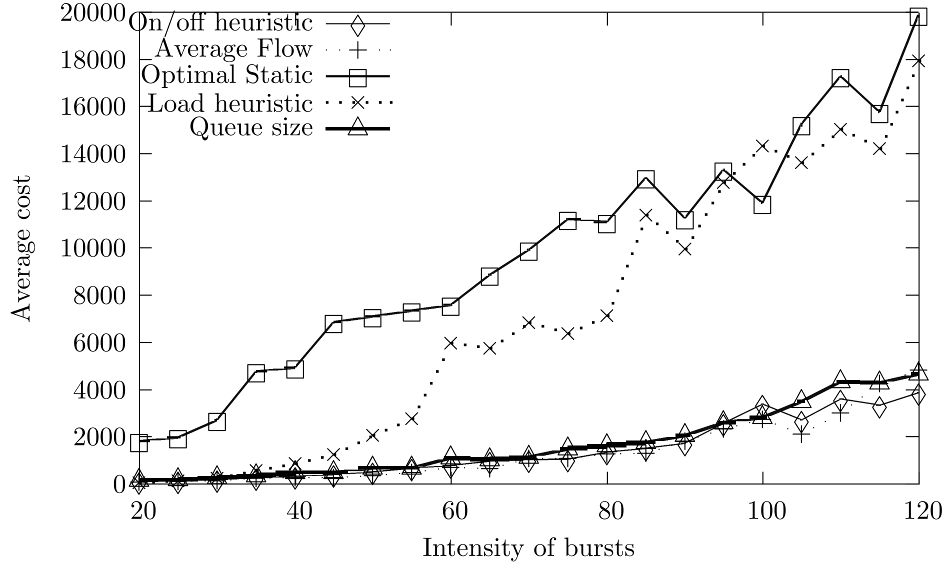

Figure 5.12: One steady and one bursty source: increasing λ_2

Three of the heuristics, namely Average Flow, On/Off and Queue Size, have a similar performance. However, the average cost achieved by any of them can be an order of magnitude lower that that of the optimal static policy. The Load heuristic is reasonable when the input is not very bursty, but becomes as bad as the optimal static policy when the burst intensity increases.

Once more, it is worth pointing out that the Queue Length heuristic, which does not need to know the values of the parameters, performs nearly as well as the policies that do.

5.6 Summary

In this chapter we introduced several heuristics. These provide us with possible dynamic policies for resource allocation in large systems. In Section 5.2 we introduced two fluid-approximation based heuristics, of which the Average Flow heuristic seemed to perform best under a variety of scenarios, as can be seen from the results in Section 5.5. Likewise in Section 5.3 we introduced two parameterless heuristics. Of those the Queue Length heuristic did very well, especially considering it does not use a priori knowledge of the parameters of the system; again this can be seen in Section 5.5. Despite those positives the results in Section 5.4 show that there is still some scope for improving the performance of the heuristics when compared to the optimal dynamic policy. This is especially true when the switching times are large.

Chapter 6

Power Management

6.1 Introduction

In this section we will examine a model that is highly related to that examined in Chapter 3 and the subsequent chapters. But here we do not consider multiple job types. Instead we have a single job type, and consider the trade off between power consumption and response time, i.e. we allow servers to be powered down and up dynamically. Powered down servers cannot serve jobs, but generate energy savings instead.

This is clearly a related problem to that considered previously but it also has some unique characteristics. In Section 6.2 we begin by discussing our model for this situation. After that we introduce some heuristic policies in Section 6.3. We then proceed to show some results with regards to the performance of these policies in Section 6.4. Section 6.5 will briefly consider non-Poisson arrival processes. And we will conclude the chapter with a summary in Section 6.6.

Some of this work has appeared previously in [STM08].

6.2 Model

As before we consider a model where there are N homogeneous servers. Instead of being assigned to different job types, these can be in one of two states: power up or power down. When a server is in power up, it can service incoming requests. When a server is in power down, it cannot process any requests, but will consume less (or no) power. The details of the power down state are expressly left ambiguous. It can mean the server is completely shutdown, in some sleep state or any other state, as long as it is less power consuming. We will refer to 'powering up' to denote the decision to switch a server from the power down state to the power up state and 'powering down' for the converse.

The service time of a request is assumed to be exponentially distributed with rate μ. The requests themselves arrive according to a two-phase Poisson process, i.e. there are 'high' and 'low' arrival periods. During a high period, denoted by $l = 1$, requests arrive as a Poisson process with rate λ_{high}. During a low period, notation $l = 0$, fewer requests arrive, with rate λ_{low}. The high and low periods themselves have durations that are distributed exponentially with mean $1/\xi$ and $1/\eta$ respectively. We consider the servers that are currently powered up to be part of one (logical) pool, called the powered servers pool, with an unbounded queue which holds the incoming requests. We denote the number of powered up servers by k_{up}. The number of jobs in the queue (including jobs currently being processed) will be denoted by j. The other servers, which are powered down, are in a pool as well, which we will call the powered down pool. We will denote their amount by k_{down}. Since all the servers are homogeneous, we do not distinguish between individual servers in each pool, but rather focus on the number of servers powered up or down.

As before we assign a cost, c_{job}, to keeping a job in the system for one unit of time. These holding costs reflect the relative value of completing a job quickly. Conversely we assign a negative cost (i.e. profit) c_{pow} to keep a server powered down for a unit of time. This should reflect the relative energy savings of not powering up a server. Please note that here we normalized energy costs of a powered up server to be zero. This means that the total cost can be negative, i.e. imply a profit, through power savings. This obviously does not reflect the real cost but serves as an equivalent optimization goal.

A server can be switched from the pool of powered servers to that of powered down servers. This will take an amount of time, assumed to be exponentially distributed with rate ζ_{down}. We will denote the number of servers powering down by m_{down}. Conversely they can be powered up again. The number of servers powering up will denoted m_{up}. The time this will take is again assumed to be exponentially distributed, now with rate ζ_{up}. During a switch a server cannot serve jobs but it does consume power, i.e. it does not accumulate a profit from energy savings. Furthermore, the powering up and down of machines can incur additional costs, e.g. through peak power consumption, which we will denote by C_{up} and C_{down} for powering up and powering down respectively.

This means we can describe a state S of the system by:

$$S = (j, l, k_{up}, k_{down}, m_{up}, m_{down}) \,. \tag{6.1}$$

There is some redundancy in the notation since the total number of servers in the system should be constant, i.e. $k_{up}+k_{down}+m_{up}+m_{down} = N$. This is done because of convenience and has no further

impact. The system behaviour is further characterized by the following transition probabilities (when no switching decision is made):

$$r(S,S') = \begin{cases} l\lambda_{high} + (1-l)\lambda_{low} & \text{if } j' = j+1 \\ \min(k_{up}, j)\mu & \text{if } j' = j-1 \\ m_{down}\zeta_{down} & \text{if } k'_{down} = k_{down}+1 \text{ and } m'_{down} = m_{down}-1 \\ m_{up}\zeta_{up} & \text{if } k'_{up} = k_{up}+1 \text{ and } m'_{up} = m_{up}-1 \\ l\xi & \text{if } l' = 0 \\ (1-l)\eta & \text{if } l' = 1 \end{cases} . \quad (6.2)$$

This characterizes a continuous time Markov chain. We can convert this into a Markov decision process by associating a set of allowed actions $\{a\}$ with each state S. Here we consider these actions to be:

- doing nothing, denoted by $a = 0$
- powering down i servers, with $0 < i \leq k_{up}$, denoted by $a = -i$
- powering up i servers, with $0 < i \leq k_{down}$, denoted by $a = i$.

These delays and costs will make the decision of when to power a server up or down non-trivial, especially in an environment with bursty arrivals. Our overall optimization goal is now to minimize the cost of the system, i.e. to find a policy of powering up/down that minimizes the average cost of the system per unit time. Here the relative value of the holding cost c_{job} versus the profit from keeping a server powered down are implicitly traded off.

Please note that the model given here is a (continuous time) Markov decision process (MDP) again. In principle these can be solved, i.e. we can find the policy that minimizes the long-term mean operating cost of the system, but as we have shown in Chapter 4 this is often very difficult. We can summarize the results we found there and apply them to this model. For example, a proper solution requires truncation of the maximum allowed queue length. We could consider a system with 25 servers and a maximum allowed queue length of 50. This number includes the jobs currently being processed, so it allows for roughly 25 waiting jobs, a rather low number. Such an MDP has 334152 states. This might seem a manageable number, but this would give rise to a system of 334152 simultaneous equations in as many different variables. These equations would then have to be repeatedly solved to find the optimal power-policy, in effect requiring the use of a (admittedly

very sparse) matrix with over 111 billion elements. For a system of 26 servers, this rise to 372708 states and a matrix of just under 139 billion elements. To make matters worse, the calculated policy is non-trivially dependent on every single parameter of the system, making even the most clever precalculation infeasible.

So in line with the results in Chapter 4, we find that we will be unable to calculate the optimal dynamic policy for interesting cases. Furthermore the ways to calculate this policy are, in principle, the same as those described in the previous chapter. So we will not discuss the optimal dynamic policy for this model any further. In the next section we will examine some other policies.

6.3 Policies

6.3.1 Introduction

In this section we will discuss some power management policies. The optimal static policy is excluded here, since in this context it does not seem as meaningful. After all, if some servers are permanently powered down, they cannot really be regarded as part of the system.

The heuristics that are considered here are quite similar to those discussed in Chapter 5. Since the model is slightly different, there are some differences. Therefore the description of the heuristics here will be somewhat elaborate.

6.3.2 Idle Heuristic

This heuristic policy follows the naïve policy of powering down any server that is idle and powering up a server, if possible, when there are jobs in the queue that are not currently being served by any server. It does not take account of switching times. Because in general the switching times are non-zero, we have to be slightly more precise. That is, we power up a server, if possible, when the number of jobs in the queue is bigger than the number of servers currently servicing a job and the number of servers being switched on, i.e. when: $j > k_{up} + m_{up}$. This assumes there are no batch arrivals, but we can easily extend the heuristic for that case by saying we power up $j - k_{up} - m_{up}$ servers.

It is worth noting that even when switching is both instantaneous and free, this 'idle heuristic' is not necessarily optimal. Consider the slightly odd situation where $i \cdot c_{job} < -c_{pow}$, i.e. the savings per unit time of having a single server powered down outweigh the penalties of having $i \geq 1$ jobs waiting. Then clearly the optimal policy is to only power up a server when there are more than i

jobs waiting to be served. Although this is a somewhat artificial situation, it does show how, even when the model is vastly simplified, finding the optimal policy is not completely trivial. In the next chapter we will discuss the optimal policy for a similar model at some length.

6.3.3 Threshold Heuristic

The Threshold Heuristic is a generalization of the Idle Heuristic. For this heuristic we choose some threshold, j_{thresh}. Servers are then powered down when there are less than j_{thresh} jobs waiting to be served. In terms of our model, this means we power down a server if $j < j_{thresh} + k_{up} + m_{up}$. Conversely we power up a server if $j > j_{thresh} + k_{up} + m_{up}$, i.e. if there are more than j_{thresh} jobs waiting to be served.

Choosing the right threshold j_{thresh} is not straightforward and should, in general, depend on both the differential between the holding cost and the power savings, and the switching times. The Idle Heuristic is equivalent to setting $j_{thresh} = 0$.

6.3.4 Semi-static Heuristic

In this heuristic we detect whether the arrival process is in the low or in the high state. Depending on what arrival state the system is in, the optimal number of servers is allocated, assuming the high/low period lasts an infinite amount of time, i.e. we allocate as many servers as would be optimal if the arrival behaves as a standard Poisson process.

Since there are N servers in the system, we can consider any distribution of n powered up servers, serving the queue, and $N - n$ powered down servers. The queue has a certain load $\rho = \frac{\lambda}{n\mu}$, where we use the appropriate λ_{high} or λ_{low} depending on whether we are in a high or low arrival phase. The formula for the mean response time, $\bar{R}$, for this $M/M/n$ queue is quite well known (see e.g. [Kle75]) and uses the famous Erlang C formula for the probability that all the servers in the queue are busy, which we will denote by Q here. The mean response time is:

$$\bar{R_n} = \frac{1}{\mu}[1 + \frac{Q}{n(1-\rho)}] \ . \tag{6.3}$$

So that we can easily find the n that minimizes:

$$c_{job}\bar{R_n} + c_{pow}(N - n) \ , \tag{6.4}$$

where we assume that the queue is stable, i.e. $\lambda < n\mu$ for the appropriate $\lambda_{high/low}$.

6.3.5 High/Low Heuristic

The High/Low Heuristic is a modified version of the On/Off heuristic introduced in the previous chapter, Section 5.2.2. It treats the queue of jobs as a deterministic fluid and assumes high/low periods last an infinite time. This means jobs arrive at rate

$$\gamma = \begin{cases} \lambda_{high} & \text{if } l = 1 \\ \lambda_{low} & \text{if } l = 0 \end{cases}, \tag{6.5}$$

where we recall that $l = 1$ denotes that the arrival stream is high and $l = 0$ denotes that it is low. Jobs are served at rate $k_{up}\mu$. We will use this approximation to calculate the cost of a decision until the queue empties. Since the system is (assumed to be) stable, this fluid approximation guarantees the queue will empty in finite time. We assume that switching servers complete their switches deterministically in the mean time indicated by the exponential distribution, i.e. when there are m_{up} servers being powered up, the first one completes its power up after $\frac{m_{up}}{\zeta_{up}}$ units of time, the second $\frac{m_{up}-1}{\zeta_{up}}$ units of time after that, etc.

Suppose there are k_{up} servers serving the queue, no servers currently being powered up or down and no powering up or down decision is made at this time. Then the queue decreases at constant rate $k_{up}\mu - \gamma$ and empties at time $\frac{j}{k_{up}\mu-\gamma}$. So the expected cost under these approximations can be shown to be the area of a triangle (see Figure 6.1, left) with as its length the time to empty the queue and as its height the queue size j at the start. More formally we obtain:

$$C_0 = \frac{c_{job}j^2}{2(k_{up}\mu - \gamma)} + \frac{c_{pow}k_{down}j}{k_{up}\mu - \gamma}\ . \tag{6.6}$$

The first part of Equation 6.6 represents the holding costs until the queue empties and the second part represents the power savings, again until the queue empties.

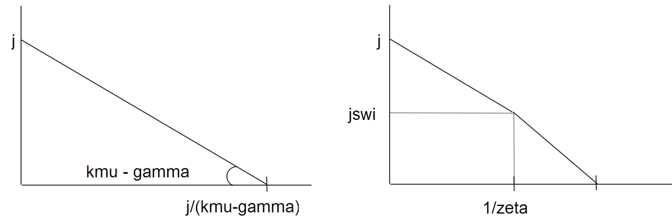

Figure 6.1: The triangle whose area represents the holding cost (left) and the cost when there is a switch present (right).

When there is already a server being powered up (see Figure 6.1, right), assuming that the queue

does not empty during the switch, its size at the point when the switch is completed, i.e. after $1/\zeta_{up}$, would be equal to j_{swi}, where

$$j_{swi} = j - (k_{up}\mu - \gamma)/\zeta_{up} \ . \tag{6.7}$$

This means that the total cost is:

$$\begin{aligned} C_1 =& \frac{c_{job} j_{swi}}{\zeta_{up}} + \frac{c_{job}(j - j_{swi})}{2\zeta_{up}} + \frac{c_{job} j_{swi}^2}{2((k_{up}+1)\mu - \gamma)} \\ &+ c_{pow} k_{down} (\frac{1}{\zeta_{up}} + \frac{j_{swi}}{(k_{up}+1)\mu - \gamma}) \end{aligned} \ . \tag{6.8}$$

Here the first two terms represent the cost of the queue until the switch is expected to be completed, the third the cost of the queue after that moment until the switch is completed and the last term represents the savings from powered down servers. Extending (6.8) to multiple switches is straightforward, although the formulae become increasingly convoluted. It should further be noted that when the *decision* is taken to power up i servers, (6.8) should be increased with the term iC_{up}.

Finally, we can consider the case where a server is already being powered down. The cost then becomes:

$$C_{-1} = \frac{c_{job} j^2}{2((k_{up}-1)\mu - \gamma)} + \frac{c_{pow} k_{down}}{\zeta_{down}} + c_{pow} k_{down} (\frac{j}{(k_{up}-1)\mu - \gamma} - \frac{1}{\zeta_{down}}) \ . \tag{6.9}$$

This too can be easily extended when multiple servers are powering down and should be increased with the term iC_{down} when it is a current decision to power i down servers, rather than an existing situation.

This heuristic now chooses the switching decision that minimizes the expected cost, at every state change. This calculation may seem prohibitively expensive; however note that we have to consider at most $N + 1$ possible decisions, assuming all our decisions will result in stable systems. Furthermore, it is entirely feasible to precalculate a table of decisions, or even to recalculate one on a very regular basis to deal with changing parameters.

6.3.6 Average Flow Heuristic

The average flow heuristic discussed here is the power modelling version of the one discussed in the Section 5.2.1. It is very similar to the High/Low heuristic. The entire analysis of the previous paragraph is applicable, with one change. For this heuristic we average out the high and low periods. This can be thought of as assuming that these periods are very short. This implies we can

use Equations 6.6, 6.8 and 6.9, but have to substitute:

$$\gamma = \frac{\lambda_{high}\eta + \lambda_{low}\xi}{\xi + \eta} \,. \tag{6.10}$$

The rest of the analysis is entirely the same. It should be noted that with this heuristic we can restrict ourselves to just considering 3 possible decisions: powering 1 server up, powering 1 server down or doing nothing. This is because there are no wholesale changes in state (like the arrival stream turning on or off) and we can thus expect the proposed switches to be much more modest.

6.4 Performance

In this section we will present some results. The heuristics in the previous section will be compared in performance under two different scenarios. We will also examine the effect of asymmetry in powering up and powering down times. And finally we will take a separate look at the performance of the threshold heuristic.

6.4.1 Increased Bursts

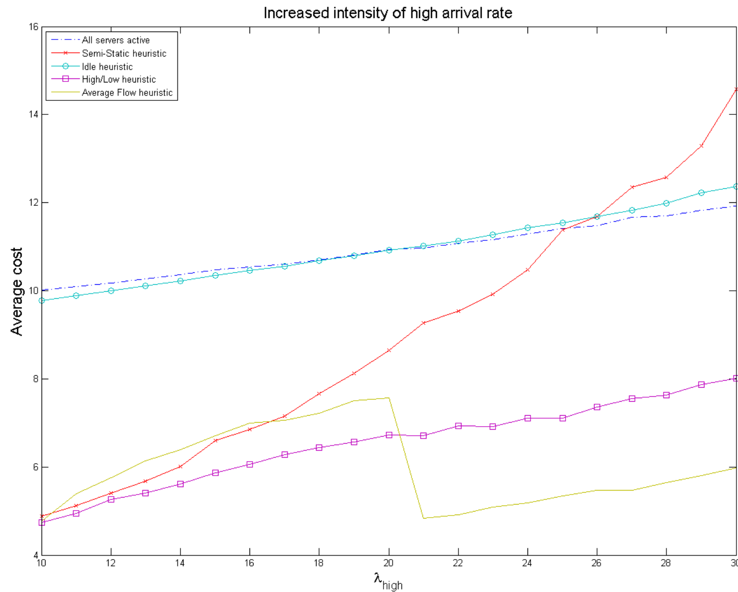

Figure 6.2: Increasingly more intensive arrivals in the high period. The x-axis shows the high arrival rate λ_{high} and the y-axis the mean cost.

For this experiment the system contains $N = 35$ servers, which process requests at a mean rate

of $\mu = 1$. The arrival rate in the low period is $\lambda_{low} = 10$ throughout the experiment, so that if all the servers are powered up, utilization is $\frac{10}{35} \approx 29\%$. The arrival rate in the high period is plotted on the x-axis and varies from $\lambda_{high} = 10$ to $\lambda_{high} = 30$. This means that if the system is a high-arrival period and all the servers are powered up, utilization varies from 29% up to 86%. Here the high arrival periods last a mean time of $\xi^{-1} = 10$ and the low arrival periods last a mean time of $\eta^{-1} = 100$. This means the highest *average* utilization is just 34%, but the peak demands mean this number is very misleading. Powering up or down is free, but takes $\zeta_{up}^{-1} = \zeta_{down}^{-1} = 1$, or the equivalent of one completion time in the mean. Finally we consider the holding cost of a job to be $c_{job} = 1$ and the benefit of powering down $c_{pow} = -0.5$ half that. These numbers are of course relative and it just signifies that having a job in the queue is twice as expensive as having a server powered up.

In Figure 6.2 we show the performance of the system under several heuristics. The costs were obtained from simulating the system for $T = 10000$ units of time and the displayed results form the averages from 50 runs. The 95 percentile of the relative error for each of these is small, typically within 5%, although the static allocations are a lot more susceptible to stochastic noise in the simulation and here the 95 percentile relative error can be higher.

The dash-dotted line represents the cost when all the servers are powered up all the time. This can be considered a baseline cost of sorts. As we can see, the idle heuristic, an obvious choice for a heuristic, does not manage to improve on this. The semi-static heuristic performs similarly almost as good as the completely dynamic heuristics at lower peak arrival rates, but its performance degrades steeply as the peak arrival rate increases.

The two fluid-approximation heuristics perform very well, even when the peak load is high. They seem to strike a good balance between the relatively high time needed to power up/down servers and the advantages of powering down servers when the system is quiet. There is a notable drop in average cost for the average flow heuristic from $\lambda_{high} = 20$ to $\lambda_{high} = 21$. In Figure 6.3 we can see this is related to the amount of switches the average flow heuristic makes. When $\lambda_{high} = 21$, the mean arrival rate γ increases to just over 11 and this seems to make the heuristic a lot less prone to switching.

6.4.2 Increasing Cost Differential

For this experiment we focus on the effect of the cost differential between the power costs and the holding costs. Please recall that the system is modeled in such a way that the (negative) power cost

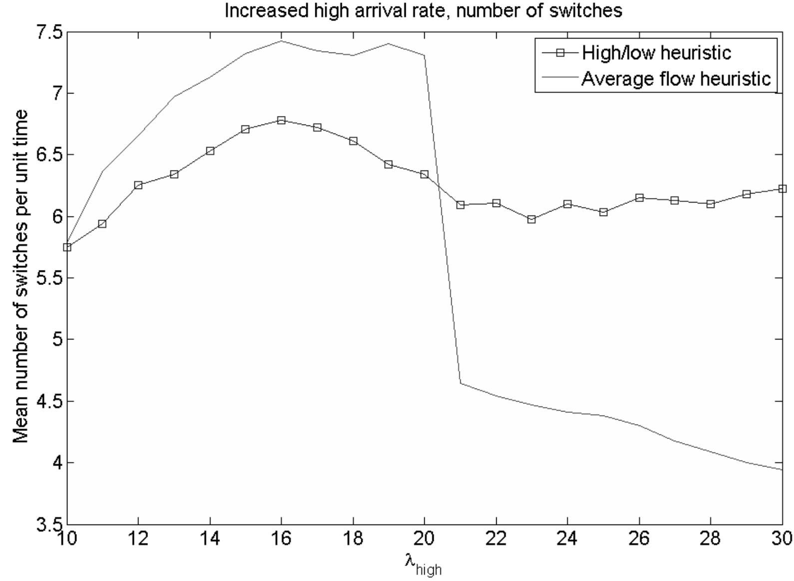

Figure 6.3: Increasingly more intensive arrivals in the high period. The x-axis shows the high arrival rate λ_{high} and the y-axis the mean number of switches per unit time.

c_{pow} of a powered down server is the difference between being (fully) powered up and powered down. This means we can add an arbitrary constant to any cost we find. This will explain the negative overall cost in the following results, since we have fixed the cost of the system when all the servers are permanently powered up at 0. We can do this, since this cost is obviously independent of the benefit we gain from any powered down server.

Here the low arrival rate is $\lambda_{low} = 10$ and the high arrival rate is $\lambda_{high} = 25$. Again, the mean duration of a high period is $\xi^{-1} = 10$ and the mean duration of a low period is $\eta^{-1} = 100$. Powering up or down is free but lasts $\zeta_{up}^{-1} = \zeta_{down}^{-1} = 1$ unit time, which is also the mean time for a job completion $\mu^{-1} = 1$, of one of the $N = 35$ available servers. Please recall that we have normalized time and therefore do not use a unit here.

In Figure 6.4 we show the cost for a relative power cost of $c_{pow} = -2$ up to $c_{pow} = -0.1$. For the latter case it makes almost no sense to ever power a server down, as the power savings will be minimal. But for the first case we will rather have 2 more jobs waiting in the queue than power up a server. Again the results are averages of 50 runs for $T = 10000$ units of time.

Here we see that the cost improvement we can get by using the Average Flow heuristic is large. The High/Low heuristic also significantly improves over having no servers powered down, especially when the cost differential between holding and power costs increases. Surprisingly the semi-static heuristic follow the High/Low heuristic very closely in this scenario. This would suggest that they

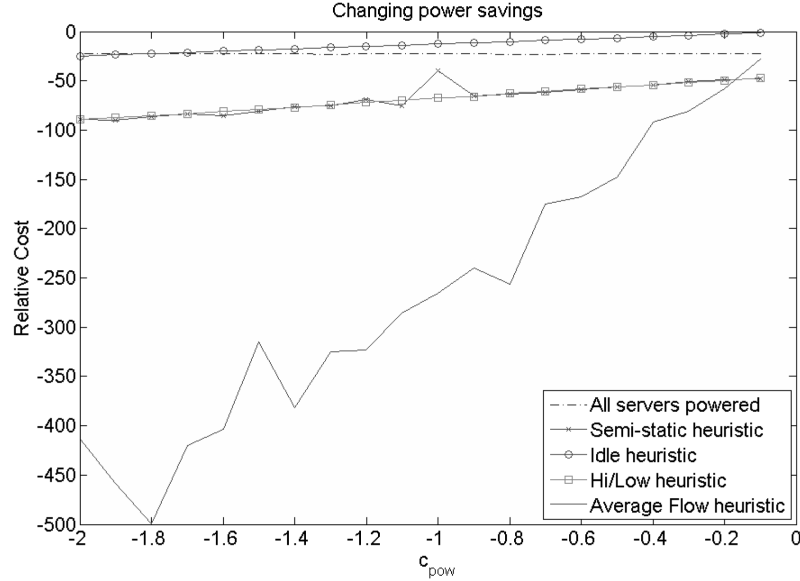

Figure 6.4: The effect of increasing cost differential between holding and power costs. The x-axis shows the (negative) power cost of a powered down server. The y-axis shows the cost savings relative to having all the servers powered up.

behave very similarly in this scenario. Indeed, Figure 6.5 seems to indicate this is true, at least for the amount of switching both heuristics do. Finally we note that the Idle heuristic continues to perform poorly.

6.4.3 Asymmetrical Switching Times

It can be noted that there is often a significant asymmetry between the time required to power up and that to power down. This depends on the mechanism used for this powering up and down, e.g. the time required to hibernate a normal desktop is much longer than the time needed to wake it from hibernation. In contrast, complete shutdown of a computer is often a lot quicker than boot up. The first asymmetry seems more attractive since we can then power up servers quickly when needed, whereas the powering down occurs when the system is under used. But we could also argue that the total time required to go through the cycle of powering up and down is the determining factor, since that determines the overall responsiveness of the system. In this subsection we show the results of an experiment where we vary the asymmetry between the powering up and down but not the total time required to power a server up and then down.

In Table 6.1 we see the result of differing asymmetry on the performance of three heuristics. The system under consideration here has $N = 35$ servers, which complete jobs at a rate $\mu = 1$ each.

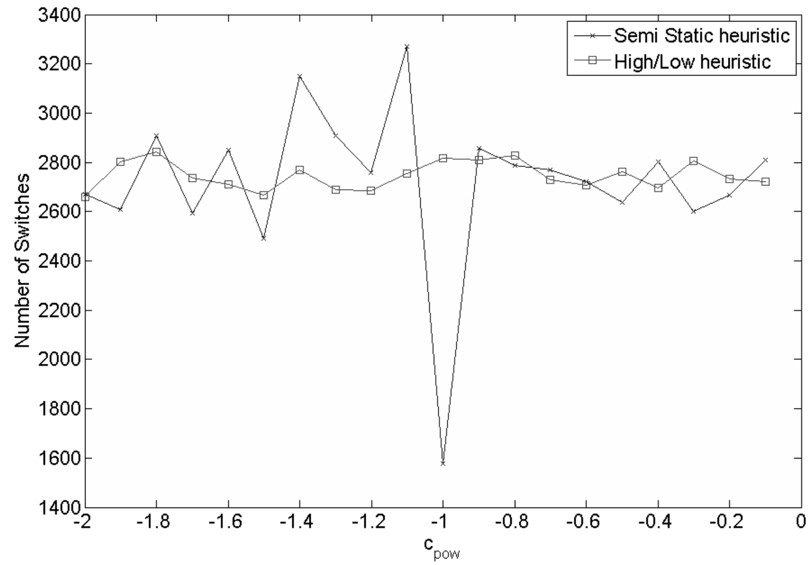

Figure 6.5: The number of switches made by the Semi Static and High/Low heuristics. The x-axis shows the (negative) power cost of a powered down server. The y-axis shows the number of switches.

	Slow up	Even	Fast up
Idle Heuristic	35.5	35.4	35.6
Average Flow Heuristic	25.9	25.8	25.7
High/Low Heuristic	26.5	26.8	26.4

Table 6.1: The impact of different asymmetry between powering up and powering down times on some heuristics.

High arrival periods last a mean time of $\xi^{-1} = 100$ and have an arrival rate of $\lambda_{high} = 30$. Low arrival periods last a mean time of $\eta^{-1} = 100$ and have an arrival rate of $\lambda_{low} = 20$. The holding cost for jobs is $c_{job} = 1$ and the negative powering down cost is $c_{pow} = -0.5$. Powering up or down is not free but costs $C_{up} = C_{down} = 0.5$. The total powering time is fixed at $\zeta_{up}^{-1} + \zeta_{down}^{-1} = 2$. But for the first column the powering up time $\zeta_{up}^{-1} = 1.5$ and $\zeta_{down}^{-1} = 0.5$, implying that we have slow powering up but quick powering down. For the second column both the powering up and down time is $\zeta_{up}^{-1} = 1 = \zeta_{down}^{-1}$, meaning both powering up and down take the same amount of time. Finally the third column has quick powering up, $\zeta_{up}^{-1} = 0.5$ but slower powering down $\zeta_{down}^{-1} = 1.5$. It is clear from Table 6.1 that the impact of the asymmetry in powering up and down times on the performance of the heuristics is negligible. This means it is the overall time it takes to complete a power up and power down cycle that matters, not just the time taken to power up.

6.4.4 The Threshold Policy

We now consider the performance of the previously described threshold heuristic. This heuristics need a parameter denoting the acceptable threshold. In Figure 6.6 we show the average cost of the threshold heuristic, given a queue length parameter from 0, i.e. it will behave as the idle heuristic, to 10, i.e. it will view a queue of at most 10 jobs as a sign to power down a server. On the other axis in the plane, the different λ_{high} are displayed, just as in the subsection 6.4.1. On the z-axis we have the average cost.

Here the cost is almost linearly increasing in the selected value of the threshold but with a minimum at 0. This implies that the threshold policy does no better than the Idle heuristic, which we now know to be poor. A similar result holds for the threshold policy under the experiment in 6.4.2. This means we can consider the threshold policy to be a poor choice for a heuristic.

6.5 Non-Poisson arrival processes

In this section we will have a very brief look at non-Markovian arrival processes and the performance of the heuristics under such conditions. The discussion will be brief since this is not really the main focus of this thesis. But at the same time it provides an interesting glimpse into how applicable our model for bursty arrivals is.

Throughout this thesis we have assumed that the arrival process is not just bursty, but bursty in such a way that the arrival process can be modeled as alternating on/off (or high/low) arrival

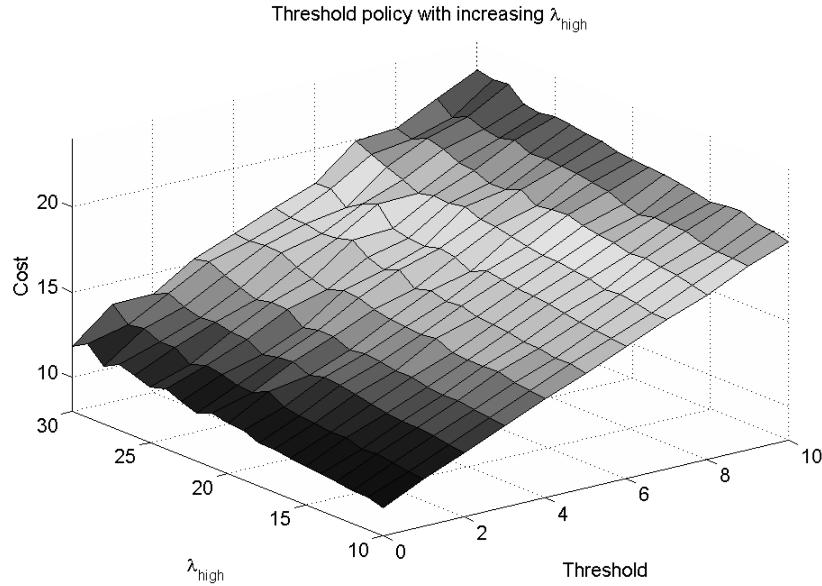

Figure 6.6: Increasingly more intensive arrivals in the high period. The x-axis shows the high arrival rate λ_{high} and the y-axis the selected threshold. The z-axis shows the mean cost per unit time.

periods. This can be considered the most basic approximation of bursty arrivals. Below we will consider two slightly more sophisticated models.

It is quite difficult to select the exact property we want to model when we discuss bursty arrivals. At least intuitively we consider the concept of burstiness related to heavy tailed-ness, self similar traffic and correlated arrivals. The interrupted Poisson model we have used so far has none of these properties. Its second moment is decidedly finite, there is no self similarity and, at least within an on/off (high/low) period, there is no correlation between arrivals.

6.5.1 Hyper-exponential Busy Times

As a first alternative arrival model we will consider the case where the second moment is significantly higher, albeit still finite. Truly heavy-tailed arrival processes are somewhat more difficult to simulate. The arrival process has two phases: a quiet phase and a busy phase. The duration of a quiet phase is exponentially distributed as before, with mean duration of $\eta^{-1} = 100$. During a quiet period, jobs will arrive as a Poisson process with rate $\lambda_{quiet} = 10$. During a busy period, jobs also arrive as a Poisson process but with rate from $\lambda_{high} = 33$. The duration of a busy period is now, however, not *exponentially* distributed, but rather *hyper-exponentially*. Meaning that with a certain probability α the busy period lasts ξ_1^{-1} and with a probability $1 - \alpha$ it lasts ξ_2^{-1}. Here we take the specific parameters such that most, from $\alpha = 0.9$ to $\alpha = 1 - 0.1 \cdot 0.9^9$, of the busy periods last an exponentially

distributed amount of time with a mean from $\xi_1^{-1} = 10$. We keep the total mean arrival rate the same, meaning that some of the busy periods last very long. The probability of such a long busy period goes down from $1 - \alpha = 0.1$ to $1 - \alpha \approx 0.04$. But the duration of such a busy period goes from $\xi_2^{-1} = 110$ to $\xi_2^{-1} \approx 268$.

The rest of the system is identical to that considered in the rest of the chapter, with $N = 35$ servers, a mean job completion time of $\mu^{-1} = 1$ and powered down servers give half the benefit compared to holding costs, i.e. $c_{pow} = -1/2$ and $c_{job} = 1$. Powering up/down costs $C_{up} = C_{down} = 1$ each way and takes an average of 1 job completion, $\zeta_{up} = \zeta_{down} = 1$.

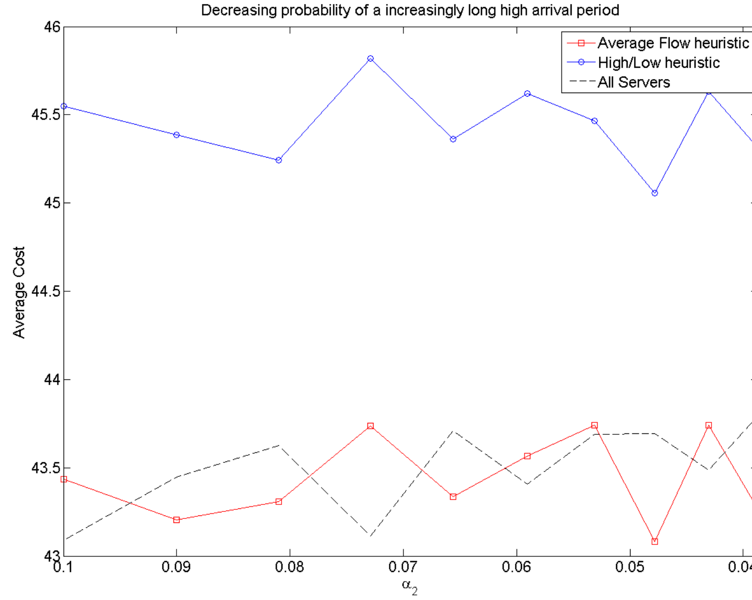

Figure 6.7: Decreasing probability of an increasingly long arrival period. On the x-axis, the probability α_2 of a long busy period. On the y-axis the average cost of the system.

In Figure 6.7 we have plotted the average cost, over 50 runs, of the system under the average flow and high/low heuristics. The cost of the system when all the servers are permanently left on is also plotted for comparison. It should be noted that the scale is quite fine grained which causes the erratic look of the line. The arrival of a very long busy period is quite rare, but the simulation time of $T = 10000$ per run should ensure they do occur.

The heuristics seems quite resilient to the exact nature of the occasional long busy period. However they do not outperform the system with all the servers on. This can be partially explained by the very heavy load of the system. Nonetheless it is quite disappointing that these heuristics perform so poorly even when the extremely long busy periods are relatively short. This seems to

suggest the extendability of these heuristics to non-Markovian arrivals with a heavy tail is very poor. We shall have another look at this in the next subsection.

6.5.2 Batch Arrivals

As a next experiment we introduce batch arrivals. In Figure 6.8, we have applied the heuristics considered in this chapter to a system with batch arrivals. There are $N = 35$ servers in the system, each with a job completion rate of $\mu = 1$. The holding cost for a job is $c_{job} = 1$, twice as much as the poer saving profit, $c_{pow} = -1/2$. Switching a server on or off takes roughly one job completion, $\zeta_{up} = \zeta_{down} = 1$ and costs $C_{up} = C_{down} = 1$ each way. Both high and low arrival periods last a mean of $\xi^{-1} = \eta^{-1} = 100$ units of time. During a low arrival period, $\lambda_{low} = 10$ jobs per second arrive. During a high arrival rate $\lambda_{high} = 10$ *batches* of jobs per second arrive. In Figure 6.8 we varied the batch size. The size of the batch is (uniformly) randomly chosen between 1 and the size displayed on the x-axis. So e.g. for the right most point, each batch had between 1 and 6 jobs, with a mean of 3.5, meaning the system is borderline unstable during a high period.

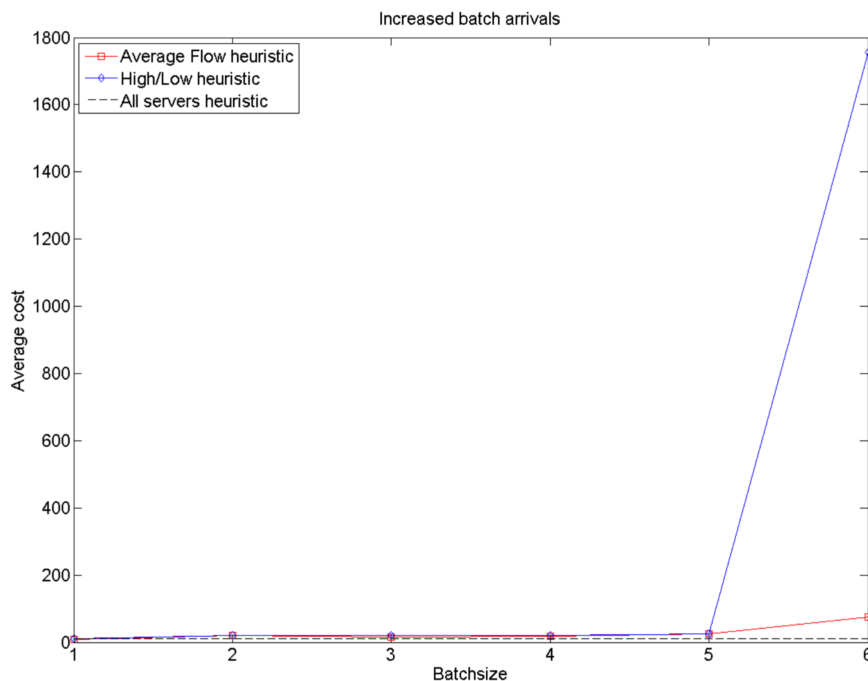

Figure 6.8: Increasing batch size arrivals. The x-axis shows the upper limit of the batch size. Each batch size is uniformly distributed between 1 and this upper limit.

The heuristics are slightly different from those described in Section 6.3. Instead of assuming a

mean arrival rate γ, both the High/Low and Average Flow heuristics were adapted to assume that during a high arrival period the mean arrival rate was:

$$\gamma_{batch} = \lambda_{high} \cdot \frac{\text{batch size} + 1}{2} \, . \tag{6.11}$$

Unfortunately the heuristics seem to cope very poorly with these batch arrivals even with this adaption. The performance of the heuristics for batch size 1, corresponding to the arrival process assumed in the rest of this thesis, is better than that of leaving all the servers on all the time. But for higher batch arrival sizes, the heuristics are outperformed by leaving all the servers on. In the case of the temporarily marginally unstable system with a possible batch size of 6, the heuristics perform spectacularly poorly. The explanation seems to be that even for relatively low batch sizes, powering down any servers is unwise since the system will be in a poor position to deal with the occasional incoming larger batches. Regardless of the explanation, the poor performance of the heuristics for non-Poisson arrival streams is again very pronounced and disappointing.

6.6 Summary

In this chapter we introduced a similar model to the one examined earlier. However, this time the trade off did not have to be made between performance of several job types, but rather between the performance of a single job type and possible energy savings by powering down servers. A model was formulated in Section 6.2 and similar policies to those examined before were introduced in Section 6.3. These policies were then tested in Section 6.4 by simulation under several different scenarios. These results were encouraging, but when we tried to apply two of the policies to a system with non-Markovian arrivals in Section 6.5 the results were disappointing.

Chapter 7

Breakdowns and Repairs

7.1 Introduction

In this chapter we introduce a final model, that is again related to the ones examined earlier. The focus of our model will be slightly different this time. The servers in question will be now be subject to breakdowns and subsequent repairs. This will add significant complexity to our model. To counteract this somewhat, we will assume the switching between job types is instantaneous. This greatly decreases the size of the state space under consideration. Throughout this chapter we will use the results from earlier chapters when appropriate and discuss the differences with this model.

The structure will be as follows. First we will describe the model formally in Section 7.2. Then we will show how to find the optimal static number of servers to allocate in Section 7.3. We will proceed by adapting the heuristics used in earlier chapters for this model in Section 7.4. The performance of these heuristics and the optimal static solution will then be compared in Section 7.5. And we will conclude this chapter with a summary in Section 7.6.

7.2 Model

The system under consideration in this model is somewhat similar to that described in Chapter 3. For some of the more technical issues of this model, we refer the reader to that chapter. In particular the formal translation of the continuous time Markov chain described in this section to a discrete time Markov process will not be addressed here. It can be done by the same uniformization process as described there. Similarly we will be brief in the translation from Markov process to Markov decision process. Details of the more formal translation are again essentially the same as those found in Chapter 3.

The model considered here has N homogeneous servers which supply service for M job types. Each server can work on one job at the time and likewise jobs can only be worked on by one server at a time. The time it takes to complete a job is exponentially distributed with rate μ_i, for a job of type i. The servers can be reassigned to serve a different job type at any time. Although we assume here that switching is instantaneous, it is not necessarily free. We will assume that there is a cost $C_{i,j}$ associated with reallocating a server from serving job type i to serving job type j. This means that, despite having instantaneous switching between job types, it still makes sense to talk about the amount of servers, denoted k_i, assigned to each job type $i = 0, 1, \cdots, M$.

The jobs in question arrive according to a two-phase Poisson process, i.e. there are 'high' and 'low' arrival periods. During a high period, denoted by $l_i = 1$, requests arrive as a Poisson process with rate $\lambda_{i,high}$. During a low period, notation $l_i = 0$, fewer requests arrive, with rate $\lambda_{i,low}$. This is the same as in the previous chapter, but for multiple job types. As before the, Poisson, high arrival periods end with rate ξ_i and the, also Poisson, low arrival periods end with rate η_i. The arrived jobs of type i join a (notional) queue j_i which they leave when service is completed. Each job has a holding cost c_i assigned to it, which reflects the relative importance of its type.

Finally we assume the servers break down occasionally and undergo subsequent repair. We model this by setting the operation period of each server to be independently distributed, lasting an exponentially distributed amount of time, with mean ζ_{down}^{-1}. Since we envision the servers switching between different job types on a regular basis, we assume the breakdown rate is independent of the current job type a server is assigned to. It would be very straightforward to change this. We denote the amount of broken down servers by k_{down}. As before, there is some redundancy in this notation since we must conserve the amount of servers in the system, i.e. $\sum_{i=1,\cdots,M} k_i + k_{down} = N$ throughout.

Repairs also take an exponentially distributed amount of time, with mean ζ_{up}^{-1}. We assume that once servers are repaired they can be freely assigned to any job type, without any additional cost. It is possible to assign a cost here, either depending on the new job type they are allocated to, or a blanket cost, independent of the new assignment. For that model we would have a pool of unassigned servers and would have to make a decision on whether to assign them, much like the powering up decision in the previous chapter. We will not examine this case in this chapter, meaning that, at least notionally, upon repair a server is always immediately assigned to a job type.

This description alone does not describe a Markov process since we have not specified whereto a repaired server is reassigned. Instead of finding a work around for this, we will introduce a relevant

Markov decision process immediately. To this aim we assume there is a policy $f(S)$, that defines where the next repaired server is assigned based on the current state. The instantaneous transition rate $r(S, S')$ (cf. Chapter 3, equation 3.4) can then be found as:

$$r(S,S') = \begin{cases} l_i\lambda_{i,high} & \text{if } \mathbf{j}' = \mathbf{j} + \mathbf{e_i} \\ (1-l_i)\lambda_{i,low} & \text{if } \mathbf{j}' = \mathbf{j} + \mathbf{e_i} \\ min(k_i, j_i)\mu_i & \text{if } \mathbf{j}' = \mathbf{j} - \mathbf{e_i} \\ l_i\xi_i & \text{if } l_i' = 0 \\ (1-l_i)\eta_i & \text{if } l_i' = 1 \\ k_i\zeta_{down} & \text{if } \mathbf{k}' = \mathbf{k} - \mathbf{e_i} \\ & \text{and } k_{down}' = k_{down} + 1 \\ k_{down}\zeta_{up} & \text{if } \mathbf{k}' = \mathbf{k} + \mathbf{e_{f(S)}} \\ & \text{and } k_{down}' = k_{down} - 1 \end{cases}, \tag{7.1}$$

where e_i denotes the ith unit vector as usual and $e_{f(S)}$ denotes the unit vector with the 1 for the job type policy $f(S)$ assigns the next repaired server to.

These transition rates relate to:

- job arrival in a high arrival period
- job arrival in a low arrival period
- job completion
- end of a high arrival period
- end of a low arrival period
- break down of a server
- repair of a server,

respectively.

If we have a policy $f(S)$ that reassigns operative servers based on a changing state, we use the technique outlined in Chapter 3 to incorporate these instantaneous transitions in the resulting Markov chain. For the formal details, including more on the required uniformization, we refer the

reader to that chapter. Informally we recall that the idea was to change the resulting transitions out of the current state as if the switch has already been made.

We will not be using this formal model explicitly in rest of this chapter. In the next Section 7.3 we will describe a slightly different relevant model. Even the simulation used to get the results in Section 7.5 will use a more convenient form of this model, not this formal definition.

Similarly we will be somewhat informal in our description of the cost function we are trying to minimize in this model. Notionally we are trying to minimize the weighted (by c_i) time jobs spend in the system. This cost is increased by a switching cost $C_{i,j}$ every time the decision to switch a server from type i to type j is made. Instead of formulating the formal cost model as in Section 3.2, we will outline a more informal, but perhaps more intuitive, cost function here.

Suppose we have a trace $\vec{S}$ of the system for a given (finite) series of events (state changes) and the time at which they occurred ($\vec{T}$), where S_0 is the initial state and T_i is the time that the system was in state S_i. We can then define the cost of that (timed) trace as follows. Set the cost of the trace, c_{trace}, initially at zero. Then we go through all events in the trace. For each interval i we increase the cost of the trace c_{trace} by the total holding cost in that epoch $\sum_{m=1,\cdots M} c_m j_m T_i$ and if a server is switched from serving job type i to type j also by $C_{i,j}$. This is how we calculate the cost in a simulation trace. The cost of a given policy can be thought of as the weighted (by probability) cost over all possible (infinitely long) traces. The formal translation is more problematic but at least intuitively it coincides with a formal cost similar to that found in Section 3.2.

7.3 Optimal Static Solution

In this section we will aim to find the optimal static allocation of servers in a system with bursty arrivals and breakdowns and repairs. This is very much parallel to what we did in Chapter 3, Section 3.3. Again we adapt the model in the previous section to only allow static allocation of servers, i.e. no switching between job types is possible. Once a server is allocated to a certain type, it will be allocated there indefinitely. We assume this includes allocation of repaired servers, i.e. they retain a 'memory' of which job type they were assigned to. The question now becomes: given that we have M job types with known parameters $\lambda_{i,high}$, $\lambda_{i,low}$, ξ_i, η_i, ζ_{down} and ζ_{up}, what is the best way to assign N servers over them? Here 'best' is defined as minimizing the overall average cost of the system.

The first thing to note is that this requires the number of servers to be at least equal to the number of job types, i.e. $N \geq M$, since otherwise the cost of the system is infinite. Secondly we

can note that this problem is easily solved if we can find the average queue length of any job type i, given that we assign n_i servers to it, so that we will suppress the use of the index i in the rest of this section. The arguments here are similar to those found in Section 3.3. Using the same solution technique as there, i.e. direct calculation, is somewhat problematic, however. The system under consideration is significantly more complex and the analysis would involve very complex calculus resulting in rather unattractive formulae. Instead we will use the technique of spectral expansion, first proposed for semi-infinite lattice strips in [MC95], where more details on this technique can be found.

For our problem to be amenable to the spectral expansion method, we have to model our system as a Quasi-Birth-and-Death process. On first sight this is somewhat problematic, as the state space of a single queue with both breakdowns and repairs and a high/low arrival process is most naturally modeled as a three dimensional lattice strip (of dimensions $\{0, 1, \cdots, N\} \times \{\text{high}, \text{low}\} \times \{0, 1, \cdots\}$), rather than the required two dimensional one. We can however relabel the states such that we have a two dimensional lattice strip $\{1, \cdots, 2N + 2\} \times \{0, 1, \cdots\}$, e.g. by labeling such that state i has low arrival rate if i is even and a high arrival rate if i is odd. And such that the amount of operational servers of the ith state is given as $\lfloor \frac{i-1}{2} \rfloor$. Under this labeling the first state corresponds to no operative servers and a high arrival rate, the second state has no operative servers but a low arrival rate, the third state has one operative server and a high arrival rate, etc.

With this model it is now possible to find the three families of matrices A_j, B_j and C_j needed by the spectral expansion method. For more details we refer the reader to [MC95]. We will note that the index j refers to a parameter relating to our position on the second (infinite dimensional) part of the lattice strip. Here this corresponds to the queue length. Matrix A_j is now the matrix whose i,k-th element corresponds to the transition rate from state (i, j) to state (k, j). In this model these are the state transitions corresponding to either busy/quiet period changes or servers breaking down and being repaired. Clearly this is not related to the amount of jobs in the queue in our model. As an example we can look at the simple case with just one server. Then the matrices A_j we find are:

$$A_j = A = \begin{pmatrix} 0 & \xi & \zeta_{up} & 0 \\ \eta & 0 & 0 & \zeta_{up} \\ \zeta_{down} & 0 & 0 & \xi \\ 0 & \zeta_{down} & \eta & 0 \end{pmatrix} . \tag{7.2}$$

In general we find that:

$$\begin{aligned} A_j = A = & \operatorname{diag}^{2N+2}_{(+1)}[\xi, 0, \xi, 0, \cdots, \xi] + \operatorname{diag}^{2N+2}_{(-1)}[\eta, 0, \eta, 0, \cdots, \eta] \\ & + \operatorname{diag}^{2N+2}_{(+2)}[N\zeta_{up}, N\zeta_{up}, (N-1)\zeta_{up}, (N-1)\zeta_{up}, \cdots, \zeta_{up}, \zeta_{up}] \ . \\ & + \operatorname{diag}^{2N+2}_{(+2)}[\zeta_{down}, \zeta_{down}, 2\zeta_{down}, 2\zeta_{down}, \cdots, N\zeta_{down}, N\zeta_{down}] \end{aligned} \tag{7.3}$$

Here $\operatorname{diag}^{j}_{(\pm i)}[\vec{v}]$ denotes the $j \times j$ matrix with $\vec{v}$ on the i-th super (sub) diagonal.

Similarly we can find the matrices B_j corresponding to the job arrival transitions. For the case with just one server we find:

$$B_j = B = \begin{pmatrix} \lambda_{high} & 0 & 0 & 0 \\ 0 & \lambda_{low} & 0 & 0 \\ 0 & 0 & \lambda_{high} & 0 \\ 0 & 0 & 0 & \lambda_{low} \end{pmatrix} . \tag{7.4}$$

And similarly the general form is:

$$B_j = B = \operatorname{diag}^{2N+2}[\lambda_{high}, \lambda_{low}, \cdots, \lambda_{high}, \lambda_{low}] \ . \tag{7.5}$$

The third family of matrices, C_j relates to job completions. Obviously this depends on the current queue length as the completion rate for jobs is proportionate to the minimum of operative servers and jobs in the queue. So we find, for one server, that:

$$C_0 = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} , \tag{7.6}$$

and

$$C_j = C = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & \mu & 0 \\ 0 & 0 & 0 & \mu \end{pmatrix} . \tag{7.7}$$

In general we find that for $j < N$:

$$C_j = \text{diag}^{2N+2}[0, 0, \min(j,1)\mu, \min(j,1)\mu, \cdots \min(j,N)\mu, \min(j,N)\mu] \ , \tag{7.8}$$

and for $j \geq N$:

$$C_j = C = \text{diag}^{2N+2}[0, 0, \mu, \mu, \cdots, N\mu, N\mu] \ . \tag{7.9}$$

Using these matrices we can find the exact expression for the probability vectors $\vec{v_j}$, where:

$$\vec{v_j} = (p_{1,j}, p_{2,j}, \cdots, p_{2N+2,j}) \ j = 0, 1, \cdots \ , \tag{7.10}$$

and $p_{i,j}$ denotes the probability of being in state (i, j) of the lattice strip, i.e. being in arrival/operative server state i with queue length j. In particular this will allow us to find the mean queue length L, i.e.

$$L = \sum_{j=0}^{\infty} [j \sum \vec{v_j}] \ . \tag{7.11}$$

The details of this process are rather technical and we gladly refer the reader to the original article [MC95].

It should be noted that the calculated mean queue length L is exact (up to numerical precision) and this calculation is extremely quick even for very large numbers of job types and servers. Using this queue length calculation we can find the optimal number of servers to allocate to each job type, simply by minimizing the average cost of the system $\sum_{i=1,\cdots,M} c_i L_i(n_i)$, subject to $\sum_{i=1,\cdots,M} n_i = N$. We can use either a brute force search for this (which will be sufficiently fast in most cases) or use the steepest descent algorithm outlined in Section 3.3 if we are optimizing very large systems.

In this chapter we have to also model another stochastic process, that of the breakdowns and repairs. We will this introduce two approximations of this. The first assumes each server that is currently operational will remain operational for ever. Likewise each broken down server will remain inoperational for ever. This means that if out of N servers, one breaks down the system is considered to have just $N - 1$ servers and treated as such for all decisions made after that point until another breakdown or repair occurs.

The second model of the breakdown and repair process would be to assume all servers have a capacity that is multiplied by their fraction of operational time. So if e.g. a server on average breaks down every 100 units of time and then, again in the mean, needs a 10 units of time to be repaired, it is considered to permanently have $100/110 \approx 0.91$ of its operational capacity. There are two

immediate problems with this approach. The first one is that it seems quite unrealistic. After all, it assumes the servers break down so often that this is a reasonable approximation, which is probably not acceptable in any real system. The second problem is that we also assume *inoperational* servers to have a capacity and indeed do not distinguish between operational and inoperational servers. This means it is quite problematic to assign servers since we can no longer get away with just assigning a random server. For significant repair times it will, after all, matter a great deal whether that server is currently operational or being repaired. Due to these problems we will not use this model.

The third obvious model would be to assume servers break down exactly after the mean amount of time and get repaired exactly after the mean amount of time. This seems more reasonable but again runs into the problem that we are assigning servers that are currently broken down. This seems an unproductive strategy since it is clearly more efficient to only assign a server once it is operational. After all, we have more information about the current system state. So a slightly different model seems in order. We do assume servers break down according to the mean rate, but we assume broken down servers remain inoperative. The problem with this model, however, is that for fairly heavily loaded systems, the approximation quickly turns unstable. And when the system is perceived to be unstable, it is impossible to make a good decision.

Consequently we can also discard this model for breakdowns and repairs, leaving us with only one appropriate model for breakdowns and repairs: until a breakdown occurs we assume that server will be operational indefinitely. Once it has occurred we will assume the server will not be operational again. When a server is repaired, we make a new allocation decision. Using this approximation we can easily adapt two previously considered fluid-approximation based heuristics.

7.4 Heuristics

In this section we will discuss some heuristic policies for dynamically allocating servers between job types. We will again use the ideas set out in the previous chapters, particularly Chapter 5. The heuristics used here are again fluid-approximation based. It can be noted that those heuristics, Average Flow and On/Off, were in essence only distinguished by their assumption on the duration of the arrival periods. The Average Flow heuristic assumed both on and off periods were infinitely short, whereas the On/Off heuristic assumed it lasted infinitely long. Both of them modeled the (stochastic) switching process in yet another way. There it was assumed that the mean time to switch a server over was the exact time it would take. Since we assume here switching is instantaneous, we do not have to model that process here.

7.4.1 Average Flow

As before we will consider an average flow heuristic. The details can be found in Chapter 5, subsection 5.2.1. The only difference to the heuristic described there is that we no longer have to consider delays when switching servers.

7.4.2 High/Low

The second heuristic will be a High/Low adaptation, as in Chapter 6, Subsection 6.3.5, of the On/Off heuristic described in Chapter 5, subsection 5.2.2.

7.5 Experiments

7.5.1 Introduction

In this section we will compare the performance of two heuristics outlined above, Average Flow and High/Low, to that of the optimal static solution, outlined in Section 7.3. Please recall that for the optimal static solution, the method used to find it, spectral expansion, already gives the mean queue length for each job type. We will use this analytical result to calculate the cost for this policy. For the two dynamic heuristics we will use simulation results.

7.5.2 Increasing Arrival Rate

In this first scenario we vary the high arrival rate of one of the job types, while leaving the rest of the system untouched. The system under consideration has $N = 50$ identical servers that provide service for $M = 2$ job types. Jobs of both types are completed with a mean of 1 per second, $\mu_1 = \mu_2 = 1$, and have a holding cost of $c_1 = 1$ and $c_2 = 2$ respectively. Both the high and low arrival periods for each job type are exponentially distributed with rate $\xi_1 = \xi_2 = \eta_1 = \eta_2 = 1/50$. During a low arrival period, the arrival rate for each job type is $\lambda_{low,1} = \lambda_{low,2} = 10$. The high arrival rate for job type 2 is set at $\lambda_{high,2} = 20$. We vary the high arrival rate for the first job type from $\lambda_{high,1} = 10.5$ to $\lambda_{high,1} = 20$. Servers of both types break down on average once every $\zeta_{down,1,2}^{-1} = 500$ units of time. It then takes an average of $\zeta_{up}^{-1} = 10$ units of time to repair them. While a server is being repaired, it cannot service any jobs. After repair it can be assigned to any job type without cost.

In Figure 7.1 the resulting average cost is displayed. For the two heuristics, the plotted line displays the mean of 50 runs for a duration of 10000 units of time each. The size of the confidence

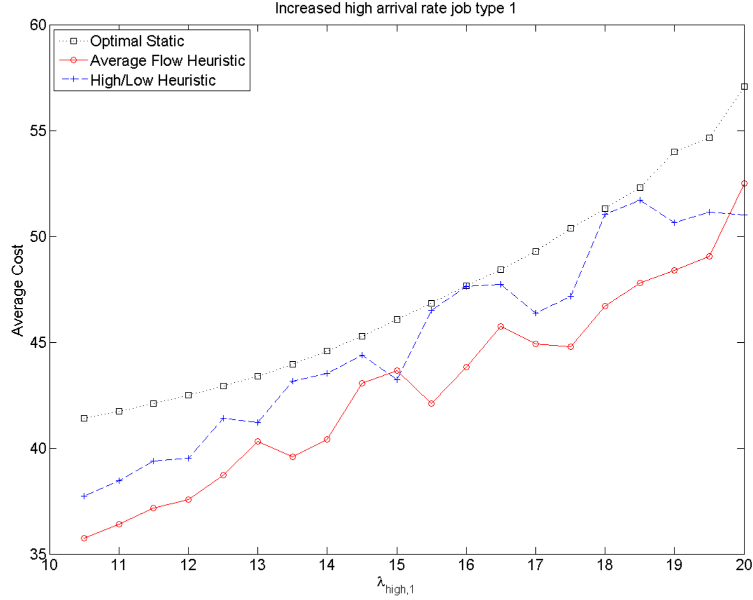

Figure 7.1: The influence of higher busy-period arrivals. Increasing high arrival rate for job type 1 on the x-axis. Mean cost on the y-axis.

interval is at most $\pm 5\%$ and typically well below that value. It should be noted that the analytically determined cost for the optimal static allocation seems to be less smooth in the last few points. This is probably due to numerical problems since MATLAB does warn of these for some of the calculations.

The behavior of all three policies is much as would be expected. The two dynamic heuristics outperform the static allocation and behave very similarly as in previous chapters, i.e. a more or less linear increase. The cost for the optimal static allocation, however, does not increase linearly but seems to grow more rapidly.

7.5.3 Increasing Breakdown Rate

In the previous experiment the breakdown rate was not that high: each server broke down once every 500 units of time, in the mean. For this experiment we increase this breakdown rate from $\zeta_{down,2} = 1/100$ to $\zeta_{down,2} = 1/5$ for job type 2. For job type 1 we keep it at a steady $\zeta_{down,1} = 1/500$. The other parameters are very similar to the ones used previously. There are $M = 2$ job types and $N = 50$ servers with a completion rate of $\mu_1 = \mu_2 = 1$ for both job types. Busy periods last a mean of $\xi_{1,2}^{-1} = 50$ units of time for each job type, during which jobs arrive with a rate of $\lambda_{high,1,2} = 20$. Likewise quiet periods last a mean of $\eta_{1,2}^{-1} = 50$ units of time for each job type, during which jobs

arrive with a rate of $\lambda_{low,1,2} = 10$. Servers are again repaired after a mean of $\zeta_{repair}^{-1} = 10$ units of time. Job type two is twice as expensive as job type one, $c_1 = 1$ and $c_2 = 2$.

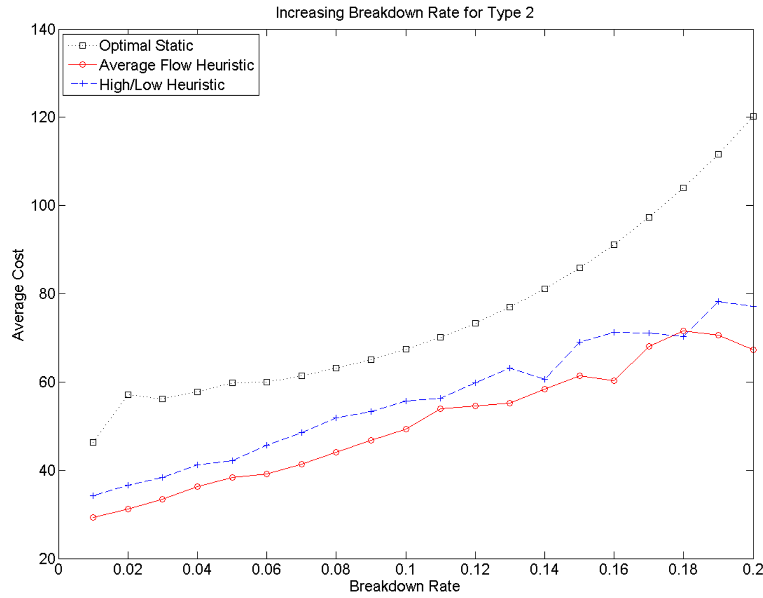

Figure 7.2: The influence of increasing breakdown rate. On the x-axis the breakdown rate for job type 2 is varied. The y-axis shows the average cost.

In Figure 7.2 we display the relationship between this increasing breakdown rate and the average cost of the system. It should be noted again that the costs for the heuristics are the mean values generated by simulation and that the 95% confidence interval is typically around ±2.5% but always below 5%. We can also note some further numerical inaccuracies in the analytically calculated cost for the optimal static allocation.

The trend for the cost is again somewhat predictable: higher breakdown rate means the cost increases. For the static allocation this has a more severe impact since it cannot dynamically reassign servers from job type 1 to handle these server shortages. As throughout this thesis, it also seems that the average flow heuristic outperforms the high/low heuristic.

7.5.4 Increasing Repair Time

As the penultimate experiment of this chapter and the final experiment aimed mainly at understanding the behavior of the optimal static allocation, we vary the repair time for servers rather than make breakdowns increasingly common.

For this we use parameters much like those before. There are $M = 2$ job types and $N = 50$

servers with a completion rate of $\mu_1 = \mu_2 = 1$ for both job types. Busy periods last a mean of $\xi_{1,2}^{-1} = 50$ units of time for each job type, during which jobs arrive with a rate of $\lambda_{high,1,2} = 20$. Likewise quiet periods last a mean of $\eta_{1,2}^{-1} = 50$ units of time for each job type, during which jobs arrive with a rate of $\lambda_{low,1,2} = 10$. Both server types break down every $\zeta_{down,1,2}^{-1} = 500$ units of time in the mean. The time it takes to repair a server is now varied from $\zeta_{up}^{-1} = 5$ to $\zeta_{up}^{-1} = 100$. Job type two is twice as expensive as job type one, $c_1 = 1$ and $c_2 = 2$.

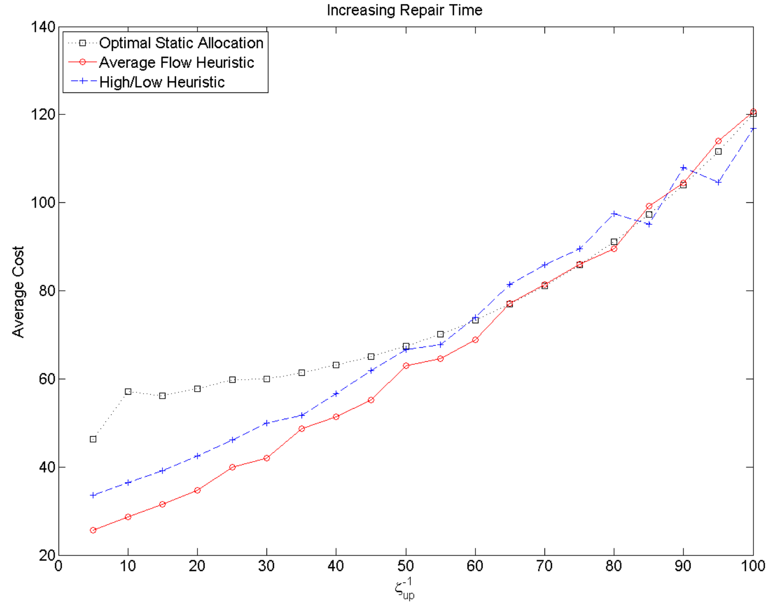

Figure 7.3: The influence of increasing repair time. On the x-axis the repair time for servers is increased. The y-axis shows the average cost.

Figure 7.3 shows the effect of increased repair time of the servers on the average cost of the system. We can note that both dynamic heuristics have very similar behavior: they significantly outperform the optimal static allocation for relatively low repair times. When the repair time grows significantly they rapidly approach the cost of the static allocation. This is probably because for high repair times there are occasional periods of time where the system is unstable. During these periods the dynamic heuristics are very poorly defined and more or less revert to static behavior. These periods also contribute very heavily to the average cost of the system.

7.5.5 Increased Switching Cost

Another scenario in which the dynamic policies converge on the static policy is that where the switching cost is increased. Please recall that throughout this chapter we assume instantaneous

switching. Also note that for the previous experiments switching was free. In this last experiment we will increase the switching cost.

The parameters used are again very much like before. There are $M = 2$ job types in the system again and the $N = 50$ servers have a completion rate of $\mu_1 = \mu_2 = 1$ for each job types. Busy periods last a mean of $\xi_{1,2}^{-1} = 50$ units of time for each job type, during which jobs arrive with a rate of $\lambda_{high,1,2} = 20$. Likewise quiet periods last a mean of $\eta_{1,2}^{-1} = 50$ units of time for each job type, during which jobs arrive with a rate of $\lambda_{low,1,2} = 10$. Both server types break down every $\zeta_{down,1,2}^{-1} = 500$ units of time in the mean. The time it takes to repair a server is fixed at $\zeta_{up}^{-1} = 10$. Job type two is twice as expensive as job type one, $c_1 = 1$ and $c_2 = 2$. We vary the switching cost from $C_{swi} = 0$ to $C_{swi} = 9.5$.

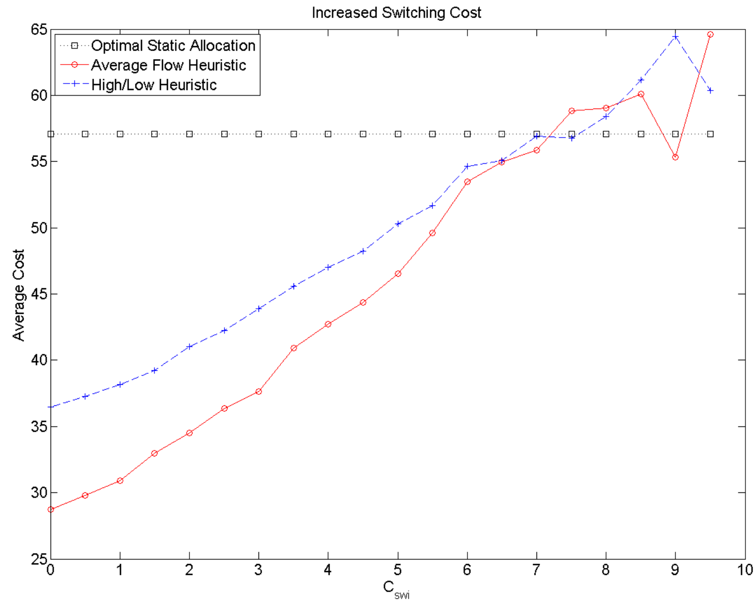

Figure 7.4: The influence of increasing switching cost. On the x-axis the switching cost is increased. The y-axis shows the average cost of the system.

As we can see from Figure 7.4, a high switching cost indeed makes the dynamic allocations ineffective. It can even make them worse than the optimal static allocation. This should correspond to the case where the heuristics overestimate the badness of the current queue size. Switching a server is a good decision in the short term but a poor one in the longer term since the cost to switch a server back is not taken into account.

7.6 Summary

In this chapter we introduced a model for a set of servers that experience breakdowns and repairs in Section 7.2. We also showed, in Section 7.3 how to use this model and the spectral expansion technique to calculate what the optimal static distribution of servers over multiple job types is. After that we introduced two fluid based heuristics for dynamic allocation in Section 7.4. The performance of these three allocation policies were then compared in several experiments in Section 7.5. Although the dynamic heuristics left scope for improvement, especially in the presence of large switching costs and repair times, they also provided a decent improvement over the optimal static allocation.

Chapter 8

Conclusions and Future Work

8.1 Conclusions

In this thesis we have considered three related models. The first, outlined in Chapter 3, concerns the allocation of servers between different job types when experiencing bursty arrivals. The system was formally modelled. The static optimal solution was found in Section 3.3 together with a quick, and under heavy traffic asymptotically exact, approximation method. We suggested a necessary and sufficient stability condition of the optimal dynamic solution in Section 3.4, and discussed how extensible this model is in Section 3.5.

We then turned our attention to finding the optimal dynamic solution in Chapter 4. Two solution methods were discussed at length in Sections 4.3 and 4.4, as well as two different optimization goals in Section 4.2. We compared the solutions found by different methods and optimization goals in Section 4.5, showing how problematic finding the optimal dynamic solution is, even for small systems. Possible speedup methods were also considered in Section 4.6 and shown to be insufficient.

This led us to introduce heuristics in Chapter 5. Two of these, see Section 5.2, were fluid-approximation based. Two others, discussed in Section 5.3, did not require any knowledge of the parameters of the system. Their performance was compared to that of the optimal dynamic solution for some small systems and found to be reasonably good. The heuristic policies were also compared to each other in Section 5.5. There we found that the two fluid based heuristics performed best, but that a parameterless one, the Queue Length heuristic, performed very well too.

A different model was considered in Chapter 6. Here the trade off is not between different job types, but between providing service for a single job type and reducing power consumption by powering down servers. Similar policies as before were introduced in Section 6.3 together with a couple of new ones. In Section 6.4 their performance was compared for several scenarios. The fluid

approximation based heuristics far outperformed the others, including the base case where servers are never powered down. A brief look in Section 6.5 at how these heuristics perform when the arrival process is not an interrupted Poisson process was, however, somewhat disappointing. The heuristics did poorly in that case.

In Chapter 7, we modeled a system of servers subject to breakdowns and repairs, again in the presence of bursty arrivals. We showed how to calculate the optimal static allocation of servers over job types for this scenario and adapted two fluid based heuristics from Chapter 5 for this scenario. The results of experiments for these policies found in Section 7.5 seem to suggest that although dynamic policies can improve performance of the system, these specific fluid based heuristics are not entirely well suited as they are outperformed by the static allocation in some cases.

Finally we have show that we can use this work as an inspiration for similar modeling of other systems. Application to e.g. component level modeling or more extensive power modeling in particular would be possible and very interesting.

8.2 Future Work

The scope for future work is rather large.

On a more theoretical front it would be interesting to formally prove the stability result of Section 3.4. On a more general level the connection between the systems considered here and polling systems can be examined at greater detail, especially if we assume switching is both free and instantaneous but the servers experience breakdowns and repairs as in Chapter 7. For that simplified case the results in polling systems are rather strong, making it tempting to connect the models considered here to those polling systems.

There are many possible extensions to the model described in Chapter 3. Most obvious perhaps is the addition of the features of Chapters 6 and 7, i.e. powering down servers and breakdowns and repairs. More complicated arrival and completion processes, especially those involving heavy tailed characteristics, would also be of interest. Although this could possibly make the model too complicated for detailed analysis.

It would also be of interest to consider a similar system, yet requiring entirely different modelling techniques, where the optimization goal is not driven by minimizing the average time a job spends in the system. Using percentile-based SLAs and stream based contracts would be particularly interesting. Especially in combination with admission control on the part of the provider, allowing rejection of jobs if the system is too heavily loaded or the offered reward is not attractive enough.

Further work on the heuristics would also be of interest. Here it would, however, be very useful to have a particular system with particular characteristics in mind. For a more specific system it might be possible to develop more sophisticated heuristics and verify them to some extend by comparing them for a few select cases to the optimal dynamic solution.

This holds in particular for the situation considered in Chapter 6. The power management problem seems to have a solid economical and ecological rationale. It can therefore serve as a very good concrete scenario for the future modelling extensions outlined above.

Likewise the results in Chapter 7 seem to suggest that the heuristics considered there can be improved on if server breakdowns and repairs are a major feature of the system.

Bibliography

[AAV98] O. Abu-Amsha and J.-M. Vincent, *An algorithm to bound functionals of Markov chains with large state space*, 4th INFORMS Conference on Telecommunications, Boca Raton, FL, 1998.

[BD98] O. Boxma and V. Dumas, *The busy period in the fluid queue*, SIGMETRICS (1998), 100–110.

[BR04] R. Bianchini and R. Rajamony, *Power and energy management for server systems*, Computer **37** (2004), no. 11, 68–74.

[BT89] D. Bertsekas and J. Tsitsiklis, *Parallel and distributed computation*, Prentice-Hall International, London, 1989.

[BVW85] C. Buyukkoc, P. Varaiya, and J. Walrand, *The $c\mu$ rule revisited*, Advances in Applied Probability **17** (1985), 237–238.

[CAT+01] J. Chase, D. Anderson, P. Thakar, A. Vahdat, and R. Doyle, *Managing energy and server resources in hosting centers*, ACM SIGOPS Operating Systems Review **35** (2001), no. 5, 103–116.

[CXHJ08] A.P. Chester, J.W.J. Xue, L. He, and S.A. Jarvis, *A system for dynamic server allocation in application server clusters*, UKPEW Proceedings, 2008, pp. 199–216.

[DO95] I. Duenyas and M.P. Van Oyen, *Stochastic scheduling of parallel queues with set-up costs*, Queueing Systems Theory and Application **19** (1995), 421–444.

[DO96] ———, *Heuristic scheduling of parallel heterogenous queues with set-ups*, Management Science **42** (1996), 814:829.

[dSeSG01] E. de Souza e Silva and H.R. Gail, *Performability modelling*, ch. The Uniformization Method in Performability Analysis, Wiley, 2001.

[FL96] S. Foss and G. Last, *Stability of polling systems with exhaustive service policies and state-dependent routing*, Annals of Applied Probability **6** (1996), no. 1, 116–137.

[FPY07] J.-M. Fourneau, N. Pekergin, and S. Younés, *Censoring Markov chains and stochastic bounds*, Formal Methods and Stochastic Models for Performance Evaluation, proceedings of EPEW 2007 (K. Wolter, ed.), Lecture Notes in Computing Science, vol. 4748, Springer-Verlag, 2007, pp. 213–227.

[GL96] G.H. Golub and C.F. Van Loan, *Matrix computations*, Johns Hopkins University Press Balitmore, 1996.

[HB07] M. Harchol-Balter, *New perspectives on scheduling*, Performance Evaluation Review **34** (2007), no. 4.

[How60] R.A. Howard, *Dynamic programming and Markov processes*, Wiley New York, 1960.

[HR87] M. Hofri and K.W. Ross, *On the optimal control of two queues with server set-up times and its analysis*, SIAM Journal of Computing **16** (1987), 399–420.

[Kle75] L. Kleinrock, *Queueing systems*, vol. Volume 1: Theory, John Wiley & Sons, 1975.

[KM81] P.J.B. King and I. Mitrani, *The effect of breakdowns on the performance of multiprocessor systems*, 8th International Conference on Modelling and Performance Evaluation (Performance '81) (F.J. Kylstra, ed.), North-Holland, November 1981.

[Koo97] G. Koole, *Assigning a single server to inhomogeneous queues with switching costs*, Theoretical Computing Science **182** (1997), 203–216.

[Koo98] ______, *Structural results for the control of queueing systems using even-based dynamic programming*, Queueing Systems Theory and Application **30** (1998), no. 323-339.

[KW90] O. Kella and W. Whitt, *Diffusion approximations for queues with server vacations*, Advances in Applied Probability **22** (1990), 706–729.

[LNT92] Z. Liu, P. Nain, and D. Towsley, *On optimal polling systems*, Queueing Systems Theory and Application **11** (1992), 59–83.

[LS90] H. Levy and M. Sidi, *Polling systems: Applications modelling and optimization*, IEEE Transactions on Communications **38** (1990), no. 10.

[LWW07] C. Lefurgy, X. Wang, and W. Ware, *Server-level power control*, International Conference on Autonomic Computing, IEEE, 2007.

[MC95] I. Mitrani and R. Chakka, *Spectral expension solution for a class of Markov models: Application and comparison with the matrix-geometric method*, Performance Evaluation **23** (1995), 241–260.

[MM91] I. Mitrani and D. Mitra, *A spectral expansion method for random walks on semi-infinite strips*, IMACS Symposium on Iterative Methods in Linear Algebra, Brussels, 1991.

[MM08] S.P. Martin and I. Mitrani, *Analysis of job transfer policies in systems with unreliable servers*, Annals of Operations Research **162** (2008), no. 1, 127–141.

[Neu81] M.F. Neuts, *Matrix geometric solutions in stochastic models*, John Hopkins Press, 1981.

[PBCH03] E. Pinheiro, R. Bianchini, E. Carrera, and T. Heath, *Compilers and operating systems for low power*, ch. Dynamic Cluster Reconfiguration for Power and Performance, pp. 75–91, Springer, 2003.

[PM05] J. Palmer and I. Mitrani, *Optimal server allocation in reconfigurable clusters with multiple job types*, Journal of Parallel and Distributed Computing **65** (2005), no. 10, 1204–1211.

[PM06] ______, *Empirical and analytical evaluation of systems with multiple unreliable servers*, 2006 International Conference on Dependable Systems and Networks (DSN 2006), 2006, pp. 517–525.

[RLIC06] P. Ranganathan, P. Leech, D. Irwin, and J. Chase, *Ensemble-level power management for dense blade servers*, SIGARCH Comput. Archit. News **34** (2006), no. 2, 66–77.

[Ros83] S.M. Ross, *Introduction to stochastic dynamic programming*, Academic Press, 1983.

[Sen67] E. Seneta, *Finite approximations to infinite non-negative matrices*, Proc. Camb. Phil. Soc **63** (1967), 983–992.

[SMT06] J. Slegers, I. Mitrani, and N. Thomas, *Server allocation in grid systems with on/off sources*, Proceedings of ISPA 2006 Workshops (Min et. al., ed.), Lecture Notes in Computing Science, vol. 4331, Springer, 2006, pp. 897–906.

[SMT07] ______, *Optimal dynamic server allocation in systems with on/off sources*, Formal Methods and Stochastic Models for Performance Evaluation, proceedings of EPEW 2007

(K. Wolter, ed.), Lecture Notes in Computing Science, vol. 4748, Springer-Verlag, 2007, pp. 186–200.

[SMT09] ______, *Static and dynamic server allocation in systems with on/off sources*, Annals of Operations Research (2009).

[STM08] J. Slegers, N. Thomas, and I. Mitrani, *Dynamic server allocation for power and performance*, Performance Evaluation: Metrics, Models and Benchmarks (S. Kounev, I. Gorton, and K. Sachs, eds.), Lecture Notes in Computing Science, vol. 5119, Springer-Verlag, 2008, pp. 247–261.

[Szy98] D.B. Szyld, *The mystery of asynchronous iterations convergence when the spectral radius is one*, Tech. Report TR 98-102, Department of Mathematics, Temple University, 1998, http://www.math.temple.edu/˜szyld/papers.html.

[Tij94] H.C. Tijms, *Stochastic models*, Wiley, 1994.

[TM95] N. Thomas and I. Mitrani, *Routing among different nodes where servers break down without losing jobs*, 1st IEEE International Computer Performance and Dependability Symposium (IPDS '95), IEEE Computer Society Press, 1995, pp. 246–255.

[Tru97] L. Truffet, *Near complete decomposability: Bounding the error by a stochastic comparison method*, Advances in Applied Probability **29** (1997), 830–855.

[WC58] H.C. White and L.S. Christie, *Queueing with preemptive priorities or with breakdown*, Operations Research **6** (1958), 75–95.

[Whi63] D.J. White, *Dynamic programming, Markov chains and the method of successive approximations*, Journal for Mathematical Analysis and Applications **6** (1963), 373–376.

Appendix A

Notation

In this appendix we will list some notation used in this thesis. The list is not exhaustive. We have strived to be consistent in notation and often the same notation denotes the same concept throughout this thesis. When a symbol has a (slightly) different meaning in different chapters, the relevant chapter is mentioned in brackets after the explanation. For example ζ *: Switching rate (3,4,5); powering down rate (6)* means that ζ denotes the switching rate in Chapters 3, 4 and 5 but the powering down rate in Chapter 6.

$c_f(S)$	:	Cost of state S given action f is taken, with other indices it can indicate other costs
g	:	Average cost
j_i	:	Queue length of job type i
k_i	:	Number of allocated servers of job type i
l_i	:	On/off state for job type i, where 0 denotes off and 1 denotes on
$m_{i,j}$	:	Number of servers switching from job type i to job type j
M	:	Number of job types
N	:	Number of servers
$q(S, S')$	:	Transition probability from state S to state S'
$r(S, S')$	:	Transition rate from state S to state S'
v	:	Length of time between two decisions
V_n	:	n-step cost
w	:	Length of interval in which parameters are observed

α	:	Discount factor
ζ	:	Switching rate (3,4,5); powering down rate (6); breakdown rate (7)
η	:	Rate with which off-periods end, i.e. inverse of mean off-time
λ	:	Job arrival rate
Λ	:	Uniformization constant
μ	:	Job completion rate
ξ	:	Rate with which on-periods end, i.e. inverse of mean on-time
ρ	:	Load