A New Toolbox for Scheduling Theory

Ziv Scully

CMU-CS-22-132
August 2022

Computer Science Department
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA

Thesis Committee

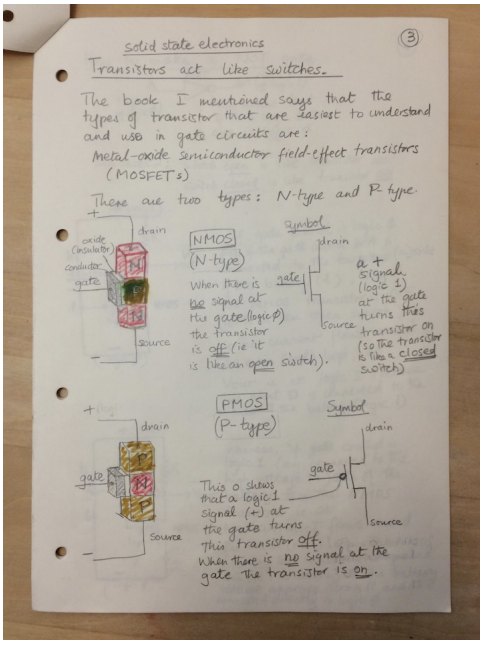
Mor Harchol-Balter, *co-chair*
Guy E. Blelloch, *co-chair*
Alan Scheller-Wolf
Anupam Gupta
Adam Wierman (Caltech)
Balaji Prabhakar (Stanford)

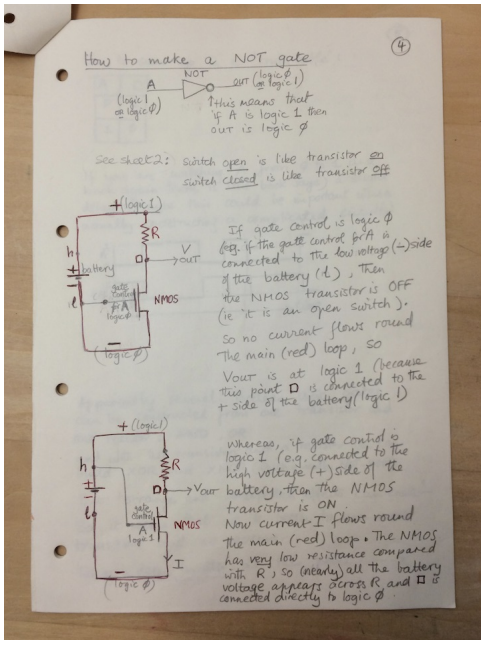
Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy.

This research was sponsored by the National Science Foundation under Grant Nos. CMMI-1334194, CMMI-1538204, CMMI-1938909, CSR-180341, CSR-1763701, and XPS-1629444; an NSF Graduate Fellowhip under Grant Nos. DGE-125222 and DGE-1745016; an ARCS Foundation scholarship; and the CMU SCS Bryant Fellowship. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of any sponsoring institution, the U.S. government, or any other entity.

Keywords: queueing theory; scheduling; response time; sojourn time; M/G/1; M/G/k; heavy tails; light tails; priority queues; Least Attained Service (LAS); Shortest Remaining Processing Time (SRPT); Shortest Expected Remaining Processing Time (SERPT); Gittins index policy; Multi-Level Processor Sharing (MLPS); limited priority levels; preemption checkpoints; job size estimates

To all of my teachers:
at school, at synagogue,
at camps math and jazz,
and most of all at home.





My grandmother Celia wrote these notes for middle-school me when I asked how to make logic gates out of transistors.

Acknowledgments

While writing a PhD thesis is a solitary endeavor, I am delighted to report that every other aspect of my PhD experience has been anything but. I have had a great time at CMU, for which I have a long list of colleagues, friends, and family to thank.

To begin, I would like to thank Mor Harchol-Balter, one of my advisors. When I first met Mor, I got the sense that this was the right person to train me to be the best researcher I could be. That turned out to be 100% correct. Mor has given me and her other students a veritable fire hose of advice on choosing research directions, writing, presenting, and academic career planning. She leads her group with an infectious enthusiasm. I will miss our group's weekly SQUALL (Scheduling and QUeueing At LLunch) seminars.

I would next like to thank Guy Blelloch, my other advisor. Guy has a very broad set of research interests and has built a group, which he guides with gentle but carefully considered advice, with the same breadth. It is thanks to this breadth that I had a brief foray into computer architecture, which was near the bottom of the list of topics I thought I might study during my PhD. Even though my core research ended up drifting in a pure queueing theory direction, which is outside even Guy's broad research umbrella, Guy has always been there with valuable advice at some of the most important decision points of my PhD and job search.

I am very grateful to the other members of my thesis committee. Alan Scheller-Wolf gave me some my first sips of the queueing theory Kool-Aid and has been a great mentor in research and beyond. Anupam Gupta helped me connect some of the ideas in this thesis to problems in (more traditional) theoretical computer science. Adam Wierman invited me to Caltech in 2018 for a very productive visit and has been dispensing research and career wisdom ever since. And Balaji Prabhakar, in addition to hosting me for a virtual visit to Stanford in 2021, has provided feedback, ideas for new directions, and encouragement throughout my thesis process.

There are a number of other faculty colleagues who I would like to thank. Michael Mitzenmacher and Sid Banerjee have been wonderful collaborators and bastions of support. Mark Squillante and Soumyadip Ghosh mentored me during an internship at IBM research, during which we attacked problems ambitious enough to appear in the future work section of this thesis's conclusion. Gauri Joshi, Weina Wang, and Ben Mosely were always willing to take time out of their busy schedules to meet with me. Onno Boxma and Jan-Pieter Dorsman have now hosted me in Eindhoven and Amsterdam twice, and having experienced a warm reception, stimulating conversations, and an enviable national rail network both times, I am eager to make it thrice.

I have been blessed to work with a number of my fellow students. Isaac Grosof deserves special thanks. Isaac brings their creativity and uncanny intuition every time we meet, whether over research, dinner, or a social deduction game. Bouncing ideas off of each other has led to some of my fondest research memories (to say nothing of some of our most important research breakthroughs), and I hope our future holds more bouncy ideas. I am

also grateful to Haotian Jiang, with whom I spent much of a happy summer swimming and discussing queueing problems, even though those problems turned out to be equivalent to famously difficult open questions. I hope all of my future collaborators are as amiable and insightful as Naama Ben-David, Lucas van Kreveld, Sahil Singla, Kunhe Yang, and Yige Hong. Another special thanks goes to Ben Berg. It was great to go through the PhD ordeal with such a knowledgeable and friendly academic twin. Ben, good luck as you start your new faculty position, and I am sure we will see each other around.

I have come to believe more and more over the course of my PhD that the most important output of CS research is not code or proofs, but rather the research's communication. There are countless folks with whom I have had great research conversations, from casual hallway banter to approaching breakthroughs while skipping a conference's business meeting. Thank you to everyone who has taken the time to engage.

No research at CMU would happen without the support of CMU's great staff. Nancy Conway, Pat Loring, and Ben Cook have helped me with the administrative side of CMU every step of the way. I would particularly like to thank Pat for helping me organize weekly lunches at SQUALL this past year. Charlotte Yano makes CSD's open house happen, and it's a highlight of every year. It was great working together to design the open house website, with the fun atmosphere of our meetings making up for the occasional stress of my amateur webmastery. And Deb Cavlovich, our CSD graduate program manager, has my and every other grad students' gratitude for serving double duty as both CSD's atlas and CSD's Atlas.

Outside of research, I have been fortunate to have met a number of kind and generous friends, a woefully inexhaustive set of whom I list here. Thanks to Charlie, Alex, and David for the many evenings we spent absorbed in Gloomhaven. Thanks to Bailey for conversations that are equally likely to have a laugh track, a genuine moment of insight, or both. I promise we will turn our class project into a paper some day. Thanks to Sol and Yvonne for, among many other things, spontaneously helping me move. Thanks to Daniel and Dhruv, with whom I interned at IBM Research, for keeping in touch, welcoming me to New York, and always-thoughtful conversations. Thanks to Ben Blum for a meaningful game of Spirit Island, sage advice, and foisting yummy things in my direction. For and out of the health and goodness of your and my hearts, respectively, I will always be happy to take home some brownies. Finally, thanks to all of my Avalon good buddies and spy buddies. It has been a pleasure being your Percival.

This last paragraph is for my family. Tal and Noam, thank you for all of the burritos, board games, and long talks we've shared. I feel so proud whenever I visit a space and am identified as "Tal's brother" or "Noam's brother". As for Mum and Dad, it says something about the way they raised Tal, Noam, and me that the thought of thanking them feels strange. They have supported the three of us to a degree that dwarfs attempts at thanks to the point of dissuading them, making it clear that no thanks are expected. As such, they will have to settle for the unexpected: thanks, Mum and Dad 😊.

This thesis was supported by beverages from Round Table Coffee under Product No. 04T-M1LK-L4TT3, and by tasty treats from Five Points Artisan Bakeshop under Product Nos. S33D3D-F0UG4553 and P3C4N-GR4N0L4.

Abstract

Queueing delays are ubiquitous in many domains, including computer systems, service systems, communication networks, supply chains, and transportation. Queueing and scheduling theory provide a rigorous basis for understanding how to reduce delays with scheduling, including evaluating policy performance and guiding policy design. Unfortunately, state-of-the-art theory fails to address many practical concerns. For example, scheduling theory seldom treats nontrivial preemption limitations, and there is very little theory for scheduling in multiserver queues.

We present two new, broadly applicable tools that greatly expand the reach of scheduling theory, using each to solve multiple open problems. The first tool, called "SOAP", is a new unifying theory of scheduling in single-server queues, specifically the M/G/1 model. SOAP characterizes the delay distribution of a broad space of policies, most of which have never been analyzed before. Such policies include the Gittins index policy, which minimizes mean delay in low-information settings, and many policies with preemption limitations. The second tool, called "WINE", is a new queueing identity that complements Little's law. WINE enables a new method of analyzing complex queueing systems by relating them to simpler systems. This results in the first delay bounds for SRPT (shortest remaining processing time) and the Gittins index policy in multiserver queues, specifically the M/G/k model.

Contents

List of Figures

Part I

Introduction

Chapter 1

Introduction

Queueing delays occur in any system where multiple entities contend for a shared but limited resource. The main way we measure the performance of such *queueing systems* is via the delay jobs experience, with lower delay generally being more desirable. Queueing systems occur in a plethora of domains, including computing, service operations, transportation, and healthcare. This thesis is in the field of *queueing theory*, which studies queueing systems in the abstract in an attempt to gain insights that apply across multiple domains. We call the entities contending for the limited resource *jobs*, and we suppose the resource they need is *service* provided by a *server*.

How can system designers reduce the queueing delay jobs experience? The most obvious approach is to acquire more or faster servers, but this may be costly. Another idea is to reduce the amount of service each job needs, e.g. by inventing a more efficient algorithm, but such innovation is not always possible. This thesis focuses on a third approach: *scheduling*, namely altering the strategy by which we allocate resources to clients. Scheduling is appealing in that it is virtually free, and it can be done with the resources and know-how one already has.

Smart scheduling can indeed significantly reduce queueing delays, but figuring out which scheduling policy is the "smart" one can be tricky. It is not always clear how changing a queueing system's scheduling policy will impact delays. Scheduling design thus requires guidance from analysis and evaluation of scheduling policies, whether theoretical or empirical. Experiments, both real-world and in simulation, can be very helpful for comparing a handful of policies. But the space of possible scheduling policies is vast. The only way to gain insight all at once into the entire space of policies, or at least large subsets thereof, is by developing the *theory of scheduling in queues*, the subject of this thesis.

Given the challenge and potential impact of smart scheduling, it should come as no surprise that queueing theorists have been studying scheduling for more than half a century. Why, then, do we need more scheduling theory? The issue is (and has always been, and will always be) that the scheduling theory we have does not adequately match scheduling practice. Some examples of areas where scheduling theory is lacking are the following:

- Scheduling under *uncertainty*, particularly regarding how much service a job needs.
- Scheduling with *multiple servers*.
- Scheduling with practical *preemption constraints*, meaning restrictions on the server's ability to switch from serving one job to serving another.

We would like to develop scheduling theory for these and other practical concerns. However, each of these concerns makes theoretically analyzing and optimizing scheduling policies, an already difficult endeavor, even more complicated.

This thesis contributes *two new queueing-theoretic tools*, which we apply to study scheduling with concerns like uncertainty, multiple servers, and preemption constrains.

The two tools are called *SOAP* and *WINE*. In addition to developing these tools, we apply them to prove numerous theorems that were previously intractable.

SOAP is a *unifying theory of single-server scheduling*. SOAP provides a universal analysis of a broad set of scheduling policies, thereby greatly expanding the set of policies we can analyze theoretically. We use SOAP to study scheduling under uncertainty and scheduling with practical preemption constraints. Chapter 3 gives an overview of SOAP and its applications, which together constitute Part II of this thesis.

WINE is a *new queueing identity* that synergizes with Little's law, another famous queueing identity [84]. WINE enables a method of analyzing complex systems by relating them to similar but much simpler systems. Most notably, we use WINE to analyze scheduling policies for multiserver systems, which are notoriously complicated, by relating them to analogous policies in single-server systems, which are much simpler. Chapter 4 gives an overview of WINE and its applications, which together constitute Part III of this thesis.

The rest of this chapter is structured as follows:

- (§ 1.1) We give some basic scheduling theory background, describing the scope of our work and the types of questions scheduling theory can answer.
- (§ 1.2) We discuss questions related to uncertainty, multiple servers, and preemption constraints that existing scheduling theory falls short of answering.
- (§ 1.3) We explain how SOAP and WINE enable us to answer many of these previously intractable questions.
- (§ 1.4) We give a chapter-by-chapter overview of the rest of the thesis.

1.1 What Is Scheduling Theory?

Scheduling is a vast field. This thesis is focused on scheduling in certain types of queueing systems, which we briefly describe below (§ 1.1.1). We then give a taste of the sorts of questions that one can answer with scheduling theory (§ 1.1.2).

1.1.1 Scope: Scheduling in Stochastic Queueing Models

In order to theoretically study queueing systems, we need to mathematically model them. There are a wide variety of possible modeling choices, reflecting the fact that there are a wide variety of queueing systems in practice. The queueing models we study in this thesis work in the following way:

- *Jobs* arrive to the system over time. We assume arrivals occur according to a *stochastic* process,[1] as opposed to considering worst-case arrival sequences.
- A *server* can serve one job at a time.
- A single central *queue* holds jobs that are waiting for service.
- Each job has an amount of work, called its *size*, to be done by the server.

[1] Specifically, we focus throughout on *M/G arrival processes*. See Chapter 5 for a full description of the queueing model.

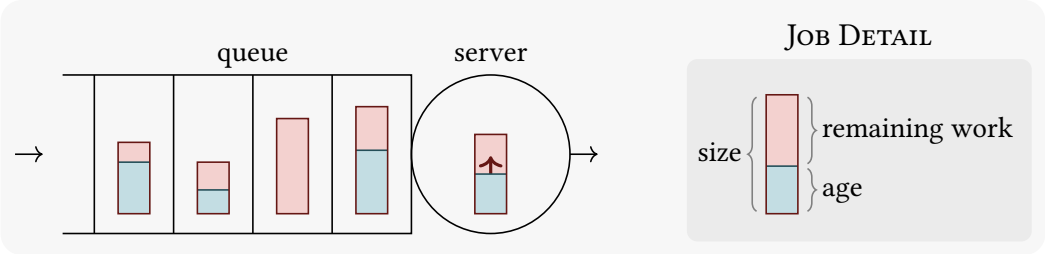


Figure 1.1. Single-server queueing model we use throughout this thesis. Jobs of varying sizes arrive over time according to a stochastic process. During service, a job's age increases and its remaining work decreases (red arrow). Once a job completes, meaning once its remaining work reaches zero, it exits the system.

- A server does work at a constant rate, so while a job is in service, its *remaining work* decreases at a constant rate. We can therefore think of work as being measured in units of time.

Figure 1.1 illustrates these concepts in the context of a single-server queueing system.

How do we evaluate the performance of a queueing model? The main metric we consider in this work is *response time*, a.k.a delay or latency. A job's response time is the amount of time the job spends in the system, meaning the amount of time between its arrival and the moment its remaining work reaches zero. We generally want jobs to have low response times.

Jobs arrive stochastically over time, with different jobs experiencing different response times. This means the system has a *response time distribution*, which we denote by T. We therefore evaluate performance using this response time distribution, looking at metrics like *mean response time* $\mathbf{E}[T]$ and the *response time tail* $\mathbf{P}[T > t]$ for various thresholds t.

There are many factors that affect a system's response time distribution. The factor we focus on in this thesis is the system's *scheduling policy*, which decides which jobs to serve at every moment in time. We consider other factors, like the number of servers or the statistics of the stochastic arrival process, to be fixed and out of our control.

1.1.2 Questions We Can Answer with Queueing-Theoretic Analysis

There is a single question at the core of much of the queueing-theoretic work on scheduling:

> How does a system's scheduling policy affect jobs' response times?

In this section, we give a taste of some specific instances of this question that scheduling theory can give insight into. For concreteness, we focus for now on a setting where

- the scheduler has knowledge of each job's size, and thus knowledge of each job's remaining work;
- there is a single server; and

- the scheduler may freely preempt jobs, meaning interrupt one job to start serving another, with no overhead.

Given that we generally want low response times, a natural objective is to minimize mean response time $\mathbf{E}[T]$. The scheduling policy that accomplishes this is *Shortest Remaining Processing Time (SRPT)*, the policy that always serves the job of least remaining work [116]. The intuition for SRPT's optimality for $\mathbf{E}[T]$ is that by working on the job of least remaining work, the next job completion happens as soon as possible, thus reducing the total amount of waiting that happens across all jobs.

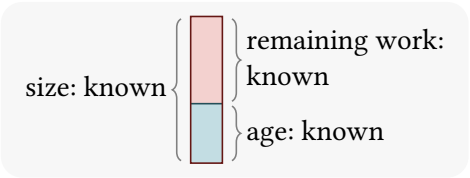
SRPT is appealing in that it minimizes mean response time $\mathbf{E}[T]$, but minimizing $\mathbf{E}[T]$ is far from the only concern system designers have in practice. There are several questions we might want to answer about SRPT before deploying it.

- There are policies that are simpler than SRPT, such as *First-Come, First-Served (FCFS)*, which is the default scheduling policy of most systems. Is the $\mathbf{E}[T]$ difference between SRPT and FCFS worth the trouble of implementing SRPT?
- In addition to improving the average job's experience, we would like to avoid jobs having especially large response times. That is, we want to make sure SRPT does not harm the tail $\mathbf{P}[T > t]$ for large thresholds t. How does SRPT perform in terms of response time tail?
- SRPT prioritizes small jobs, but this means large jobs might experience longer response times than they would under a policy that did not take into account job sizes. Is SRPT unfair to large jobs?

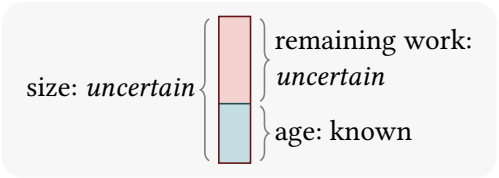
Fortunately, queueing theory can give us insight into each of these questions. The key ingredient is the *queueing-theoretic analysis* of SRPT, which was done by Schrage and Miller [117] in 1966. This analysis gives a detailed characterization of SRPT's response time distribution T in terms of the stochastic arrival process. While the analysis of SRPT does not by itself answer all of the questions above, it lays a theoretical foundation for studying them. Indeed, follow-up work on SRPT has continued into the 2000s, applying the analysis of Schrage and Miller [117] to better understand SRPT's mean response time [13, 14, 83, 146], response time tail [101, 103, 104], and fairness properties [15, 143–145].

1.2 Where Existing Scheduling Theory Falls Short

As we discuss in our review of prior work (Ch. 2), SRPT is far from the only policy that has been queueing-theoretically analyzed. Nevertheless, the set of scheduling policies we can analyze is in many ways limited, and several practical concerns remain outside the reach of existing scheduling theory. We review three such concerns below. For each, we ask multiple open questions, all of which we make progress on or solve in this thesis, as indicated by forward references.



(a) Job with known size, and thus known remaining work.



(b) Job with uncertain size, and thus uncertain remaining work.

Figure 1.2. Known vs. uncertain job sizes.

1.2.1 Scheduling under Uncertainty

SRPT requires knowing each job's exact size to implement. But it is often the case in practice that job sizes are *uncertain*, meaning we do not know how much service a job will need to complete, as illustrated in Figure 1.2. Uncertainty can mean having no size information at all, having good but imperfect estimates of each job's size, or something in between, like very noisy size estimates.

How should one schedule in light of job size uncertainty? This question has been studied, and there is a policy, called the *Gittins* policy, that is known to minimize mean response time $\mathbf{E}[T]$ in preemptive single-server queueing systems when job sizes are uncertain [44]. But, as discussed in Section 1.1.2, minimizing $\mathbf{E}[T]$ is not the only design goal one might have, and there are number of questions about Gittins that remain open.

- (Chs. 10 and 11) Gittins is a complicated policy that requires some potentially intensive computation. Is the $\mathbf{E}[T]$ reduction Gittins provides worth the implementation complexity, or do simpler alternatives suffice?
- (Ch. 16) One way we might simplify Gittins is to accept approximations in its underlying computations. Would such approximation significantly degrade $\mathbf{E}[T]$?
- (Ch. 13) How does Gittins perform in terms of response time tail $\mathbf{P}[T > t]$?

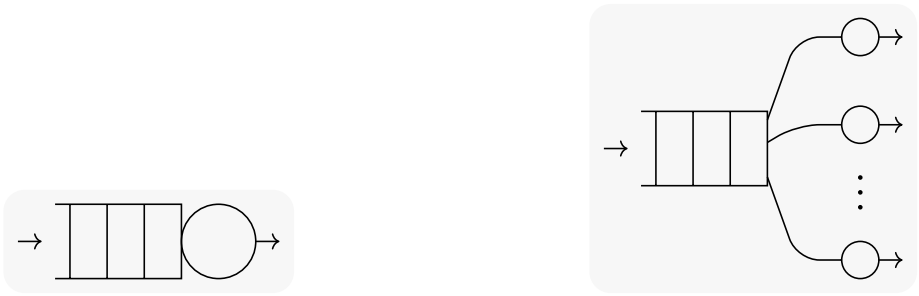
Unfortunately, unlike SRPT, Gittins has never been queueing-theoretically analyzed, making it difficult to study these questions.

1.2.2 Scheduling in Multiserver Systems

The vast majority of existing analyses of scheduling policies are for single-server queueing systems. But multiserver queueing systems are ubiquitous. For instance, in computing, multiserver systems occur at every scale, from the multiple CPU cores in a mobile device's to the thousands of machines in a data center.

How should we schedule in multiserver systems? Unfortunately, queueing theory has relatively little to say about multiserver systems. This is because even the very simplest scheduling policies, like FCFS, become intractable to exactly analyze in multiserver queueing models, like the one shown in Figure 1.3.

As an example of the sort of question we might hope to answer, suppose we want to minimize mean response time $\mathbf{E}[T]$ in a setting with known job sizes and unrestricted



(a) Single-server system. **(b)** Multiserver system.

Figure 1.3. Single-server vs. multiserver queueing systems.

preemption. In a single-server setting, SRPT is optimal. In a multiserver setting, it is known that SRPT is not perfectly optimal,[2] but beyond that, we know very little about SRPT's response time. Is SRPT nearly optimal for $\mathbf{E}[T]$ in multiserver systems, or is another policy much better (Ch. 17)? Unfortunately, unlike single-server SRPT, multiserver SRPT has never been queueing-theoretically analyzed.

We can of course ask an analogous question for the setting of uncertain job sizes. Is a multiserver version of Gittins nearly optimal for $\mathbf{E}[T]$ in multiserver systems (Ch. 17)? Given that there is no analysis of Gittins in single-server systems, let alone multiserver systems, scheduling theory is even further from answering this question.

1.2.3 Scheduling with Practical Preemption Constraints

Most queueing-theoretic analyses of scheduling policies take one of two extreme stances on preemption.

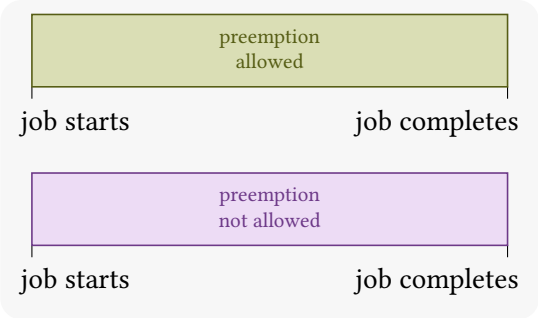
- Some work considers the *fully nonpreemptible* case, where once a job begins service, it must stay in service until it completes.
- Other work considers the *fully preemptible* case, where jobs may be interrupted at any time with no overhead or loss of work. For example, the single-server analysis of SRPT [117] considers this case.

But many systems in practice lie between these two extremes. It might be that jobs are only sometimes preemptible, as illustrated in Figure 1.4, or preemption may incur an overhead.

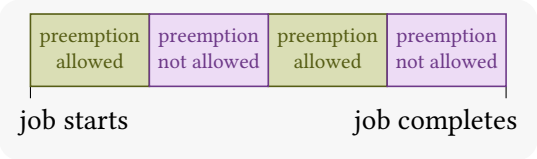
How should we schedule in light of practical preemption constraints? This is a very broad question, because there are many possible preemption practicalities. Some more concrete examples are the following:

- (Ch. 9) In many some settings, jobs can only preempted at certain *checkpoints*. An example is scheduling packet flows in a network, where it is undesirable to stop transmission in the middle of a packet. Frequent checkpoints permit flexible

[2] SRPT's suboptimality for $\mathbf{E}[T]$ in multiserver systems follows from prior work [81], but the details are somewhat subtle. We discuss this further in our review of prior work in Chapter 2.



(a) Simple preemption constraints: preemption is either always allowed or never allowed.



(b) Complex preemption constraints: preemption is sometimes but not always allowed.

Figure 1.4. Simple vs. complex preemption constraints.

preemption, but checkpoints may incur an overhead, e.g. a packet header. How do we balance the tradeoff between preemption flexibility and avoiding overhead?

- (Ch. 9) In network switches and other computing contexts, it is often the case that scheduling policies must be designed to work with only a limited number of priority levels [57, 96]. This precludes perfectly implementing SRPT, which uses a continuum of priority levels. How should we adapt SRPT to fit into a small number of priority levels? How many levels do we need for good mean response time?
- (Ch. 16) In the fully preemptible setting with uncertain job sizes, the Gittins policy is known to minimize mean response. Can we generalize Gittins and its optimality proof to settings with preemption constraints?

1.3 Two New Theoretical Tools: SOAP and WINE

The main contribution of this thesis is introducing two new queueing-theoretic tools, SOAP (§ 1.3.1) and WINE (§ 1.3.2), which we can use to at least begin to answer all of the questions mentioned throughout Section 1.2.

1.3.1 SOAP: Unifying Theory of Single-Server Scheduling

SOAP consists of two new contributions.

- (Ch. 6) *SOAP policies:* a newly identified broad class of scheduling policies.
- (Ch. 7) *SOAP analysis:* a universal formula that characterizes the response time distribution of a single-server system using any given SOAP policy.

SOAP stands for *Scheduled Ordered by Age-based Priority*, a phrase which briefly describes the class of SOAP policies.

SOAP policies include a significant fraction of policies that have previously been analyzed in single-server systems, such as FCFS and SRPT. But SOAP policies also includes an infinite array of policies that have never previously analyzed, such as Gittins. The SOAP

analysis thus unifies many previous queueing-theoretic analyses while also generalizing them far beyond what they could previously handle.

We use SOAP to answer numerous questions about scheduling in single-server systems. For instance, Gittins is a SOAP policy, so we can use the SOAP analysis as a basis for answer questions about Gittins. These include understanding Gittins's response time tail (Ch. 13) and determining when we can achieve near-optimal performance with simpler alternative policies (Chs. 10 and 11). For example, we show that in systems with noisy size estimates, if the noise is not too large, one can get near-optimal mean response time with very simple policies (Ch. 12).

SOAP policies also include many policies that operate under preemption limitations, so we can apply the SOAP analysis to answer questions about such policies (Ch. 9). These include policies that use only a limited number of priority levels and policies that only preempt jobs at certain checkpoints. For example, we characterize how the spacing between checkpoints can affect the response time tail (Ch. 13).

1.3.2 WINE: New Queueing Identity

WINE is a new queueing identity that comes in many "flavors". Its simplest flavor gives a formula for the number of jobs in a queueing system, which we denote by N. WINE expresses N in terms of a quantity which is related to *system work*, the total remaining work of all jobs in the system (Ch. 15). Crucially, WINE holds in any queueing system, whether single-server or multiserver. WINE stands for *Work Integral Number Equality*, a phrase which briefly describes the identity.

Why might we want to know the number of jobs N in a queueing system? In some systems, we might directly care about analyzing or optimizing metrics related to N, e.g. to ensure adequate buffer size. But even if we only care about response time T, understanding N still helps. Another queueing identity, Little's law [84], states that $\mathbf{E}[T]$ and $\mathbf{E}[N]$ are directly proportional, so minimizing $\mathbf{E}[T]$ and minimizing $\mathbf{E}[N]$ are equivalent goals.

WINE relates the number of jobs N, and therefore also mean response time $\mathbf{E}[T]$, to system work. Why is this a useful relationship? It turns out that this is a crucial step for analyzing multiserver systems. Determining $\mathbf{E}[T]$ for multiserver systems is very challenging, even under simple scheduling policies like FCFS [73, 82], and using more complex policies like SRPT and Gittins only increases the challenge. But it turns out that analyzing system work in multiserver systems is more tractable. We derive new bounds on system work (Ch. 8), which we use to give the first analyses of SRPT and Gittins in multiserver systems.

While our main motivation for developing WINE is analyzing multiserver systems, WINE also proves useful in single-server settings (Ch. 16). For example, we use WINE to show that approximately computed Gittins still has approximately optimal mean response time. This approximation result, in addition to being an interesting result in its own right, turns out to itself have multiple applications throughout this thesis (Chs. 12 and 13).

1.4 Organization of This Thesis

We give a chapter-by-chapter summary of this thesis below. If in doubt, we recommend that most readers start with Chapters 3 and 4, which give overviews of SOAP and WINE.

1.4.1 Parts I and IV: Motivation, Context, and Main Ideas

- Chapter 1, the chapter you are reading now, gives a high-level description of the problems this thesis aims to solve.
- Chapter 2 reviews prior work. It also explains the relationship between the work presented in this thesis and my prior publications on that work.
- Chapter 3 gives an overview of SOAP, explaining the problem SOAP solves, one of the key ideas behind how SOAP works, and the impact SOAP has in terms of questions it helps us answer.
- Chapter 4 gives an analogous overview of WINE.
- Chapter 18 concludes by summarizing the problems solved in this thesis and discussing future directions.

1.4.2 Part II: SOAP and Its Applications

- Chapter 5 describes the queueing model we work with throughout Part II.
- Chapter 6 defines the class of SOAP policies.
- Chapter 7 carries out the SOAP analysis. The end result is a generic formula that works for any SOAP policy.
- Chapter 9 numerically applies the SOAP analysis to answer questions about scheduling in systems with practical preemption limitations. These limitations include having a limited number of priority levels and being restricted to preempt only at certain checkpoints.
- Chapter 10 numerically applies the SOAP analysis to investigate scheduling under uncertainty. The specific question we focus on is whether we can achieve near-optimal mean response time with policies that are simpler than the theoretically optimal but complex Gittins policy.
- Chapter 11 theoretically applies the SOAP analysis to prove that a newly proposed policy achieves mean response time within a constant factor of optimal, despite being much simpler than Gittins.
- Chapter 12 theoretically applies the SOAP analysis to investigate how to design policies that are robust to job size estimation errors. The policies we propose can make use of high-quality job size estimates when they are available, and their performance degrades gracefully as estimate quality decreases.
- Chapter 13 theoretically applies the SOAP analysis to analyze the response time tail $\mathbf{P}[T > t]$ of SOAP policies for large thresholds t. One of our results is that Gittins, in addition to optimizing mean response time, sometimes also optimizes the response

time tail's asymptotic decay.

For brevity, while we give the full technical details for the development of SOAP itself, the chapters with theoretical applications summarize results and key ideas, with full proofs outsourced to my prior publications.

1.4.3 Part III: WINE and Its Applications

- Chapter 14 describes the queueing model we work with throughout Part III, which is somewhat more general than the model we use in Part II. We also generalize the Gittins policy to work in this more general model.
- Chapter 15 presents the WINE queueing identity.
- Chapter 16 applies WINE to solve problems in single-server scheduling. One of the main results is that Gittins's performance degrades gracefully if its underlying computations are approximated.
- Chapter 17 applies WINE to analyze SRPT and Gittins in multiserver systems, proving bounds on their mean response times. The bounds are tight enough to imply that both policies, which are optimal in single-server systems, are in some sense near-optimal in multiserver systems.

Chapter 2

Prior Work

The two tools introduced in this thesis, SOAP and WINE, build on an extensive literature of prior work in queueing theory and related fields. Most of this work fits into one of the following categories.

- (§ 2.1) Analyzing scheduling policies in the M/G/1. SOAP builds on this line of work, unifying and generalizing a significant portion of it.
- (§ 2.2) Defining and proving optimality of different versions of the *Gittins* policy, which optimally solves several scheduling problems in the M/G/1. Unifying and generalizing this line of work is one of the key steps in developing WINE.
- (Ch. 2) Work on central-queue multiserver systems, and in particular the M/G/k. Most of this assumes First-Come, First-Served (FCFS) scheduling, but there is some work on other scheduling policies. We apply WINE to contribute new results to this line of work.

There are also a number of other topics which are either related to individual chapters or are more tangentially related (§ 2.4). We conclude the chapter by clarifying the relationship between this thesis and the publications of mine that it builds upon (§ 2.5).

2.1 Scheduling in the M/G/1

The M/G/1 is one of the canonical single-server queueing models [67], and there is an extensive body of queueing theory literature that analyzes scheduling policies in the M/G/1. See Cox and Smith [28] and Conway et al. [27] for influential early treatments of the subject, and see Harchol-Balter [55, Part VII] for a modern overview.

This section begins by giving a history of analyzing scheduling policies in the M/G/1 (§ 2.1.1). We then review several ways these analyses have been applied (§§ 2.1.2–2.1.4). Finally, we describe how SOAP builds upon this prior work, pointing out some important precursors to its unifying analysis (§ 2.1.5).

A few of the results mentioned throughout apply to models more general than the M/G/1, but for simplicity of exposition, we discuss only their implications for the M/G/1.

2.1.1 A Brief History of M/G/1 Scheduling

Static Priority

The first analyses of the M/G/1 are the seminal works of Pollaczek [108, 109] and Khintchine [70] in the early 1930s. Both results are for *First-Come, First-Served (FCFS)*, which is one of the simplest scheduling policies.

From the perspective of this thesis, FCFS is a policy where all jobs always have the same static priority, with ties within a priority broken by arrival order. One can imagine other ways of breaking ties, which results in other scheduling policies. These include *Last-Come, First-Served (LCFS)* [134], *Random Order of Service (ROS)* [72, 134], *Preemptive Last-Come, First-Served (PLCFS)* [65, 100], and *Processor Sharing* [66, 74, 112, 153, 154], which were analyzed in the 1960s and 1970s. Some of these policies, particularly PS, are significantly more complicated to analyze than FCFS.

The 1950s and 1960s saw the first analyses of scheduling policies where different jobs have different priorities. This includes the *Nonpreemptive Priority (NP-Prio)* [68, 69, 135] and *Preemptive Priority (P-Prio)* [89, 135] policies, depending on whether a job is preempted if a new job of higher priority arrives during its service. While both NP-Prio and P-Prio assign different jobs different priorities, we can still view them as static priority policies, because a job's priority does not change after it arrives.

Dynamic Priority

The 1960s also saw the first analyses of scheduling policies with dynamic priorities, where a job's priority can change after it arrives. The two most notable examples are the following

- *Shortest Remaining Processing Time (SRPT)*, analyzed by Schrage and Miller [117] in 1966. SRPT preemptively serves the job of least remaining work. SRPT is significant in that it minimizes mean response time [116].
- *Least Attained Service (LAS)*, analyzed by Schrage [115] in 1967. LAS preemptively serves the job of least *age*, meaning the job that has been served the least so far. This can result in sharing the server equally between multiple jobs if there is a tie for least age.

Both of these policies have dynamic priority in that a job's priority changes during service. Under SRPT, a job's priority gets better as its age increases, while under LAS, a job's priority gets worst as its age increases.

A number of other policies where a job's priority varies during service have been analyzed since the analyses of SRPT and LAS [48, 49, 74, 75, 105, 146]. All of these are in the class of policies analyzed by SOAP, so we cover them in Section 2.1.5 when discussing precursors to SOAP.

The past decade has seen progress on analyzing scheduling policies with a different type of dynamic priority called *accumulating* priority, where a job's priority improves over time even if it is not in service. Stanford et al. [131] analyze *Nonpreemptive Accumulating Priority (NP-Acc-Prio)*, and Fajardo and Drekic [38] analyze *Preemptive Accumulating Priority (P-Acc-Prio)*. These developments are complementary to SOAP, as both policies fall outside the class of policies SOAP can analyze.

How Does Analyzing a Policy Help?

In the above discussion, analyzing a scheduling policy in the M/G/1 means characterizing its response time distribution, typically through a Laplace-Stieltjes transform. In many

cases, conditional response time distributions are also characterized, such as the response time distribution of jobs of a given size. However, these characterizations are often given in an implicit form, which makes them just the first step to gaining insight into scheduling design. In the following sections, we review several ways in which the analyses above have been applied to answer questions about scheduling design.

2.1.2 Heavy-Traffic Scaling of Mean Response Time

A famous fact in queueing theory is that the mean response time of an M/G/1 under FCFS increases as a function of the system's *load* (a.k.a. utilization), a parameter $\rho \in [0, 1)$ describing how busy the system is. One interpretation is that ρ is the fraction of time that the server is busy, which must be less than 1 to ensure stability. For a fixed job size distribution, FCFS's mean response time $\mathbf{E}[T_{\text{FCFS}}]$ scales as

$$\mathbf{E}[T_{\text{FCFS}}] = \Theta\left(\frac{1}{1-\rho}\right)$$

in the $\rho \to 1$ limit, which is known as the *heavy-traffic* regime. The same scaling applies to several other policies, such as LCFS, ROS, PLCFS, and PS.

The above state of affairs prompts a question: is it possible to schedule in a way that improves upon the $\Theta\left(\frac{1}{1-\rho}\right)$ scaling? A natural place to start is SRPT, which minimizes mean response time [116]. Bansal [13] gives the first result on the heavy-traffic scaling of SRPT, showing that for exponential job sizes,

$$\mathbf{E}[T_{\text{SRPT}}] = \Theta\left(\frac{1}{(1-\rho)\log\frac{1}{1-\rho}}\right) < o\left(\frac{1}{1-\rho}\right).$$

Further work shows similar results for general size distributions with infinite support [14, 83].

Heavy-traffic analysis of mean response time under other scheduling policies with dynamic priorities is relatively scarce. We are aware of only two instances: Kamphorst and Zwart [64] and predecessors [14, 102] characterize the heavy-traffic scaling of LAS, and Bansal et al. [16] characterize the heavy-traffic scaling of a policy called *Randomized Multi-Level Feedback (RMLF)* [17, 63].

We have used SOAP to obtain new heavy-traffic analyses in the M/G/1 [119], though we do not cover these results in this thesis.

2.1.3 Asymptotic Response Time Tail

The response time characterizations of Section 2.1.1 do not usually yield nice formulas for the response time tail $\mathbf{P}[T > t]$. As such, queueing theorists turn to a more tractable problem: analyzing the *asymptotic response time tail*, namely the behavior of $\mathbf{P}[T > t]$ in the $t \to \infty$ limit. We refer the reader to Boxma and Zwart [23] for a survey of this topic, reviewing just the main takeaways below.

Heavy-Tailed Size Distributions

When the job size distribution is heavy-tailed, meaning (roughly speaking) Pareto-like, Núñez-Queija [101] shows that SRPT, LAS, and PS are all *tail-optimal* [101, 103], meaning $\mathbf{P}[T > t]$ decays as quickly as possible in a big-Θ sense in the $t \to \infty$ limit. In contrast, FCFS is *tail-pessimal* [20, 25], meaning $\mathbf{P}[T > t]$ decays as slowly as possible.

We use SOAP to provide a simple sufficient condition which implies tail optimality in the heavy-tailed case, which greatly expands the set of policies that are known to be tail-optimal (Ch. 13). Most notably, we show that the Gittins policy satisfies the condition, so it is tail-optimal.

Light-Tailed Size Distributions

The situation for light-tailed job sizes is essentially the reverse of the heavy-tailed case: FCFS is tail-optimal [23, 133], while SRPT, LAS, and PS are all tail-pessimal [102, 103]. Wierman and Zwart [149] show that this is inevitable: policies can be tail-optimal for either heavy-tailed or light-tailed job size distributions, but not both.

Wierman and Zwart [149] also conjecture that FCFS is tail-optimal not just in a big-Θ sense, but that the leading constant is also optimal. However, Grosof et al. [53] show that the leading constant can be improved. The policy they use to do so, called *Nudge*, is not a SOAP policy, and our results suggest that no SOAP policy can match Nudge's leading constant (Ch. 13).

2.1.4 Fairness

It is known that SRPT minimizes mean response time [116]. However, because SRPT prioritizes small jobs, there is a concern that SRPT might treat large jobs unfairly. This gives rise to the study of several *fairness* metrics [15, 143–145], which attempt to quantify the degree to which a scheduling policy pays adequate attention to jobs of all sizes. One result is that for the purposes of mean response time, SRPT is, perhaps surprisingly, often as fair as PS [144], which due to its symmetry is a common benchmark for fairness.

One could in principle use the SOAP analysis as a basis for proving results about fairness, but we have not yet done so.

2.1.5 Precursors to SOAP

SOAP (Chs. 3, 6, and 7) greatly expands the set of scheduling policies we can analyze in the M/G/1. Roughly speaking, SOAP can analyze any policy where a job's priority varies during service, but stays the same as long as the job is not in service, a class we call *SOAP policies*. There are two ways in which SOAP is notable:

- The class of SOAP policies is very broad and includes many relatively complex scheduling policies.
- SOAP gives a *universal analysis* that applies to all SOAP policies at once.

With that said, SOAP is neither the first analysis of a relatively complex scheduling policy, nor is it the first universal analysis of an entire class of scheduling policies. This section reviews precursors to SOAP in both of these categories.

Analyzing Increasingly Complex Scheduling Policies

The first analyses of policies with dynamic priorities were for relatively simple policies, namely SRPT and LAS [115, 117]. These initial analyses use a technique called the *tagged job approach*, which analyzes response time by following a generic "tagged" job in its journey through the system.

Queueing theorists were quick to recognize that the tagged job approach could be used for a wide variety of scheduling policies, many more complex than SRPT and LAS. To name just a few, the tagged job approach has been used to analyze

- a version of SRPT with preemption checkpoints [48],
- versions of LAS with limited priority levels [86, 115],
- a version of P-Prio that uses SRPT within each class [49],
- policies that use noisy job size estimates instead of exact job sizes [36, 92], and
- special cases of the Gittins policy [105].

All of these analyses follow a similar strategy, though the details are different in each case. It turns out that these similarities are not coincidental: all of the above policies are SOAP policies. The SOAP analysis thus unifies all of the above analyses.

The MLPS Class

There are two significant classes of policies that have been analyzed prior to SOAP in a unified way. The first of these is the *Multi-Level Processor Sharing (MLPS)* class, introduced and analyzed by Kleinrock and Muntz [75]. MLPS consists of policies that combine FCFS, LAS, and PS in the following way. We partition $\mathbb{R}_{\geq 0}$ into a number of intervals called *levels*, with the ith level denoted by $[a_i, a_{i+1})$. Jobs are prioritized by the level their age is in, with lower levels having better priority. This means that much like LAS, a job's priority generally gets worse as its age increases. However, each level i uses one of FCFS, LAS, or PS to analyze jobs within that level. The analysis of MLPS is generic in the sense that with a single analysis, it gives the mean or Laplace-Stieltjes transform of response time in terms of the level boundaries a_i and the policy used in each level. See Kleinrock [74, § 4.7] for a comprehensive account.

SOAP contains all MLPS policies that use only FCFS or LAS within each level. However, the MLPS analysis of Kleinrock and Muntz [75] requires a restriction on the job size distribution when a level uses PS. As such, SOAP strictly generalizes the MLPS analysis for the case for fully general job size distributions.

The SMART Class

The second significant class of policies analyzed prior to SOAP in a unified way is the *SMAll Response Times (SMART)* class, introduced and analyzed by Wierman et al. [146]. Roughly speaking, SMART includes policies that, like SRPT, prioritize smaller jobs over larger jobs, with a job's priority possibly getting better during service. Among the results of Wierman et al. [146] is that all SMART policies have mean response time within a factor of 2 of SRPT.

SOAP contains only a subset of SMART policies. However, the SMART analysis of Wierman et al. [146] gives only bounds on response time. As such, for most of the subset of SMART that SOAP includes, SOAP gives the first exact response time analysis.

We note that Wierman and Nuyens [147] introduce and analyze an extension to the SMART class called *ε-SMART*, which includes policies that use inexact job size estimates. Unfortunately, the response time bounds obtained for ε-SMART are significantly looser than the corresponding bounds for SMART. SOAP can be used to compute exact response time results for many ε-SMART policies.

2.2 The Gittins Policy in Queues

The *Gittins* policy [44], named after its principal inventor [45], is a policy that minimizes mean response time in the M/G/1 queue when job sizes are unknown. Actually, Gittins is somewhat more general than this: for a wide variety of M/G/1 models, Gittins minimizes the mean total *holding cost* of jobs in the system, where each job may have a different holding cost. This is because Gittins is not really a single policy but rather a policy construction: given a stochastic model of what the scheduler knows about each job and a holding cost function, we can construct a version of Gittins that minimizes mean holding cost. By Little's law [84], minimizing mean response time corresponds to the case where all jobs have the same constant holding cost.

We note that versions of the Gittins policy have also been used extensively outside of queueing. See Gittins et al. [44] for a recent treatment of the topic.

2.2.1 Analyzing Gittins's Response Time

There has been some work on characterizing properties of Gittins [3, 4], but aside from some special cases [105, 141], the actual mean response time or mean holding cost achieved by Gittins is not known, let alone other properties of Gittins's response time distribution.

Fortunately, Gittins is a SOAP policy, so we are able to use SOAP to obtain the first full analysis of Gittins's response time distribution, which plays a role in many of our results (Chs. 10, 11, and 13).

2.2.2 Proofs of Gittins's Optimality

Given that Gittins is more of a policy construction than a single policy, it should come as no surprise that there are many proofs of Gittins's optimality in the M/G/1 [18, 28, 29, 43, 76, 79, 116, 128, 128, 136, 141]. Each proof makes different assumptions on what information the scheduler knows about each job, what structure the job size distribution has, when preemption is allowed, and when a job's holding cost can change. Many of the proofs also have technical limitations which are seldom acknowledged in the literature. See Scully and Harchol-Balter [122, Section II] for a detailed review of these prior proofs.

We use WINE to give a unified account of Gittins's optimality in the M/G/1, unifying and generalizing the prior proofs cited above (Ch. 16). Moreover, our approach naturally extends to give performance bounds on variations of Gittins where a job's priority is computed only approximately. Key to our approach is a very general job model that can capture many types of uncertainty the scheduler might have about a job's remaining work (Ch. 14).

2.2.3 Precursors to WINE

WINE (Chs. 4 and 15) is a queueing identity that gives a formula for the number of jobs in the system, or more generally for the total holding cost of jobs in the system. WINE is intimately tied to the Gittins policy. Roughly speaking, each version of Gittins gives rise to a new version of WINE.

Just as prior proofs of Gittins's optimality are special cases of our proof (Ch. 16), there have a number of previously introduced identities that are special cases of WINE. These include results of Glazebrook and Niño-Mora [47, Lem. 3] and Glazebrook [46, Thm. 1(a)], who, as we discuss further in Section 2.3.2, analyze versions of Gittins in multiserver queues. In addition to our presentation of WINE being more general, we believe our statement and proof is also easier to understand. With that said, one advantage of the presentations of Glazebrook and Niño-Mora [47] and Glazebrook [46] is that they make clear the connection between WINE and a previous technique called the *achievable region method* [18, 29], which lurks behind our presentation without making itself fully visible.

There are other identities that are similar to WINE, though not special cases of it, that have been used to analyze scheduling policies. These include results of Righter et al. [111, Lem. (3.12)] and Banerjee et al. [12, Lem. 13], the latter of which has a common special case with WINE but generalizes it in a different direction.

2.3 Scheduling in Multiserver Systems

Multiserver systems like the M/G/k have posed a significantly greater challenge to queueing theory than single-server systems like the M/G/1. For example, Kingman [73] highlights analyzing the M/G/k as a challenge to revisit as queueing theory enters its second century.

2.3.1 Approximate Analyses of the M/G/k under FCFS

While much progress towards understanding the M/G/k has been made in queueing theory's first century, the vast majority of it is for FCFS scheduling. This progress has come partly in the form of response time bounds, for which Li and Goldberg [82] provide both both an excellent overview and a remarkable recent example. Other work on the M/G/k includes determining when response time moments are finite [113, 114, 138], diffusion approximations [71, 139, 152], and heavy-traffic analyses [77, 78].

While we prove results for SRPT and Gittins in the M/G/k, we have relatively little to say about the M/G/k under FCFS, except in the few special cases where Gittins reduces to FCFS. This suggests that using policies designed to lower mean response time might actually make the M/G/k *easier* to analyze.

2.3.2 Scheduling in the M/G/k and Similar Models

There has been some progress on analyzing scheduling policies in the M/G/k, but it is limited to relatively simple policies like P-Prio [56, 91, 130]. Moreover, these results assume phase-type size distributions.

We use WINE to provide the first analyses of SRPT and Gittins in the M/G/k, without any assumptions on the size distribution (Ch. 17). There is some prior work on SRPT and Gittins in multiserver settings, which we review below.

Multiserver SRPT

SRPT has been studied in central-queue multiserver systems with *adversarial* arrival processes, as opposed to the stochastic arrival processes of traditional queueing models like the M/G/k. Specifically, it is known that even though SRPT minimizes mean response time in single-server systems with adversarial arrivals [116], SRPT becomes suboptimal with multiple servers. Specifically, Leonardi and Raz [81] show that the competitive ratio of SRPT. Nevertheless, Leonardi and Raz [81] also show that SRPT is in some sense the best one can do in multiserver systems.

One can show that even with the stochastic arrivals of the M/G/k, SRPT is still suboptimal.[1] However, the existing bounds on SRPT for adversarial arrivals [81] are too loose to give a good sense of whether SRPT is close to optimal. We resolve this question by showing that SRPT is indeed close to optimal in the M/G/k (Ch. 17).

Concurrently with the work that led to this thesis, Dong and Ibrahim [34] have analyzed the heavy-traffic behavior of SRPT in a setting with customer abandonment. This work is complementary to ours. For instance, they consider a steady-state overloaded regime, which exists due to abandonment, while we only consider systems with load less than the service capacity.

[1] Personal communication with Isaac Grosof, August 2022. Roughly speaking, even though arrivals are stochastic in the M/G/k, there is a positive probability of arrival sequences similar to those Leonardi and Raz [81] construct to show SRPT's suboptimality under adversarial arrivals.

Multiserver Gittins

Both Glazebrook and Niño-Mora [47] and Glazebrook [46] analyze particular cases of the Gittins policy in the M/G/k, bounding its mean response time or, more generally, mean holding cost. However, both results make restrictive assumptions on the job size distributions, and both consider cases where, roughly speaking, jobs are always in one of finitely many states.[2] The means that there is a "worst state" in which jobs have their maximum possible expected remaining work, and this maximum possible expected remaining work plays a role in the bound. But there are plenty of scenarios where there is no upper bound on jobs' expected remaining size, such as scheduling with unknown sizes under a heavy-tailed size distribution. Our work on Gittins in the M/G/k overcomes this obstacle with an analysis that works for any size distribution, without assuming finitely many job states (Ch. 17).

Multiserver Gittins has also received some attention in the adversarial scheduling literature. For example, Megow and Vredeveld [87] study scheduling with adversarial arrival times but stochastic job sizes. The prove that Gittins is a constant-factor approximation for mean *completion time*. However, completion time is a subtly different metric than response time: a job's response time is its completion time minus its arrival time. This means that while minimizing mean completion time and mean response time are equivalent objectives, approximation ratios for completion time do not carry over to response time. Our work on Gittins in the M/G/k thus avoids working with completion time, instead reasoning directly in terms of response time (Ch. 17).

2.3.3 Scheduling in More Complicated Multiserver Models

In addition to central-queue multiserver systems, scheduling and related problems have been considered in more complicated multiserver architectures. These include

- immediate-dispatch systems [61, 62, 81],
- systems with redundant job replicas [6, 7, 41, 42],
- polling systems [19, 21, 148],
- input-queued switches [37, 59, 60, 85, 132], and
- systems where jobs occupy multiple servers [58, 137, 151].

We have only just begun to understand scheduling in these systems. For instance, the literature on input-queued switches provides sophisticated algorithms for deciding which of many possible subsets of queues to serve, but the scheduling policy within each queue is always assumed to be FCFS.

We have taken a first step in this direction, studying immediate-dispatch systems with SRPT scheduling at each queue [51], though we do not cover this result in this thesis.

[2] Specifically, Glazebrook and Niño-Mora [47] analyze the preemptive M/M/1 with Bernoulli feedback, in which jobs can be modeled as finite-state Markov chains; and Glazebrook [46] analyzes (a generalization of) the nonpreemptive M/G/1 with Bernoulli feedback, in which jobs are piecewise-deterministic Markov processes with continuous state spaces, but preemption is allowed in only finitely many states. See Chapter 14 for more on modeling jobs as Markov processes.

2.4 Other Related Work

2.4.1 Work Decomposition Laws

Chapter 8 is about *work decomposition laws*. These are theorems that decompose the steady-state amount of work in a queueing system into a sum of two random variables, one of which is the amount of work in a simpler queueing system. Our results follow in the tradition of a long line of work on decomposition laws [22, 39, 40, 46, 47, 94]. While we believe our results are new, they are a natural extension of known decomposition laws. Roughly speaking, we apply the techniques of Miyazawa [94] to a wider variety of systems. This yields decomposition results similar to those of Glazebrook and Niño-Mora [47, Thm. 1] and Glazebrook [46, Lem. 1], but our results are more general, and our proofs are significantly simpler.

2.4.2 Size Estimates

Chapters 10 and 12 both study scheduling with *noisy size estimates*. This is a setting where the scheduler learns an estimate of each job's size when it arrives and must make do with these estimates when scheduling.

Most work on scheduling with size estimates has used simulations, with Dell'Amico et al. [32] starting a fruitful series of simulation studies [5, 30, 31, 36, 92, 93]. While there has been some analytical work on scheduling with size estimates [36, 92], the analysis is still applied numerically.

Our study of size estimates uses the SOAP analysis applied both numerically and theoretically, building on the prior work in two ways. First, thanks to the flexibility of SOAP, we can compare against the policy that minimizes mean response time, namely Gittins, resulting in new insights (Ch. 10). Second, we use SOAP to prove mean response time guarantees on policies for scheduling with size estimates, yielding theoretical results that hold for a range of noise models (Ch. 12).

We note that the ε-SMART policies of Wierman and Nuyens [147] are related to scheduling with size estimates, but they are not quite the same. This is because ε-SMART policies assume *dynamically updating* estimates for a job's *remaining work*, as opposed to a *static* estimate of a job's *initial size*.

2.4.3 Practical Preemption Limitations

Limited Priority Levels

Some systems only give the scheduler a *limited number of priority levels* to work with. A notable example is network switches, which might have, say, eight priority levels. One of the challenges to using policies like SRPT in systems with limited priroity levels is deciding how to partition the range of possible remaining work amounts, or more generally the range of possible ideal priorities, into a small number of levels. Work in the systems

literature has contended with this question in settings like network switches [96], web servers [57], GPU cluster scheduling [54], and TCP flow scheduling [86].

We use SOAP to study scheduling with limited priority levels (Ch. 9). While we do not model all of the concerns of the aforementioned systems, we arrive at similar conclusions, particularly for adapting SRPT to limited priority levels [57, 96].

Preemption Checkpoints

Some systems do not allow preempting jobs at any time but instead have *preemption checkpoints*. Preempting a job between checkpoints may be disallowed, or it may be undesirable due to the risk of losing progress. Most queueing-theoretic prior work on preemption checkpoints studies the problem of placing checkpoints to minimize lost progress [8, 33, 99].

To the best of our knowledge, the only queueing-theoretic work on preemption checkpoints in scheduling is that of Goerg [48], who analyzes a version of SRPT with preemption checkpoints. Using the analysis, Goerg [48] studies the question of how frequent preemption checkpoints should be to ensure good mean response time, assuming that each checkpoint incurs some overhead. We use SOAP to study an analogous question about LAS with preemption checkpoints (Ch. 9).

2.4.4 Scheduling in Adversarial Models

This chapter has focused almost exclusively on scheduling in the queueing theory community. However, scheduling has also been studied extensively by the algorithms community, typically with adversarial arrivals or all arrivals in a single initial batch. See Pinedo [107] and Lenstra and Shmoys [80] for recent treatments of this area. Throughout this chapter, we have mentioned a few scheduling results from the algorithms community that are particularly related to problems studied in this thesis [10, 11, 17, 63, 81, 87, 110].

2.5 Publications Covered in This Thesis

This thesis is based on a number of publications of which I was a coauthor. These publications are listed below, along with the chapters based on them. We also comment on ways where either prior publications or the thesis goes beyond what the other covers.

- (Chs. 6 and 7) Scully et al. [124] introduces SOAP. We have slightly extended SOAP in a subsequent publication [121], but we do not cover the extension here.
- (Chs. 9 and 10) Scully and Harchol-Balter [123] numerically applies the SOAP analysis to study practical scheduling questions. The second half of Chapter 10 is new material not covered in the paper.
- (Ch. 11) Scully et al. [125] introduces a simple scheduling policy whose mean response time is comparable to Gittins's. We give just an overview here, referring the reader to the paper for full proofs.

- (Ch. 13) Scully et al. [127] and Scully and van Kreveld [126] characterize the asymptotic response time tail of Gittins and other SOAP policies. We give just an overview here, referring the reader to the papers for full proofs.
- (Ch. 12) Scully et al. [120] studies scheduling with noisy size estimates. We give just an overview here, referring the reader to the paper for full proofs.
- (Chs. 8 and 17) Scully et al. [118] analyzes SRPT and Gittins in the M/G/k. This paper is the culmination of other work of ours on multiserver scheduling [50, 51, 119], but we do not cover these precursors here.[3] The results of Chapter 8 are more general than those in the paper.
- (Chs. 14–16) Scully and Harchol-Balter [122] introduces WINE and applies it to proving Gittins's optimality in the M/G/1, though the name "WINE" is new as of this thesis. A prior publication of ours [118] develops many of the same ideas but in a less general setting. The second half of Chapter 10 is new material not covered in the paper, though special cases of it have appeared in other publications of ours [120, 126].

[3] Grosof et al. [50], which gives the first analysis of SRPT in the M/G/k, deserves a brief discussion. This paper started our work on multiserver scheduling by analyzing SRPT in the M/G/k. In particular, it does so *without* using WINE. As such, our claim that we use WINE to give the first analysis of SRPT in the M/G/k is a simplification: we give the first analysis of SRPT in the M/G/k, and the version of that analysis presented in this thesis uses WINE, even though the original analysis did not use WINE. In contrast, WINE seems to be essential for analyzing Gittins in the M/G/k [119, Appx. A].

Chapter 3

SOAP Overview

A three-point summary of SOAP:

- (§ 3.1) Our ability to analyze the response time distribution of scheduling policies is limited to a small set of relatively simple policies. Beyond our reach are policies designed to deal with important concerns, such as job size uncertainty and practical preemption limitations. But the state of the art advances slowly, because each new policy generally requires its own custom-tailored analysis.
- (§ 3.2) SOAP defines a *broad class* of scheduling policies and gives a *universal analysis* of all policies in the class. The SOAP policy class covers most previously analyzed policies, but also many policies that have never been analyzed before.
- (§ 3.3) SOAP gives the first response time analyses of scheduling policies that deal with job size uncertainty, practical preemption limitation, and more. We use these analyses to answer many theoretically and practically motivated questions about scheduling policy design.

3.1 Problem: Can Analyze Only a Small Set of Scheduling Policies

3.1.1 Why Analyzing Scheduling Policies Matters

Scheduling can significantly improve a queueing system's response time, but it can be hard to determine the precise impact a particular scheduling design decisions will have. For example, consider using *Shortest Remaining Processing Time (SRPT)* in a single-server queueing system. SRPT, which always serves whichever job has the least remaining work, is known to minimize mean response time [116], but there are other questions we might have about SRPT's performance.

- How well does SRPT perform on non-mean response time metrics, such as response time tail?
- How are SRPT's response time benefits distributed across different types of jobs? Is SRPT unfair to large jobs?

Fortunately, thanks to the fact that SRPT has been analyzed [117], queueing theorists can answer these and many other questions about SRPT [13–15, 83, 101, 103, 104, 143–146].

What Is "Analyzing" A Policy?

When we say that SRPT has been "analyzed", we mean that its response time distribution has been characterized in a single-server queueing model, specifically the *M/G/1 queue*

(Ch. 5). However, this response time characterization is often still rather complicated, making it just the first step towards answering questions about the policy's performance. For instance, while Schrage and Miller [117] analyzed SRPT in 1966, it was not until decades later that the analysis was used to investigate SRPT's response time tail [103] and fairness [142].

3.1.2 Prior Scheduling Analyses Fall Short

Many scheduling policies have been analyzed in the M/G/1 in the years since Schrage and Miller [117] analyzed SRPT, as illustrated in Figure 3.1(a). However, as we discuss in our review of prior work (§ 2.1), this set of policies is still limited, and M/G/1 scheduling theory has little to say even slightly beyond the borders of what has already been analyzed.

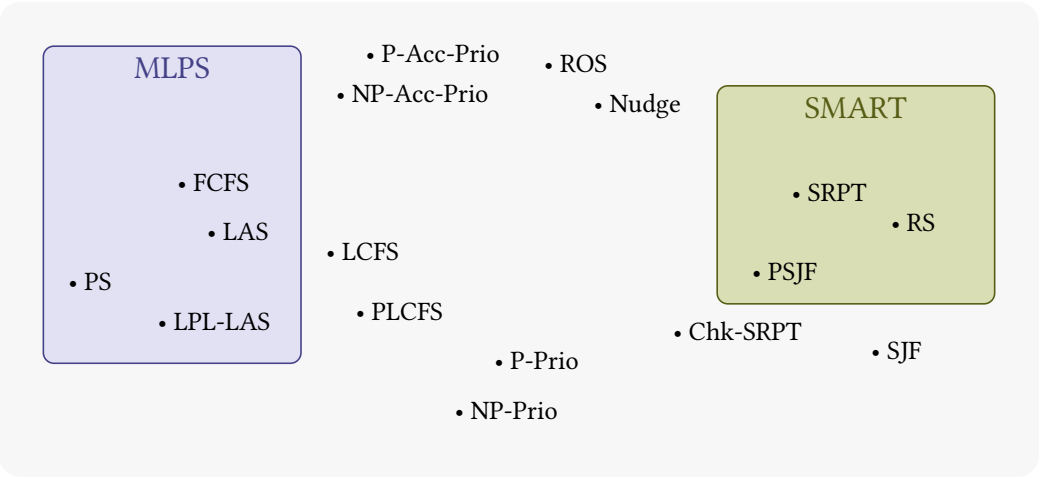
Practical Concerns Are Out of Reach

For example, we saw in Section 3.1.1 how SRPT's analysis helped us answer some questions a system designer might want to know before implement SRPT. However, in practice, few systems will perfectly implement SRPT, leading to a number of further questions.

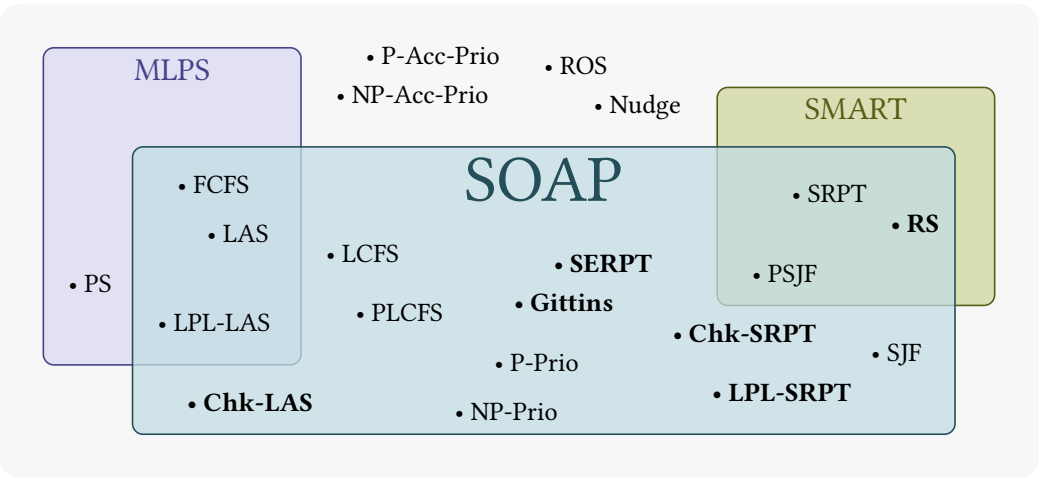
- SRPT assumes knowledge of each job's exact size. But in practice, some systems can provide only a noisy estimate of each job's size. How should we adapt SRPT to be robust to noisy size estimates? How does performance degrade as a function of the amount of noise?
- SRPT assumes arbitrary granularity in how it prioritizes jobs. But in practice, some systems have only a limited number of priority levels available to the scheduler. How should we adapt SRPT to settings with a limited number of priority levels? How many levels do we need for good performance?
- SRPT assumes jobs may be preempted at any time. But in practice, some systems allow jobs to be preempted only at certain checkpoints, and due to overhead, such checkpoints cannot be too frequent. How should we adapt SRPT to settings with preemption checkpoints? What checkpoint frequency balances good performance with low overhead?

Notice that these questions are not about SRPT itself, but about *practical variations* that deviate from true SRPT. Such practical variations are generally outside the scope of what we can currently analyze. In particular, the first and second questions above have not been theoretically studied in the M/G/1 prior to SOAP, though as we will discuss below, there is some work on the third question [48].

There are a number of other questions that are out of reach of current analysis. We might ask analogues of the second and third questions for preemptive policies other than SRPT. Or, as a variation on the first question, we might consider scheduling in a setting where size estimates are extremely noisy or nonexistent. In such a setting, we likely want to abandon SRPT altogether. One alternative is to use historical job size data to compute the *expected* remaining work of each job, then prioritize jobs based on that. This policy is called *Shortest Expected Remaining Processing Time (SERPT)*, but it has never been analyzed.



(a) Scheduling policies analyzed in the M/G/1, excluding those analyzed with SOAP. See Chapter 2 for further discussion on many of the policies shown.



(b) Scheduling policies analyzed in the M/G/1, including those analyzed with SOAP, which are highlighted in **bold**. This includes two policies, Chk-SRPT and RS, for which mean response time, but not distribution of response time, was previously known.

Figure 3.1. SOAP expands the set of policies we know how to analyze in the M/G/1, a single-server queueing model. (a) Prior to SOAP, most policies were analyzed one-by-one, with a few relatively small classes of policies analyzed (§ 2.1.5). (b) SOAP unifies and generalizes the state of the art with a *single universal analysis* for a broad class of policies, subsuming much of what was already known while also analyzing many policies for the first time.

Most Prior Analysis Happens One Policy at a Time

Some good news about the questions above is that there has been work analyzing SRPT with preemption checkpoints, with Goerg [48] providing a mean response time formula. Unfortunately, the analysis is specific to SRPT, so we are out of luck if we want to adapt a different policy for a system with preemption checkpoints.

This is just one instance of a larger issue with the state of the art in M/G/1 scheduling: for the most part, scheduling policies are analyzed one by one. There are numerous publications analyzing a single policy or small number of similar policies (§ 2.1). Analyzing a policy or few evidently takes a publication-sized amount of research effort. But expending so much effort on every single policy is unsustainable in light of the vast space of possible scheduling policies.

Ideally, we would hope to analyze many scheduling policies at once. Two examples of this are analyses of *classes* of policies: *Multi-Level Processor Sharing (MLPS) policies* [74] and *SMAll Response Times (SMART)* [146]. Both classes represent significant steps that increase our ability to analyze many policies at once. Unfortunately, both classes are somewhat limited in scope (§ 2.1.5). Roughly speaking, SMART includes only relatives of SRPT, and MLPS includes only specific combinations of simple policies like FCFS and LAS (§ 3.2.1).

3.2 Key Idea: Unifying Language for Policies Enables a Universal Analysis

Despite the fact that the state of the art in M/G/1 scheduling is analyzing policies one by one, there are common ideas that appear across multiple analyses. For example, part of Harchol-Balter [55], an introductory queueing text, is devoted to analyzing ten different scheduling policies in the M/G/1. But *nine out of ten* of the analyses follow a common overall strategy, albeit with different details. Can we unify the analyses of these nine policies? If so, can that help us analyze even more policies?

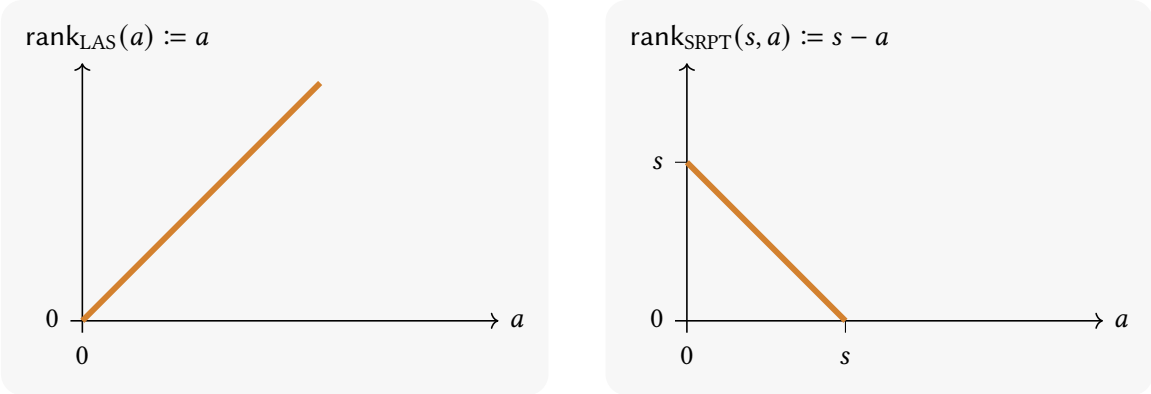
SOAP, which stands for *Schedule Ordered by Age-based Priority*, answers both questions affirmatively. SOAP consists of two parts:

- (§ 3.2.1) *SOAP policies:* a broad class of scheduling policies, which can all be described in a single unifying language.
- (§ 3.2.2) *SOAP analysis:* a single universal analysis that applies to all SOAP policies.

SOAP thus unifies and generalizes prior analyses, as shown in Figure 3.1(b).

3.2.1 Unifying Language: Rank Functions

The key idea behind SOAP is its unifying language for describing scheduling policies, which we call *rank functions*. A rank function assigns each job a *rank*, or numerical priority, based on the job's *age*, which is the amount of time the job has been served so far. We use the convention that lower rank is better. A rank function thus encodes a scheduling policy:



(a) Rank function of LAS.

(b) Rank function of SRPT for job of size s.

Figure 3.2. A *rank function* describes a scheduling policy in the following way. Each job, based on its age a and possibly other static characteristics, is assigned a *rank*, or numerical priority, by the rank function. (a) Under LAS, a job's rank is simply its age. (b) Under SRPT, a job's rank is its remaining work, which depends on both its age and its initial size.

the scheduler serves the job of minimal rank at every moment in time. In the case of ties, we usually use the convention that ties are broken in first-come, first-served (FCFS) order.[1]

A *SOAP policy* is any scheduling policy that can be described using a rank function. We give some examples of rank functions below. For many more examples, as well as a more formal definition of SOAP policies and rank functions, see Chapter 6.

Simple Examples of Rank Functions

Perhaps the simplest example of a policy that can be described as a rank function is *Least Attained Service (LAS)*, which always serves the job of least age. LAS can be represented by the rank function

$$\text{rank}_{\text{LAS}}(a) := a.$$

See Figure 3.2(a) for an illustration.

Another example is the aforementioned *Shortest Remaining Processing Time (SRPT)*. Here a job's rank is its remaining work, which is difference between the job's initial size s and the work done so far. The work done so far is simply the job's age a, so

$$\text{rank}_{\text{SRPT}}(s, a) := s - a.$$

See Figure 3.2(b) for an illustration.

One last example of a rank function is that of *First-Come, First-Served (FCFS)*, which serves jobs in arrival order. FCFS can be represented by many rank functions. In particular, thanks to our FCFS tiebreaking convention, any constant function $\text{rank}_{\text{FCFS}}(a) = c$ suffices.

[1] We actually also allow for last-come, first-served tiebreaking (LCFS) as well. Whether using FCFS or LCFS tiebreaking, there are several subtleties to consider, which Chapter 6 explains in detail.

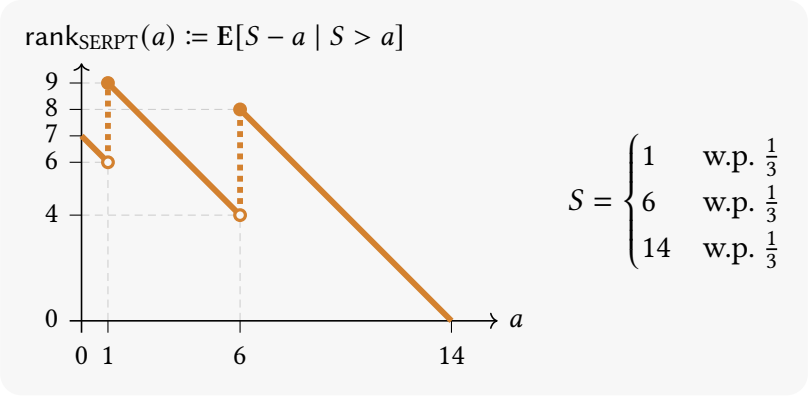


Figure 3.3. The SERPT policy always serves the job of least expected remaining work. We show the rank function of SERPT for the given job size distribution S, meaning jobs have size 1, 6, or 14. Initially, a job's expected size is $\mathbf{E}[S] = 7$, and its expected remaining work decreases while it is served. This continues until age 1, at which point one of two things happens: either the job completes and exits the system, or we learn from its continued presence that it is not size 1. Its expected remaining work then jumps up to $\mathbf{E}[S - 1 \mid S > 1] = 9$. A similar jump occurs at age 6.

More Complex Examples of Rank Functions

LAS, SRPT, and FCFS all have relatively simple rank functions. It is therefore unsurprising that all three of these policies have been analyzed. But, as we will see in examples below, policies with more complex rank functions arise in practical situations, and these complex rank functions are harder to analyze (§ 3.2.2).

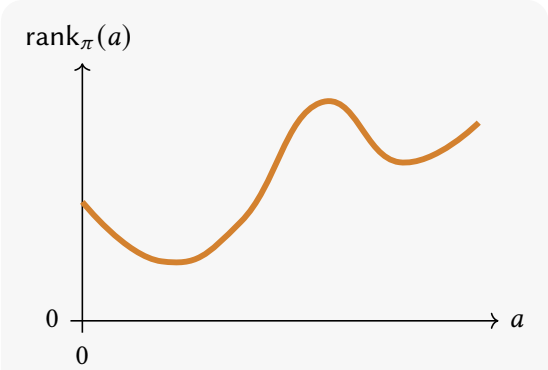
As a first example, suppose we are scheduling in a system with unknown job sizes with the goal of lowering mean response time. Ideally, we would use SRPT, but we cannot implement SRPT without knowing job sizes. How can we mimic SRPT using only jobs' ages? One policy that does this is *Shortest Expected Remaining Processing Time (SERPT)*. SERPT uses the job size distribution to compute the *expected* remaining work of each job based on its age. Specifically, for job size distribution S, SERPT's rank function is[2]

$$\text{rank}_{\text{SERPT}}(a) := \mathbf{E}[S - a \mid S > a].$$

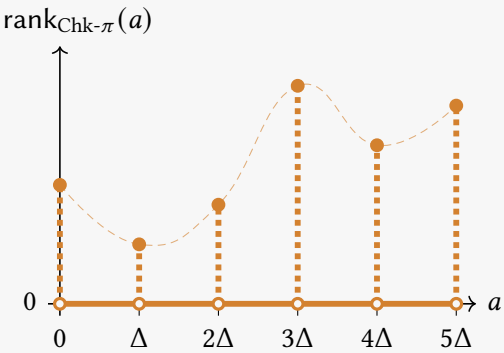
The way to understand the right-hand side is as follows. The job's size is a random variable S. When the job has age a, the remaining work is the difference $S - a$. But if the job is not yet complete, it must be that the job's remaining work is positive, so we condition on $S > a$.

We illustrate and explain SERPT's rank function for an example job size distribution in Figure 3.3. Notice that SERPT's rank function is *nonmonotonic*, unlike the previously mentioned rank functions of LAS, SRPT, and FCFS. It turns out that with a single exception (discussed below), all SOAP policies that have been analyzed in the past have nonmonotonic rank functions. As we explain in Section 3.2.2, rank nonmonotonicity is the main technical obstacle we overcome in the SOAP analysis.

[2] Abusing notation slightly, below, we write S for both the size *distribution* and also for the *random variable* representing a generic job's size. See (Ch. 5) for details regarding this and other notation conventions.



(a) Rank function of generic SOAP policy π.



(b) Rank function of Chk-π, with rank of π shown for reference (dashed orange line).

Figure 3.4. Preemption limitations can be thought of as restrictions on a SOAP policy's rank function. For example, given (a) a generic SOAP policy π, we can define (b) a new policy Chk-π, which is like π but only preempts jobs at checkpoint ages. These checkpoint ages are evenly spaced with gap Δ.

We have seen in SERPT that nonmonotonic rank functions can arise from unknown job sizes. They can also arise from other practical concerns, such as preemption limitations. Suppose we wish to schedule in a system where jobs can only be preempted at certain *checkpoint ages*. Perhaps these checkpoints occur every Δ units of time while serving a job. One example of this is scheduling packet flows in a network switch where Δ is the packet size, because we want to avoid interrupting transmission in the middle of a packet.

We can model scheduling with checkpoint ages using rank functions as follows. Let π be a generic SOAP policy.[3] We can define a variant of π, which we call *Checkpointed π (Chk-π)*, which is essentially a version of π that only preempts jobs at checkpoint ages. The rank function of Chk-π is

$$\text{rank}_{\text{Chk-}\pi}(a) \coloneqq \text{rank}_\pi(a)\, \mathbb{1}(a = n\Delta \text{ for some } n \in \mathbb{N}).$$

We illustrate an example in Figure 3.4. Notice that Chk-π can have a nonmonotonic rank function even if π's rank function is monotonic. For example, Chk-SRPT, which was analyzed by Goerg [48], has a nonmonotonic rank function. This makes Chk-SRPT the only nonmonotonic rank function to be analyzed prior to SOAP.[4] But Chk-LAS and essentially every other policy with checkpoints is also nonmonotonic, and none of these have been analyzed prior to SOAP.

[3]For simplicity of notation, we assume that π's rank function is always positive and takes only a job's age as input, as opposed to also needing a job's size or other properties.

[4]With that said, Goerg [48] only derives the *mean* response time of Chk-SRPT, so the SOAP analysis still provides the first *distributional* characterization of Chk-SRPT's response time.

3.2.2 Main Obstacle to Universal Analysis: Nonmonotonic Rank Functions

The SOAP analysis characterizes the response time of any SOAP policy in the M/G/1. Here we describe the main obstacles we overcome in the analysis. See Chapter 7 for the full analysis, and in particular Theorem 7.15 for the main result.

A number of SOAP policies have been analyzed in the past, but virtually all of them have monotonic rank functions. The SOAP analysis improves on this state of the art in two ways.

- SOAP *unifies* prior analyses. While LAS, SRPT, FCFS, and other SOAP policies were previously each analyzed separately, the SOAP analysis gives one formula that works for any rank function.
- SOAP *generalizes* prior analyses. In particular, virtually all prior analyses were for monotonic rank functions, but the SOAP analysis also applies to nonmonotonic rank functions.

What are the main obstacles to these improvements, and how do we overcome them? For unifying prior analyses, the answer is more subjective, but I believe the main obstacle is the lack of a uniform way of defining scheduling policies. This is why the idea of representing policies as rank functions is important: without a unifying language for policies, we have no hope of a universal analysis that applies to all of them.

For generalizing prior analyses, there is a much more concrete obstacle: nonmonotonic rank functions. In the rest of this section, we explain why policies with nonmonotonic rank functions are hard to analyze, then give the main insight that enables us to analyze them.

Why Nonmonotonic Rank Functions Are Challenging to Analyze

Perhaps the most common method of analyzing a scheduling policy's response time in the M/G/1 is the *tagged job approach*. The approach works by considering a single "tagged" job's journey through the system. By appropriately randomizing the tagged job's size and other attributes, the jobs already in the system when the tagged job arrives, and the arrivals that occur after the tagged job, the distribution of the tagged job's response time becomes the system's response time distribution [150].

How do we determine the tagged job's response time? This boils down to determining how long each other job *delays* the tagged job, meaning receives service while the tagged job is in the system. The tagged job's response time is the sum of these delays, plus its own size.

The SOAP analysis thus boils down to determining how long each other job delays the tagged job. Under monotonic policies like SRPT, this is not too hard to do. For example, suppose the tagged job has size s, and another job has remaining work r when the tagged job arrives.

- If $r \leq s$, then the other job outranks the tagged job both now and hereafter, so the other job delays the tagged job by r.

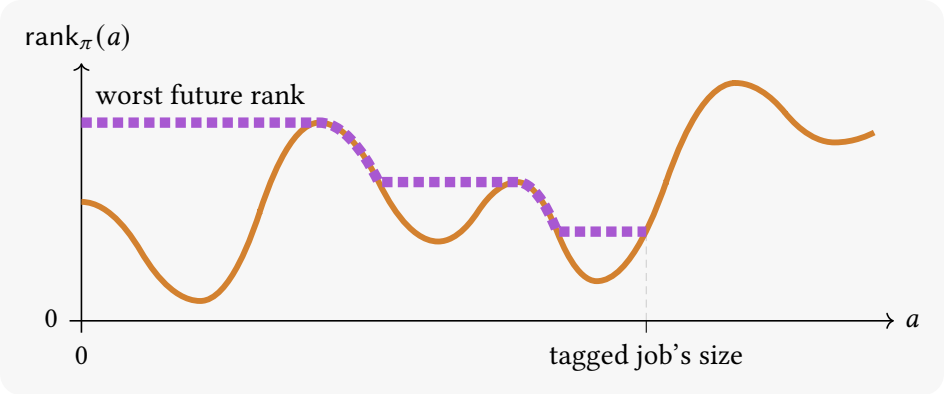


Figure 3.5. The Pessimism Principle states that the tagged job's response time is unaffected if instead of following the ordinary rank function (orange solid line), we increase its rank at each age to its worst future rank (magenta dotted line).

- If $r > s$, then the tagged job outranks the other job both now and hereafter, so the other job does not delay the tagged job at all.

However, under nonmonotonic policies, determining the delay due to another becomes more complicated. Whether the other job outranks the tagged job can change over time, with alternating periods of the tagged job and the other job having better priority, then the other job having better priority.

Insight: Remove Some Nonmonotonicity from the Analysis

The SOAP analysis uses the tagged job approach described above, but in a way that avoids dealing with the worst of the complexities introduced by nonmonotonic rank functions. It turns out that a single observation effectively removes some of the nonmonotonicity from the analysis. This observation, which we call the *Pessimism Principle*, is the following:

> The tagged job's response time is unaffected if we increase its current rank to its *worst future rank*.

Figure 3.5 shows the relationship between the rank function and the tagged job's worst future rank.

Why is the Pessimism Principle helpful? Without the Pessimism Principle, with of the tagged job and other job has better rank can alternate back and forth. With the Pessimism Principle, the tagged job's rank never increases, as shown in Figure 3.5. This means the other job outranks the tagged job until it is served for some amount of time t, after which it never outranks the tagged job again. Having a single interval of delay makes the tagged job's response time much easier to analyze.

But why is the Pessimism Principle true? The key is that the amount of time t the other job is served under the Pessimism Principle is the amount of time until it completes or reaches the tagged job's worst future rank. But without the Pessimism Principle, the total

delay due to the other job is this same amount t, because the tagged job will *eventually* reach its worst future rank. Increasing the tagged job's rank to its worst future rank thus does not create extra delay. It simply "clumps together" smaller delays, which simplifies the analysis.

3.3 Impact: Broad Class of Policies Analyzed for the First Time in the M/G/1

The SOAP analysis boils down to a single result, Theorem 7.15, that characterizes the response time distribution in the M/G/1 under any SOAP policy. But the breadth of SOAP policies makes it a high-impact result with a wide variety of applications.

Our applications of SOAP can be grouped into roughly three settings.

- (§ 3.3.1) We apply SOAP to scheduling with *unknown job sizes*. This involves analyzing policies like SERPT (Fig. 3.3).
- (§ 3.3.3) We apply SOAP to scheduling with *noisy job size estimates*. For instance, we analyze multiple SRPT-like policies to determine which are most robust to noise.
- (§ 3.3.2) We apply SOAP to scheduling with *practical preemption limitations*. One example of such a limitation is only allowing preemption at checkpoints (Fig. 3.4).

In each of these settings, we apply SOAP in two distinct ways.

- *Numerical:* We evaluate formulas from the SOAP analysis for specific rank functions, size distributions, and loads. Sometimes the rank functions involved are parameterized, allowing us to numerically optimize those parameters to achieve some objective. Despite their simplicity, computational applications teach us many important lessons about scheduling policy design.
- *Theoretical:* We use the SOAP analysis as a starting point for proving a theorem that holds for a broad class of rank functions, size distributions, or loads. Theoretical applications give more rigorous guarantees than what we can learn from computational applications. However, they are much harder to obtain, as they require novel theoretical insight on top of the already complex SOAP analysis.

Because the SOAP analysis is for the M/G/1, all of the applications described in this section are in single-server systems.

3.3.1 Unknown Job Sizes

How Good Is SERPT's Mean Response Time?

Recall from Figure 3.3 that SERPT is the policy that always serves the job of least expected remaining work. SERPT is one way to generalize SRPT to settings with unknown job sizes.

Given that SRPT minimizes mean response time when job sizes are known, it is natural to ask: does SERPT minimize mean response time when job sizes are unknown? Unfortunately, it is not so simple. Instead, a different, more complicated policy called the *Gittins*

policy is known to be optimal with unknown sizes (§ 2.2).

Despite its suboptimality for mean response time, SERPT has an intuitive appeal, and it is significantly simpler than Gittins. Does SERPT achieve mean response time close enough to Gittins to serve as a substitute for it in practice? We use SOAP to investigate this question both numerically and theoretically.

- (Ch. 10) Numerically, we find that SERPT does indeed seem to have near-optimal mean response time in a variety of situations.
- (Ch. 11) Theoretically, we unfortunately are unable to prove results about SERPT itself. In light of this, we propose a new variant of SERPT, called *monotonic SERPT (M-SERPT)*, that is even simpler to implement. We are able to prove a mean response time guarantee for M-SERPT, specifically that its mean response time is always within a factor of 5 of Gittins's.

How Good Are SERPT's, M-SERPT's, and Gittins's Response Time Tails?

We have seen above that SERPT, M-SERPT, and Gittins have good mean response time. However, mean response time is not the be-all and end-all of queueing metrics. Another important metric is the *response time tail*, namely the probability that a job experiences response time larger than some threshold. By looking at the *asymptotic* behavior of the response time tail, we can understand the probability with which jobs have especially long response time.

We theoretically apply the SOAP analysis to investigate the asymptotic response time tail of SERPT, M-SERPT, and Gittins (Ch. 13). Our main question is whether any of these policies have asymptotically optimal response time tail, because if so, we would have a policy that performs well for both the mean and tail of response time.

- For heavy-tailed size distributions, all three policies have optimal asymptotic tail.
- For light-tailed size distributions, all three policies can have asymptotic tail that is optimal, pessimal, or in between. But we can tweak either policy to avoid the pessimal case without significantly harming mean response time.

3.3.2 Practical Preemption Limitations

Queueing theoretic study of preemptive scheduling typically assumes that preemption is unrestricted and incurs no overhead, but preemption can be a lot messier in practice. We have already seen that we can use SOAP policies to model some of these practicalities, such as only being able to preempt jobs at certain checkpoints, as in the Chk-π policy. Another type of preemption limitation that occurs in practice is having only a limited number of priority levels to work with when designing a policy. This happens, for instance, when scheduling packet flows in network switches [96], as there are typically a small number of priority levels baked into a switch's hardware.

How should we adapt preemptive scheduling policies to handle practical preemption concerns like limited priority levels and preemption checkpoints? We use SOAP to investigate this question both numerically and theoretically.

- (Ch. 9) Numerically, we find rules of thumb for scheduling in systems with limited priority levels and preemption checkpoints. For example, we find that whether job sizes are known or unknown, one can often achieve near-optimal mean response time with just 5 or 6 priority levels.
- (Ch. 13) Theoretically, we investigate how frequent checkpoints need to be to ensure asymptotic optimality of the response time tail. We find that a constant gap between checkpoints never harms tail optimality, but we also find that tail optimality is still possible even when the checkpoint gap grows as a function of age.

3.3.3 Noisy Size Estimates

SRPT minimizes mean response time assuming access to perfect job size information. However, information provided in a practical system will virtually always be imperfect. This prompts a question: is SRPT robust to noise in the job size information it is given? If not, can we design a new policy that is? We use SOAP to investigate this question both numerically and theoretically.

- (Ch. 10) Numerically, we find that naively using SRPT with noisy size estimates can lead to surprisingly poor performance. Fortunately, we find a promising alternative in a policy called *Preemptive Shortest Job First (PSJF)*, which prioritizes jobs by (original) size instead of remaining work. PSJF is known to nearly match SRPT's performance when given perfect job size information [146], but we find that as estimation noise increases, PSJF is much more robust than SRPT.
- (Ch. 12) Theoretically, we use the SOAP analysis to prove theorems that explain the aforementioned non-robustness of SRPT and robustness of PSJF. We also propose a new variant of SRPT, whose design is guided by the SOAP analysis, that we prove has robustness properties similar to PSJF.

Chapter 4

WINE Overview

A three-point summary of WINE:

- (§ 4.1) We do not know how to schedule jobs in multiserver systems to *optimize* response time objectives. Are policies like SRPT and Gittins, which are good in single-server systems, also good in multiserver systems? We cannot answer this question because we do not even know how to *analyze* the response time of a given policy. Existing techniques for single-server systems, such as SOAP Chapter 3, do not generalize to multiserver systems.
- (§ 4.2) WINE is a *new queueing identity* that relates a system's performance to a much simpler quantity: the *relevant work* in the system. WINE is helpful because it turns bounds on relevant work, which can be obtained using other techniques (Ch. 8), into bounds on mean response time. WINE works in any queueing system, unlike SOAP, which is limited to single-server systems.
- (§ 4.3) We use WINE to bound the mean response time of SRPT and Gittins in multiserver systems. The bounds are tight enough to imply near-optimality for mean response time in the multiserver setting. And although we developed WINE for multiserver systems, it turns out to have several applications in single-server scheduling, too.

4.1 Problem: Analyzing and Optimizing Scheduling in Multiserver Systems

Multiserver queueing systems are ubiquitous in practice. For example, virtually all computer systems today have multiple processing units, from smartphones with multiple cores to datacenters with thousands of machines. Unfortunately, while queueing theorists have studied scheduling in single-server systems for decades, there is currently very little queueing theory that can help us analyze or optimize scheduling policies in multiserver systems.

For concreteness, consider the problem of minimizing mean response time in a system with known job sizes. In single-server systems, it has long been known that SRPT, which always serves the job of least remaining work (§ 3.2.1), is the optimal policy [116]. SRPT is actually proven optimal not just in the M/G/1, which has stochastic arrivals, but also in single-server systems with *adversarial* arrivals (§ 5.1.3).

Although SRPT's optimality proof only holds with a single server, the general idea of serving jobs that will complete soon seems wise even we have multiple servers. We therefore ask:

> Is SRPT also optimal, or at least near-optimal, in multiserver systems?

We focus for now on central-queue systems, like the M/G/k, in which $k \geq 2$ servers are connected to a single queue.

When necessary for disambiguation, we append a "-k" to a policy's name when discussing its k-server version. That is, SRPT-1 is the single-server policy that always serves the job of least remaining work, and SRPT-k is the k-server policy that always serves the k jobs with the k least amounts of remaining work, or all jobs if there are fewer than k.

4.1.1 Prior Work on SRPT-k Is Not Enough

Scheduling in multiserver systems has received some study, though the core question of whether SRPT-k performs well in the M/G/k is still open. Below we discuss the most immediately relevant prior work, which studies SRPT-k in an adversarial model, and explain why it does not give a strong indication as to whether SRPT-k performs well in the M/G/k. Chapter 2 gives a more comprehensive account of prior work.

SRPT-k Can Perform Poorly under Adversarial Arrivals

Leonardi and Raz [81] show that under adversarial arrivals, SRPT-k is suboptimal with *unbounded competitive ratio*. This means that an *offline* policy π-k, meaning one that knows the sizes and arrival times of past *and future* jobs, can achieve mean response time much better than SRPT-k's for some arrival sequences, making $\mathbf{E}[T_{\text{SRPT-}k}]/\mathbf{E}[T_{\pi\text{-}k}]$ arbitrarily large for those sequences. With that said, Leonardi and Raz [81] also show that no other policy can have bounded competitive ratio, and SRPT-k's competitive ratio is essentially the best possible.

How Good is SRPT-k under Stochastic Arrivals?

The proof that SRPT-k has unbounded competitive ratio under adversarial arrivals uses an arrival sequence specifically constructed to be bad for SRPT-k. But such arrival sequences are unlikely to occur naturally in practice. Therefore, to determine whether SRPT-k is a policy one should use in practice, it is helpful to analyze its performance in stochastic models, such as the M/G/k.

Unfortunately, the above results for adversarial arrivals give us little clue as to how well SRPT-k performs in the M/G/k. On one hand, SRPT-k has the best possible competitive ratio, which seems to be a good sign. But on the other hand, that competitive ratio is unbounded, indicating that SRPT-k can perform poorly. It is unclear whether some other policy with the same competitive ratio as SRPT-k, or maybe even a worse competitive ratio, might significantly outperform SRPT-k in the M/G/k.

Unknown Job Sizes and the Gittins Policy

We have thus far focused our discussion on SRPT-k, which requires known job sizes to implement. But we can ask the same questions for scheduling with unknown job sizes, or

more generally any level of partial information about job sizes.

In settings with any amount of job size information,[1] mean response time is minimized by a policy called the *Gittins* policy (§ 3.3.1). In fact, SRPT can be seen as the special case of Gittins for the case where the amount information known about each job is the job's exact size.

We might hope to use Gittins in a multiserver setting like the M/G/k. Fortunately, defining a multiserver version of Gittins is straightforward. In a single-server setting, Gittins works much like SRPT: it assigns each job a numerical rank and always serves the job of least rank (§ 3.2.1). We can thus define Gittins-k to be the policy that serves the k jobs with the k least ranks, much like SRPT-k.

Given that Gittins-1 minimizes mean response time in the M/G/1, it is natural to ask: is Gittins-k also optimal or near-optimal in the M/G/k? Unfortunately, there is no prior work on the mean response time of Gittins-k, so this question is wide open.

4.1.2 Why the M/G/k Is Harder to Analyze than the M/G/1

We use SOAP, the other tool in this thesis (Ch. 3), to analyze a wide variety of policies in the M/G/1. What prevents us from doing a similar style of analysis in the M/G/k? Answering this requires describing how the SOAP analysis works at a high level.

Background: Relevant Work

The SOAP analysis uses a technique called the *tagged job approach* which is common in M/G/1 scheduling theory (§ 2.1). The basic idea is to follow a generic "tagged" job on its journey through the system, tracking how long it spends in service and how long it spends waiting in the queue. As we discuss in Chapter 3, this is challenging because each job's rank can go up and down, including the tagged job's rank, and jobs arrive and depart over time.

Fortunately, there is a trick that helps simplify the above story. Roughly speaking, rather than tracking the exact states of all the jobs that might delay the tagged job, we track a summary quantity called *relevant work*. Roughly speaking, relevant work is the total amount of work that currently outranks the tagged job. Put another way, if new jobs suddenly stopped arriving, then the amount of time until the tagged job would next enter service is the amount of relevant system work. If the tagged job's current rank is r, then we call relevant work $\leq r$*-work*, because it's work on jobs with rank r or better.

Why Relevant Work Helps More in the M/G/1 than in the M/G/k

Keeping track of relevant work for a given rank r is usually easier than keeping track of every job's state, because relevant work summarizes the many job states with a single

[1] Specifically, we assume job sizes follow some stochastic model, and while this stochastic model is known to the scheduler, the future size outcome for any particular job is unknown. One simple example of this is the one discussed in Section 3.3.1, where the scheduler knows the job size distribution but not any job's size.

number. And remarkably, tracking relevant work turns out to be enough to analyze SOAP policies in the M/G/1. The reason why is that in the M/G/1, the single server acts as a "choke point". This means that all relevant work is strictly prioritized over the tagged job, which in turn is strictly prioritized over any other work.

Unfortunately, the situation is more complicated in the M/G/k. Because there are multiple servers, there is no single "choke point". While the tagged job is in service at one server, other servers can serve other jobs, which may have rank better or worse than the tagged job's rank. This means that just looking at relevant work does not tell the whole story in the M/G/k, which makes a tagged job analysis much more difficult.[2]

4.2 Key Idea: Relate Response Time to Work, a Much Simpler Quantity

We have seen that the tagged job approach which is successful for the M/G/1 is difficult to translate to the M/G/k. One of the principle obstacles is that *relevant work*, the amount of work with priority over the tagged job, plays a much clearer role in determining the tagged job's response time in the M/G/1 than it does in the M/G/k. Nevertheless, relevant work remains a promising idea for summarizing the system state. Is there a way to translate relevant work into response time *without* using the tagged job approach?

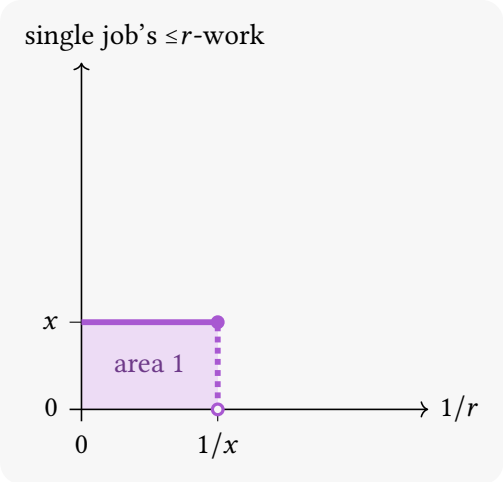
WINE, which stands for *Work Integral Number Equality*", answers this question affirmatively. WINE is a new queueing identity that directly relates the number of jobs in a queueing system to relevant work. Crucially, WINE holds in *any queueing system*, and in particular in the M/G/k. By combining WINE with Little's law [84], which relates the number of jobs to mean response time, WINE gives an *exact* formula for mean response time in the M/G/k in terms of relevant work.

Of course leaves us with a burning question: how much relevant work is in the M/G/k? This is still a much harder question than analyzing relevant work in the M/G/1. Our key idea here is to relate relevant work in the M/G/k to relevant work in an M/G/1 whose server is k times as fast. We prove a result called *Relevant Work Decomposition* (Ch. 8),[3]

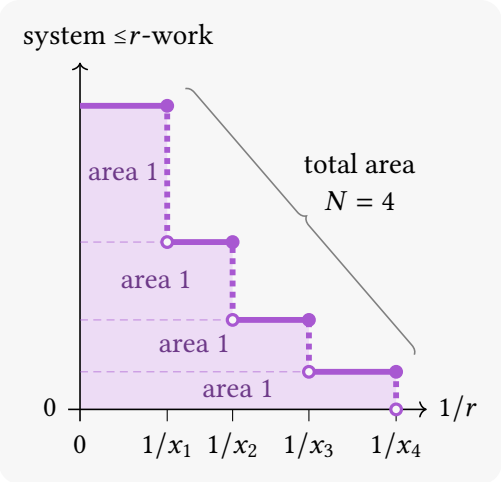
There are actually many versions, or "flavors", of WINE. The rest of this section fully illustrates one of the simplest flavors of WINE (§ 4.2.1) and gives a quick taste of other flavors (§ 4.2.2). Section 4.3 explains in more detail how WINE and Relevant Work Decomposition combine to analyze the M/G/k under SRPT-k and Gittins-k.

[2] As we discuss in our review of prior work (§ 2.5), we actually have managed to complete tagged job analyses of SRPT-k and a few other relatively simple policies in multiserver systems [51, 52, 119], but the technique has little hope of generalizing to more complex policies like Gittins-k [119].

[3] Relevant Work Decomposition appears in Part II instead of Part III because it turns out that a special case of Relevant Work Decomposition helps with the SOAP analysis, so we introduce it when needed.



(a) Integrating the $\leq r$-work of a single job with remaining work x.



(b) Integrating system $\leq r$-work with $N = 4$ jobs present.

Figure 4.1. Illustration of SRPT-flavored WINE, which relates the number-in-system N to an integral of $\leq r$-work. Recall that SRPT assigns a job rank equal to its remaining work.

4.2.1 Basic "SRPT-Flavored" WINE

How can we hope to translate between relevant work and the number of jobs? For concreteness, let us consider SRPT, under which a job's rank is its remaining work.

WI is for Work Integral

There is a hint in the WINE name: the number of jobs results from an integral of $\leq r$-work. How might we design such an integral? The key idea is to look at one job at a time: if we can make a single job's $\leq r$-work integrate to 1, then the system's total $\leq r$-work will integrate to the number of jobs.

Consider a single job with remaining work x.[4] What is this single job's $\leq r$-work under SRPT? Recall that SRPT assigns the job rank x. If $x \leq r$, then the job is relevant, so its $\leq r$-work is its remaining work x. If instead $x > r$, then the job is irrelevant, so its $\leq r$-work is 0. That is,

$$\text{single job's } \leq r\text{-work} = x\mathbb{1}(x \leq r).$$

How can we integrate this quantity to get r? Figure 4.1(a) gives one way of doing so:

$$1 = \int_0^\infty (\text{single job's } \leq r\text{-work})\, \mathrm{d}(1/r) = \int_0^\infty \frac{\text{single job's } \leq r\text{-work}}{r^2}\, \mathrm{d}r.$$

[4] To clarify, even though we are looking at one job at a time, this is *not* a tagged job analysis. As will soon become clear, instead of following this job in its journey through the system, we will examine it at a specific moment in time.

This means that if we integrate the total system $\leq r$-work instead of a single job's $\leq r$-work, we get the number of jobs N currently in the system, as illustrated in Figure 4.1(b):

$$N = \int_0^\infty (\text{system } \leq r\text{-work})\, \mathrm{d}(1/r) = \int_0^\infty \frac{\text{system } \leq r\text{-work}}{r^2}\, \mathrm{d}r.$$

This equation is *SRPT-flavored WINE*.

Under What Conditions Does SRPT-Flavored WINE Hold?

A subtle note that should be appreciated about SRPT-flavored WINE is that, despite being SRPT-flavored, it holds in any queueing system under any scheduling policy. Specifically, the only place we appealed to SRPT was in determining a job's $\leq r$-work, which we might more specifically call "SRPT-flavored" $\leq r$-work. But we can still look at the system from the perspective of SRPT's rank function, even if another scheduling policy is being used.

4.2.2 Other Flavors of WINE

We have seen SRPT-flavored WINE gives a formula for the number of jobs that holds under very general conditions. However, it requires analyzing SRPT-flavored $\leq r$-work, which seems like it might be difficult to do for policies other than SRPT. It turns out that SRPT-flavored WINE can in fact help us analyze policies other than SRPT (§ 4.3), but there is nevertheless some merit to this concern. Are there other flavors of WINE?

It turns out there are an infinite array of flavors of WINE. Roughly speaking, SRPT-flavored WINE comes from looking at the system from SRPT's perspective, which assumes each job's size is known. The key idea to generalizing WINE is to look at the system from a *less-informed perspective* where job sizes are uncertain. There are many types of uncertainty: we might have no idea as to each job's size, or we might have very accurate size estimates, or somewhere in between. Each of these "types of uncertainty", a concept we formalize in Chapter 14, yields a new flavor of WINE.

There is actually another way we can generalize WINE. So far, we have integrated relevant work to compute the number of jobs, which is useful for computing mean response time. But if we are in a setting where some jobs are more important than others, we might instead want to optimize for a *weighted* mean response time metric. We derive flavors of WINE that give different jobs different weights, yielding new formulas for weighted mean response time.

How do we derive these different flavors of WINE? It turns out the key is to look at relevant work from the perspective of the Gittins policy. That is, these other versions of WINE are "Gittins-flavored". Deriving the general form of WINE thus amounts to proving properties of Gittins's rank function.

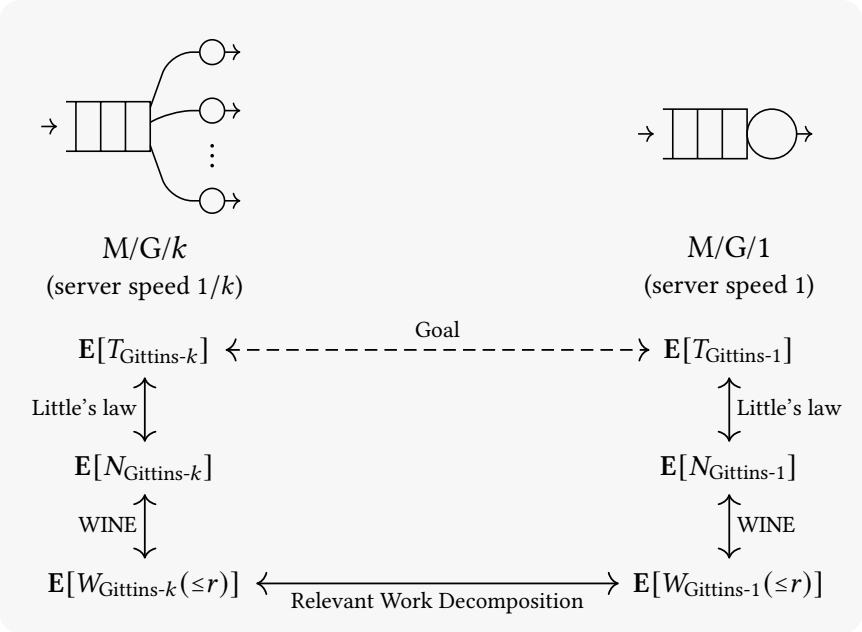


Figure 4.2. Using WINE to analyze the mean response time of Gittins-k in the M/G/k.

4.3 Impact: Near-Optimal Mean Response Time in the M/G/k, and More

Figure 4.2 illustrates how we use WINE to analyze mean response time in the M/G/k. The figure shows Gittins, but we use the same approach for SRPT.[5]

The main idea behind analyzing the M/G/k is to scale its servers' speeds so that their total speed is 1. This gives the M/G/k the same total server speed as an M/G/1. One might therefore hope that the M/G/k and M/G/1 have similar amounts of $\leq r$-work. We can use Relevant Work Decomposition (Ch. 8) show that this is indeed the case, resulting in a precise comparison between the mean relevant system work $\mathbf{E}[W(\leq r)]$ under Gittins-k to that under Gittins-1. WINE converts this comparison into a comparison of mean number-in-system $\mathbf{E}[N]$, which Little's law [84] in turn converts into a comparison of mean response time $\mathbf{E}[T]$. The end result, presented in Chapter 17, has the form

$$\mathbf{E}[T_{\text{Gittins-}k}] \leq \mathbf{E}[T_{\text{Gittins-1}}] + (k-1)(\text{“something small”}).$$

In particular, this bound is tight enough to imply that Gittins-k is in a certain sense near-optimal for mean response time, because the "something small" term usually grows more slowly as a function of load than the $\mathbf{E}[T_{\text{Gittins-1}}]$ term.

While our main motivation for developing WINE is analyzing multiserver systems, WINE is also useful in analyzing single-server systems. For example, in Chapter 16, we use

[5] In fact, SRPT can be viewed as a special case of Gittins (Pol. 6.12).

WINE to show that an approximately computed version of Gittins still has approximately optimal mean response time in the M/G/1. This proves to be a critical step of Chapter 12, which combines the WINE-based approximate Gittins result with additional analysis using SOAP to design a variant of SRPT that is robust to job size estimation error.

Part II

SOAP

Chapter 5

Core Modeling Assumptions and Queueing Theory Background

There are many types of queueing systems, and there are many ways one might mathematically model them. The goal of this chapter is to introduce the types of queueing systems we consider (§ 5.1) and specify the modeling choices we make (§ 5.2). We also introduce queueing terminology and notation we use throughout our study (§§ 5.3–5.6).

We use the model described in this chapter throughout the entire thesis. We use it as-is in Part II, and we add just one more feature (Ch. 14) in Part III. When we venture beyond this default model, which occurs only occasionally, we clearly state what assumptions change.

5.1 What Is a Queueing System?

At a high level, a queueing system consists of the following:

- *Servers*, entities that do some sort of useful work.
- *Jobs*, entities with some amount of work, called the job's *size*, to be done at a server.
- *Queues*, which hold jobs that are waiting for time at the server.

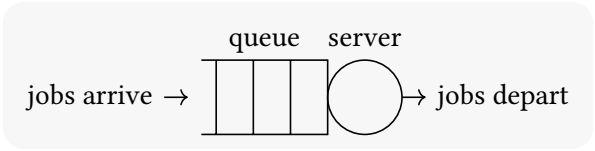
We model queueing systems that have a fixed configuration of servers and queues, some examples of which are shown in Figure 5.1.[1] Jobs, in contrast, are transient: they arrive, spend time in the system, then eventually depart when their work is complete.

While in the system, a job may spend time *queueing*, meaning in a queue, and spends the rest of its time *in service*, meaning at a server. Likewise, each server is sometimes *busy*, meaning serving a job, and is otherwise *idle*.

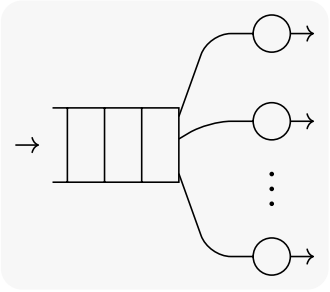
A canonical example of a queueing system is checkout in a grocery store: servers are checkout counters, each of which has a queue, and jobs are customers, and a job's size is roughly determined by the number of items they are buying. Among the queueing systems shown in Figure 5.1, the grocery store checkouts I frequent most resemble Figure 5.1(c), with customers distributing themselves among the various checkout counters.

We assume that jobs only occupy one server at a time and that, for the most part, servers serve one job at a time. Thus, whenever the system has more jobs than servers, some jobs must wait in a queue. When this occurs, the system's *scheduler* decides which jobs to serve according to a *scheduling policy* (§ 5.3). Choosing the right scheduling policy is an important aspect of designing queueing systems. Typically, a policy is chosen based

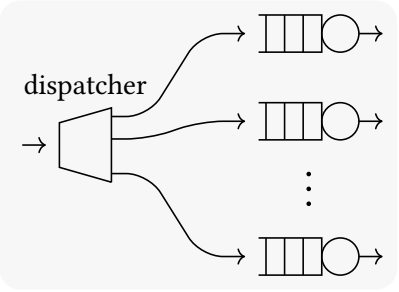
[1] For some applications, such as ride-sharing, it makes sense to also allow servers to arrive and depart, but such systems are outside the scope of this thesis.



(a) Single-server system.



(b) Central-queue multiserver system.



(c) Immediate-dispatch multiserver system.

Figure 5.1. Examples of queueing systems.

on its performance on one or multiple metrics. These metrics include quantities like *mean response time*, the average amount of time jobs spend in the system (§ 5.4.1).

In practice, all queues have a limited capacity, and jobs typically will not wait in a queue forever. Returning to the grocery store example, even the largest supermarkets can hold only so many people, and all but the most patient customers would leave the store empty-handed instead of waiting in an hours-long checkout line. However, as an idealized assumption, we assume that queues have unlimited capacity and that jobs, once they enter the system, do not leave until they have been served.

5.1.1 What do Servers and Jobs Represent?

Exactly what the servers and jobs in our model represent depend on what real-world system we wish to study. When studying queueing in data centers, we might make any of the following choices, depending on what aspect of the system we want to focus on:

- A server represents a CPU core, and a job represents a single thread.
- A server represents a continuously running program, such as a database, and a job represents a request to that program, such as a query.
- A server represents a multicore computer, and a job represents a parallel program.
- A server represents a network switch, and a job represents a packet flow.
- A server represents a repair technician, and a job represents a broken computer or network switch.
- A server represents the entire data center, and a job represents a large distributed computation.

Of course, this list is far from exhaustive: there are more examples of queueing in data

centers, to say nothing of other domains.

Instead of committing to any one choice about what servers and jobs represent, we consider queueing systems in the abstract. Many real-world systems resemble our model, and our results can in principle provide insight into any of them, though the value of this insight depends on how closely our modeling assumptions match the real-world system in question.

5.1.2 Central-Queue vs. Immediate-Dispatch Systems

A queueing system may have one or multiple queues and one or multiple servers, as illustrated in Figure 5.1. This thesis focuses on *central-queue* systems, in which all servers are connected to a single queue. This means that any job may be served at any server. Both Figures 5.1(a) and 5.1(b) show central-queue systems.

Figure 5.1(c) shows another type of system called *immediate-dispatch*: each server is connected to its own queue, and jobs must be assigned to a server upon arrival. This creates an additional scheduling constraint: jobs may only be served at the server to which they were assigned. While a few of our intermediate results apply to immediate-dispatch systems (Ch. 8), we primarily focus on central-queue systems.

5.1.3 Adversarial vs. Stochastic Arrivals

When modeling a queueing system, some of the most important choices concern how jobs enter the system. When does each job arrive? What is each job's size? What information does the scheduler learn about each arriving job?

Broadly speaking, there are two ways theorists model arrival processes:

- *Adversarial arrival models* make few, if any, assumptions about what arrivals occur.
- *Stochastic arrival models* assume that arrivals are generated according to some random process.

Each model has its strengths and weaknesses. Studying adversarial arrival models can yield guarantees about a system's performance in the worst-case scenario. However, the lessons learned from adversarial models can be misleading if such worst-case scenarios are uncommon: it may be that we optimize performance in a rare scenario but sacrifice performance in more common scenarios. This is because adversarial models have no notion of what scenarios are "rare" or "common".

Optimizing for common scenarios is where stochastic arrival models shine: the assumption of underlying randomness tells us that some scenarios are more likely than others. We can thus optimize for probabilistic quantities, such as means and percentiles of key metrics (§ 5.4). However, the lessons learned from stochastic models can be misleading if the randomly generated arrivals do not closely enough resemble the real-world arrivals being modeled.

We study stochastic arrival models. The distributional assumptions we make on the arrival process, outlined in Section 5.2 below, aim to be general enough to teach us broadly

applicable lessons. For example, while we assume each job's size is drawn randomly from some distribution (§ 5.2.2), many of our results apply no matter what that distribution is.

5.2 Primary Model: The M/G/1 with Labels

One of the main goals of this thesis is to study entire classes of scheduling policies. Instead of studying individual policies one by one, we seek generic results that yield insight into many policies at once. This approach requires a model that allows us to study a wide range of scheduling scenarios with a single vocabulary. A particular obstacle to doing so is that in different scenarios, the scheduler uses different information about the jobs in the system to make its scheduling decisions.

- There may be multiple *priority classes* of jobs, such as from different clients paying for different qualities of service.
- The scheduler might have any level of *size information* about each job. In some scenarios, job sizes might be exactly known. In others, only noisy estimates may be available.
- There may be *other metadata* attached to each job, such as its geographic origin.

The purpose of this section is to define a single model that we can use to study all of the above concerns and more.

Our model is a version of the venerable M/G/1, a stochastic single-server model with a long history in queueing theory [26, 28, 74]. We call our model the *M/G/1 with labels*. Its distinguishing feature is that each job has a *label* representing information the scheduler knows about the job, though exactly what this information is depends on the particular system being modeled. Hereafter, when we discuss the M/G/1, we mean the M/G/1 with labels unless otherwise specified.

Figure 5.2 summarizes the key features of the M/G/1 with labels. The rest of this section fills in the details by answering the following questions:

- (§ 5.2.1) What do jobs look like, and what does it mean to serve a job?
- (§ 5.2.2) How do new jobs arrive?
- (§ 5.2.3) What does the scheduler know about each job?

We conclude the section by describing the multiserver version of our model (§ 5.2.4). We defer discussing how the scheduler decides which job to serve to Section 5.3.

5.2.1 Anatomy of a Job

As mentioned in Section 5.1, each job has some amount of work to be done at the server. We follow the convention that the server serves work at rate 1. We thus measure work in units of time: a job's work is the amount of time it needs to be served.

At any moment in time, four pieces of data are associated with each job in the system:

- *Size*: the amount of work the job had initially when it arrived.
- *Age*: the amount of time the job has already been served.
- *Remaining work*: the amount of work the job has now.

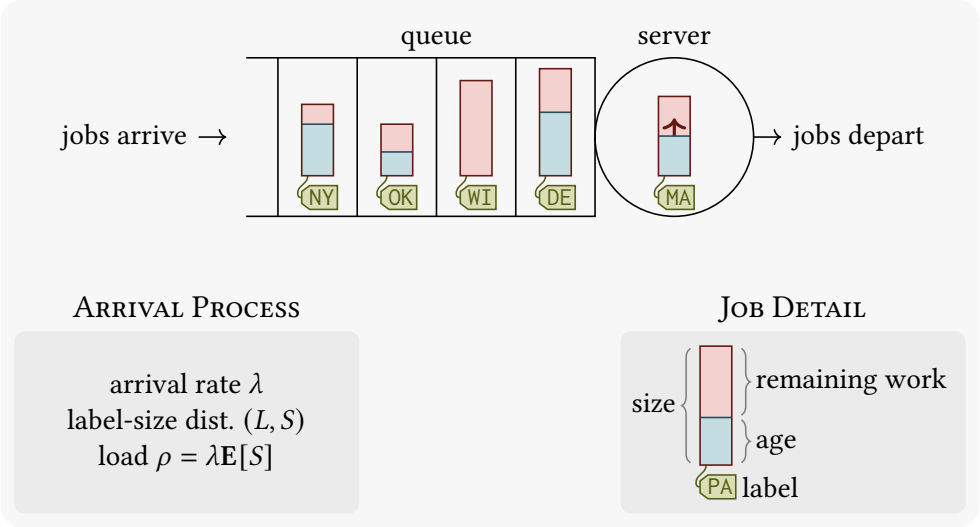


Figure 5.2. The M/G/1 with labels. Several jobs of varying size, remaining work, ages, and labels are present, with new jobs arriving over time according to a stochastic arrival process. The job labeled MA is in service, so its work decreases and age increases at rate 1 (red arrow). Each job's label communicates some information to the scheduler, which in this case is the US state the job came from.

- *Label*: information about the job that is revealed to the scheduler upon arrival.

All four of these concepts are illustrated in Figure 5.2. When discussing job sizes, we use the terms like *small*, *smaller*, *large*, and *larger*, which have their natural meanings.

A job's size and label are static: they are determined upon arrival and never change. A job's age and remaining work are dynamic: they respectively increase and decrease at rate 1 while a job is in service, though they do not change while a job is queueing. A job's size, age, and remaining work always satisfy

$$\text{size} = \text{age} + \text{remaining work}.$$

When a job completes, meaning when its remaining work reaches 0, it departs the system.

We assume that the job in service may be *preempted*, meaning paused and returned to the queue, with no overhead or loss of work. We describe preemption in more detail in Section 5.3.2.

5.2.2 The M/G Stochastic Arrival Process

The arrival process specifies when jobs arrive, what each job's size is, and what each job's label is. We first address arrival times and sizes before moving on to labels. We use the term *M/G arrivals* to refer to the arrival process described below, and we call a queueing system with M/G arrivals an *M/G system*.

The name "M/G/1" is actually formal queueing theory notation [67], with the symbol

in each of the three positions communicating something about the system, the first two of which concern the arrival process.

- The "M" in the first position means that *job arrival times* are generated by a *memoryless* process, specifically a Poisson process with constant rate. We denote this *arrival rate* by λ.
- The "G" in the second position means that *job sizes* are drawn from a *general* distribution. We denote this *size distribution* by S.
- The "1" in the third position means that there is a single server.

Both λ and S are nonnegative by definition, and to avoid dwelling on uninteresting corner cases, we assume that both are strictly positive.

It remains to explain how labels fit into this picture. We denote the set of possible labels by $\mathbb{L}$. Each job's label-size pair is drawn from a *label-size distribution* (L, S), with L and S denoting the label and size, respectively.

We assume that the arrival times and label-size pairs of all jobs are mutually independent. We also assume that future arrivals cannot be anticipated in any way, so at any moment in time, the future of the arrival process is independent of all past events. Note, however, that an individual job's label L and size S need not be independent. For example, if jobs are labeled with a noisy size estimate, we would certainly hope that L and S are correlated.[2]

Load and Stability

Given that jobs are arriving randomly over time, we might worry: can the arrivals overwhelm the server and cause the system to be unstable? The key quantity to determine whether this occurs is called *load*, which is

$$\rho := \lambda \mathbf{E}[S].$$

We can interpret load in multiple ways:

- The average rate at which work arrives is ρ.
- The average number of arrivals that occur while serving a job is ρ.
- If the system is stable, the fraction of time the server is busy is ρ.

Each of these perspectives makes it clear that we must have $\rho \leq 1$ to have any hope of stability. It turns out that for the M/G/1 to converge to some *steady-state* distribution, meaning stochastic equilibrium, it is necessary and sufficient to have the strict inequality $\rho < 1$, so we assume this throughout (§ 5.6.1). Unless otherwise stated, we assume that each queueing system we study is in steady state.

An M/G arrival process is uniquely characterized by its arrival rate λ and label-size distribution (L, S). Equivalently, one may specify the load ρ instead of the arrival rate, as one can recover the arrival rate as $\lambda = \rho / \mathbf{E}[S]$. We use the latter convention throughout. In particular, when we discuss a quantity varying as a function of changing load, unless

[2]Here we slightly abuse notation by conflating the label and size *distributions* with *random variables* that represent an individual job's label and size. As discussed in Section 5.6.2, we do so throughout this thesis.

otherwise specified, we mean that only the arrival rate is changing. For example, the *heavy-traffic limit* is the $\rho \to 1$ limit, namely the limit as the system approaches but does not quite reach instability.

5.2.3 Labels and What the Scheduler Knows

It is natural to assume that the scheduler knows each job's age, because ages can be tracked in real time. We also assume that the scheduler knows each job's label. We call the pair (ℓ, a) of a job's label ℓ and age a the job's *state*. A job's state thus encodes much of what the scheduler knows about the job.[3] We generally do *not* assume that the scheduler knows any job's size or remaining work unless they can be deduced from the job's state.

However, even when the scheduler does not know a job's size, it may have have *partial information* about the job's size based on its state. Consider a job with label ℓ that just arrived and has age 0. From the scheduler's perspective, the job's size is an unknown random variable drawn from the *label-conditional size distribution*

$$S_\ell \coloneqq (S \mid L = \ell).$$

More generally, if the job has age a, then from the scheduler's perspective, its remaining work is an unknown random variable drawn from the *state-conditional remaining work distribution*

$$S_{\ell,a} \coloneqq (S_\ell - a \mid S_\ell > a) = (S - a \mid L = \ell, S > a).$$

Of course, the scheduler needs to know the label-size distribution (L, S) to compute the above conditional distributions. We assume throughout that (L, S) is indeed known, as one can imagine inferring it from past data. However, in practice, such inference would yield only an approximation of (L, S). While we do consider one scenario in Chapter 12 where the scheduler knows (L, S) only approximately, scheduling when (L, S) is only approximately known is area with many open problems.

This thesis is written in a style where we (meaning you, the reader, and me, the author) often inhabit the role of the scheduler. Phrases like "we know each job's size" thus refer to what the scheduler knows.

The Unlabeled Case

In some scenarios, we do not have any relevant information that distinguishes one job from another other than age. This is the case where there is only one label. We call this the *unlabeled* case, and we denote the single *trivial label* by $*$, so $\mathbb{L} = \{*\}$. We often drop $*$ from notation, such as writing S_a instead of $S_{*,a}$.[4]

[3] Specifically, in addition to a job's label and age, the scheduler could keep track of other quantities, such as when the job arrived or how . Our reasons for focusing especially on a job's label and age will become clear in Chapter 6.

[4] Whether S with a single subscript denotes a label-conditional size distribution (S_ℓ for $\ell \in \mathbb{L}$) or a state-conditional remaining work distribution in the unlabeled case (S_a for $a \geq 0$) will always be clear from context.

What Do Labels Represent?

There is a sense in which saying "the scheduler knows each job's label" is slightly misleading. This is because labels are a modeling tool: we are vague about what $\mathbb{L}$ is and what information labels have because we as modelers get to decide. A better way to think about labels is the other way around: "each job's label is what the scheduler knows about it". For example, if we are modeling a system where the geographic origin of each job is known, we would want the set of labels $\mathbb{L}$ to encode that information. We might let $\mathbb{L}$ be the set of U.S. states, as in Figure 5.2.

Throughout this thesis, we typically let the set of labels be whatever is simplest for the scheduling policy we are studying. That is, we do not include irrelevant information in the label. For instance, when studying scheduling policies that use only job sizes and ages to make scheduling decisions, we let each job's label be its size, so $\mathbb{L} = \mathbb{R}_{>0}$, even though other information might be available to the scheduler. See Chapter 6 for examples of how we use different sets of labels for different scheduling policies.

Limitations of Labels

The M/G/1 with labels turns out to be a very flexible model that is suitable for studying a wide array of scheduling scenarios, as we demonstrate with many examples in Chapter 6. With that said, the model has two important limitations:

- Labels must be i.i.d. across jobs (§ 5.2.2). This means that, for instance, a job's label cannot specify what time of day the job arrived.
- Once a job arrives, its label is fixed. The only dynamic part of a job's state is its age, and so a job's state only changes while it is in service. This means that we cannot model scenarios where the scheduler dynamically learns information about jobs while they wait in the queue.

We will partially relax the second limitation in Chapter 14, in which we extend our model such that, roughly speaking, a job's label can change alongside its age while the job is in service. However, models where a job's state changes while waiting in the queue are beyond the scope of this thesis.

5.2.4 Multiserver Analogue: The M/G/k

One of the main topics we study in this thesis is scheduling in multiserver systems. Most of our work on this topic studies the *M/G/k*, the central-queue multiserver analogue of the M/G/1. The only difference between the M/G/1 and M/G/k is the servers:

- The M/G/1 has a single server with service rate 1.
- The M/G/k has $k \geq 2$ servers, each with service rate $1/k$. That is, while a job is in service, its work decreases and age increases at rate $1/k$. A job of size s thus requires ks time in service to complete.

All other aspects of the M/G/k are the same as the M/G/1: we have the same M/G arrival process, and the scheduler still knows each job's label and age.

Using service rate $1/k$ for the M/G/k's servers is, of course, an arbitrary convention. However, it plays an important intuitive role in our analysis. The M/G/k is a notoriously difficult system to analyze directly, even when using the simplest scheduling policies [73, 82]. Our general approach is to analyze the M/G/k by comparing it to an M/G/1 experiencing the same arrival process. Giving both systems total service rate 1 makes this comparison a "fair fight", meaning the systems behave similarly enough that we can gain insight into the M/G/k by studying the considerably simpler M/G/1.

The M/G/k is a central-queue system, so the scheduler is free to assign any job to any server. Most of our multiserver results are proven for the M/G/k. However, a few of our results apply to any system with M/G arrivals (Ch. 8), or even to systems with other arrival processes (Ch. 15).

5.3 Scheduling

The system's *scheduler* decides which jobs to serve at every moment in time. The procedure the scheduler uses to make this decision is called the system's *scheduling policy*, or simply *policy* when it is clear that scheduling is being discussed. Having already described what the scheduler knows about each job (§ 5.2.3), we now give some simple examples of scheduling policies (§ 5.3.1). We then discuss *preemption* and *server sharing*, which feature in several of the example policies, in more detail (§ 5.3.2).

5.3.1 Examples of Scheduling Policies

All of the policies below are described assuming a single-server system like the M/G/1, but most of them have analogues for central-queue multiserver systems like the M/G/k (Ch. 6). We allude throughout our discussion to the *mean response time* metric, which is the average amount of time jobs spend in the system (§ 5.4.1). We discuss mean response times in more detail in Section 5.5.

Policies that Treat All Jobs the Same Way

If queueing systems have a single default scheduling policy, it is unquestionably *First-Come, First-Served (FCFS)*, which serves jobs in arrival order. FCFS is a *nonpreemptive* policy: once a job enters service, it remains in service until it completes. FCFS is appealing in that it seems to treat jobs fairly. However, FCFS suffers from a drawback: a single very large job can "block" the system for a long time, delaying many other jobs. For job size distributions with high variance, this blocking leads to poor mean response time (§ 5.5.1). Other nonpreemptive policies, such as *Last-Come, First-Serve (LCFS)* and *Random Order of Service (ROS)*, suffer from the same issue.[5]

[5] After each completion, LCFS serves the job that most recently arrived, whereas ROS serves a job chosen uniformly at random.

In order to mitigate this blocking issue, a policy must be *preemptive*: it must occasionally *preempt*, or interrupt, the job in service. One of the simplest preemptive policies is *Preemptive Last-Come, First-Served (PLCFS)*, which always serves whichever job most recently arrived. That is, whenever a new job arrives, PLCFS preempts the job currently in service and starts serving the new arrival. Counterintuitively, this frequent preemption can lead to PLCFS having lower mean response time than FCFS for high-variance size distributions [55].

Another policy that is designed to avoid blocking the server is *Processor Sharing (PS)*. This policy takes preemption to the extreme: it serves jobs in round-robin fashion, serving each job for a short time δ each cycle. Specifically, we consider PS in the $\delta \to 0$ limit, which creates the effect of *sharing the server equally* among all jobs in the system. That is, if there are n jobs in the system, the server serves each job at rate $1/n$ (§ 5.3.2). By sharing the server, PS avoids getting blocked by any one very large job.

A final policy that treats all jobs the same way is *Least Attained Service (LAS)*. This policy always serves the job of *least age*. If multiple jobs are tied for least age, LAS shares the server equally among the tied jobs.[6] The idea behind the LAS policy is that by prioritizing the jobs with small ages, very small jobs will be served very quickly, without needing to wait behind larger jobs that have been in the system for a long time.

Policies with Multiple Classes of Jobs

There are many systems where some jobs are more important than others for one reason or another. Perhaps some requests are especially urgent, or perhaps some clients pay extra for expedited service. One way to model such a system is to give each job a priority *class*. In the simplest version, there are n classes of jobs $1, \ldots, n$. Class 1 has priority over class 2, which has priority over class 3, and so on.

Two natural policies for scheduling in systems with priority classes are *Preemptive Priority (P-Prio)* and *Nonpreemptive Priority (NP-Prio)*. Both policies prioritize jobs according to their class, and both serve jobs in arrival order within each class. The difference between the policies is when they make scheduling decisions.

- P-Prio makes scheduling decisions *continuously*. It will thus preempt a job of class i if a job of class $j < i$ arrives.
- NP-Prio makes scheduling decisions *at discrete points in time*, namely when jobs arrive while the server is idle and when jobs depart while the queue is nonempty. If a job of class i is in service when a job of class $j < i$ arrives, NP-Prio completes the class i job then begins serving whichever job has the best priority. This next job might be the class j job, but it could be a different job with even better priority.

The advantage of P-Prio is that it is stricter in its prioritization: each job can essentially

[6] This sharing arises naturally in the following way. Suppose LAS repeatedly served the job of least age for an interval of length δ, breaking ties arbitrarily. Given multiple jobs tied for least age, this version of LAS would alternate between them, because being served for δ time increases a job's age by δ. The $\delta \to 0$ limit thus results in sharing the server, analogous to the situation with PS.

pretend that lower priority jobs don't exist. The advantage of NP-Prio is that it does not require preemption to implement.

When modeling systems with multiple classes, one typically lets the labels be the set of classes $\mathbb{L} = \{1, \ldots, n\}$, with each job labeled by its class. However, one would use a larger set of labels if each job carried additional pertinent information in addition to its class.

We have described P-Prio and NP-Prio for a finite number of classes above, but the same descriptions easily generalize to any totally ordered set of classes.

Policies that Use Job Size Information

It turns out that to achieve low mean response time, it is crucial to use information about each job's size. The intuition is that if the scheduler serves a small job before a large job, that causes a small amount of waiting, whereas serving the large job before the small job causes a large amount of waiting.

One way to prioritize jobs by size is to use P-Prio and NP-Prio, using a job's size as its class and prioritizing smaller sizes. The resulting policies are more commonly called *Preemptive Shortest Job First (PSJF)* and *Shortest Job First (SJF)*, respectively. For most size distributions, PSJF has lower mean response time than SJF, especially when the size distribution has high variance [55].

However, it is possible to do even better than PSJF when it comes to minimizing mean response time. PSJF prioritizes each job using its *initial* amount of work, namely its size. But this may be different than the amount of work the job *currently* has, namely its remaining work. When deciding the order in which to serve two jobs, we cause the least amount of waiting by considering the jobs' remaining work, as opposed to their sizes.

The policy that always serves the job of least remaining work is called *Shortest Remaining Processing Time (SRPT)*. A classic result in scheduling theory is that SRPT minimizes mean response time in single-server queueing systems under very general conditions [116], even when arrivals are adversarial rather than stochastic (§ 5.1.3).

5.3.2 Preemption and Server Sharing

Several of the scheduling policies described in Section 5.3.1 preempt jobs, with PS and LAS using very frequent preemption to share the server between multiple jobs. We take a moment here to describe more precisely the assumptions we make about preemption and server sharing.

Preemption Is Free

We assume that when the scheduler preempts a job to begin serving a different job, the switch occurs with no overhead and no loss of work.

- *No overhead* means that preemption is immediate: the server switches instantly from serving one job to serving another.

- *No loss of work* means that when a job is preempted, its remaining work and age are unaffected.

Our default assumption is that the scheduler may preempt jobs at any time, though we occasionally make more restrictive assumptions (Chs. 9, 14, and 16).

Server Sharing Follows from Free Preemption

Our assumptions about preemption allow the server to rapidly switch between a number of jobs, effectively sharing a server between multiple jobs. It turns out that the simplest approach to modeling this phenomenon is to explicitly allow the scheduler to share the server between multiple jobs. Formally, we allow a server with service rate $u \geq 0$ may serve any number of jobs n, serving the ith such job at rate $v_i \geq 0$. Serving a job at rate v decreases its remaining work and increases its age at rate v. The total job service rate must not exceed the server's service rate, meaning $\sum_{i=1}^{n} v_i \leq u$.

For simplicity of language, we often use phrases that assume that servers serve one job at a time. For example, we often discuss "the job in service" at a server. These phrases should be understood as meaning what they would if the scheduler really were alternating rapidly between serving different jobs to simulate sharing. For example, "the job in service" at a server refers to a randomly chosen job among those that are currently sharing the server, with with each job i being chosen with probability proportional to its service rate v_i.

Server Sharing with Multiple Servers

Our above discussion of server sharing is sufficient for systems where each queue is connected to its own server, such as the M/G/1. However, there is an additional subtlety we must consider for central-queue multiserver systems, such as the M/G/k. The M/G/k has k servers, each with service rate $1/k$. We can think of the scheduler as assigning each job a service rate at every moment in time. There are two types of constraints on what the scheduler's actions:

- Because the total service rate of all the servers is 1, it must be that the total service rate of all jobs is at most 1.
- Because we intend server sharing to be the limit of rapidly alternating which job is in service at each server, it must be that each job's service rate is at most $1/k$. Otherwise, we would be effectively allowing two servers to serve the same job at the same time.

5.4 Queueing Metrics

5.4.1 Response Time

One of the most important aspects of a queueing system's performance is jobs' *response times*. A single job's response time is the amount of time it spends in the system between

arrival and departure. This includes time queueing and time in service. In most queueing systems, lower response time, and in particular less time spent queueing, is generally desirable.

Of course, the queueing models we study involve many jobs arriving over time, so the natural object to study is a system's *response time distribution*, denoted T. Intuitively, T is the distribution of response times we would observe by watching a very large number of jobs pass through the system. We give two more precise definitions in Section 5.4.2.

There are multiple response time metrics one might hope to optimize, including:

- *Mean response time* $\mathbf{E}[T]$: the average amount of time jobs spend in the system.
- *Response time tail probability* $\mathbf{P}[T > t]$: the probability jobs have response time larger than a parameter t.
- *Response time quantile* $t_q = \min\{t \geq 0 \mid \mathbf{P}[T \leq t] \geq q\}$: the threshold t_q such that (at most) a q fraction of jobs have response time at most t_q.

Which of these metrics is most important depends on the system being modeled. For example, when designing for the goal of preventing especially large response times, tail probabilities and quantiles would be more important metrics to optimize than the mean.

In this thesis, we focus primarily on mean response time and the *asymptotic response time tail*, meaning the asymptotic behavior of $\mathbf{P}[T > t]$ in the $t \to \infty$ limit. While we do not discuss quantiles specifically, results about the asymptotic tail imply results about the $q \to 1$ limiting behavior of quantiles.

What Factors Affect Response Time?

For concreteness, consider an M/G/1 with labels. Its response time distribution T is a function of the following factors:

- The M/G arrival process, namely the arrival rate λ and label-size distribution (L, S).
- The scheduling policy. For example, we have seen in Section 5.3.1 that scheduling can affect mean response time $\mathbf{E}[T]$.

Beyond the M/G/1, other aspects of the queueing system's architecture can affect its response time. For example, in an M/G/k, the number of servers k is an important factor. To reduce clutter, we often leave T's dependence on the various factors that affect it implicit in our notation. When we wish to make some dependence explicit, we write it as a subscript. For the most part, this means specifying the scheduling policy: T_π denotes the response time distribution under policy π.

The above discussion applies not just to response time, but also to many other quantities that depend on the system parameters, such as the distribution of the number of jobs in the system (§ 5.4.3). In particular, we use the same subscript notation convention.

Some of the above factors affect response time in obvious ways. For instance, increasing the arrival rate essentially always increases response times. But other effects are less straightforward. For instance, in an M/G/1, suppose we change S by increasing its variance $\mathbf{Var}[S]$ while keeping its mean $\mathbf{E}[S]$ fixed.

- Under FCFS, mean response time $\mathbf{E}[T]$ *increases* as a function of $\mathbf{Var}[S]$ (§ 5.5.1).

- Under SRPT, the specifics of S matter, but as a rule of thumb, mean response time $\mathbf{E}[T]$ *decreases* as a function of $\mathbf{Var}[S]$ [146].

The fact that scheduling can counterintuitively affect response time is one of the main reasons I believe scheduling theory is so important (Ch. 1).

5.4.2 Characterizing Response Time with PASTA

We introduced T earlier as the distribution of response times one would observe by watching a large number of jobs pass through the system. More formally, if T_n is the distribution of the first n jobs' response times, then T_n converges in distribution to T in the $n \to \infty$ limit (§ 5.6.1).

For systems with M/G arrivals, we can define T in another way. Consider a *generic job*, meaning one with label-size pair drawn from (L, S), arriving to a steady-state system. This new arrival's response time is a random variable distributed as T.

The equivalence of the above two ways of defining T follows from the *Poisson Arrivals See Time Averages (PASTA)* property [150] and our assumption that the systems we study are in steady state (§ 5.6.1). While the first perspective is the reason we really care about T, the second perspective is often the more useful one when it comes to characterizing T theoretically (Ch. 7). In particular, we often compute T by way of the *conditional response time distribution*

$$T(\ell, s) \coloneqq (T \mid L = \ell, S = s),$$

which is the response time distribution of a job with given label ℓ and size s arriving to a steady-state system.[7]

5.4.3 Number-in-System

Another important aspect of a queueing system's performance is the number of jobs in the system, which we call the *number-in-system*, denoted N.

By default, N refers to the number-in-system of a steady-state system, but we occasionally look at non-steady-state scenarios. We use the same convention for other quantities defined in terms of the system's current state (§ 5.6.1).

Just as one can define multiple metrics from the response time distribution T, one can define multiple metrics based in the number-in-system distribution N. In this thesis, we primarily concern ourselves with the *mean number-in-system* $\mathbf{E}[N]$. This is largely because the mean number-in-system is related to mean response time via *Little's law* [84], which states

$$\mathbf{E}[N] = \lambda \mathbf{E}[T].$$

Little's law holds for a wide variety of queueing systems, even those with arrival processes other than M/G arrivals, so long as there is a well-defined average arrival rate λ. Its main

[7] The definition of $T(\ell, s)$ contains a mild abuse of notation: we use the notation for *distributions* to stand for *random variables* with those distributions. We use similar abuse of notation throughout this thesis, as discussed in more detail in Section 5.6.2.

implication is that given a fixed arrival process, minimizing mean response time and minimizing mean number-in-system are equivalent objectives.

5.4.4 Weighted Response Time and Holding Cost

Metrics based on response time T or number-in-system N are limited in that they treat all jobs the same way. However, there are various reasons we might want a performance metric that gives different jobs different values.

- In a system with multiple priority classes, we might care more about the response times of jobs with better priority.
- The amount of delay that is acceptable for a job may depend on its size. For example, delaying a 100-minute job for 10 minutes may barely be noticed, whereas delaying a 1-minute job for 10-minutes may feel like a major delay.

In these situations, a metric that assigns different jobs different weights may be more appropriate. One notable example of a weighted response time metric is *slowdown* T/S, the ratio of a job's response time to its size.

More generally, we can consider each job to have a *holding cost*, and we might aim to minimize the mean *system holding cost*, namely the mean total holding cost of all jobs in the system. We do not discuss holding cost in detail until Part III, so we defer further details to Chapter 14.

5.5 M/G/1 Crash Course

We have thus far discussed a variety of scheduling policies (§ 5.3) and performance metrics (§ 5.4). One of the main goals of this thesis is to answer many versions of the following question:

> How well does a given scheduling policy perform on a given metric in a given queueing system?

As a warm-up to answering many difficult instances of this question in the rest of the thesis, we take a moment to answer some simple versions of it by reviewing the M/G/1 response times of two basic scheduling policies: FCFS and PLCFS. With that said, the true purpose of this section is to introduce two key queueing theory concepts, *system work* (§ 5.5.1) and *busy periods* (§ 5.5.2). We use these concepts not just to analyze FCFS and PLCFS here but to analyze a great deal of scheduling policies throughout this thesis.

5.5.1 System Work

As discussed in Section 5.4.2, one way to analyze a system's response time is to imagine a job arriving to a steady-state system. To that end, consider a generic job arriving to a steady-state M/G/1 using FCFS. We can divide the job's response time into two parts: time in service and time queueing.

- The amount of time the job spends in service is simply its size S.
- Because jobs are being served in arrival order, the amount of time the job spends queueing is the total remaining work of all the other jobs that in the system when it arrives. We call this quantity the *system work*, denoted W.

Because future arrivals are independent of the past (§ 5.2.2), the arriving job's size S and system work W it observes are independent, so we can write response time as an independent sum

$$T_{\text{FCFS}} =_{\text{st}} W + S, \tag{5.1}$$

where $=_{\text{st}}$ denotes equality of distributions (§ 5.6.2).

Of course, we have only shifted the problem of determining FCFS's response time to the problem of characterizing the steady-state system work W. Fortunately, characterizations of W are among the earliest results obtained for the M/G/1. We give a derivation of our own in Chapter 8, but for now, we focus on just its mean, which is [55]

$$\mathbf{E}[W] = \frac{\frac{\lambda}{2}\mathbf{E}[S^2]}{1-\rho} = \frac{\frac{\rho}{2}\mathbf{E}[S] + \frac{\lambda}{2}\mathbf{Var}[S]}{1-\rho}.$$

We thus obtain a formula for FCFS's mean response time:

$$\mathbf{E}[T_{\text{FCFS}}] = \frac{\frac{\lambda}{2}\mathbf{E}[S^2]}{1-\rho} + \mathbf{E}[S] = \frac{\left(1-\frac{\rho}{2}\right)\mathbf{E}[S] + \frac{\lambda}{2}\mathbf{Var}[S]}{1-\rho}. \tag{5.2}$$

Takeaways for FCFS's Response Time

The most important things to notice about $\mathbf{E}[T_{\text{FCFS}}]$ is how it is affected by load ρ and job size variance $\mathbf{Var}[S]$.

- As load ρ approaches 1, which is the maximum load under which the system is stable (§ 5.6.1), mean response time $\mathbf{E}[T_{\text{FCFS}}]$ diverges. This occurs under any scheduling policy, although some have a growth rate slower than FCFS's $\Theta\left(\frac{1}{1-\rho}\right)$ (§ 2.1.2).[8]
- As job size variance $\mathbf{Var}[S]$ increases while $\mathbf{E}[S]$ and ρ remain fixed, mean response time $\mathbf{E}[T_{\text{FCFS}}]$ increases. In particular, arbitrarily large variance can yield arbitrarily large mean response times, even at low loads. We will soon see that not all scheduling policies suffer from this issue (§ 5.5.2).

System Work Is Scheduling-Invariant

Above, we reduced the problem of characterizing FCFS's response time T_{FCFS} to the problem of characterizing the steady-state system work W. Why is this a helpful step?

One of the main reasons system work is such a helpful concept is that in single-server systems like the M/G/1, system work is *scheduling-invariant*, meaning the same under

[8] Recall from Section 5.2.2 that when we discuss a quantity varying as a function of changing load, we mean that the arrival rate is changing while the size distribution remains fixed.

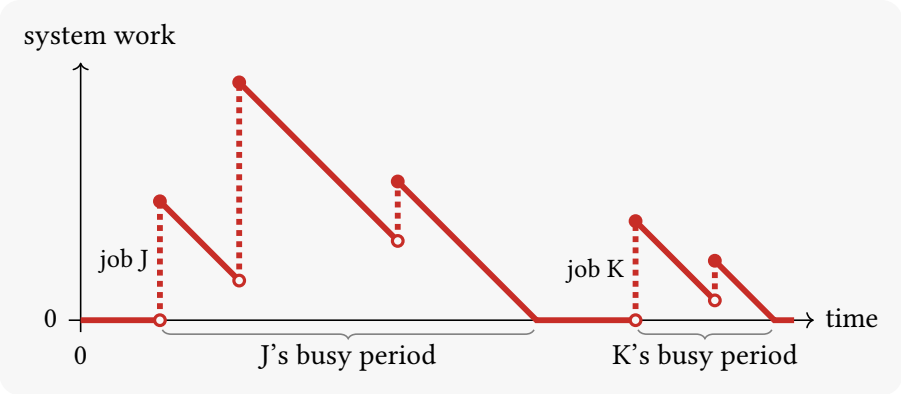


Figure 5.3. System work over time under a non-idling scheduling policy in an M/G/1. Each arriving job causes an upward jump, increasing system work by the arriving job's size. During busy periods, meaning whenever system work is nonzero, system work decreases at rate 1.

any scheduling policy, under a very mild condition: the system must be using a *non-idling* policy, meaning one that never leaves the server idle while system work is nonzero. This is because system work undergoes the same upward jumps (due to arrivals) and downward slopes (due to service) under any non-idling policy, as illustrated in Figure 5.3.

In the interest of transparency, I should mention that analyzing T_{FCFS} and analyzing W are tasks of essentially equivalent difficulty, as witnessed by (5.1). But the general strategy of reducing analysis of a scheduling policy's response time to scheduling-invariant quantities will serve us well throughout this thesis, including in the very next section.

5.5.2 Busy Periods

We now move on to analyzing the response time of PLCFS. Just as system work helped us analyze FCFS, another queueing theory concept will help us analyze PLCFS: *busy periods*. Roughly speaking, a busy period is an interval of time during which a system has nonzero system work, as illustrated in Figure 5.3. After defining busy periods more precisely, we see they are exactly what we need to understand to analyze PLCFS.

Somewhat confusingly, the term "busy period" actually refers to multiple related but slightly different concepts. In an attempt to clarify, we introduce some more specific terms below. Consider a particular job J that at some point arrives to an M/G/1. The *busy period started by job J*, or simply *J's busy period*, consists of the following two entities:

- The busy period's *tree* is the following random rooted tree. Let every job be a vertex, and draw a directed edge from a job K to another job L if L arrives while K is in service.[9] The busy period's tree is all vertices accessible from J, as illustrated in Figure 5.4.

[9] If the server is sharing between multiple jobs when a job L arrives (§ 5.3.2), we choose which job from which to draw the edge to L randomly from among those in service, picking each job with probability proportional to its service rate when L arrives.

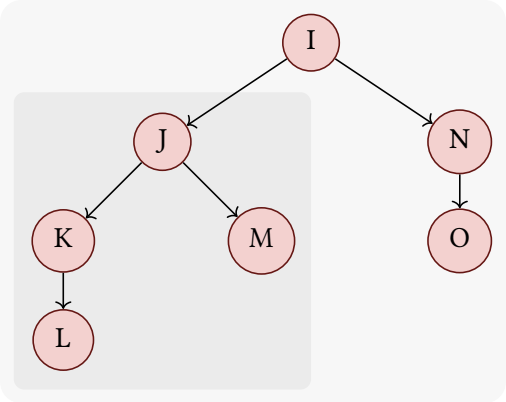


Figure 5.4. Tree of job I's busy period. An edge from a job A to another job B indicates that B arrives while A is in service. Job J's busy period's tree (gray box) is the subtree rooted at J.

- The busy period's *work* is the total size of all jobs in the busy period's tree. We can divide work into *initial work*, which is J's size, and *new work*, which is the total size of the other jobs in the tree.

When unambiguous, we occasionally use just "busy period" to refer to a busy period's tree or work. Some busy periods are highlighted in Figure 5.3.

We write $B(s)$ for the distribution of the work of a busy period started by a job of size s, and we write $B \coloneqq B(S)$ for the work of a *generic busy period*, meaning a busy period started by a generic job with size distributed as S. Characterizations of $B(s)$ and B are standard results in queueing theory. For now, we show just their means [55]:

$$\mathbf{E}[B(s)] = \frac{s}{1-\rho},$$
$$\mathbf{E}[B] = \frac{\mathbf{E}[S]}{1-\rho}.$$

PLCFS Response Time Is a Busy Period

What is PLCFS's response time distribution? Recall from Section 5.3.1 that PLCFS is the policy that always serves whichever job most recently arrived. By PASTA (§ 5.4.2), it suffices to consider the response time of a generic job, which we will call J, arriving to a steady-state M/G/1.

What is job J's response time? When J arrives, it becomes the most recently arrived job in the system, so it immediately begins service. However, new arrivals may preempt J before it completes, and still newer arrivals may preempt those new arrivals, and so on. The key observation is that PLCFS prioritizes another job over J if and only if the other job

is in J's busy period's tree. This means J's response time is its busy period's work, so

$$T_{\text{PLCFS}} =_{\text{st}} B,$$
$$\mathbf{E}[T_{\text{PLCFS}}] = \frac{\mathbf{E}[S]}{1-\rho}.$$

In particular, note that PLCFS's mean response time does *not* depend on the job size distribution beyond its mean. This is in contrast to FCFS, which, as shown in (5.2), depends also on the variance.

Busy Periods are Scheduling-Invariant

One important feature of busy periods in the M/G/1 is that they are *scheduling-invariant*. Strictly speaking, as we explain below, busy periods are only scheduling-invariant in a distributional sense, but this is adequate for our purposes. Busy periods thus join system work as scheduling-invariant quantities that prove useful for a variety of analyses throughout this thesis.

To consider how scheduling decisions affect the busy period of some job J, let Y_1 be the list of label-size pairs of arrivals that occur while J is in service under some scheduling policy, and let Y_2 be the same for another policy. On one hand, changing when J is served will affect which jobs arrive while serving J, so $Y_1 \neq Y_2$ in general. This means busy periods are not scheduling-invariant for a fixed arrival sequence. But on the other hand, we do not have a fixed arrival sequence: M/G arrivals are a stochastic process. In particular, arrivals are distributed, roughly speaking, "time-invariantly". The upshot is that under M/G arrivals, we have $Y_1 =_{\text{st}} Y_2$, meaning Y_1 and Y_2 have the same distribution. Applying this observation recursively, we find that the distribution of J's busy period is scheduling-invariant.[10]

An important consequence of scheduling invariance of busy periods is that we can broaden our view on what can start a busy period. Instead of limiting ourselves to busy periods started by a job, we can consider busy periods started by an *amount of initial work*. This work can come from multiple jobs, and it can even include parts of jobs. Busy periods started by initial work v are essentially the same as busy periods started by a job of size v.[11] In particular, such a busy period's work distribution is also $B(v)$.

[10] In the queueing theory literature, scheduling invariance of busy periods under M/G arrivals is sometimes phrased as a system having *job-linked arrivals* [44, Section 4.8]. The idea is that we can imagine that because the arrivals during a job's service have a scheduling-invariant distribution, we may view the arrivals during a job as being predetermined.

[11] The only difference is that when discussing the busy period's tree, the root represents the initial work, as opposed to representing a specific job.

5.6 Additional Preliminaries

5.6.1 Convergence to Stochastic Equilibrium

We assume that the systems we study in this thesis are ergodic and converge to a stochastic equilibrium. To make this precise, we need to specify what the state of the system is.

Recall that we write N for the number of jobs in the system. The state of the system consists of the following:

- The number of jobs N.
- The state (L_i, A_i) of the ith job in the system for all $i \in \{1, \ldots, N\}$.
- Any additional state the scheduling policy needs to maintain to make its scheduling decisions.

Provided the scheduling policy's state is detailed enough, an M/G system with the above state description becomes a continuous-time Markov process.

All the specific M/G systems we study are either M/G/1 or M/G/k queues, and their scheduling policies do not require tracking any additional state at all. This makes them (non-lattice) renewal processes, so $\rho < 1$ ensures positive recurrence and thereby our ergodicity and equilibrium assumptions. However, the ergodicity and equilibrium assumptions are implicit in results that apply to all M/G systems.

5.6.2 (Abuse of) Notation for Distributions and Random Variables

Our notation throughout this thesis does not distinguish between two related but subtlety different concepts:

- *distributions*, namely probability measures; and
- *random variables*, which have distributions.

Given a distribution V, we also write V for a new random variable with that distribution. Unless otherwise specified, this new random variable is independent of all other random variables. A notable exception is the label L and size S of a generic job, which we assume to be correlated according to the joint label-size distribution (L, S).

In the other direction, given a random variable V to do with the system state (e.g. system work W), we also write V to denote its steady-state distribution. More generally, random variables that depend on the system state refer to steady-state systems unless otherwise noted.

We use the following notation for some common distributions:

- Bernoulli(p): a "probability-p coin flip" which is 1 with probability p and is 0 otherwise.
- Geo(p): geometric distribution with parameter p and support at 0. This is the number of independent probability-p coin flips before a 1 occurs, not counting the final 1.
- Normal(μ, σ): Gaussian distribution with mean μ and standard deviation σ.

We write $=_{\text{st}}$ for equality in distribution. That is, $U =_{\text{st}} V$ means $\mathbf{P}[U > t] = \mathbf{P}[V > t]$ for all $t \in \mathbb{R}$. Similarly, $\leq_{\text{st}}$ denotes the usual stochastic partial order, so $U \leq_{\text{st}} V$ means $\mathbf{P}[U > t] \leq \mathbf{P}[V > t]$ for all $t \in \mathbb{R}$.

5.6.3 Excess Distributions

In a steady-state queueing system, there is some probability the server is in the middle of serving a job. Analyzing the system often involves determining the distribution of the amount remaining work of the job in service. This is given by the *excess* of the job size distribution S, as defined below.

Definition 5.1. The *excess* of distribution V, denoted $\mathcal{E}V$, is the distribution defined by the tail function

$$\mathbf{P}[\mathcal{E}V > t] \coloneqq \frac{1}{\mathbf{E}[V]} \int_t^\infty \mathbf{P}[V > u] \, \mathrm{d}u.$$

When we wish to denote n independent samples of the excess distribution $\mathcal{E}V$, we denote them $(\mathcal{E}V)_1, \ldots, (\mathcal{E}V)_n$.

5.6.4 Laplace-Stieltjes Transforms

Definition 5.2. The *Laplace-Stieltjes transform (LST)* of distribution V, denoted $\mathcal{L}[V]$, is the function

$$\mathcal{L}[V](\theta) \coloneqq \mathbf{E}[\exp(-\theta V)].$$

Below, we review the well-known LST formulas for the excess of a distribution, the system work in an M/G/1, and an M/G/1 busy period's work.

Proposition 5.3. *Let V be a nonnegative distribution. The LST of the excess $\mathcal{E}V$ is*

$$\mathcal{L}[\mathcal{E}V](\theta) = \frac{1 - \mathcal{L}[V](\theta)}{\theta \mathbf{E}[V]}.$$

Proposition 5.4. *The LST of the system work W in an M/G/1 is*

$$\mathcal{L}[W](\theta) = \frac{1 - \rho}{1 - \rho \mathcal{L}[\mathcal{E}S](\theta)}.$$

Proposition 5.5. *The LST of the work of a busy period started by a random amount of initial work V is*

$$\mathcal{L}[B(V)](\theta) = \mathcal{L}[V](\eta(\theta)),$$

where $\eta(\theta)$ is the principal solution to[12]

$$\eta(\theta) = \theta + \lambda\big(1 - \mathcal{L}[S](\eta(\theta))\big).$$

[12]Specifically, when $\theta > 0$, there is a unique positive real solution, and taking the analytic continuation covers other values of θ.

While the busy period LST is given in terms of the implicitly defined $\eta(\theta)$, the derivatives of busy period LSTs yield equations that can be solved to obtain explicit expressions for any integer moment.

One can show using Proposition 5.5 that busy periods are additive.

Corollary 5.6. *Busy periods are additive in the following sense: for any independent nonnegative random variables U and V,*

$$B(U+V) =_{\text{st}} B(U) + B(V),$$

where the two busy periods on the right-hand side are independent.

5.6.5 Neglected Measure-Theoretic Technicalities

This thesis plays somewhat fast and loose with measure theory. For example, we ought to assume that the set of labels $\mathbb{L}$ is a measure space, we should maybe assume that S has no singular component, and we should verify that various functions on $\mathbb{L} \times \mathbb{R}_{>0}$ are measurable. There are even more measure-theoretic technicalities to consider for the more general job model we use in Part III.

I find it easier to communicate the main ideas in this thesis without worrying about measure-theoretic technicalities, and I hope that most readers will appreciate this choice. Those concerned about the details should rest assured that in a great many cases of practical interest, the technicalities can be worked out easily. More conservatively, all of the sufficiently general theoretical results in this thesis, particularly those in Part III, can be understood as "reductions to measure theory". See Chapter 14 for further discussion.

5.6.6 Other Notes on Notation and Terminology

We conclude with some miscellaneous notes on notation and terminology.

- We use $f(x-)$ and $f(x+)$ to refer to right and left limits of a function f at x.
- We use $(x)^+ \coloneqq \max\{x, 0\}$ to denote the positive part of x.
- When it can be done without loss of clarity, we often omit parentheses around objects like tuples to reduce clutter. For instance, for a function f applied to a tuple (x, y), we might write $f(x, y)$ instead of $f((x, y))$.
- The terms "increasing", "decreasing", and "monotonic" are meant in their weak sense unless preceded by "strictly".
- In addition to the usual numbering of theorems, lemmas, definitions, etc., we also have several numbered *policies*, which we use for definitions of scheduling policies.

Most importantly, Appendix A contains an index of notation.

Chapter 6

SOAP Policies: Describing Scheduling with Rank Functions

There is a large body of literature analyzing the response time distributions of various scheduling policies in the M/G/1. Unfortunately, this prior work is lacking in two significant ways.

- The M/G/1 scheduling literature is limited to relatively "simple" policies. This is a problem because more "complex" policies naturally arise in practice, such as when scheduling with uncertain job sizes or preemption limitations.
- The M/G/1 scheduling literature, for the most part, analyzes policies one at a time, with occasional analyses of classes of related policies [74, 146]. This is a problem because there is a huge space of possible scheduling policies

The next few chapters introduce *SOAP: Schedule Ordered by Age-based Priority*, a framework that takes a big step towards solving both of the above problems. The SOAP framework consists of two things:

- *SOAP policies*: a broad class of scheduling policies that includes many complex policies we would like to analyze.
- *SOAP analysis*: a generic analysis of the response time distribution in an M/G/1 using any SOAP policy.

This chapter introduces SOAP policies, and Chapters 7 and 8 present the SOAP analysis.

The key idea behind SOAP is to define a *unifying language* for expressing scheduling policies. We want this language to be flexible enough to describe a wide variety of policies, but we also want it to be restrictive enough to admit a generic analysis of any policy that can be expressed in the language. We thus begin this chapter by introducing a new unifying language which provides a good balance of these features: *rank functions* (§ 6.1). A SOAP policy is any policy that can be represented as a rank function. We spend most of the chapter showcasing the breadth of the class of SOAP policies, using rank functions to express a large fraction of the simple policies that have been analyzed previously (§ 6.2), as well as many more complex policies that were never analyzed prior to SOAP (§ 6.3). We conclude with a discussion of what policies are not SOAP (§ 6.4).

Throughout this section, by default, we discuss scheduling in the M/G/1 with labels (§ 5.2), though we will briefly discuss SOAP policies in the M/G/k (§ 6.1.5).

This chapter is based on material from Scully et al. [124].

6.1 What Is a SOAP Policy?

Given that essentially all of Part II relies on the definition of SOAP policies, let us immediately answer the question posed in the section title.

A *SOAP policy* is a scheduling policy that works by always assigning each job a *rank*, or numerical priority (lower is better), based on just its *label* and *age* (§ 5.2.3). A SOAP policy is thus specified by a *rank function*

$$\text{rank} : \overbrace{\mathbb{L}}^{\text{label}} \times \overbrace{\mathbb{R}_{\geq 0}}^{\text{age}} \to \overbrace{\mathbb{R}_{\geq 0}}^{\text{rank}}$$

All SOAP policies thus work in the same way:

> At all times, serve the job of *minimal rank*, breaking ties in FCFS order.

The differences between SOAP policies thus rest entirely in the choice of rank function.

We give a more formal definition of SOAP policies later in this section (§ 6.1.2). The rest of the section motivates this definition (§ 6.1.1), discusses its significance (§ 6.1.3), and clarifies some technical concerns (§ 6.1.4).

6.1.1 Motivation: General, But Not Too General, Policy Class

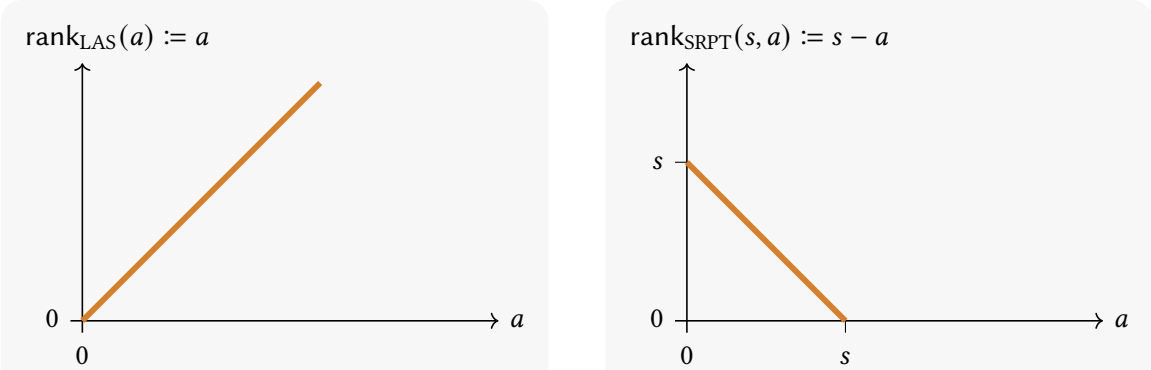
There are many scheduling policies that work by assigning each job a numerical priority, then serve the job with least or greatest numerical priority. Such policies are known as *index policies*. For example, LAS and SRPT (§ 5.3) are both index policies: LAS always serves the job of least age, and SRPT always serves the job of least remaining work.

Can we hope for a generic analysis of all index policies? Unfortunately, this is almost certainly too much to ask for, because any policy can be framed as an index policy by "retroactively" assigning indices: give the job the policy would serve priority 0, and give jobs in the queue priority 1, then serve the job with the least numerical priority. This construction tells us that for the notion of index policies to be helpful, we need to specify what a job's priority is allowed to depend on.

We can thus reduce our search for a suitable unifying language for scheduling policies to a single question: to balance the concerns of modeling flexibility and theoretical tractability, *what should we allow a job's priority to depend on?* SOAP provides a new answer to this question: letting a job's priority depends only on its *label* and *age* turns out to strike a good balance. The rest of this section defines the class of SOAP policies in more detail.

6.1.2 Defining SOAP Policies and Rank Functions

SOAP policies are the subset of index policies where a job's priority depends only on its label and age. We call a job's priority its *rank*, and we use the convention that *lower rank means better priority*, so a SOAP policy always serves the job of minimal rank. We formalize this below.



(a) Rank function of LAS.

(b) Rank function of SRPT for job of size s.

Figure 6.1. Rank functions representing LAS (Pol. 6.4) and SRPT (Pol. 6.10), both of which are SOAP policies.

Definition 6.1.

(a) A *rank function* on label set $\mathbb{L}$ is a function $\text{rank} : \mathbb{L} \times \mathbb{R}_{\geq 0} \to \mathbb{R}_{\geq 0}$ that maps each label-age pair (ℓ, a) to a number $\text{rank}(\ell, a)$, which we call a *rank*. If a job is in state (ℓ, a), then we call $\text{rank}(\ell, a)$ that job's rank.
- In the unlabeled case, meaning when $\mathbb{L}$ is a singleton (§ 5.2.3), we drop the trivial label from the notation, writing just $\text{rank}(a)$. We also call the rank function itself *unlabeled* in this case.

(b) A *SOAP policy* is a scheduling policy π such that there exists some rank function rank_π on some label set such that π always serves the job assigned minimal rank by rank_π, breaking ties in FCFS order by serving whichever job arrived earliest among the tied jobs. We say that rank_π *represents* policy π.
- When a job J has priority over another job K, we say J *outranks* K. This happens when J's rank is strictly less than K's, or when J and K have equal ranks but the tiebreaker favors J.
- When we wish to disambiguate with (c) below, we use the term *SOAP policy with FCFS tiebreaking*.

(c) A *SOAP policy with LCFS tiebreaking* is the same as an ordinary SOAP policy as defined in (b), except ties are broken in LCFS order by serving whichever job arrived latest among the tied jobs.

As simple examples, LAS is the SOAP policy represented by $\text{rank}_{\text{LAS}}(a) := a$, and SRPT is the SOAP policy represented by $\text{rank}_{\text{SRPT}}(s, a) := s - a$, where a job's label s is its size. These rank functions are illustrated in Figure 6.1. We give many more examples of SOAP policies in Sections 6.2 and 6.3.

For the most part, whenever we define a SOAP policy π, we specify a canonical rank function rank_π representing it, which we call "the" rank function of π. This is despite the fact that many other rank functions can represent π. For example, we define $\text{rank}_{\text{LAS}}(a) := a$

above, but any strictly increasing function of a represents LAS.

Whenever we are discussing the rank function of a specific SOAP policy π, we typically include π in the subscript, as in rank_π. When discussing a generic rank function of an unspecified SOAP policy, we typically omit the subscript.

6.1.3 "Simple" vs. "Complex" SOAP Policies

As we will see in Section 6.2, many classic scheduling policies that have been previously analyzed the queueing literature are SOAP policies. However, these policies are all relatively simple compared to other SOAP policies. We take a moment here to outline several features rank functions that allow for expressing policies that are significantly more complex than those previously analyzed.

Complexity from Nonmonotonicity

Definition 6.2.

(a) A rank function rank is *increasing* if $\mathsf{rank}(\ell, \cdot)$ is increasing for all $\ell \in \mathbb{L}$. We define *decreasing* rank functions similarly. A rank function is *monotonic* if it is either increasing or decreasing. Otherwise, it is *nonmonotonic*.

(b) A SOAP policy π is *increasing* if it can be represented by an increasing rank function. We define *decreasing* and *monotonic* SOAP policies similarly. If a SOAP policy is not monotonic, meaning it can only be represented by nonmonotonic rank functions, then it is *nonmonotonic*.

Virtually all previously analyzed SOAP policies[1] are monotonic, including all of the examples in Section 6.2. This is likely because, as we explain in Chapter 7, nonmonotonic policies are significantly more difficult to analyze. Nevertheless, nonmonotonic policies arise naturally in many scenarios, such as scheduling with uncertain job sizes (§ 6.3.1) or under preemption limitations (§ 6.3.2). The fact that we can analyze the M/G/1 response time of any SOAP policy, and in particular nonmonotonic ones, is one of the main ways SOAP advances the state of the art in queueing theory.

Complexity from Policy Mixture and Multiple Types of Labels

SOAP policies that were previously analyzed tend to be represented by rank functions where the set of labels $\mathbb{L}$ contains, roughly speaking, one type of label. For instance, LAS uses a single trivial label, and SRPT has all jobs labeled by their size. But what if one only knows the sizes of certain jobs? One could model this scenario using a label set like[2]

$$\mathbb{L} = \{\mathsf{unknown}\} \cup \{\mathsf{known}(s) \mid s > 0\}.$$

[1] While we refer to "previously analyzed SOAP policies" throughout this chapter, we emphasize that such policies were not identified as SOAP policies when they were first analyzed. SOAP policies and the SOAP analysis were not introduced until 2018 [124].

[2] Below, unknown and known are purely symbolic, making $\mathbb{L}$ analogous to an algebraic data type in a programming language like ML, Haskell, and Rust. Being symbolic means unknown is equal to itself and

Using multiple types of labels allows one to express mixtures of policies. For instance, the rank function

$$\mathsf{rank}(\mathsf{unknown}, a) \coloneqq a,$$
$$\mathsf{rank}(\mathsf{known}(s), a) \coloneqq s - a.$$

is a mixture of LAS and SRPT. We are aware of only a few previously analyzed SOAP policies that mix policies together [49, 105], and none use multiple types of labels like in the example above.

Complexity from Multidimensional Ranks

Definition 6.1 defines rank functions to always have codomain $\mathbb{R}_{\geq 0}$. However, one can in principle use a set of ranks with a richer ordering than $\mathbb{R}_{\geq 0}$, such as $\mathbb{R}^2_{\geq 0}$ ordered lexicographically. We are not aware of any previously analyzed SOAP policies that requires multidimensional ranks to represent, but the SOAP analysis we present in Chapter 7 applies essentially verbatim to multidimensional ranks as well. See Scully et al. [124] for more details on multidimensional ranks.

With that said, while one can construct SOAP policies that do require multidimensional ranks, we find that single-dimensional ranks are expressive enough for the policies we wish to study. So for simplicity of presentation, we use $\mathbb{R}_{\geq 0}$ as the set of ranks throughout.

6.1.4 Clarifying Remarks on SOAP Policies and Rank Functions

Rank Ties and Server Sharing

Section 5.3.2 outlines how rank ties in LAS naturally lead to server sharing: when many jobs are tied for minimal rank, whichever job is favored by the tiebreaking rule stops being tied for minimal rank after an instant of service, due to its age increasing. Similar server sharing can happen under any SOAP policy whose rank function at some point increases with age. We outline how this works in detail in Algorithm 6.1. The algorithm is the natural result of imagining that the server has a minimum service quantum δ, then taking the $\delta \to 0$ limit.

Is SOAP Only for the M/G/1 with Labels?

Unless otherwise stated, all the examples in this chapter assume an M/G/1 with labels. While labels are central to the definition of SOAP policies, nothing in the definition explicitly mentions any assumptions about the arrival process. Can SOAP be defined for systems other than the M/G/1 with labels? Extending SOAP to other M/G arrival systems, like the M/G/k, is generally straightforward (§ 6.1.5). But even beyond that, the answer is at

nothing else, and $\mathsf{known}(s)$ is equal only to other terms of the form $\mathsf{known}(s')$ for some $s' > 0$, with equality if and only if $s = s'$. We reserve `typewriter` font for symbolic terms like these. A similar example arises in Policy 6.18.

Algorithm 6.1. SOAP policy tiebreaking with server sharing.

INPUT A rank function rank and a set $\mathcal{J}$ of jobs currently tied for minimal rank.

OUTPUT Subset of jobs in $\mathcal{J}$ to serve, with service rates if sharing the server between multiple jobs.

PROCEDURE

- Let $\mathcal{K}$ be the subset of $\mathcal{J}$ consisting of jobs with decreasing rank, meaning they are in states (ℓ, a) with $\frac{\mathrm{d}}{\mathrm{d}a}\mathsf{rank}(\ell, a) \leq 0$.
- If $\mathcal{K}$ is nonempty, use FCFS tiebreaking within $\mathcal{K}$, meaning serve the job in $\mathcal{K}$ that arrived earliest.
- Otherwise, $\mathcal{K}$ is empty, so share the server among all jobs in $\mathcal{J}$, choosing service rates such that all the jobs' ranks increase at the same rate. That is, serve a job in state (ℓ, a) at rate proportional to $1 \Big/ \frac{\mathrm{d}}{\mathrm{d}a}\mathsf{rank}(\ell, a)$, normalizing such that the total service rate is 1.

least in principle yes: one can imagine a variety of arrival processes, both stochastic and adversarial (§ 5.1.3), where each job is assigned a static label when it arrives.

However, it turns out that for the SOAP analysis, it is crucial that labels, and more generally label-size pairs, are *i.i.d. across jobs*. This is one of the core assumptions of the M/G/1 with labels (§ 5.2.3), and it turns out to be one of the main limits on what policies the SOAP analysis applies to. As such, for the purposes of this thesis, policies that could be represented by a rank function but require non-i.i.d. labels are *not* SOAP policies. We give some examples of such policies in Section 6.4.3.

Terminology: "Rank" vs. "Index"

In Definition 6.1, we refer to a job's numerical priority as its "rank", with lower rank indicating better priority. This is somewhat at odds with conventional terminology for index policies, where a job's numerical priority is called its "index", with higher index indicating better priority. A notable example is the literature on the Gittins policy [44], which is in fact usually called the Gittins *index* policy and defined as serving the job of maximal Gittins index.

In light of this, why do we introduce the additional term "rank" and use the lower-is-better convention? Why not stick with "index" and higher-is-better? There are two reasons, though both are ultimately matters of personal preference.

- It is useful to have separate terms for numerical priority in general, for which we use the term "index", and numerical priority that specifically only depends on a job's label and age, for which we use the term "rank".[3]
- While I find that each of lower-is-better and higher-is-better can be more intuitive in different cases, for most of the scheduling policies in this thesis, I prefer lower-is-

[3] In Chapter 14, we introduce a more general job model where a job's state might not be its label-age pair, but we continue to use "rank" to refer to priroity that depends only on that more general job state.

better. In particular, SRPT is nicely framed as lower-remaining-work-is-better, and several other policies we study are, in one way or another, generalizations of SRPT.

Technical Restrictions on Rank Functions

When discussing server sharing above, we implicitly assumed the existence of (right) derivatives $\frac{\mathrm{d}}{\mathrm{d}a}\,\mathsf{rank}(\ell, a)$. We also assume throughout Chapters 7 and 8 that various expectations related to rank functions are well defined. The following assumptions suffice and hold in essentially all cases of practical interest, so we assume them throughout.

Assumption 6.3.

(a) The label set $\mathbb{L}$ is (a metric space isomorphic to) a subset of $\mathbb{R}^n$ for some integer $n \geq 0$.
(b) For all ages $a \geq 0$, the function $\mathsf{rank}(\cdot, a)$ is piecewise continuous.
(c) For all labels $\ell \in \mathbb{L}$, the function $\mathsf{rank}(\ell, \cdot)$ is piecewise differentiable and is upper semi-continuous, meaning $\mathsf{rank}(\ell, a) \geq \max\{\mathsf{rank}(\ell, a-), \mathsf{rank}(\ell, a+)\}$.

These assumptions become even less restrictive if we allow a richer space of ranks than $\mathbb{R}_{\geq 0}$ (§ 6.1.3). They can also likely be relaxed by stating them in measure-theoretic language, but doing so is outside the scope of this thesis (§ 5.6.5).

6.1.5 SOAP Policies in the M/G/k

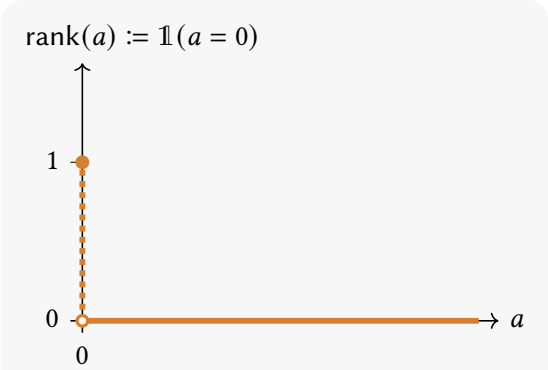
We have thus far discussed SOAP policies as they apply in a single-server setting like the M/G/1 with labels. SOAP policies have natural interpretations in multiserver systems as well. In the M/G/k, a SOAP policy serves the k jobs of minimal k minimal ranks, serving all jobs if there are $k-1$ or fewer. There are a few subtle details to note in the M/G/k case: server sharing can become somewhat intricate (§ 5.3.2), and multiple rank functions that represent the same policy in the M/G/1 may represent different policies in the M/G/k. However, these details will not concern us.

When we wish to disambiguate between a SOAP policy π being used in a single-server system and π being used in a multiserver system, we write π-1 and π-k for the single-server and multiserver versions, respectively. This disambiguation is useful in Part III, where we compare an M/G/k using a multiserver policy π-k to an M/G/1 with the same total service rate (§ 5.2.4) using the analogous single-server policy π-1.

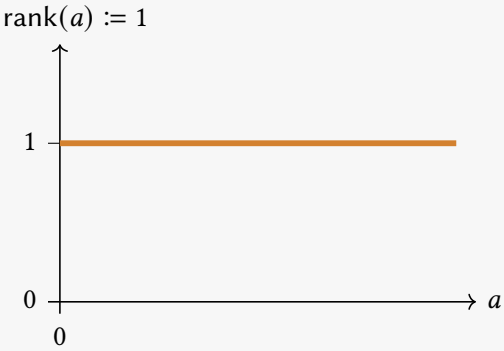
6.2 Previously Analyzed "Simple" SOAP Policies

6.2.1 Simple SOAP Policies that Treat All Jobs the Same Way

For a SOAP policy, treating all jobs the same way corresponds to the unlabeled case.



(a) Rank function of FCFS (or LCFS, if we use LCFS tiebreaking).



(b) Rank function of FCFS (or PLCFS, if we use LCFS tiebreaking).

Figure 6.2. Two rank functions that represent FCFS (Pol. 6.5). Both rely on the fact that we use FCFS tiebreaking when multiple jobs have the same rank. If we use LCFS tiebreaking instead, then the two rank functions instead represent two different policies: LCFS (Pol. 6.6) and PLCFS (Pol. 6.7).

Policy 6.4. The *Least Attained Service (LAS)* policy always serves the job of least age. It is a SOAP policy with rank function

$$\mathsf{rank}_{\mathrm{LAS}}(a) := a.$$

See Figure 6.1(a) for an illustration.

Policy 6.5. The *First-Come, First-Served (FCFS)* policy nonpreemptively serves jobs in arrival order. It is a SOAP policy that is represented by any rank function rank such that $\mathsf{rank}(0) \geq \mathsf{rank}(a)$ for all $a > 0$. See Figure 6.2 for illustrations of two such rank functions.

We take a moment below to explain in more detail why the property $\mathsf{rank}(0) \geq \mathsf{rank}(a)$ results in FCFS, because we will see similar ideas in other rank functions later.

Suppose for now that $\mathsf{rank}(0) > \mathsf{rank}(a)$ for all $a > 0$, as in Figure 6.2(a). We can think of the $\mathsf{rank}(0) > \mathsf{rank}(a)$ property as *encoding nonpreemptiveness*. This is because only time multiple jobs are tied for minimal rank is immediately after a departure, when all the jobs present have age 0. Once a job begins service, it outranks the jobs that remain at age 0, as well as later arrivals which have initial age 0, so it is never preempted. FCFS tiebreaking ensures that when choosing a job at age 0 to start serving, the scheduler starts the one that arrived earliest, thus resulting in FCFS.

FCFS tiebreaking has another effect: the scheduler will not preempt the job in service even if the rank function only satisfies the weaker property $\mathsf{rank}(0) \geq \mathsf{rank}(a)$, as in Figure 6.2(b). However, we will see in Policies 6.6 and 6.7 below that this weaker property does not ensure nonpreemptiveness under LCFS tiebreaking.

Policy 6.6. The *Last-Come, First-Served (LCFS)* policy nonpreemptively serves jobs in reverse arrival order. That is, after each departure, LCFS serves whichever job most recently

arrived. It is a SOAP policy with LCFS tiebreaking represented by any rank function rank such that $\mathsf{rank}(0) > \mathsf{rank}(a)$ for all $a > 0$, such as that the one shown in Figure 6.2(a). As discussed above, such a rank function encodes the fact that the scheduling policy is nonpreemptive, so LCFS tiebreaking results in LCFS.

Policy 6.7. The *Preemptive Last-Come, First-Served (PLCFS)* policy always serves the job that most recently arrived. It is a SOAP policy with LCFS tiebreaking represented any constant rank function, such as the one shown in Figure 6.2(b). This means the LCFS tiebreaking rule is always in effect, as opposed to in Policy 6.6, when LCFS tiebreaking is only relevant after departures.

6.2.2 Simple SOAP Policies with Multiple Classes of Jobs

The following example considers a system with multiple classes of jobs, each representing a different priority level. Each job is labeled by its class. For simplicity of notation, we assume classes are nonnegative numbers, so $\mathbb{L} \subseteq \mathbb{R}_{\geq 0}$, where lower numbers denote better priority.

Policy 6.8. The *Nonpreemptive Priority (NP-Prio)* and *Preemptive Priority (P-Prio)* policies both prioritize jobs according to their priority class. Their eponymous difference is that NP-Prio is nonpreemptive, while P-Prio will preempt the job in service if a new job with better priority class arrives. They are SOAP policies with respective rank functions

$$\begin{aligned}\mathsf{rank}_{\text{NP-Prio}}(\ell, a) &\coloneqq \ell \mathbb{1}(a = 0),\\ \mathsf{rank}_{\text{P-Prio}}(\ell, a) &\coloneqq \ell.\end{aligned}$$

Due to FCFS tiebreaking, within each class, NP-Prio and P-Prio both serve jobs in FCFS order.

The queueing literature traditionally considers systems with finitely many classes, meaning $\mathbb{L} = \{1, \ldots, n\}$ for some positive integer n, but the above examples of NP-Prio and P-Prio work just as well with infinitely many classes.

6.2.3 Simple SOAP Policies that Use Job Size Information

The following examples consider a system where the scheduler knows each job's size. Each job is thus labeled with its size s, so $\mathbb{L} = \mathbb{R}_{>0}$.

Policy 6.9. The *Shortest Job First (SJF)* and *Preemptive Shortest Job First (PSJF)* policies both prioritize jobs according to their size, with smaller sizes having better priority. In fact, SJF and PSJF are just NP-Prio and P-Prio, respectively, in the case where each job's class is its size. They are therefore SOAP policies with respective rank functions

$$\begin{aligned}\mathsf{rank}_{\text{SJF}}(s, a) &\coloneqq s \mathbb{1}(a = 0),\\ \mathsf{rank}_{\text{PSJF}}(s, a) &\coloneqq s.\end{aligned}$$

Policy 6.10. The *Shortest Remaining Processing Time (SRPT)* policy always serves the job of least remaining work. It is a SOAP policy with rank function

$$\mathsf{rank}_{\mathrm{SRPT}}(s, a) \coloneqq s - a.$$

See Figure 6.1(b) for an illustration. SRPT has the distinction of minimizing mean response time in single-server systems [116].

6.3 Newly Analyzed "Complex" SOAP Policies

6.3.1 Complex SOAP Policies for Job Size Uncertainty

SRPT minimizes mean response time, but to implement SRPT, we need to know each job's size. What can we do instead if the scheduler does not know each job's size? Under SRPT, a job's rank is its remaining work, so a natural idea is to try to estimate each job's remaining work, then use those estimates to schedule.

What does the scheduler know about each job? This is subject of Section 5.2.3, part of which we briefly review here. The scheduler knows each job's *state* (ℓ, a), which is the pair of its label ℓ and age a. We also assume that we know the characteristics of the M/G arrival process, and in particular the label-size distribution (L, S). From this, we can compute two important types of distributions:

- the *label-conditional size distribution* of a label ℓ, defined as $S_\ell = (S \mid L = \ell)$; and
- the *state-conditional remaining work distribution* of a state (ℓ, a), defined as

$$S_{\ell,a} \coloneqq (S_\ell - a \mid S_\ell > a) = (S - a \mid L = \ell, S > a).$$

SERPT: A Natural but Suboptimal Idea

From the state-conditional remaining work distribution comes a natural scheduling idea: instead of prioritizing by remaining work, which is unknown, prioritize by *expected* remaining work.

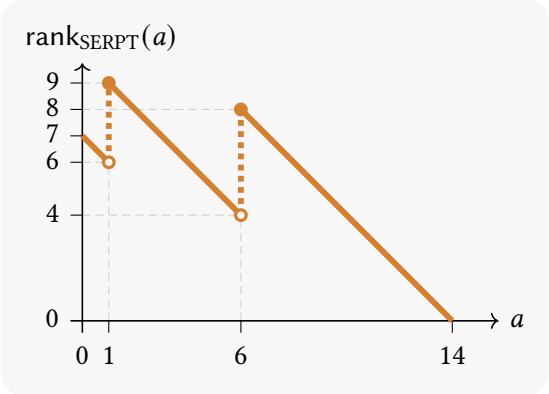
Policy 6.11. The *Shortest Expected Remaining Processing Time (SERPT)* policy always serves the job with the least *expected* remaining work. It is a SOAP policy with rank function

$$\begin{aligned}\mathsf{rank}_{\mathrm{SERPT}}(\ell, a) &\coloneqq \mathbf{E}[S_{\ell,a}] \\ &= \mathbf{E}[S_\ell - a \mid S_\ell > a] \\ &= \mathbf{E}[S - a \mid L = \ell, S_\ell > a].\end{aligned}$$

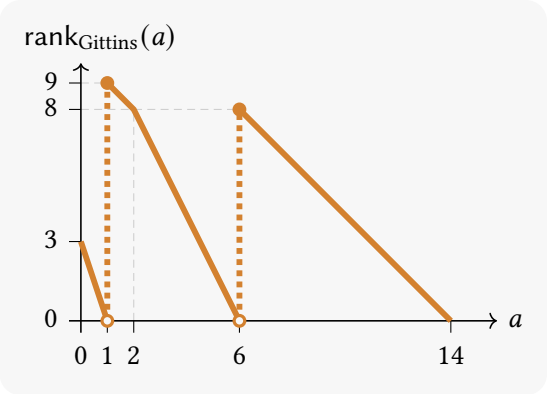
In the unlabeled case, this becomes

$$\mathsf{rank}_{\mathrm{SERPT}}(a) \coloneqq \mathbf{E}[S_a] = \mathbf{E}[S - a \mid S > a],$$

and in the known-size case, where a job's label is its size, SERPT reduces to SRPT.



(a) Rank function of SERPT.



(b) Rank function of Gittins.

Figure 6.3. Rank functions of two policies designed to deal with job size uncertainty: SERPT (Pol. 6.11) and Gittins (Pol. 6.12) . We show both policies in the unlabeled case with job size distribution $S =$ "1, 6, or 14, each with probability 1/3". Notice that both policies are nonmonotonic for this job size distribution. Gittins minimizes mean response time in the M/G/1 with labels (Ch. 16). In particular, Gittins improves upon SERPT by giving a job rank approaching 0 as its age approaches 1 and 6, two ages at which the job could potentially complete.

See Figure 6.3(a) for an example of SERPT's rank function in the unlabeled case. In particular, the example shows that SERPT's rank function can be nonmonotonic, in contrast to all of the policies in Section 6.2.

Given that SRPT is optimal for mean response time when job sizes are known to the scheduler, we might wonder: is SERPT optimal when job sizes are unknown? Perhaps surprisingly, the answer is no: there is room for improvement. To see where this improvement might come from, consider the example from Figure 6.3, where all jobs have size 1, 6, or 14. Suppose there are two jobs present in the system: job J has age 10, and job K has age $1 - \varepsilon$ for some small $\varepsilon > 0$.[4] SERPT treats these two jobs as follows (Fig. 6.3(a)).

- Job J (age 10) has passed ages 1 and 6, implying it is size 14, so its remaining work is 4.
- Job K (age $1 - \varepsilon$) could still be any size, so its expected remaining work is $6 + \varepsilon$.
- Seeing as J has lower expected remaining work than K, we should serve J.

But is serving J the right decision? Here is an alternative line of reasoning that suggests otherwise.

- Job J (age 10) has remaining work 4, as previously discussed.
- Job K (age $1 - \varepsilon$) has a 1/3 chance of having only remaining work ε.
- Seeing as K has a significant chance of being almost done while J definitely has significant remaining work, we should serve K. Specifically, we should serve K at least until it reaches age 1: the cost of taking ε time is very small, but we benefit

[4] It turns out that having two jobs of these ages cannot actually occur under SERPT, but the example is informative regardless.

greatly if K turns out to be size 1.

The second line of reasoning turns out to be correct: serving K is indeed better in this example. In fact, even if we alter the example so that job K is either size 1, 300, or 400, it remains correct to serve job K until it reaches age 1. The key observation is that thanks to preemption, when we serve a job, we are only committing to serving it in the short term. SERPT neglects this observation: a job's remaining work is the amount of service it takes to complete the job, but we might preempt the job before it completes, so expected remaining work is not always the right quantity to focus on.

Gittins: Optimal Mean Response Time

The following scheduling policy, called *Gittins*, improves upon SERPT by taking into account the above observation that we it can help to preempt jobs before they complete.

Policy 6.12. The *Gittins policy for mean response time (Gittins)* is the SOAP policy with rank function

$$\begin{aligned}\mathsf{rank}_{\text{Gittins}}(\ell, a) &:= \inf_{\Delta>0} \frac{\mathbf{E}[\min\{S_{\ell,a}, \Delta\}]}{\mathbf{P}[S_{\ell,a} \le \Delta]} \\ &= \inf_{b>a} \frac{\mathbf{E}[\min\{S_\ell, b\} - a \mid S_\ell > a]}{\mathbf{P}[S_\ell \le b \mid S_\ell > a]} \\ &= \inf_{b>a} \frac{\mathbf{E}[\min\{S, b\} - a \mid L = \ell, S > a]}{\mathbf{P}[S \le b \mid L = \ell, S_\ell > a]}.\end{aligned}$$

In the unlabeled case, this becomes

$$\begin{aligned}\mathsf{rank}_{\text{Gittins}}(a) &:= \inf_{\Delta>0} \frac{\mathbf{E}[\min\{S_a, \Delta\}]}{\mathbf{P}[S_a \le \Delta]} \\ &= \inf_{b>a} \frac{\mathbf{E}[\min\{S, b\} - a \mid S > a]}{\mathbf{P}[S \le b \mid S > a]},\end{aligned}$$

and in the known-size case, where a job's label is its size s, Gittins reduces to SRPT:

$$\begin{aligned}\mathsf{rank}_{\text{Gittins}}(s, a) &:= \inf_{\Delta>0} \frac{\mathbf{E}[\min\{s - a, \Delta\}]}{\mathbf{P}[s - a \le \Delta]} \\ &= s - a && \text{[by minimizing at } \Delta = s - a\text{]} \\ &= \mathsf{rank}_{\text{SRPT}}(s, a). && \text{[by Pol. 6.10]}\end{aligned}$$

Gittins has the distinction of minimizing mean response time in the M/G/1 with labels (Ch. 16). We can thus see Gittins as a way of optimally generalizing SRPT to scenarios where the scheduler does not known each job's size.

See Figure 6.3(b) for an example of Gittins's rank function in the unlabeled case. Notice that, like SERPT, Gittins is nonmonotonic in this example.

The intuition behind Gittins is that the Δ in the infimum is an amount of time we might "commit" to serving the job for. Equivalently, we can see this as committing to serving the job until it reaches age $b = a + \Delta$. Under this commitment, we serve the job for $\mathbf{E}[\min\{S_{\ell,a}, \Delta\}]$ in expectation, and the job completes with probability $\mathbf{P}[S_{\ell,a} \leq \Delta]$. We can think of the ratio $\mathbf{E}[\min\{S_{\ell,a}, \Delta\}]/\mathbf{P}[S_{\ell,a} \leq \Delta]$ as being the average *time-per-completion ratio* of the Δ service commitment. While SERPT only considers the time-per-completion ratio for $\Delta = \infty$, Gittins considers the time-per-completion ratios of all $\Delta > 0$, choosing the minimum such ratio as a job's rank. Considering all $\Delta > 0$ is the right choice because we can choose to preempt jobs at any time.

The version of Gittins defined above is actually a special case of a more general definition, which we discuss in Chapter 14. In particular, one can construct versions of Gittins that optimize objectives other than mean response time, such as mean weighted response time or mean holding cost (§ 5.4.4).

How Does SERPT Compare to Gittins?

Comparing Policies 6.11 and 6.12, we see that the the Gittins policy is significantly more complicated to define than SERPT.[5] Yet the rank functions for SERPT and Gittins shown in Figure 6.3 look very similar. We therefore might wonder: what is Gittins's complexity buying us? Is SERPT's performance much worse than Gittins?

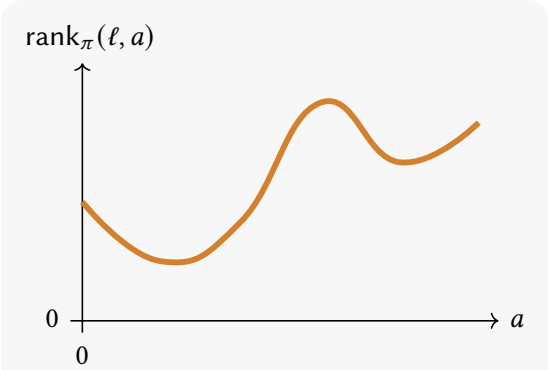
The question of how well SERPT does compared to Gittins is one we could not answer prior to SOAP, because the mean response time of neither policy was known. Armed with the SOAP analysis (Chs. 7 and 8), we are able to compare SERPT to Gittins for any given label-size distribution. We do this in Chapter 10 for many examples, and we find that SERPT does indeed have mean response time close to that of Gittins.

However, the question of whether SERPT is nearly as good as Gittins for *all* label-size distributions remains a challenge, even with the SOAP analysis. This is because SERPT and Gittins are not really individual SOAP policies, but rather SOAP policy *constructions*. Specifically, different label-size distributions yield different SERPT and Gittins rank functions, as we can see from the appearance of $S_{\ell,a}$, the state-conditional remaining work distribution, in Policies 6.11 and 6.12. We take a first step towards solving this difficult problem in Chapter 11, showing that in the unlabeled case, a new variant of SERPT achieves mean response time within a constant factor of Gittins's for *all* job size distributions.

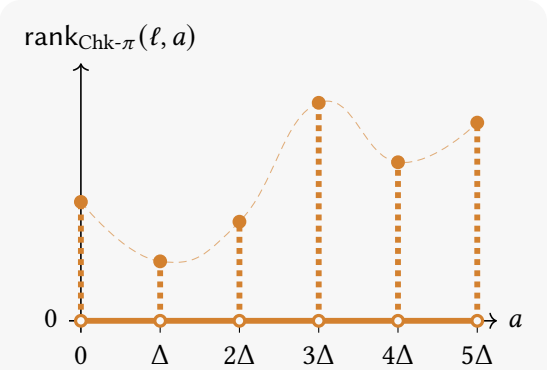
6.3.2 Complex SOAP Policies for Preemption Limitations

For the most part, the M/G/1 scheduling literature focuses on one of two extreme cases with regards to preemption: it is either disallowed entirely or completely unrestricted. However, there are a variety of ways that practical systems may lie between these two extremes. We can use rank functions to model many of these scenarios as SOAP policies.

[5] Whether this translates into greater computational complexity remains an open problem, but with known algorithms, computing SERPT is faster than computing Gittins [125, Appx. B].



(a) Rank function of generic SOAP policy π.



(b) Rank function of Chk-π, with rank of π shown for reference (dashed orange line).

Figure 6.4. Rank function of Chk-π (Pol. 6.13) for some SOAP policy π, with checkpoint ages $A = \{0, \Delta, 2\Delta, \ldots\}$.

Preemption Checkpoints

In some systems, jobs can be preempted, but not at any time. Imagine, for instance, a computer program that has periodic *checkpoints* at which it saves its work. It could be that to prevent loss of progress, we only allow preempting a job immediately after a checkpoint. As another example, computer networks transmit *packet flows* that consist of multiple *packets* of data. Most network hardware works at packet-level granularity, so while we might preempt transmission of one packet flow to start transmitting another, we would not preempt transmission in the middle of a packet. The following example shows how to model this scenario using SOAP policies.

Policy 6.13. Let π be a SOAP policy, and let $A \subseteq \mathbb{R}_{\geq 0}$ be a set of ages, which we call *checkpoint ages*. The *Checkpointed π (Chk-π)* policy is, roughly speaking, a version of π that only preempts jobs at checkpoint ages. It is a SOAP policy with rank function

$$\mathrm{rank}_{\text{Chk-}\pi}(\ell, a) \coloneqq \mathrm{rank}_\pi(\ell, a)\, \mathbb{1}(a \in A).$$

As a concrete example, to model scheduling packet flows with packet size Δ, we would let the checkpoint ages be $A = \{n\Delta \mid n \in \mathbb{N}\} = \{0, \Delta, 2\Delta, \ldots\}$. See Figure 6.4 for an illustrated example with this set of checkpoint ages. Because each checkpoint causes jumps to and from rank 0, even if the original policy π is monotonic, Chk-π is generally nonmonotonic.[6]

We note that Goerg [48] analyzes the mean response time of Chk-SRPT with checkpoint ages $A = \{0, \Delta, 2\Delta, \ldots\}$ for some $\Delta > 0$. But analyzing the full response time distribution

[6]There are two corner cases where Chk-π is monotonic. First, in the corner case where $\mathrm{rank}_\pi(\ell, a) = 0$ for all $\ell \in \mathbb{L}$ and $a \in A$, the rank function $\mathrm{rank}_{\text{Chk-}\pi}$ is constant, and so Chk-π is equivalent to FCFS. Second, it may be that some monotonic rank function represents Chk-π, even if the rank function $\mathrm{rank}_{\text{Chk-}\pi}$ defined here is nonmonotonic. For example, Chk-NP-Prio is equivalent to ordinary NP-Prio.

has not been done in prior work, nor have any other policies of the form Chk-π been analyzed.

When designing a system where jobs have preemption checkpoints, an important design question is: how frequent should checkpoints be? On one hand, more frequent checkpoints allow for more frequent preemption, which allows for more flexibility when scheduling. But on the other hand, it is usually the case that each checkpoint actually adds some amount of overhead to a job's size. For example, in networking, every packet has a header containing metadata, so using smaller packet sizes means having more headers and thus larger packet flows.

We use the SOAP analysis to answer the question of how frequent checkpoints should be. In Chapter 9, we study many examples of checkpointed policies with a constant gap between checkpoints, resulting in a rule of thumb for optimizing the tradeoff between scheduling flexibility and avoiding overhead. One might wonder whether one could do better than constant checkpoint gaps, such as by having the gaps grow for larger ages. We study this question in Chapter 13, showing that if gaps between checkpoints grow too quickly, then the checkpointed policy can have poor performance.

Limited Priority Levels

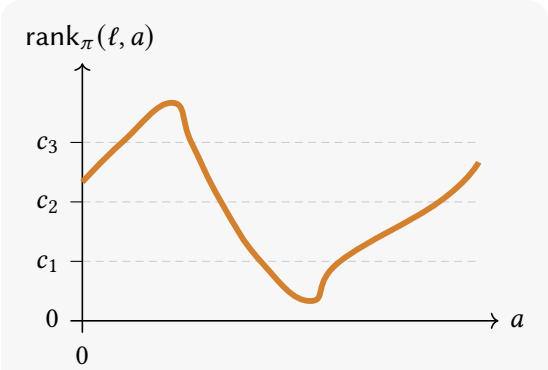
Some systems limit preemption in a more subtle way than age checkpoints: they have *limited priority levels (LPL)*. For instance, network switches often have a fixed number of priority levels built into their hardware [96], and other computer systems have similar constraints [54, 57, 86]. In terms of SOAP policies, LPL means that a rank function's range must be a finite set. This rules out using policies like LAS and SRPT, where a job's rank changes continuously with age. The following example shows how to model LPL systems using SOAP policies.

Policy 6.14. Let π be a SOAP policy, and let $R \subseteq \mathbb{R}_{\geq 0}$ be a set of *cutoff ranks*. The *Limited-Priority-Level π (LPL-π)* policy is, roughly speaking, a version of π that has only $|R| + 1$ priority levels, where the ranks in R serve as cutoffs between the levels. It is a SOAP policy with rank function

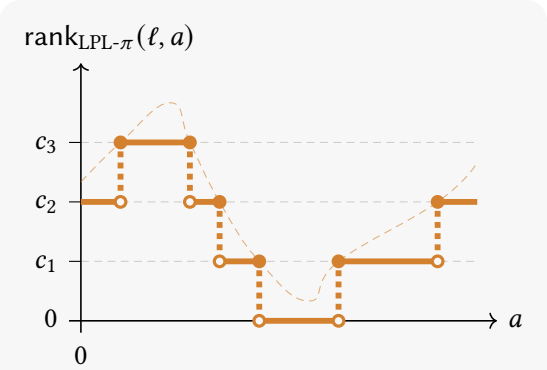
$$\mathsf{rank}_{\text{LPL-}\pi}(\ell, a) \coloneqq \sup\big((\{0\} \cup R) \cap [0, \mathsf{rank}_\pi(\ell, a)]\big).$$

That is, if $R = \{c_1, \ldots, c_{n-1}\}$ contains $n - 1$ cutoff ranks, then LPL-π "rounds down" the ranks π assigns to one of $0, c_1, \ldots, c_{n-1}$, so LPL-π uses n priority levels. See Figure 6.5 for an illustrated example with four priority levels.

When designing a scheduling policy for an LPL system, policies of the form LPL-π are a natural choice. For instance, if we wish to minimize mean response time and have access to job size information, we might use LPL-SRPT, but we are immediately confronted with a question: how should we choose the rank cutoffs? We use the SOAP analysis to answer this question, as well as other design questions for LPL systems, in Chapter 9.



(a) Rank function of generic SOAP policy π.



(b) Rank function of LPL-π, with rank of π shown for reference (dashed orange line).

Figure 6.5. Rank function of LPL-π (Pol. 6.14) for some SOAP policy π, with four priority levels separated by rank cutoffs $R = \{c_1, c_2, c_3\}$.

Only Preempting for Large Priority Differences

For our final examples of preemption limitations, we return to the class-based priority setting of Section 6.2.2. For concreteness, suppose there are a finite number of classes $\mathbb{L} = \{1, \ldots, n\}$. In this setting, NP-Prio and P-Prio represent the two extremes of never allowing preemption and always allowing preemption. But perhaps we want to allow preemption only when the job in service is significantly less important than the job preempting it. The following two policies show how to model such constraints using SOAP policies.

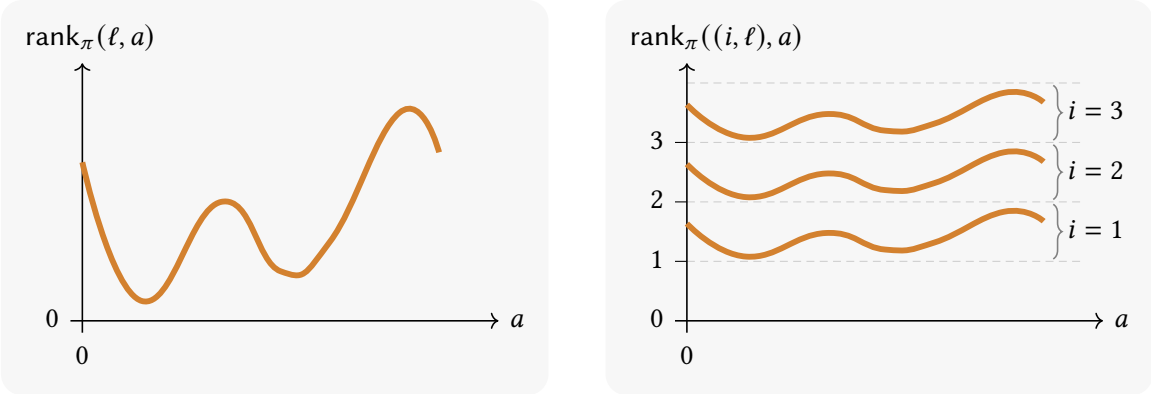
Policy 6.15. The *Preempt-Large-Difference Priority (Large-Diff)* policy is a partially preemptive version of class-based priority where once a job begins, it can only be preempted by a job which is at least d priority classes better. Put another way, its rank decreases by d once it begins service, so this is a SOAP policy with rank function

$$\text{rank}_{\text{Large-Diff}}(\ell, a) = \ell - d\mathbb{1}(a > 0).$$

Policy 6.16. The *Red-Preempts-Blue Priority (Red-Blue)* policy is a partially preemptive version of class-based priority. Jobs from classes $1, \ldots, m$, which we call *red* classes, may preempt jobs from classes $m+1, \ldots, n$, which we call *blue* classes, but otherwise, preemption is not allowed. This is a SOAP policy with rank function

$$\text{rank}_{\text{Red-Blue}}(\ell, a) = \begin{cases} \ell\mathbb{1}(a = 0) & \text{if } \ell \in \{1, \ldots, m-1\}. \\ \ell\mathbb{1}(a = 0) + (m+1)\mathbb{1}(a > 0) & \text{if } \ell \in \{m+1, \ldots, n\}. \end{cases}$$

The policies in Policies 6.15 and 6.16 are relatively simple as far as SOAP policies go. Both are monotonic, and so they would be tractable to analyze with pre-SOAP techniques.



(a) Rank function of generic SOAP policy π.

(b) Rank function of P-Prio-π.

Figure 6.6. Rank function of P-Prio-π (Pol. 6.17) for some SOAP policy π. We show the ranks for classes $i \in \{1, 2, 3\}$.

Nevertheless, the SOAP analysis is still valuable in this context, because it provides a "one-size-fits-all" solution that works not just for these two policies, but also for many variations on the same theme of allowing preemption only for large priority differences.

6.3.3 Complex SOAP Policies from Mixtures of Policies

Most previously analyzed SOAP policies are, roughly speaking, "one-idea" policies. For instance, NP-Prio and P-Prio only care about each job's priority class, while SJF, PSJF, and SRPT only care about each job's size. But what if we want to use both a job's priority class and size to make scheduling decisions? For example, perhaps we want to preemptively prioritize by class like P-Prio, but we want to use SRPT within each class. We can model this and similar policies as SOAP policies.

Policy 6.17. Consider a system where each job is labeled by one of n priority classes and a label from set $\mathbb{L}'$, so $\mathbb{L} = \{1, \ldots, n\} \times \mathbb{L}'$, and let π be a SOAP policy with label set $\mathbb{L}'$. The *Preemptive Priority Around π (P-Prio-π)* policy is a variant of P-Prio that uses π within each class. It is the SOAP policy with rank function

$$\mathsf{rank}((i, \ell), a) = i + \mathsf{squish}\big(\mathsf{rank}_\pi(\ell, a)\big).$$

where squish is a strictly increasing function $\mathbb{R}_{\geq 0} \to [0, 1)$, such as $\mathsf{squish}(r) = 1 - \exp(-r)$. See Figure 6.6 for an illustration.

We note that Goerg and Pham [49] analyze the mean overall and per-class response times of P-Prio-SRPT. But analyzing the full response time distributions has not been done in prior work, nor have any other policies of the form P-Prio-π been analyzed.

Below is one last example showcasing the flexibility of rank functions as a modeling tool, featuring a mixture of FCFS and SRPT. It is also another example of modeling a

preemption limitation using SOAP (§ 6.3.2), because some jobs are preemptible while others are not.

Policy 6.18. Consider a system with two classes of customers: humans and robots.

- Humans, unpredictable and easily offended, have unknown job sizes, are nonpreemptible, and insist on FCFS service relative to other humans. We label humans with the symbol human.
- Robots, precise and ruthlessly efficient, have known job sizes, are preemptible, and insist on SRPT service relative to other robots. We label a robot of size s with the symbolic expression $\mathsf{robot}(s)$.

One scheduling policy we could use in this system is the SOAP policy with rank function

$$\begin{aligned}\mathsf{rank}(\mathsf{human}, a) &= c\mathbb{1}(a = 0),\\ \mathsf{rank}(\mathsf{robot}(s), a) &= s - a,\end{aligned}$$

where $c \geq 0$ is a constant. This policy mixes FCFS (among humans) with SRPT (among robots), letting robots with size below c have priority over humans that have not yet started service.

6.4 What Policies Are Not SOAP?

We have seen the breadth of the class of SOAP policies throughout Sections 6.2 and 6.3. Nevertheless, there are some types of scheduling policies that fall squarely outside the SOAP class. This section describes four ways scheduling policies can fail to be SOAP, giving examples of each:

- (§ 6.4.1) The policy does not fit into the general paradigm of assigning each job a priority based only on its own characteristics.
- (§ 6.4.2) The policy assigns jobs priorities based on their own characteristics, but a job's priority can change while it is waiting in the queue.
- (§ 6.4.3) The policy could be represented using a rank function, but it would require assigning labels to jobs in a non-i.i.d. manner.
- (§ 6.4.4) The policy would be a SOAP policy if not for the fact that it breaks ties in an order other than FCFS or LCFS.

The lines between these aspects are admittedly blurry. For instance, we will cover the *Earliest Deadline First (EDF)* policy as an example for both the second and third reason.

6.4.1 Priorities Not Determined Job-by-Job

SOAP policies are in some sense "local": they assign each job its priority based only on its own characteristics. But not all policies fit into this general paradigm. For instance, many policies from the worst-case scheduling literature use the entire system state to make scheduling decisions. These include recently developed scheduling policies for scheduling with job size estimates under adversarial arrivals [10, 11].

One example of a scheduling policy from the queueing literature is the recently introduced *Nudge* policy [53]. Nudge serves jobs in FCFS order by default, but it occasionally swaps the order of adjacent jobs in the queue. Exactly when these swaps occur depends on the history of past swaps, so the resulting prioritization of jobs is not determined locally job-by-job. Nudge's significance has to do with its response time tail $\mathbf{P}[T > t]$, as we discuss further in Chapter 13.

6.4.2 Priorities Changing in the Queue

SOAP policies only change a job's priority while it is in service, because a job's label never changes, and its age changes only while in service. But some policies do change a job's priority while it waits in the queue. Perhaps the simplest examples of this are *Accumulating Priority* policies [38, 131]. Under these policies, a job's priority gets better and better the longer it waits in the queue.

Another example of a policy changing jobs' priorities while they are in the queue is the *Earliest Deadline First (EDF)* policy. EDF is used in settings where each job is assigned a *deadline* when it arrives, and the scheduler always serves the job with the least time until its deadline. We would like to say that under EDF, a job's rank could be the amount of time between now and its deadline, but this amount of time depends not on the job's age but rather the total time the job is in the system, so this strategy does not work for representing EDF.

6.4.3 Labels Not Independently and Identically Distributed

Here is another attempt at representing EDF with a rank function: let a job's rank be simply its deadline, represented as a "wall clock" time. A job's assigned deadline never changes, so this avoids the issue discussed in Section 6.4.2 above. However, this rank function would require each job's label to be its deadline, or at least contain information from which the deadline could be deduced. This is an issue because deadlines are not i.i.d. in general, as jobs that arrive later in time tend to have later deadlines.

Another example of a scheduling policy that requires non-i.i.d. labels to represent using a rank function is the *Randomized Multi-Level Feedback (RMLF)* policy [16, 17, 63]. RMLF labels each job with a value in the interval $\mathbb{L} = [0, 1)$ and has rank function[7]

$$\mathsf{rank}_{\mathrm{RMLF}}(\ell, a) = 2^{\lfloor \ell + \log_2 a \rfloor}. \tag{6.1}$$

One could in principle use this rank function to schedule in the M/G/1 with labels, and the resulting policy would be a SOAP policy. However, the random assignment of labels is actually part of the RMLF policy, and typically different jobs have different label distributions. This is partly because RMLF is designed for guaranteeing good mean response

[7] There are multiple variants of RMLF in prior work. This rank function is for the eRMLF variant introduced by Bansal et al. [16].

time under *adversarial* arrival processes (§ 5.1.3), a setting where careful randomization is critical for good worst-case performance.

Even though we can represent RMLF using the above rank function, its lack of i.i.d. labels means it is "just barely" not a SOAP policy. Is there any hope of using SOAP to analyze it anyway? Scully and Harchol-Balter [121] give a partial answer to this by extending the SOAP analysis to policies which specify a range of ranks, as opposed to a single rank, for each job state. This approach works for RMLF because plugging in $\ell = 0$ and $\ell = 1$ give lower and upper bounds, respectively, in (6.1). The cost of using only rank function bounds is that we obtain only response time bounds, as opposed to an exact analysis, but the bounds are tight enough to characterize RMLF's response time tail (Ch. 13).

6.4.4 Tiebreaking Not FCFS or LCFS

Perhaps the most important policy that is not SOAP is the *Processor Sharing (PS)* policy, which always shares the server equally among all jobs in the system. This seems simple enough to represent as a rank function: all jobs have the same rank. However, PS requires a tiebreaking rule other than FCFS or LCFS. For example, with (continuously invoked) random tiebreaking, we could represent PS using the rank function in Figure 6.2(b).

Another example of a policy that requires random tiebreaking is the *Random Order of Service (ROS)* policy, a nonpreemptively that chooses a job uniformly at random to serve after each completion. With random tiebreaking, we could represent ROS using the rank function in Figure 6.2(a).

Chapter 7 presents the SOAP analysis assuming FCFS or LCFS tiebreaking. One might hope to extend the analysis to random tiebreaking. However, given the fact that the prior analyses of PS are significantly more complicated than prior analyses of SOAP policies [55], extending SOAP to random tiebreaking is likely to require significant new insights.

Chapter 7

SOAP Analysis: One Response Time Formula for All Rank Functions

Having introduced SOAP policies and rank functions in Chapter 6, we now move on to presenting our main result about SOAP: a universal response time analysis that applies to any SOAP policy in the M/G/1.

The significance of the SOAP analysis is two-fold: it *unifies* and *generalizes* prior M/G/1 response time analyses. Its unifying power is in formalizing the common ideas that occur in many prior analyses, such as most of those in Harchol-Balter [55, Part VII]. In particular, the SOAP analysis demonstrates that many prior analyses are all essentially the same argument, but applied to different rank functions. And therein lies its generalizing power: by analyzing a generic rank function, we wind up with a result that applies to all SOAP policies, which go far beyond previously analyzed policies. In particular, SOAP provides the first analysis for policies with nonmonotonic rank functions.

This section presents the SOAP analysis. As a warmup, we begin by analyzing P-Prio, a simple SOAP policy which has been analyzed before (§ 7.1). This case is simple because a job's rank never changes under P-Prio, but it leaves us with a major question: how do we deal with the fact that a job's rank can change as a function of its age? The general SOAP analysis boils down to answering this question. Some key definitions from the P-Prio analysis generalize relatively straightforwardly to general SOAP policies (§ 7.2), but to carry out the full analysis, we need two new ideas. Roughly speaking, the first idea helps us handle rank increases (§ 7.3), and the second idea helps us handle rank decreases (§ 7.4). The result is formulas for the mean and LST of response time under any SOAP policy (Thm. 7.15).

There is one key part of the SOAP analysis that we defer to Chapter 8. We do so for two reasons. First, it is the most technical part of the analysis, so deferring the details streamlines our presentation. Second, it turns out that generalizing slightly beyond what is necessary for this chapter's analysis gives a result that is useful in other contexts, as we will see in Part III.

This chapter is based on material from Scully et al. [124].

7.1 Warmup with Constant Ranks: Analyzing P-Prio

Our goal is to determine $T_{\text{P-Prio}}$, the response time distribution of an M/G/1 using P-Prio. Recall that P-Prio is the policy with rank function $\text{rank}_{\text{P-Prio}}(\ell, a) = \ell$, meaning each job is labeled with its rank (Pol. 6.8). Our result will actually characterize the conditional response time distribution $T_{\text{P-Prio}}(\ell, s)$ (§ 5.4.2), but $T_{\text{P-Prio}}$ can be fully determined from $\mathcal{L}[T_{\text{P-Prio}}(\ell, s)]$ and the label-size distribution (L, S).

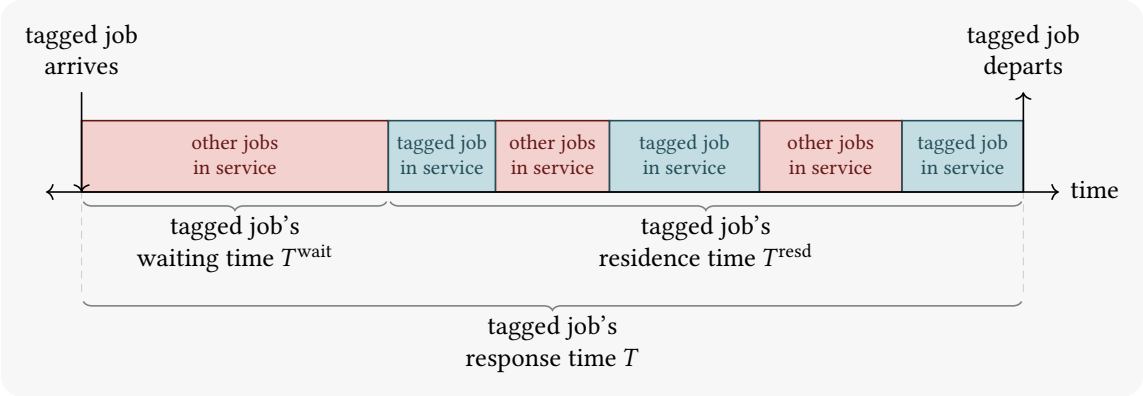


Figure 7.1. Response time is the sum of waiting time and residence time.

This section only discusses P-Prio, so to reduce notational clutter, we usually omit the subscript P-Prio in the rest of this section.

We define many new concepts in this section. For now, we define them in a way that works for P-Prio and prioritize intuition over formality. We defer the more general versions of the definitions, which are more complicated, to Section 7.2.

7.1.1 Overall Analysis Strategy

To analyze P-Prio, we use the *tagged job approach*. We consider a generic "tagged" job arriving to a steady-state system and analyze the tagged job's response time. By PASTA (§ 5.4.2), the tagged job's response time is a random variable distributed according to T. Slightly abusing notation (§ 5.6.2), we hereafter use T to denote both the overall response time distribution as well as the random variable that is the tagged job's response time.

We analyze the tagged job's response time T by decomposing it into two parts:

- *Waiting time*, denoted T^{wait}, is the amount of time between the tagged job's arrival and its first instant in service.
- *Residence time*, denoted T^{resd}, is the amount of time between the tagged job's first instant in service and its departure.

We can thus write response time as

$$T = T^{\text{wait}} + T^{\text{resd}}, \tag{7.1}$$

as illustrated in Figure 7.1.

Analyzing T amounts to analyzing each of T^{wait} and T^{resd}, then determining how the correlation between them. The first step of doing so is to condition on the tagged job's label-size pair. Suppose for the rest of this section that the tagged job has label ℓ and size s. Conditioning on this, (7.1) becomes

$$T(\ell, s) = T^{\text{wait}}(\ell, s) + T^{\text{resd}}(\ell, s), \tag{7.2}$$

where $T^{\text{wait}}(\ell, s)$ and $T^{\text{resd}}(\ell, s)$ denote conditional waiting and residence times, respectively, which are defined analogously to conditional response time $T(\ell, s)$ (§ 5.4.2).

It may seem that we have not made much progress with (7.2) compared to (7.1), but there is an important difference: it turns out that $T^{\text{wait}}(\ell, s)$ and $T^{\text{resd}}(\ell, s)$ are independent. That is, all correlation between T^{wait} and T^{resd} is due to the fact that the tagged job's label-size pair affects both quantities, so they are conditionally independent given the label-size pair.[1] This conditional independence will become evident in the rest of this section as we analyze each of waiting time (§ 7.1.2) and residence time (§ 7.1.3).

7.1.2 Waiting Time of P-Prio

Old Jobs and System Relevant Work

When the tagged job arrives, there may be other jobs already in the system. We call any jobs present when the tagged job arrives *old jobs*, because they arrived prior to the tagged job. Whether each old job is served before the tagged job depends on whether or not it outranks the tagged job. That is, writing $r := \ell$ for the tagged job's rank during its waiting time,[2] we classify old jobs as follows:

- *Old relevant jobs* outrank the tagged job, meaning they have rank at most r. These jobs are served before the tagged job begins service, so they contribute to its waiting time.
- *Old irrelevant jobs* are outranked by the tagged job, meaning they have rank greater than r. These jobs are not served until after the tagged job completes, so they contribute to neither waiting time nor residence time.

To determine the tagged job's waiting time, we need to know the total remaining work of old relevant jobs.

Let the *relevant system work*, denoted $W(\leq r)$, be the total remaining work of old relevant jobs in the system when the tagged job arrives. We can think of relevant system work as being the system work in an M/G/1 where we modify the arrival process to remove jobs that would be irrelevant, replacing the label-size distribution (L, S) by $(L, S\mathbb{1}(L \leq r))$. This allows us to apply known results about M/G/1 system work to relevant system work. In particular (§ 5.5.1),

$$\mathbf{E}[W(\leq r)] = \frac{\frac{\lambda}{2}\mathbf{E}[S^2\mathbb{1}(L \leq r)]}{1 - \rho(\leq r)}. \tag{7.3}$$

where $\rho(\leq r) = \lambda\mathbf{E}[S\mathbb{1}(L \leq r)]$. See Definition 7.3 for the full definition of relevant system work.

[1] For analyzing P-Prio specifically, we actually only need to condition on the tagged job's label. But conditioning on the tagged job's size is necessary for the full SOAP analysis, so we do the same here.

[2] We introduce the notation r to emphasize that what matters here is the rank of the tagged job, which in general might be a more complicated function of the job's label and age.

New Jobs and Relevant Busy Periods

We have seen so far that old jobs contribute $W(\leq r)$ to the tagged job's waiting time. But *new jobs*, those that arrive after the tagged job, can also contribute to waiting time. Whether a new job is served before the tagged job depends on whether or not it outranks the tagged job, so we classify new jobs as follows:

- *New relevant jobs* outrank the tagged job, meaning they have rank (strictly) less than r.
- *New irrelevant jobs* are outranked by the tagged job, meaning they have rank at least (and possibly equal to) r.

To determine the tagged job's waiting time, we need to know the total size of new relevant jobs that arrive during the waiting time. This is recursive: if a relevant new job arrives during waiting time, it extends that waiting time, thus creating the possibility for more new relevant jobs to arrive. The recursion suggests waiting time is a type of busy period (§ 5.5.2).

Let a *relevant busy period started by initial work v* be the same as an ordinary a busy period started by initial work v, except we only include relevant jobs. We can think of a relevant busy period as a busy period in an M/G/1 where we modify the arrival process to remove jobs which would be irrelevant, replacing the label-size distribution (L, S) by $(L, S\mathbb{1}(L < r))$. This allows us to apply known results about M/G/1 busy periods to relevant busy periods. For example, writing $B(<r, v)$ for the work of a relevant busy period started by initial work v, we have (§ 5.5.2)

$$\mathbf{E}[B(<r, v)] = \frac{v}{1 - \rho(<r)}. \tag{7.4}$$

where $\rho(<r) = \lambda \mathbf{E}[S\mathbb{1}(L < r)]$. See Definition 7.6 for the full definition of relevant busy periods.

Putting the Pieces Together

To review: when the tagged job arrives, it observes relevant system work due to old relevant jobs, all of which outranks the tagged job. New relevant jobs continue to arrive, and these also outrank the tagged job. These are exactly the jobs that contribute to the tagged job's waiting time. Therefore, the waiting time is a relevant busy period started by the relevant system work:

$$T^{\text{wait}}_{\text{P-Prio}}(\ell, s) =_{\text{st}} B(<r, W(\leq r)),$$

recalling that $r := \ell = \mathsf{rank}_{\text{P-Prio}}(\ell, 0)$. Combining (7.3) and (7.4) yields the expectation:

$$\mathbf{E}[T^{\text{wait}}_{\text{P-Prio}}(\ell, s)] = \frac{\frac{\lambda}{2}\mathbf{E}[S^2\mathbb{1}(L \leq r)]}{(1 - \rho(\leq r))(1 - \rho(<r))}.$$

7.1.3 Residence Time of P-Prio

The first important fact about residence time is that, at least under P-Prio, the tagged job's rank is the same as it was during waiting time, namely $r := \ell$. We can therefore use the same definition of "relevant" as we did for waiting time.

At the start of residence time, there are no relevant jobs, new or old, in the system. However, new relevant jobs can still arrive, preempting the tagged job and delaying its completion. This means that, much like waiting time, the tagged job's residence time is a relevant busy period, but this time started by the tagged job's size s. Therefore,

$$\begin{aligned} T^{\text{resd}}_{\text{P-Prio}}(\ell, s) &=_{\text{st}} B(<r, s), \\ \mathbf{E}[T^{\text{resd}}_{\text{P-Prio}}(\ell, s)] &= \frac{s}{1-\rho(<r)}, \qquad \text{[by (7.4)]} \end{aligned}$$

where $r := \ell = \mathsf{rank}_{\text{P-Prio}}(\ell, a)$ for all ages a.

7.1.4 Response Time of P-Prio

Having analyzed the tagged job's mean waiting and residence times, mean conditional response time follows immediately:

$$\begin{aligned} \mathbf{E}[T_{\text{P-Prio}}(\ell, s)] &= \mathbf{E}[T^{\text{wait}}_{\text{P-Prio}}(\ell, s)] + \mathbf{E}[T^{\text{resd}}_{\text{P-Prio}}(\ell, s)] \\ &= \frac{\frac{\lambda}{2}\mathbf{E}[S^2 \mathbb{1}(L \le r)]}{(1-\rho(\le r))(1-\rho(<r))} + \frac{s}{1-\rho(<r)}. \end{aligned}$$

To determine the distribution and not just the mean, we need to determine the dependence between waiting time and residence time.

Fortunately, as alluded to in Section 7.1.1, it turns out that waiting time and residence time are independent.[3] This is because when the tagged job enters service, there must be no relevant jobs in the system. Otherwise, the tagged job would not have the best rank and thus would not be starting service. This means we effectively start residence time with a "clean slate". Irrelevant jobs that were present during waiting time might still be present during residence time, but these irrelevant jobs do not affect the tagged job's response time.

Independence of waiting and residence times allows us to characterize the response time distribution of the tagged job in terms of relevant system work and relevant busy periods:[4]

$$\begin{aligned} T_{\text{P-Prio}}(\ell, s) &=_{\text{st}} B(<r, W(\le r)) + B(<r, s) \\ &=_{\text{st}} B(<r, W(\le r) + s). \qquad \text{[by Cor. 5.6]} \end{aligned}$$

[3] More precisely, they are conditionally independent given the tagged job's label-size pair. We say simply "independent" because it is clear that we are considering a tagged job with a given label-size pair (ℓ, s). We do the same throughout this chapter.

[4] Recall from our notation conventions that the terms in the following sum are independent (§ 5.6.2).

We can use this to characterize the LST (Def. 5.2) of $T_{\text{P-Prio}}(\ell, s)$ in terms of the system work and busy period LSTs under modified arrival processes. Because busy period LSTs are characterized recursively (Prop. 5.5), the LST of $T_{\text{P-Prio}}(\ell, s)$ also has a recursive component. But like busy period LSTs, the recursion can be solved to obtain moments $\mathbf{E}\left[\left(T_{\text{P-Prio}}(\ell, s)\right)^n\right]$ for all $n \in \mathbb{N}$.

7.2 The Relevant System: What Delays the Tagged Job

Having warmed up by analyzing P-Prio in Section 7.1, we now turn to analyzing general SOAP policies, where a job's rank can change during service. Just like P-Prio, the basic idea of our general SOAP analysis is to look at the system through the lens of what is relevant to a tagged job. Informally, we call this lens the *relevant system*. Concepts like relevant jobs, relevant system work, and relevant busy periods are all aspects of the relevant system.

7.2.1 Why Is Generalizing from P-Prio to All SOAP Policies Hard?

There are three main steps to generalizing the P-Prio analysis to all SOAP policies. The first step is the topic of this section. The second and third, which we review very briefly below, are treated in detail in Sections 7.3 and 7.4.

The first step is to generalize definitions of relevant system concepts. This is the topic of this section: we define relevant jobs (§ 7.2.2), relevant work (§ 7.2.3), and relevant busy periods (§ 7.2.4). The second and third steps of generalizing the P-Prio analysis have to do with overcoming two obstacles:

- (§ 7.3) The fact that *the tagged job's* rank might *increase* makes it more complicated to *apply* relevant system concepts. Specifically, we can no longer simply use the tagged job's rank when deciding what is relevant.
- (§ 7.4) The fact that *other jobs'* ranks might *decrease* makes it more complicated to *analyze* the relevant system. Specifically, the steady-state relevant system work no long.

In light of the first obstacle above, this section takes no strong position about exactly which jobs should count as "relevant". Instead, we define the relevant system in a way that works for any subset of states $\mathbb{Y} \subseteq \mathbb{L} \times \mathbb{R}_{\geq 0}$ we deem relevant. We will generally take $\mathbb{Y}$ to be the set of states whose rank is below some threshold, but the definitions work just as well for any set $\mathbb{Y}$, and we find some uses for this extra generality throughout the thesis (Chs. 8, 12, and 16).

7.2.2 Relevant Jobs

Definition 7.1. Let $\mathbb{Y} \subseteq \mathbb{L} \times \mathbb{R}_{\geq 0}$ be a set of job states.

(a) A job is $\mathbb{Y}$-*relevant* if its state is in $\mathbb{Y}$. Otherwise, it is $\mathbb{Y}$-*irrelevant*.

(b) A server is $\mathbb{Y}$-*relevant-busy* if it is currently serving a $\mathbb{Y}$-relevant job. Otherwise, it is $\mathbb{Y}$-*relevant-idle*.

Definition 7.2. We use the shorthands below when discussing $\mathbb{Y}$-relevant system concepts, such as those defined in Definitions 7.1 and 7.3–7.6.

(a) When discussing a SOAP policy π that is clear from context, we write

$$\begin{aligned} \leq r &:= \{(\ell, a) \in \mathbb{L} \times \mathbb{R}_{\geq 0} \mid \mathsf{rank}_\pi(\ell, a) \leq r\}, \\ < r &:= \{(\ell, a) \in \mathbb{L} \times \mathbb{R}_{\geq 0} \mid \mathsf{rank}_\pi(\ell, a) < r\} \end{aligned}$$

to denote the sets of states with rank at most r and rank less than r, respectively.

(b) When discussing an unspecified set $\mathbb{Y}$, or when $\mathbb{Y}$ is clear from context, we omit the "$\mathbb{Y}$-" prefix, writing just *relevant* or *irrelevant*.
- For example, throughout Section 7.1, a relevant job is one that is either old and $\leq r$-relevant or new and $< r$-relevant.

(c) When we do specify the set $\mathbb{Y}$, we often omit "relevant".
- For example, a $\leq r$-idle server is one that is either idle or serving a job of rank greater than r.
- We never similarly omit "irrelevant".

7.2.3 Relevant Work

Definition 7.3. Let $\mathbb{Y} \subseteq \mathbb{L} \times \mathbb{R}_{\geq 0}$ be a set of job states.

(a) The $\mathbb{Y}$-*relevant remaining work* (or *remaining* $\mathbb{Y}$-*work*) of a job in state (ℓ, a), denoted $S_{\ell,a}(\mathbb{Y})$, is the amount of service the job requires to stop being a $\mathbb{Y}$-job, which happens when it either completes or its state exits $\mathbb{Y}$. Formally, the job's remaining $\mathbb{Y}$-work is the random variable

$$\begin{aligned} S_{\ell,a}(\mathbb{Y}) &:= \min\{S_{\ell,a}, \Delta_{\ell,a}(\mathbb{Y})\} \\ &= \big(\min\{S_\ell - a, \Delta_{\ell,a}(\mathbb{Y})\} \;\big|\; S_\ell > a\big) \\ &= \big(\min\{S - a, \Delta_{\ell,a}(\mathbb{Y})\} \;\big|\; L = \ell, S > a\big). \end{aligned}$$

where

$$\Delta_{\ell,a}(\mathbb{Y}) := \inf\{\Delta \geq 0 \mid (\ell, a + \Delta) \notin \mathbb{Y}\}$$

is the distance from the job's age a to the earliest age at which the job would exit $\mathbb{Y}$.

(b) The $\mathbb{Y}$-*relevant system work* (or *system* $\mathbb{Y}$-*work*), denoted $W(\mathbb{Y})$, is the total remaining $\mathbb{Y}$-work of all jobs in the system:

$$W(\mathbb{Y}) := \sum_{i=1}^{N} S_{L_i,A_i}(\mathbb{Y}).$$

When we wish to specify or clarify the scheduling policy π whose system $\mathbb{Y}$-work we are discussing, we include it as a subscript, as in $W_\pi(\mathbb{Y})$.

(c) More generally, just as we use the term "work" in many ways in informal discussion, we use the term $\mathbb{Y}$-*relevant work* (or $\mathbb{Y}$-*work*) to refer to any work that involves serving a job while its state is in $\mathbb{Y}$.

Some clarifying remarks about $\mathbb{Y}$-work:

- Although a job's state may enter and exit $\mathbb{Y}$ multiple times over the course of its service, remaining $\mathbb{Y}$-work measures only the service needed to complete or exit $\mathbb{Y}$ *for the first time*. Visits to $\mathbb{Y}$ after this first exit do not contribute to the job's remaining $\mathbb{Y}$-work.
- If the job is not a $\mathbb{Y}$-job, meaning its state is not in $\mathbb{Y}$, then its remaining $\mathbb{Y}$-work is 0.
- Unlike system work, system $\mathbb{Y}$-work is *not* scheduling-invariant. As we will see in Chapter 8, the steady-state distribution of system $\mathbb{Y}$-work depends on, roughly speaking, the degree to which the scheduling policy prioritizes $\mathbb{Y}$-jobs. With that said, in this chapter, it is usually clear what scheduling policy is being discussed, in which case we write $W(\mathbb{Y})$ without a subscript despite its dependence on the scheduling policy.

How Relevant Work Enters the System

Definition 7.4. Let $\mathbb{Y} \subseteq \mathbb{L} \times \mathbb{R}_{\geq 0}$ be a set of job states.

(a) A $\mathbb{Y}$-job whose state has been in $\mathbb{Y}$ for its entire time in the system is called $\mathbb{Y}$-*fresh*.

(b) The $\mathbb{Y}$-*relevant size* of a job is its remaining $\mathbb{Y}$-work at age 0. The label-size distribution (L, S) gives the system a $\mathbb{Y}$-*relevant size distribution*

$$S(\mathbb{Y}) \coloneqq \min\{S, \Delta_{L,0}(\mathbb{Y})\},$$

where $\Delta_{\ell,a}(\mathbb{Y})$ is as in Definition 7.3(a). We can think of $S(\mathbb{Y})$ as the amount of service a generic job receives while it is $\mathbb{Y}$-fresh.

(c) The $\mathbb{Y}$-*fresh load* of the system is

$$\rho(\mathbb{Y}) \coloneqq \lambda \mathbf{E}[S(\mathbb{Y})],$$

namely the average rate $\mathbb{Y}$-work is added to the system due to new arrivals.

Definition 7.5. Let $\mathbb{Y} \subseteq \mathbb{L} \times \mathbb{R}_{\geq 0}$ be a set of job states.

(a) A job is $\mathbb{Y}$-*recycled* if it is currently $\mathbb{Y}$-relevant but was $\mathbb{Y}$-irrelevant at some point in the past.

(b) The moment a job switches from $\mathbb{Y}$-irrelevant to $\mathbb{Y}$-relevant is called a $\mathbb{Y}$-*recycling* of the job. That is, $\mathbb{Y}$-recyclings are when a job's state enters $\mathbb{Y}$ from outside $\mathbb{Y}$. We write

(c) The $\mathbb{Y}$-*recycling rate*, denoted $\lambda_{\text{rcy}}(\mathbb{Y})$, is the time-average rate at which $\mathbb{Y}$-recyclings occur:

$$\lambda_{\text{rcy}}(\mathbb{Y}) \coloneqq \lambda \mathbf{E}[\text{number of } \mathbb{Y}\text{-recyclings of a generic job}].$$

(d) The $\mathbb{Y}$-*recycling size distribution*, denoted $S_{\text{rcy}}(\mathbb{Y})$, is the distribution of remaining $\mathbb{Y}$-work of a job immediately after its $\mathbb{Y}$-recycling.

(e) The $\mathbb{Y}$-*recycled load* of the system, denoted $\rho_{\text{rcy}}(\mathbb{Y})$, is the average rate $\mathbb{Y}$-work is added to the system due to $\mathbb{Y}$-recyclings:

$$\rho_{\text{rcy}}(\mathbb{Y}) \coloneqq \lambda_{\text{rcy}}(\mathbb{Y})\, \mathbf{E}[S_{\text{rcy}}(\mathbb{Y})].$$

Some clarifying remarks about $\mathbb{Y}$-recyclings:

- Because a job's state only changes while it is in service (§ 5.2.3), a job's $\mathbb{Y}$-recycling must occur while the job is in service.
- A job may undergo zero, a finite number of, or even infinitely many $\mathbb{Y}$-recyclings over the course of its time in service, depending on $\mathbb{Y}$ and the label-size pair of the job in question.
- Spending even an instant outside of $\mathbb{Y}$ can cause a $\mathbb{Y}$-recycling. For example, consider the Chk-π policy for scheduling with preemption checkpoints (Pol. 6.13). When a job passes an isolated checkpoint age without being preempted, it still undergoes a ≤ 0-recycling, even though its rank is greater than 0 for only an instant.
- We generally assume that $\lambda_{\text{rcy}}(\mathbb{Y})$ is finite for simplicity of exposition and notation, but this is not essential for any of our results.

7.2.4 Relevant Busy Periods

Definition 7.6. Let $\mathbb{Y} \subseteq \mathbb{L} \times \mathbb{R}_{\geq 0}$ be a set of job states. A $\mathbb{Y}$-*relevant busy period* (or $\mathbb{Y}$-*busy period*) started by initial work v is, roughly speaking, a busy period where each new arrival is "cut short" as soon as it stops being a $\mathbb{Y}$-job, even if it has not yet left the system. More formally, a $\mathbb{Y}$-busy period consists of the following two entities:

(a) The $\mathbb{Y}$-busy period's *work*, denoted $B(\mathbb{Y}, v)$, is the sum of the *initial work* v and the *new work* contributed by arrivals in the $\mathbb{Y}$-busy period's tree (see below). Each arrival contributes its $\mathbb{Y}$-size to new work.

(b) The $\mathbb{Y}$-busy period's *tree* is defined analogously to the tree of an ordinary busy period (§ 5.5.2). The root vertex represents the initial work, and non-root vertices represent arriving jobs, and we draw edges as follows:
   - We draw an edge from the root vertex to all jobs that arrive while the initial work is being served.
   - We draw an edge from job K to job L if L arrives "during K's $\mathbb{Y}$-size", meaning while K is both in service and $\mathbb{Y}$-fresh.

A clarifying remark about $\mathbb{Y}$-busy periods: while new work consists entirely of $\mathbb{Y}$-work, we allow for the possibility that the initial work is *not* all $\mathbb{Y}$-work. As an example of why this is useful, consider the residence time of P-Prio (§ 7.1.3): it is a $<r$-busy period started by the tagged job, which has rank exactly r and thus has $<r$-size 0.

7.3 Handling Rank Increases: The Pessimism Principle

This section and the next analyze the response time of an arbitrary SOAP policy π with rank function rank_π. We continue to use the tagged job approach introduced in Section 7.1.1, analyzing the response time of a tagged job with label-size pair (ℓ, s) to obtain the conditional response time distribution $T_\pi(\ell, s)$. We will occasionally consider specific policies as illustrative examples.

It turns out that increases and decreases in the rank function each present their own obstacle. This section focuses on handling rank increases, explaining the main obstacle (§ 7.3.1) and how we overcome it (§ 7.3.2). The result is a characterization of response time in terms of steady-state relevant system work (§ 7.3.3). However, rank decreases make analyzing steady-state relevant system work complicated in its own right, so we defer this last task to Section 7.4.

7.3.1 What Obstacles Do Rank Increases Create?

We illustrate the obstacles created by rank increases with perhaps the simplest example of an increasing rank function. Let *One-Jump* be the policy with unlabeled rank function

$$\mathsf{rank}_{\text{One-Jump}}(a) \coloneqq \mathbb{1}(a \geq b),$$

where $b > 0$ is a constant. Even though it does not use labels, One-Jump effectively divides jobs into two types: "small", meaning size at most b, and "large", meaning size greater than b. Jobs have rank 0 as long as the might be small, but once a job is known to be large, its rank jumps up to 1.

It turns out that if the tagged job has small size $s \leq b$, then we can proceed nearly identically to the analysis of P-Prio. This is because the tagged job's rank is always 0, and other jobs' ranks never decrease. Combining the overall strategy we used for P-Prio (§ 7.1) with the new relevant system definitions (§ 7.2), one can show that for all small sizes $s \leq b$,

$$\begin{aligned}
T^{\text{wait}}_{\text{One-Jump}}(s) &=_{\text{st}} B(<0, W(\leq 0)),\\
T^{\text{resd}}_{\text{One-Jump}}(s) &=_{\text{st}} B(<0, s),\\
T_{\text{One-Jump}}(s) &=_{\text{st}} B(<0, W(\leq 0) + s) =_{\text{st}} W(\leq 0) + s.
\end{aligned}$$

We can make the last simplification because jobs never have rank less than 0, so $B(<0, v) = v$ for any initial work v.[5]

Suppose now that the tagged job has large size $s > b$. Until the tagged job reaches age b, its rank is 0, as if it were small. It thus takes the tagged job $W(\leq 0) + b$ time to reach age b, which covers its waiting time and an initial portion of its residence time. At age b, the tagged job's rank jumps up to 1. The rest of the tagged job's residence time thus involves serving the rest of the tagged job, which has remaining work $s - b$ at age b, as well as any new <1-jobs that arrive. This means the remaining portion of the tagged job's residence time is a <1-busy period, specifically $B(<1, s - b)$. This gives response time

$$T_{\text{One-Jump}}(s) \stackrel{?}{=}_{\text{st}} W(\leq 0) + b + B(<1, s - b).$$

However, as indicated by the "?", this expression is not right. What went wrong?

[5]For the One-Jump policy, $W(\leq 0)$ is also not too hard to analyze. We discuss this in Section 7.4.1.

The problem is that when the tagged job's rank increases to 1 at age b, other jobs that are already in the system may begin outranking the tagged job. We call these other jobs "ghosts". There are two types of ghosts:[6]

- Old jobs with rank 1.
- New jobs with rank 0 that arrive before the tagged job reaches age b.

Accounting for the extra delay due to ghosts creates multiple obstacles, including the following:

- It is not clear how much relevant work ghosts contribute to residence time.
- Because ghosts include old jobs and new jobs that arrived during waiting time, ghosts create a dependence between waiting time and residence time.

These present a significant difficulty even for One-Jump, let alone policies with more complex rank functions. Rather than attack these obstacles head-on, we take a different approach that avoids them entirely.

7.3.2 Key Insight: Use Worst Future Rank, Not Current Rank

Response Time of "Large" Jobs under the One-Jump Policy

We continue our example from Section 7.3.1, considering a "large" tagged job's response time under the One-Jump policy.

Let us consider carefully the amount of service each old job receives while the (large) tagged job is in the system. Suppose an old job is in arbitrary state (ℓ', a') when the tagged job arrives. How much service will this job receive before the tagged job completes? Although the tagged job has rank 0 when it arrives, it will *eventually* have rank 1. Therefore, the server will complete the old job's remaining ≤ 1-work $S_{\ell',a'}(\leq 1)$ by the time the tagged job leaves the system. This ≤ 1-work is split between waiting and residence time:

- The old job's remaining ≤ 0-work $S_{\ell',a'}(\leq 0)$ is served during the tagged job's waiting time.
- The rest of the old job's remaining ≤ 1-work $S_{\ell',a'}(\leq 1) - S_{\ell',a'}(\leq 0)$ is served during the tagged job's residence time.[7]

This means that the total amount of service old jobs receive while the tagged job is in the system is equal to the system ≤ 1-work $W(\leq 1)$.

We can go through essentially the same reasoning for new jobs. Even if a new job arrives while the (large) tagged job has rank 0, because the tagged job will *eventually* have rank 1, the new job will be served for time equal to its <1-size before the tagged job leaves. This means that we can think of the tagged job's response time as a <1-busy period started by the system ≤ 1-work $W(\leq 1)$ plus the tagged job's size s. Thus, for all large sizes $s > b$,

$$T_{\text{One-Jump}}(s) =_{\text{st}} B(<1, W(\leq 1) + s).$$

[6] Recall from Section 7.1.2 that old jobs are those that are present when the tagged job arrives, and new jobs are those that arrive after the tagged job.

[7] In the expression $S_{\ell',a'}(\leq 1) - S_{\ell',a'}(\leq 0)$, both random variables refer to the same old job, so they are not independent.

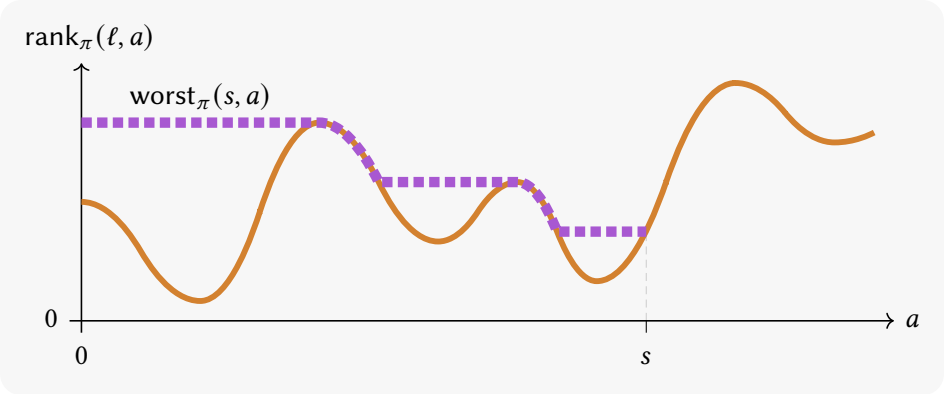


Figure 7.2. Relationship between worst future rank (magenta dotted line) and the underlying rank function (orange solid line) for a job with label-size pair (ℓ, s).

One important aspect of the above argument is the following. Suppose that instead of following the rank function $\mathsf{rank}_{\text{One-Jump}}$, the (large) tagged job instead had rank 1 for its entire time in the system. Then we could reason exactly the same way: old ≤ 1-work and new <1-work would have priority over the tagged job, resulting in the same response time distribution $B(<1, W(\leq 1) + s)$. That is, for the purposes of figuring out the tagged job's response time, *eventually* having rank 1 is equivalent to *currently* having rank 1.

General Case: The Pessimism Principle

The lesson from the above example is this: when analyzing the tagged job's response time, at any given age, we should focus not on the tagged job's current rank but its *worst future rank*. For a tagged job of label-size pair (ℓ, s) under general SOAP policy π, the worst future rank at age a, illustrated in Figure 7.2, is

$$\mathsf{worst}_\pi(\ell, s, a) \coloneqq \text{“sup”} \underset{a \leq b < s}{} \mathsf{rank}_\pi(\ell, a).$$

We put quotes around “sup” because, as we will see later (Defs. 7.8 and 7.9), it turns out we need to distinguish between the cases where the supremum is attained or not attained.

When the tagged job has age a, we can, for the purposes of analyzing its response time, pretend that its rank is $\mathsf{worst}_\pi(\ell, s, a)$ instead of $\mathsf{rank}_\pi(\ell, a)$. We call this observation the *Pessimism Principle*, stated in full below. It essentially follows from the discussion so far. For a more formal argument, see Scully et al. [124, Appx. D].

Proposition 7.7 (Pessimism Principle)**.** *Consider an M/G/1 under SOAP policy π, and consider a tagged job with label-size pair (ℓ, s).*

(a) Each other job contributes the following amount to the tagged job's response time.

- *Old job: its remaining $\leq \mathsf{worst}_\pi(\ell, s, 0)$-work when the tagged job arrives.*
- *New job: its $< \mathsf{worst}_\pi(\ell, s, a)$-size, where a is the age of the tagged job when the new job arrives.*

(b) The tagged job's response time distribution is $T_\pi(\ell, s)$ in both of the following scenarios:
- *All jobs are assigned ranks by π as usual.*
- *The tagged job's rank is modified so that at age a, its rank is* $\mathrm{worst}_\pi(\ell, s, a)$. *No other jobs' ranks are modified.*

The Pessimism Principle is useful because it effectively removes rank increases, at least for the tagged job, thus alleviating the obstacles discussed in Section 7.3.1.

To formalize the Pessimism Principle, we need to formally define the tagged job's worst future rank. As mentioned above, it turns out we need to be careful about whether the worst future rank actually occurs. To see why, suppose there is an old job in the system with rank r. If the tagged job ever enters a state of rank at least r, then the old job outranks the tagged job at that point. But if the tagged job instead approaches rank r from below without ever reaching it, then the old job never outranks the tagged job. In the latter case, we say the tagged job has worst future rank "$r-$" instead of r. The following definitions formalize this.

Definition 7.8.

(a) A *limit rank* is an expression of the form $r-$ for some rank $r \geq 0$. A limit rank $r-$ is ordered "just below" rank r. That is, $r' < r- < r$ for any ranks $r' < r$.
- We can easily extend SOAP scheduling to the case where jobs can be assigned limit ranks.[8] We make use of this in the Pessimism Principle (Prop. 7.7), in which the tagged job may be assigned a limit rank.
- The notation from Definition 7.2(a) can be used with limit ranks: both $\leq r-$ and $< r-$ are equivalent to $< r$.
- Formally, we can define the set of ranks and limit ranks to be $\mathbb{R}_{\geq 0} \times \{-1, 0\}$ ordered lexicographically, with $(r, -1)$ corresponding to $r-$ and $(r, 0)$ corresponding to r.

(b) The *rank maximum* of a set of ranks $R \subseteq \mathbb{R}$, denoted $\operatorname{rank\,max} R$, is

$$\operatorname{rank\,max} R = \begin{cases} \sup R & \text{if } \sup R \in R \\ (\sup R)- & \text{if } \sup R \notin R. \end{cases}$$

That is, if $r = \sup R$, then the rank maximum is r if the supremum is attained and $r-$ otherwise.

Definition 7.9. Let π be a SOAP policy, and consider a job with label-size pair (ℓ, s).

(a) The *worst future rank* of the job at age a, denoted $\mathrm{worst}_\pi(\ell, s, a)$, is

$$\mathrm{worst}_\pi(\ell, s, a) \coloneqq \operatorname*{rank\,max}_{a \leq b < s} \mathrm{rank}_\pi(\ell, a).$$

That is, if the job has a maximum future rank r, then its worst future rank is r, but if instead the job approaches but never attains a supremum future rank r, then its worst future rank is $r-$ (Def. 7.8). See Figure 7.2 for an illustration.

(b) The *worst ever rank* of the job is its worst future rank at age 0, namely $\mathrm{worst}_\pi(\ell, s, 0)$.

[8] It is a special case of using a set of ranks with a richer ordering than $\mathbb{R}_{\geq 0}$ (§ 6.1.3).

7.3.3 Using the Pessimism Principle to Compute Response Time

We hereafter consider a modified system, which we call the *pessimism-adjusted system*, in which the tagged job's rank is assigned rank $\text{worst}_\pi(\ell, s, a)$ at each age a. We refer to the system without this modification as the *standard system*.

By the Pessimism Principle (Prop. 7.7), the tagged job's response time distribution is the same in the pessimism-adjusted and standard systems, but working with the pessimism-adjusted system is simpler. This is because even when rank function is nonmonotonic, the tagged job's worst future rank is decreasing as a function of age (Fig. 7.2). The Pessimism Principle thus partially reduces the problem of analyzing nonmonotonic SOAP policies to the simpler monotonic case, which has been well studied (§ 2.1.5).

We analyze the tagged job's response time in the pessimism-adjusted system by splitting it into waiting and residence times (§ 7.1.1). However, waiting and residence times in the pessimism-adjusted system may not be the same as in the standard system. Specifically, waiting time is longer, and residence time is shorter. This is because we effectively shift delays due to "ghost" jobs (§ 7.3.1) from residence time to waiting time. Hereafter, the notations T^{wait} and T^{wait} always refer to the pessimism-adjusted system.

Lemma 7.10. *Consider an M/G/1 under SOAP policy π, and consider a tagged job with label-size pair (ℓ, s). The tagged job's pessimism-adjusted waiting time $T_\pi^{\text{wait}}(\ell, s)$ and pessimism-adjusted residence time $T_\pi^{\text{wait}}(\ell, s)$ are independent.*

Proof. Consider the moment the tagged job enters service, namely the boundary between its waiting time and residence time. There may be some other jobs in the system at this time. Call these *end-of-waiting* jobs. It suffices to show that the end-of-waiting jobs do not affect the tagged job's residence time.

Because the tagged job's residence time starts with the tagged job being served, the tagged job must outrank all end-of-waiting jobs at the start of its residence time. But the tagged job's worst future rank only decreases, so it continues outranking end-of-waiting jobs for the remainder of its residence time. Moreover, under a SOAP policy, end-of-waiting jobs have no impact on how any new jobs that arrive during residence time are scheduled. This means end-of-waiting jobs do not affect the tagged job's residence time. □

Lemma 7.11. *Consider an M/G/1 under SOAP policy π, and consider a tagged job with label-size pair (ℓ, s). The tagged job's pessimism-adjusted waiting time is*

$$T_\pi^{\text{wait}}(\ell, s) = B(<r, W(\leq r)),$$

where $r \coloneqq \text{worst}_\pi(\ell, s, 0)$ is the tagged job's worst ever rank.

Proof. By the Pessimism Principle (Prop. 7.7), the tagged job is delayed by the remaining $\leq r$-work of old jobs it observes when it arrives, plus the $<r$-size of new jobs that arrive during waiting time. This means that, analogous to our analysis of P-Prio (§ 7.1.2), waiting time is a $<r$-busy period started by system $\leq r$-work. □

Lemma 7.12. *Consider an M/G/1 under SOAP policy π, and consider a tagged job with label-size pair (ℓ, s). The tagged job's pessimism-adjusted residence time is*

$$T_\pi^{\text{resd}}(\ell, s) =_{\text{st}} \int_0^s B(<r(a), \mathrm{d}a),$$

where $r(a) \coloneqq \text{worst}_\pi(\ell, s, a)$ is the tagged job's worst future rank at age a.

Proof. We begin by noting that the integral on the right-hand side can be formalized as integration with respect to a random measure. However, the formality distracts from the key ideas, so we instead reason informally. We can think of $B(<r(a), \mathrm{d}a)$ as simply a $<r(a)$-busy period started by an infinitesimal amount of work $\mathrm{d}a$. Consistent with our notation conventions (§ 5.6.2), all of these $<r(a)$-busy periods are mutually independent.

To that end, consider the amount of time it takes the tagged job to go from being served at age a to being served at age $a + \mathrm{d}a$ for infinitesimal $\mathrm{d}a$. Call this amount of time the *slice at age a*, or simply the *slice*. It suffices to show that the slice at age a is independent of past slices and has distribution $B(<r(a), \mathrm{d}a)$.

Under our assumption that rank functions are piecewise continuous (Asm. 6.3), we may assume that the tagged job has worst future rank $r(a)$ for the entirety of the slice at age a. Because the tagged job is in service at the start of the slice, it outranks all other jobs present. Reasoning as in the proof of Lemma 7.10, this means those other jobs do not affect the slice at age a, so the slice is independent of past slices.

Recalling that the tagged job has worst future rank $r(a)$ during the slice at age a, any new jobs that arrive during it contribute their $<r(a)$-size to the slice. This means the slice is at most a $<r(a)$-busy period started by initial work $\mathrm{d}a$. We say "at most" because it may be that the tagged job reaches age $a + \mathrm{d}a$ before the end of this $<r(a)$-busy period. But all the new $<r(a)$-jobs in the busy period tree outrank the tagged job, so the tagged job is only served when none of these $<r(a)$-jobs are left. This means the tagged job reaches age $a + \mathrm{d}a$ at the end of the $<r(a)$-busy period, as desired. □

7.4 Handling Rank Decreases: Analyzing the Impact of Recycled Jobs

With Lemmas 7.10–7.12 in hand, only one task remains to determine a SOAP policy's response time: given a rank r, determine the steady-state distribution of the system $\leq r$-work $W(\leq r)$, which determines the tagged job's waiting time (Lem. 7.11). For P-Prio, we were able to reduce this task to analyzing the system work of an M/G/1 with a modified size distribution. However, under general SOAP policies, the fact that jobs' ranks can decrease makes $W(\leq r)$ harder to compute (§ 7.4.1). This section explains how we overcome this obstacle (§ 7.4.2), yielding a formula for steady-state system $\leq r$-work (§ 7.4.3). We defer some of the technical details to Chapter 8.

7.4.1 What Obstacles Do Rank Decreases Create?

We illustrate the obstacles created by rank decreases with a simple example. Recall the *One-Jump* policy from Section 7.3.1, which has rank function

$$\mathsf{rank}_{\text{One-Jump}}(a) \coloneqq \mathbb{1}(a \geq b),$$

where $b > 0$ is a constant. We compare One-jump to *Two-Jump*, the policy with rank function

$$\mathsf{rank}_{\text{Two-Jump}}(a) \coloneqq \mathbb{1}(a \in [b, c)),$$

where $c > b > 0$ are constants. Even though it does not use labels, Two-Jump effectively divides jobs into three types: "small", meaning size in $[0, b)$; "medium", meaning size in $[b, c)$; and "large", meaning size in $[c, \infty)$.

Consider the waiting time of a small tagged job under One-Jump or Two-Jump. The tagged job's worst future rank is 0, and no jobs are <0-relevant, so by Lemma 7.11, the waiting time in both cases is simply the system ≤ 0-work.

The steady-state system ≤ 0-work of One-Jump, namely $W_{\text{One-Jump}}(\leq 0)$, is simple to characterize. Under One-Jump, a job is ≤ 0-relevant until it reaches age b and never again thereafter. That is, all ≤ 0-relevant jobs are ≤ 0-*fresh* (Def. 7.4). We can thus view $W_{\text{One-Jump}}(\leq 0)$ as the system work in an M/G/1 where we replace the size distribution S by $S(\leq 0) = \min\{S, b\}$. This means[9]

$$W_{\text{One-Jump}}(\leq 0) =_{\text{st}} \sum_{i=0}^{\text{Geo}(1-\rho(\leq 0))} (\mathcal{E}S(\leq 0))_i.$$

In fact, we can reason similarly for any rank r and any increasing SOAP policy π, obtaining

$$W_\pi(\leq r) =_{\text{st}} \sum_{i=0}^{\text{Geo}(1-\rho(\leq r))} (\mathcal{E}S(\leq r))_i \qquad \text{if } \pi \text{ is increasing.}$$

However, $W_{\text{Two-Jump}}(\leq 0)$ stochastically dominates $W_{\text{One-Jump}}(\leq 0)$ as long as some jobs are larger than c. This is because large jobs, in addition to being ≤ 0-relevant before age b, also become ≤ 0-relevant again after age c. These large ≤ 0-*recycled* jobs (Def. 7.5), meaning jobs that become ≤ 0-relevant again after being ≤ 0-irrelevant, constitute a source of ≤ 0-work that we do not have to worry about when computing $W_{\text{One-Jump}}(\leq 0)$, or more generally when computing $W_\pi(\leq r)$ for increasing SOAP policies π. That is, $\leq r$-recycled jobs only exist if the rank function at some point decreases as a function of age.

How do we quantify the impact $\leq r$-recycled jobs have on system $\leq r$-work? This question appears difficult because it is not a priori clear when $\leq r$-recyclings occur. Recyclings are generally not a Poisson process, so we cannot simply treat them as additional arrivals.

[9] Recall from Section 5.6.2 that $\text{Geo}(q)$ is a geometric distribution with parameter q supported at zero, namely $\mathbf{P}[\text{Geo}(q) = n] = q(1-q)^n$; and recall from Section 5.6.3 that the $(\mathcal{E}V)_i$ are i.i.d. drawings from $\mathcal{E}V$, the excess of distribution V. See also Appendix A for an index of notation.

7.4.2 Key Insight: Before Recyclings, No Relevant System Work

The main obstacle to analyzing system $\leq r$-work is that it is hard to characterize $\leq r$-recyclings as a stochastic process. Nevertheless, we still know something about when $\leq r$-recyclings occur, and fortunately, this turns out to be enough to determine $W(\leq r)$.

Proposition 7.13. *Consider an M/G/1 under SOAP policy π, and consider a rank r.*
(a) Immediately before an $\leq r$-recycling occurs, the system $\leq r$-work is zero.
(b) No two $\leq r$-recyclings occur simultaneously.

Proof. Consider the instant before an $\leq r$-recycling. there must be an $\leq r$-irrelevant job in service, namely the one that is about to be $\leq r$-recycled. This means there must be no $\leq r$-jobs in the system, because any $\leq r$-job would outrank the $\leq r$-irrelevant job. Without $\leq r$-jobs, system $\leq r$-work is zero, implying (a).

Can two jobs become $\leq r$-recycled at the same time? For this to happen, the server would need to be sharing between two $\leq r$-irrelevant jobs in the instant before an $\leq r$-recycling. As described in Algorithm 6.1, sharing occurs only when the rank function is increasing. But a job's rank needs to decrease for it to become $\leq r$-recycled, so only one $\leq r$-recycling occurs at a time, implying (b).

There is one corner case glossed over in the argument for (b) that deserves spelling out. Consider, for example, an unlabeled rank function with a downwards jump at age a, namely $\mathsf{rank}_\pi(a-) > \mathsf{rank}_\pi(a+)$.[10] If rank is increasing leading up to age a, then we might worry that two jobs share the server as both approach up to age a, causing them to simultaneously jump from $\mathsf{rank}_\pi(a-)$ to $\mathsf{rank}_\pi(a+)$. Fortunately, this is ruled out by our assumption that rank functions are upper semi-continuous with respect to age (Asm. 6.3). Upper semi-continuity ensures that at age a, both jobs have rank $\mathsf{rank}_\pi(a) \geq \mathsf{rank}_\pi(a-) > \mathsf{rank}_\pi(a+)$. Therefore, at age a, both jobs' ranks are decreasing, so by Algorithm 6.1, their ranks jump down one at a time, with FCFS tiebreaking controlling which goes first. We can argue analogously in the presence of labels. □

7.4.3 Using Recycled Jobs to Compute Relevant Work

Proposition 7.13 tells us how much system $\leq r$-work there is immediately before $\leq r$-recyclings. How does this help us understand steady-state system $\leq r$-work? Bridging this gap is the topic of Chapter 8. Its main result, the *Relevant Work Decomposition Law*, gives the LST of system $\leq r$-work in terms of the LST of system $\leq r$-work at the instants before $\leq r$-recyclings. Proposition 7.13 characterizes the latter, implying the following result.

[10] For simplicity, we reason in terms of two jobs approaching the same age a in the unlabeled case, but the argument easily generalizes to the general case of two jobs approaching downward jumps, possibly in two different states.

Lemma 7.14. *Consider an M/G/1 under SOAP policy π, and consider a rank r. The steady-state system $\leq r$-work distribution is*[11]

$$W(\leq r) =_{\text{st}} \sum_{i=0}^{\text{Geo}(1-\rho(\leq r))} (\mathcal{E}S(\leq r))_i + \text{Bernoulli}\left(\frac{\rho_{\text{rcy}}(\leq r)}{1-\rho(\leq r)}\right) \mathcal{E}S_{\text{rcy}}(\leq r),$$

its LST is

$$\mathcal{L}[W(\leq r)](\theta) = \frac{1-\rho(\leq r)-\rho_{\text{rcy}}(\leq r)+\rho_{\text{rcy}}(\leq r)\,\mathcal{L}[\mathcal{E}S_{\text{rcy}}(\leq r)](\theta)}{1-\rho(\leq r)\,\mathcal{L}[\mathcal{E}S(\leq r)](\theta)},$$

and its mean is

$$\mathbf{E}[W(\leq r)] = \frac{\rho_{\text{rcy}}(\leq r)\,\mathbf{E}[\mathcal{E}S(\leq r)] + \rho_{\text{rcy}}(\leq r)\,\mathbf{E}[\mathcal{E}S_{\text{rcy}}(\leq r)]}{1-\rho(\leq r)}.$$

Proof. As discussed above, this follows from the Relevant Work Decomposition Law (Thm. 8.8(a)) and Proposition 7.13. □

7.5 SOAP Response Time Formulas

7.5.1 Main Result

Theorem 7.15. *Consider an M/G/1 under SOAP policy π, and consider label-size pair (ℓ, s). Let $r(a) := \text{worst}(\ell, s, a)$ and $r := r(0)$.*

(a) The conditional response time distribution is

$$T_\pi(\ell,s) =_{\text{st}} \overbrace{B\left(<r, \sum_{i=0}^{\text{Geo}(1-\rho(\leq r))} (\mathcal{E}S(\leq r))_i + \text{Bernoulli}\left(\frac{\rho_{\text{rcy}}(\leq r)}{1-\rho(\leq r)}\right)\mathcal{E}S_{\text{rcy}}(\leq r)\right)}^{T_\pi^{\text{wait}}(\ell,s)} + \underbrace{\int_0^s B(<r(a), \mathrm{d}a)}_{T_\pi^{\text{resd}}(\ell,s)}.$$

(b) The LST of conditional response time is

$$\mathcal{L}[T_\pi(\ell,s)](\theta) = \overbrace{\frac{1-\rho(\leq r)-\rho_{\text{rcy}}(\leq r)+\rho_{\text{rcy}}(\leq r)\,\mathcal{L}[\mathcal{E}S_{\text{rcy}}(\leq r)](\eta(<r,\theta))}{1-\rho(\leq r)\,\mathcal{L}[\mathcal{E}S(\leq r)](\eta(<r,\theta))}}^{\mathcal{L}[T_\pi^{\text{wait}}(\ell,s)](\theta)} \underbrace{\times \exp\left(\int_0^s \Big(\theta + \lambda\big(1-\mathcal{L}[S(\leq r)](\eta(<r(a),\theta))\big)\Big)\,\mathrm{d}a\right)}_{\mathcal{L}[T_\pi^{\text{resd}}(\ell,s)](\theta)},$$

[11] The notation below is defined in Section 5.6.2 and Definitions 7.4 and 7.5. See also Appendix A for an index of notation.

where $\eta(\mathbb{Y}, \theta)$ is the principal solution of[12]

$$\eta(\mathbb{Y}, \theta) = \theta + \lambda\big(1 - \mathcal{L}[S(<r)](\eta(\mathbb{Y}, \theta))\big).$$

(c) The mean conditional response time is

$$\mathbf{E}[T_\pi(\ell, s)] = \overbrace{\frac{\rho(\leq r)\,\mathbf{E}[\mathcal{E}S(\leq r)] + \rho_{\text{rcy}}(\leq r)\,\mathbf{E}[\mathcal{E}S_{\text{rcy}}(\leq r)]}{(1-\rho(\leq r))(1-\rho(<r))}}^{\mathbf{E}[T_\pi^{\text{wait}}(\ell,s)]} + \overbrace{\int_0^s \frac{1}{1-\rho(<r(a))}\,\mathrm{d}a}^{\mathbf{E}[T_\pi^{\text{wait}}(\ell,s)]}.$$

Proof. Combining Lemmas 7.10–7.12 and 7.14 yields (a). From this, (b) and (c) follow from standard results for M/G/1 busy periods (Prop. 5.5 and Cor. 5.6), because we can view a $\mathbb{Y}$-busy period as an ordinary busy period in a modified M/G/1 with size distribution $S(\mathbb{Y})$. □

7.5.2 More Explicit Formulas for Relevant Size and Recycled Size

Theorem 7.15 is written in terms of the distributions of relevant size $S(\leq r)$ and recycled size $S_{\text{rcy}}(\leq r)$. These distributions depend on just the label-size distribution (L, S) and the rank function. That is, they can be determined without worrying about queueing at all. With that said, it is still helpful to have a more explicit formula for these quantities. We give a few such formulas below, with different formulas being useful in different contexts.

Throughout the following, let r be either a rank or a limit rank (Def. 7.8). This allows us to focus on the non-strict case $\leq r$, as the strict case $<r$ for ordinary ranks r is equivalent to the non-strict case for a limit rank, namely $<r-$.

The first step to writing more explicit formulas for $S(\leq r)$ and $S_{\text{rcy}}(\leq r)$ is to write a more explicit description of the set of relevant states $\leq r$, which we recall from Definition 7.2(a) is

$$\leq r \coloneqq \{(\ell, a) \in \mathbb{L} \times \mathbb{R}_{\geq 0} \mid \mathsf{rank}_\pi(\ell, a) \leq r\}.$$

The following definition writes $\leq r$ in terms of the ages at which the rank function crosses r.

Definition 7.16. Let $\ell \in \mathbb{L}$ be a label, π be a SOAP policy, and r be a rank or limit rank.

(a) The *ith $\leq r$-relevant interval* (or *0th $\leq r$-interval*) of label ℓ, denoted $\mathbb{A}_{\ell,i}(\leq r)$, is the ith interval of ages during which the rank of a job with label ℓ is at most r. For $i = 0$, we define

$$\begin{aligned} b_{\ell,0}(\leq r) &\coloneqq 0, \\ c_{\ell,0}(\leq r) &\coloneqq \inf\{c \geq 0 \mid \mathsf{rank}(\ell, c) > r\}, \\ \mathbb{A}_{\ell,0}(\leq r) &\coloneqq \big[b_{\ell,0}(\leq r), c_{\ell,0}(\leq r)\big). \end{aligned}$$

[12] Specifically, when $\theta > 0$, there is a unique positive real solution, and taking the analytic continuation covers other values of θ.

and for $i \geq 1$, we define[13]

$$\begin{aligned} b_{\ell,i}(\leq r) &:= \inf\{b > c_{\ell,i-1}(\leq r) \mid \mathsf{rank}(\ell, b) \leq r\}, \\ c_{\ell,i}(\leq r) &:= \inf\{c \geq b_{\ell,i} \mid \mathsf{rank}(\ell, a) > r\}, \\ \mathbb{A}_{\ell,0}(\leq r) &:= \big(b_{\ell,0}(\leq r), c_{\ell,0}(\leq r)\big). \end{aligned}$$

If $b_{\ell,i}(\leq r) = c_{\ell,i}(\leq r) = \infty$, then we define the ith $\leq r$-interval to be empty.

(b) The *service in the ith $\leq r$-relevant interval*, denoted $d_{\ell,i}(\leq r, a)$, is the length of the largest interval (b, a) such that a job with label ℓ has rank at most r for all ages in (b, a):

$$d_{\ell,i}(\leq r, a) := a - \inf\Big\{b \in [0, a] \;\Big|\; (b, a) \cap \mathbb{A}_{\ell}(\leq r) = (b, a)\Big\}.$$

For any age a, it can be that $d_{\ell,i}(\leq r, a) > 0$ for at most one value of i. We call this value the *service while $\leq r$-relevant*:

$$d_{\ell}(\leq r, a) := \max_{i \geq 0} d_{\ell,i}(\leq r, a).$$

Note that $d_{\ell}(\leq r, a) = 0$ if $\mathsf{rank}_{\pi}(\ell, a) > r$.

We can use Definition 7.16 to more explicitly define relevant size and recycled size. The following lemma gives a nice formula for a common case that occurs when computing mean response time (Thm. 7.15(c)) and higher moments of response time.

Lemma 7.17. *Consider an M/G/1 under SOAP policy π, and r be a rank or limit rank. For any differentiable $f\colon \mathbb{R}_{\geq 0} \to \mathbb{R}$ such that $f(0) = 0$,*

$$\rho(\leq r)\, \mathbf{E}[f(\mathcal{E}S(\leq r))] = \lambda \mathbf{E}\left[\int_0^{\infty} f'(d_{L,0}(\leq r, a))\, \mathbf{P}[S_L \geq a]\, \mathrm{d}a\right],$$

$$\begin{aligned} &\rho(\leq r)\, \mathbf{E}[f(\mathcal{E}S(\leq r))] + \rho_{\mathrm{rcy}}(\leq r)\mathbf{E}[f(\mathcal{E}S_{\mathrm{rcy}}(\leq r))] \\ &\qquad = \lambda \mathbf{E}\left[\int_0^{\infty} f'(d_{L}(\leq r, a))\, \mathbf{P}[S_L \geq a]\, \mathrm{d}a\right]. \end{aligned}$$

In both cases, the expectation is taken over a random label L.

Proof. We prove the formula for $\rho(\leq r)\, \mathbf{E}[f(\mathcal{E}S(\leq r))]$. The other formula follows from a very similar argument, with the main difference being that we consider not just $\leq r$-fresh jobs but also $\leq r$-recycled jobs.

Our approach is to interpret $\rho(\leq r)\, \mathbf{E}[f(\mathcal{E}S(\leq r))]$ as the expectation of a random variable V. Imagine sampling a steady-state system. If the server is idle, we let $V := f(0) = 0$. Otherwise, the job in service is in state (ℓ, a). We want V to be $f(a)$ if the job is $\leq r$-fresh and

[13] It is worth clarifying the distinction between the 0th and 1st $\leq r$-intervals. If age 0 has rank at most r, then the 0th $\leq r$-interval is nonempty. If age 0 has rank greater than r, then the 0th $\leq r$-interval is empty, with $c_{\ell,0}(\leq r) = 0$. In this latter case, it is still possible for the 1st $\leq r$-interval to start at age 0, e.g. if there is a jump discontinuity with $\mathsf{rank}_{\pi}(\ell, 0) > r \geq \mathsf{rank}_{\pi}(\ell, 0+)$.

$f(0) = 0$ otherwise. Using Definition 7.16(b), one can easily show that $V \coloneqq f(d_{\ell,0}(\leq r, a))$ accomplishes this.

We claim that

$$\rho(\leq r)\,\mathbf{E}[f(\mathcal{E}S(\leq r))] = \mathbf{E}[V]. \tag{7.5}$$

From (7.5), the desired formula follows from computing $\mathbf{E}[V]$, which, after applying ideas from the following discussion, is a simple exercise in integration by parts. To show (7.5), recall from Section 5.6.3 that we can think of the excess $\mathcal{E}S(\leq r)$ in the following way. Imagine a renewal process where each cycle has length distributed as $S(\leq r)$. If we sample this process in steady-state, the time since the last cycle start is distributed as $\mathcal{E}S(\leq r)$. But this is exactly the situation we see in defining V in the case where the system is busy with an $\leq r$-fresh job, which happens with probability $\rho(\leq r)$. □

We note that both Definition 7.16 and Lemma 7.17 can be generalized. Both could use any set of job states $\mathbb{Y} \subseteq \mathbb{L} \times \mathbb{R}_{\geq 0}$ instead of one of the form $\leq r$, and, at the cost of some additional notation, Lemma 7.17 can be altered to work for functions with $f(0) \neq 0$.

Chapter 8

Work Decomposition Laws

The goal of this chapter is to analyze relevant system work (§ 7.2) in the M/G/1, M/G/k, and other systems. We have two main motivations for doing so:

- The SOAP analysis in Chapter 7 reduces the problem of analyzing a SOAP policy's response time to analyzing the system relevant work in an M/G/1 using that SOAP policy.
- In Part III, we will introduce a technique that allows us to use relevant work to analyze mean response time in systems beyond the M/G/1, including multiserver systems like the M/G/k.

As such, in this chapter, we consider general *M/G systems*, meaning any queueing system with M/G arrivals (§ 5.2.2). Our results apply not only to relevant system work, but also to ordinary system work. To emphasize when we are discussing ordinary system work as opposed to relevant system work, we call the former *total* system work.

This chapter's results continue the tradition of *work decomposition laws* (§ 2.4.1) in the sense that they all have the form

$$\text{work in generic M/G system} = \text{work in an M/G/1} + \text{some gap},$$

where the M/G/1 experiences the same arrival process as the generic M/G system. While the M/G/1 terms is explicit in each case, we can only characterize the gap term implicitly. Nevertheless, the gap can be tractable to bound for the M/G/k, which will serve us well in Part III.

We begin by characterizing total system work (§ 8.1). In addition to serving as a nice warmup, the results are of interest in their own right, as they provide a step towards characterizing the mean response time of FCFS in the M/G/k (§ 8.2). We then characterize relevant system work, which is slightly more complicated because we have to account for recyclings (§ 8.3).

This chapter is based on material from Scully et al. [118], though the results here are significantly more general.

8.1 Total Work Decomposition

8.1.1 Rate Conservation Law for Total System Work under M/G Arrivals

System work W goes up and down over time.

- Work increases according to the M/G arrival process.

- Work decreases according to the *service rate* of the system, namely the sum of the rates at which each jobs' remaining work is decreasing. We denote the system's service rate by J.

As long as the system is stable and has a steady-state distribution (§ 5.6.1), the average rate at which system work increases should equal the average rate at which it decreases. The average increase rate is $\lambda \mathbf{E}[S] = \rho$, so this tells us

$$\mathbf{E}[J] = \rho. \tag{8.1}$$

This, of course, is a standard result in queueing which can be derived using Little's law [84].

The key idea of this chapter is that we can apply the above reasoning not just to system work W, but also to *any function of* W. This is an instance of the *Rate Conservation* from Palm calculus [95], which is a general statement of the principle that increases balance decreases in steady-state stochastic processes. Applied to system work under M/G arrivals, the Rate Conservation Law yields the following.

Proposition 8.1. *Let $f : \mathbb{R}_{\geq 0} \to \mathbb{R}_{\geq 0}$ be differentiable. In any steady-state system with M/G arrivals,*

$$\mathbf{E}[Jf'(W)] = \lambda \mathbf{E}[f(W + S) - f(W)],$$

where the job size S on the right-hand side is independent of the system work W.[1]

Proof. The left-hand side is the average rate at which $f(W)$ decreases, and by PASTA (§ 5.4.2), the right-hand side is the average rate at which $f(W)$ increases. The Rate Conservation Law [95, Theorem 2.1] states that these are equal. □

8.1.2 Mean Total Work Decomposition

By applying Proposition 8.1 applied to a quadratic function, we obtain a formula for the mean amount of work in any system with M/G arrivals.

Theorem 8.2 (Mean Total Work Decomposition)**.** *Consider an M/G system. The mean system work is*

$$\mathbf{E}[W] = \mathbf{E}[W_{\text{min}}] + \mathbf{E}[W_{\text{gap}}] = \frac{\frac{\lambda}{2}\mathbf{E}[S^2] + \mathbf{E}[(1 - J)W]}{1 - \rho},$$

where $\mathbf{E}[W_{\text{min}}]$ is the mean system work in an M/G/1 under a work-conserving scheduling policy, namely

$$\mathbf{E}[W_{\text{min}}] = \frac{\frac{\lambda}{2}\mathbf{E}[S^2]}{1 - \rho} = \frac{\rho \mathbf{E}[\mathcal{E}S]}{1 - \rho},$$

[1] Specifically, in the expectation on the right-hand side, W represents the system work immediately prior to a job's arrival, and S represents the new arrival's size. The arriving job cannot affect the system prior to its arrival, so S and W are independent.

and W_{gap} is a random variable with mean[2]

$$\mathbf{E}[W_{\text{gap}}] = \mathbf{E}[W] - \mathbf{E}[W_{\min}] = \frac{\mathbf{E}[(1-J)W]}{1-\rho}.$$

Proof. We apply Proposition 8.1 with $f(w) = \frac{1}{2}w^2$, so $f'(w) = w$. This yields

$$\begin{aligned}\mathbf{E}[JW] &= \lambda \mathbf{E}\left[\tfrac{1}{2}(W+S)^2 - \tfrac{1}{2}W^2\right] \\ &= \tfrac{\lambda}{2}\mathbf{E}[S^2] + \lambda \mathbf{E}[SW].\end{aligned}$$

Because S and W are independent above (Prop. 8.1), we have $\lambda \mathbf{E}[SW] = \rho \mathbf{E}[W]$. The result follows from adding $\mathbf{E}[W]$ to both sides and rearranging terms. □

Interpreting Work Decomposition

To understand Theorem 8.2, let us first consider an M/G/1 that is *not* work-conserving but still stable. We can interpret $1 - J$ as an indicator of whether the server is idle. We have $\mathbf{E}[1-J] = 1-\rho$ by (8.1), which leads to the following interpretation of the mean work gap:

$$\mathbf{E}[W] - \mathbf{E}[W_{\min}] = \mathbf{E}[W \mid \text{the server is idle}]. \tag{8.2}$$

More generally,

More generally, consider a system with maximum service rate 1 but that does not always fully use this maximum service rate. One example is the M/G/k with servers of service rate $1/k$ (§ 5.2.4). We can now view $1-J$ as the "idleness" of the system, meaning the fraction of the service rate that remains unused. We end up with an interpretation similar to (8.2), but instead of conditioning on the server being idle, we take an "idleness-weighted" sample of the work. One way to formalize this is to condition on a coin flip with probability determined by the system's idleness:

$$\mathbf{E}[W] - \mathbf{E}[W_{\min}] = \mathbf{E}[W \mid \text{Bernoulli}(1-J) = 1]. \tag{8.3}$$

Both of the above interpretations of Theorem 8.2 assume a maximum service rate of 1. While the result still holds for systems with greater maximum service rate, it may be harder to characterize the $\mathbf{E}[(1-J)W]$ term in such cases.

8.1.3 Distributional Total Work Decomposition

We can characterize the distribution of system work in much the same way as its mean. The key is to apply Proposition 8.1 with an exponential function, as opposed to a quadratic function. This yields the LST of system work, which happens to factor nicely.

[2] We define the random variable W_{gap} in Theorem 8.3.

Theorem 8.3 (Distributional Total Work Decomposition). *Consider an M/G system. Let $W_{\min}$ be the system work in an M/G/1 under a work conserving scheduling policy.*

(a) The system work can be written as an independent sum[3]

$$W =_{\text{st}} W_{\min} + W_{\text{gap}},$$

where $W_{\min}$ is the system work in a work-conserving M/G/1, namely

$$W_{\min} =_{\text{st}} \sum_{i=1}^{\text{Geo}(1-\rho)} (\mathcal{E}S)_i,$$

and W_{gap} is a random variable with LST given in (b) below. If the system's maximum service rate is 1, we may interpret W_{gap} as "idleness-weighted" system work, namely

$$W_{\text{gap}} =_{\text{st}} (W \mid \text{Bernoulli}(1 - J) = 1).$$

(b) Consider an M/G system. The system work LST is

$$\mathcal{L}[W](\theta) = \mathcal{L}[W_{\min}](\theta)\, \mathcal{L}[W_{\text{gap}}](\theta) = \frac{\mathbf{E}[(1 - J)\exp(-\theta W)]}{1 - \rho \mathcal{L}[\mathcal{E}S](\theta)},$$

where $W_{\min}$ is the system work in a work-conserving M/G/1, which has LST

$$\mathcal{L}[W_{\min}](\theta) = \frac{1 - \rho}{1 - \rho \mathcal{L}[\mathcal{E}S](\theta)},$$

and W_{gap} is the random variable with LST

$$\mathcal{L}[W_{\text{gap}}](\theta) \coloneqq \frac{\mathbf{E}[(1 - J)\exp(-\theta W)]}{1 - \rho}.$$

Proof. We prove (b), from which standard results for LSTs yield (a). We apply Proposition 8.1 with $f(w) = \exp(-\theta w)$, so $f'(w) = -\theta \exp(-\theta w)$. This yields

$$\begin{aligned} -\theta \mathbf{E}[J \exp(-\theta W)] &= \lambda \mathbf{E}[\exp(-\theta(W + S)) - \exp(-\theta W)] \\ &= -\lambda \mathbf{E}\big[\exp(-\theta W)\,\big(1 - \exp(-\theta S)\big)\big]. \end{aligned}$$

Because S and W are independent above (Prop. 8.1), we have

$$\begin{aligned} \mathbf{E}\big[\exp(-\theta W)\,\big(1 - \exp(-\theta S)\big)\big] &= \mathbf{E}[\exp(-\theta W)]\, \mathbf{E}[1 - \exp(-\theta S)] \\ &= \mathcal{L}[W](\theta)\,\big(1 - \mathcal{L}[S](\theta)\big). \end{aligned}$$

After adding $\mathcal{L}[W](\theta) = \mathbf{E}[\exp(-\theta W)]$ and using the fact that

$$\frac{\lambda\big(1 - \mathcal{L}[S](\theta)\big)}{\theta} = \rho \mathcal{L}[\mathcal{E}S](\theta), \qquad \text{[by Prop. 5.3]}$$

rearranging terms yields the desired formula for $\mathcal{L}[W](\theta)$. □

[3]The definition we give for W_{gap} is in terms of an expression that itself involves the system work. Nevertheless, the two terms on the right-hand side are independent random variables. One way to think about this is that W on the left-hand side and W_{gap} on the right-hand side are each defined using independent copies of the system, and $W_{\min}$ is defined using an M/G/1 that is independent of both copies.

8.2 Why Work Decomposition Is Useful for the M/G/k

8.2.1 Mean Response Time of FCFS in terms of System Work

There is no simple expression known for mean response time $\mathbf{E}[T_{\text{FCFS-}k}]$ of an M/G/k under FCFS-k, namely the k-server version of FCFS. But Goldberg[4] has observed that we can write $\mathbf{E}[T_{\text{FCFS-}k}]$ in terms of $\mathbf{E}[W_{\text{FCFS-}k}]$, the mean system work under FCFS-k:[5]

$$\mathbf{E}[T_{\text{FCFS-}k}] = \overbrace{\frac{\mathbf{E}[W_{\text{FCFS-}k}]}{\rho} - k\mathbf{E}[\mathcal{E}S]}^{\text{E[time in queue]}} + \overbrace{k\mathbf{E}[S]}^{\text{E[time in service]}}. \tag{8.4}$$

To see why this holds, first note that $\mathbf{E}[W_{\text{FCFS-}k}] - k\rho\mathbf{E}[\mathcal{E}S]$ is the mean total size of jobs in the queue. All of these jobs have not yet begun service, implying they have expected size $\mathbf{E}[S]$, so dividing by $\mathbf{E}[S]$ gives the mean number of jobs in the queue. Applying Little's law [84] then yields the mean time jobs spend in the queue. Mean time in service is simply mean job size $\mathbf{E}[S]$ over the service rate $1/k$.

Combining Theorem 8.2 and (8.4) reduces the problem of analyzing FCFS-k's mean response time to the problem of analyzing $\mathbf{E}[(1 - J_{\text{FCFS-}k})W_{\text{FCFS-}k}]$. As an example of how we might do this, we show below how we can sometimes bound the analogue of this term in an M/G/k under any non-idling scheduling policy π-k, of which FCFS-k is an example.

8.2.2 Bounding System Work with Work Decomposition

Consider an M/G/k with a non-idling scheduling policy π-k, meaning π-k never leaves servers idle while there are jobs in the queue. We can apply (8.3) to this system. While $\mathbf{E}[W_{\pi\text{-}k} \mid \text{Bernoulli}(1-J_{\pi\text{-}k}) = 1]$ is hard to characterize exactly, we can make an observation about it: whenever $\text{Bernoulli}(1 - J_{\pi\text{-}k}) = 1$, there must be at least one idle server, in which case there are at most $k - 1$ jobs in the system. This means the mean work gap can be bounded, roughly speaking, as

$$\mathbf{E}[W_{\pi\text{-}k}] - \mathbf{E}[W_{\min}] \leq \mathbf{E}[\text{“remaining work of } k-1 \text{ jobs”}].$$

For some size distributions, such as exponential distributions, we can uniformly bound the expected remaining work of a job by some constant $s_{\max}$ no matter what state the job is in. In these situations, we can formalize the above bound, obtaining

$$\mathbf{E}[W_{\pi\text{-}k}] - \mathbf{E}[W_{\min}] \leq (k-1)s_{\max}. \tag{8.5}$$

[4] Personal communication with David A. Goldberg, April 2022. The argument below is due to him.

[5] Unlike in the work-conserving M/G/1, in the M/G/k, the steady-state system work depends on the scheduling policy, even among non-idling scheduling policies.

8.2.3 Looking Ahead to Relevant Work Decomposition

Equation (8.5) is only useful if there is a uniform bound $s_{\max}$ on a job's expected remaining work, which is only the case for some size distributions. However, as we will see in Section 8.3.2, we can derive an analogue of (8.5) for *relevant* system work which demans a uniform bound on *relevant* remaining work. For some sets of relevant states $\mathbb{Y}$, it may be that a job's expected remaining $\mathbb{Y}$-work is easy to bound. For example, consider $\leq r$-work under SRPT's rank function. Any job's remaining $\leq r$-work is at most r, because jobs with greater remaining work have rank greater than r and are thus $\leq r$-irrelevant.

Of course, under policies other than FCFS, it is less clear how to relate total or relevant system work to response time. Part III introduces a new technique that does exactly this (Ch. 15), allowing us to analyze SRPT and Gittins in the M/G/k (Ch. 17).

8.3 Relevant Work Decomposition

8.3.1 Rate Conservation Law for Relevant System Work under M/G Arrivals

Analogous to J, we write $J(\mathbb{Y})$ for the $\mathbb{Y}$-*relevant service rate* of the system, namely the total rate at which system $\mathbb{Y}$-work is decreasing, or equivalently the total service rate of all $\mathbb{Y}$-jobs in the system.

We write $\mathbf{E}_{\text{rcy}}(\mathbb{Y})[V]$ for the expectation of random variable V, which may be a function of the system state, sampled at the instants immediately before $\mathbb{Y}$-recyclings.[6] Inside such an expectation, $S_{\text{rcy}}(\mathbb{Y})$ represents the recycled size of the job undergoing $\mathbb{Y}$-recycling.

Proposition 8.4. *Let $\mathbb{Y} \subseteq \mathbb{L}\times\mathbb{R}_{\geq 0}$ be a set of job states and $f : \mathbb{R}_{\geq 0} \to \mathbb{R}_{\geq 0}$ be differentiable. In any steady-state system with M/G arrivals,*

$$\begin{aligned}\mathbf{E}\big[J(\mathbb{Y})\, f'\big(W(\mathbb{Y})\big)\big] = \lambda \mathbf{E}\big[f\big(W(\mathbb{Y}) + S(\mathbb{Y})\big) - f\big(W(\mathbb{Y})\big)\big] \\ + \lambda_{\text{rcy}}(\mathbb{Y})\, \mathbf{E}_{\text{rcy}}(\mathbb{Y})\big[f\big(W(\mathbb{Y}) + S_{\text{rcy}}(\mathbb{Y})\big) - f\big(W(\mathbb{Y})\big)\big],\end{aligned}$$

where the job $\mathbb{Y}$-size $S(\mathbb{Y})$ on the right-hand side is independent of the system $\mathbb{Y}$-work $W(\mathbb{Y})$.[7]

Proof. The left-hand side is the average rate at which $f(W(\mathbb{Y}))$ decreases. By PASTA (§ 5.4.2), the first term on the right-hand side is the average rate at which $f(W(\mathbb{Y}))$ increases. The second term on the right-hand side is the average rate at which $f(W(\mathbb{Y}))$ increases due to $\mathbb{Y}$-recyclings. The Rate Conservation Law [95, Theorem 2.1] states that the average decrease rate equals the sum of these average increase rates. □

[6] If multiple recyclings occur simultaneously, we assume they happen in some order at the same moment in time. The state immediately after the first recycling is the state immediately before the second, the state immediately after the second is the state immediately before the third, and so on.

[7] In contrast, the $\mathbb{Y}$-recycled size $S_{\text{rcy}}(\mathbb{Y})$ is *not* necessarily independent of the system $\mathbb{Y}$-work $W(\mathbb{Y})$ when sampled immediately before the $\mathbb{Y}$-recycling.

Corollary 8.5. *Let $\mathbb{Y} \subseteq \mathbb{L} \times \mathbb{R}_{\geq 0}$ be a set of job states. In any steady-state system with M/G arrivals,*

$$\mathbf{E}[J(\mathbb{Y})] = \rho(\mathbb{Y}) + \rho_{\text{rcy}}(\mathbb{Y}).$$

Proof. We apply Proposition 8.4 with $f(w) = w$, then simplify the right-hand side using Definitions 7.4(c) and 7.5(e). □

8.3.2 Mean Relevant Work Decomposition

Definition 8.6. Let $\mathbb{Y} \subseteq \mathbb{L} \times \mathbb{R}_{\geq 0}$ be a set of job states.

(a) An M/G/1 scheduling policy is $\mathbb{Y}$-*relevant-prioritizing* (or $\mathbb{Y}$-*prioritizing*) if
- it always serves a $\mathbb{Y}$-job if there is one in the system, and
- it never causes simultaneous $\mathbb{Y}$-recyclings.

For example, a SOAP policy is $\leq r$-prioritizing for all ranks r (Prop. 7.13).

(b) The *minimal M/G/1* $\mathbb{Y}$-*relevant system work* (or *minimal M/G/1* $\mathbb{Y}$-*work*), denoted $W_{\text{min}}(\mathbb{Y})$, is the system $\mathbb{Y}$-work in a steady-state M/G/1 under any $\mathbb{Y}$-prioritizing scheduling policy. That this quantity does not depend on the particular $\mathbb{Y}$-prioritizing scheduling policy chosen follows from Theorem 8.8, which we prove later in this section.

Theorem 8.7 (Mean Relevant Work Decomposition)**.** *Let $\mathbb{Y} \subseteq \mathbb{L} \times \mathbb{R}_{\geq 0}$ be a set of job states.*

(a) The mean minimal M/G/1 $\mathbb{Y}$-work is

$$\begin{aligned}\mathbf{E}[W_{\text{min}}(\mathbb{Y})] &= \frac{\frac{\lambda(\mathbb{Y})}{2}\mathbf{E}[S(\mathbb{Y})^2] + \frac{\lambda_{\text{rcy}}(\mathbb{Y})}{2}\mathbf{E}[S_{\text{rcy}}(\mathbb{Y})^2]}{1-\rho(\mathbb{Y})} \\ &= \frac{\rho(\mathbb{Y})\,\mathbf{E}[\mathcal{E}S(\mathbb{Y})] + \rho_{\text{rcy}}(\mathbb{Y})\,\mathbf{E}[\mathcal{E}S_{\text{rcy}}(\mathbb{Y})]}{1-\rho(\mathbb{Y})}.\end{aligned}$$

(b) Consider an M/G system. The mean system $\mathbb{Y}$-work is

$$\mathbf{E}[W(\mathbb{Y})] = \mathbf{E}[W_{\text{min}}(\mathbb{Y})] + \frac{\mathbf{E}\big[\big(1-J(\mathbb{Y})\big)\,W(\mathbb{Y})\big] + \lambda_{\text{rcy}}(\mathbb{Y})\,\mathbf{E}_{\text{rcy}}(\mathbb{Y})\big[S_{\text{rcy}}(\mathbb{Y})\,W(\mathbb{Y})\big]}{1-\rho(\mathbb{Y})}.$$

Proof. Throughout, we omit "$(\mathbb{Y})$" to reduce clutter.

We apply Proposition 8.4 with $f(w) = \frac{1}{2}w^2$, so $f'(w) = w$. This yields

$$\begin{aligned}\mathbf{E}[JW] &= \lambda\mathbf{E}\big[\tfrac{1}{2}(W+S)^2 - \tfrac{1}{2}W^2\big] + \lambda_{\text{rcy}}\mathbf{E}_{\text{rcy}}\big[\tfrac{1}{2}(W+S_{\text{rcy}})^2 - \tfrac{1}{2}W^2\big] \\ &= \tfrac{\lambda}{2}\mathbf{E}[S^2] + \tfrac{\lambda_{\text{rcy}}}{2}\mathbf{E}[S_{\text{rcy}}^2] + \lambda\mathbf{E}[SW] + \lambda_{\text{rcy}}\mathbf{E}_{\text{rcy}}[S_{\text{rcy}}W].\end{aligned}$$

Because S and W are independent above (Prop. 8.4), we have $\lambda\mathbf{E}[SW] = \rho\mathbf{E}[W]$. Adding $\mathbf{E}[W]$ to both sides and rearranging terms, we obtain

$$\mathbf{E}[W] = \frac{\frac{\lambda}{2}\mathbf{E}[S^2] + \frac{\lambda_{\text{rcy}}}{2}\mathbf{E}[S_{\text{rcy}}^2] + \mathbf{E}[(1-J)W] + \lambda_{\text{rcy}}\mathbf{E}_{\text{rcy}}[S_{\text{rcy}}W]}{1-\rho}.$$

The result follows from observing the following facts about an M/G/1 under a relevant-prioritizing scheduling policy.

- Because the policy always prioritizes relevant jobs if any are present, we have
$$\mathbf{E}[(1-J)W] = \mathbf{E}[\mathbb{1}(W=0)\,W] = 0.$$
- Because the policy never recycles an irrelevant job if a relevant job is present, $W = 0$ immediately prior to recyclings, so $\mathbf{E}_{\text{rcy}}[S_{\text{rcy}}W] = 0$. □

8.3.3 Distributional Relevant Work Decomposition

Theorem 8.8 (Distributional Relevant Work Decomposition)**.** *Let $\mathbb{Y} \subseteq \mathbb{L} \times \mathbb{R}_{\geq 0}$ be a set of job states.*

(a) The minimal M/G/1 $\mathbb{Y}$-work distribution is
$$W_{\min}(\mathbb{Y}) =_{\text{st}} \sum_{i=0}^{\text{Geo}(1-\rho(\mathbb{Y}))} (\mathcal{E}S(\mathbb{Y}))_i + \text{Bernoulli}\left(\frac{\rho_{\text{rcy}}(\mathbb{Y})}{1-\rho(\mathbb{Y})}\right) \mathcal{E}S_{\text{rcy}}(\mathbb{Y}),$$
and its LST is
$$\mathcal{L}[W_{\min}(\mathbb{Y})](\theta) = \frac{1-\rho(\mathbb{Y})-\rho_{\text{rcy}}(\mathbb{Y})+\rho_{\text{rcy}}(\mathbb{Y})\,\mathcal{L}[\mathcal{E}S_{\text{rcy}}(\mathbb{Y})](\theta)}{1-\rho(\mathbb{Y})\,\mathcal{L}[S(\mathbb{Y})](\theta)}$$

(b) Consider an M/G system. The system $\mathbb{Y}$-work can be written as an independent sum
$$W(\mathbb{Y}) =_{\text{st}} W_{\min}(\mathbb{Y}) + W_{\text{gap}}(\mathbb{Y}),$$
where $W_{\text{gap}}(\mathbb{Y})$ is a random variable with LST
$$\mathcal{L}[W_{\text{gap}}(\mathbb{Y})](\theta) := \frac{\left(\begin{array}{l}\mathbf{E}\big[\big(1-J(\mathbb{Y})\big)\exp(-\theta W(\mathbb{Y}))\big] \\ \quad + \rho_{\text{rcy}}(\mathbb{Y})\,\mathbf{E}_{\text{rcy}}(\mathbb{Y})\Big[\frac{1-\exp(-\theta S_{\text{rcy}}(\mathbb{Y}))}{\theta\mathbf{E}[S_{\text{rcy}}(\mathbb{Y})]}\exp(-\theta W(\mathbb{Y}))\Big]\end{array}\right)}{1-\rho(\mathbb{Y})-\rho_{\text{rcy}}(\mathbb{Y})+\rho_{\text{rcy}}(\mathbb{Y})\mathcal{L}[\mathcal{E}S_{\text{rcy}}(\mathbb{Y})](\theta)}.$$

Proof. We apply Proposition 8.4 with $f(w) = \exp(-\theta w)$, incorporating recyclings as in the proof of Theorem 8.7. We then compute similarly to the proof of Theorem 8.3. The result follows from the fact that in an M/G/1 under a $\mathbb{Y}$-prioritizing scheduling policy, we have $\mathcal{L}[W_{\text{gap}}(\mathbb{Y})](\theta) = 1$, and thus $W_{\text{gap}}(\mathbb{Y}) =_{\text{st}} 0$, by reasoning similar to the end of the proof of Theorem 8.7. □

8.3.4 A Useful Corollary: Relevant-Prioritizing Policies Minimize Relevant Work

Corollary 8.9. *Let $\mathbb{Y} \subseteq \mathbb{L} \times \mathbb{R}_{\geq 0}$ be a set of job states. In the M/G/1, system $\mathbb{Y}$-work is stochastically minimized, and thus its mean is minimized, by any $\mathbb{Y}$-prioritizing scheduling policy. That is, letting π be a $\mathbb{Y}$-prioritizing policy and π' be another policy,*
$$W_{\min} =_{\text{st}} W_{\pi}(\mathbb{Y}) \leq_{\text{st}} W_{\pi'}(\mathbb{Y}).$$

In particular, for any SOAP policy π, any other policy π', and any rank r,

$$W_\pi(\mathsf{rank}_\pi \le r) \le_{\mathrm{st}} W_{\pi'}(\mathsf{rank}_\pi \le r),$$
$$W_\pi(\mathsf{rank}_\pi < r) \le_{\mathrm{st}} W_{\pi'}(\mathsf{rank}_\pi < r),$$

where "$\mathsf{rank}_\pi \le r$" is "$\le r$" with π's rank function, and similarly with $<$ in place of $\le$.[8]

Proof. The result for a general set of states $\mathbb{Y}$ follows immediately from Theorem 8.8 and a fact used in its proof, namely that $W_{\mathrm{gap}}(\mathbb{Y}) =_{\mathrm{st}} 0$ under any $\mathbb{Y}$-prioritizing scheduling policy. The SOAP policy result then follows from the fact that any SOAP policy π is $(\mathsf{rank}_\pi \le r)$-prioritizing and $(\mathsf{rank}_\pi < r)$-prioritizing. □

[8] We write "$\mathsf{rank}_\pi \le r$" as opposed to simply "$\le r$" to emphasize that it is π's rank function being used on both sides of the inequality. We formally define this notation in Chapter 15 (Def. 15.1).

Chapter 9

Practical Preemption Limitations

When it comes to job preemption, most queueing theory literature studies one of two extreme cases: either preemption is completely unrestricted and incurs no overhead,[1] or preemption is entirely disallowed. However, many practical systems lie somewhere between these two extremes. This chapter numerically applies the SOAP analysis from Chapter 7 to two case studies with intermediate limitations on preemption.

- (§ 9.1) We study scheduling in settings where the scheduler can only use a limited number of priority levels. Jobs in the same priority level cannot preempt each other, so having limited priority levels restricts preemption.
- (§ 9.1) We study scheduling in settings where jobs can only be preempted at certain checkpoints. We assume that each checkpoint incurs some overhead. This creates a tricky tradeoff: more frequent checkpoints allow for more flexibility in preempting jobs, but at the cost of additional load due to overhead.

This chapter is based on material from Scully and Harchol-Balter [123].

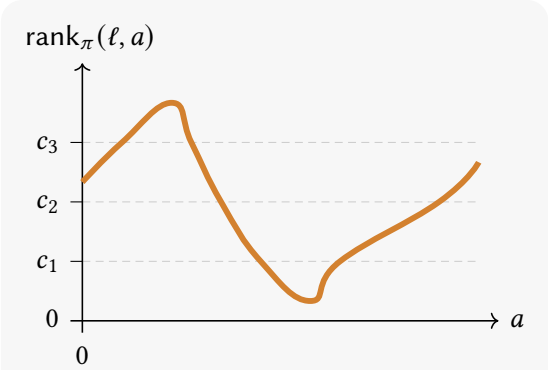
9.1 Limited Priority Levels

Many practical computer systems permit only a small finite number of priority levels. For example, network switches have a small number of priority levels, typically at most 8, built into their hardware [96]. Similarly, the Linux kernel packet scheduler has a configurable number of priority levels, with a default of 3 [57]. We call this the *Limited-Priority-Level* (LPL) setting.

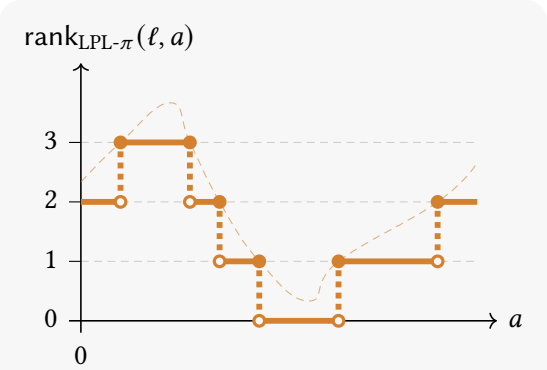
When using a SOAP policy (Ch. 6), a priority level corresponds to a rank. Unfortunately, many SOAP policies assume a continuum of ranks. For example, under SRPT (Pol. 6.10), a job's rank is its remaining work, for which there are infinitely many possible values. System designers who would like to implement policies like SRPT in LPL settings are thus confronted with the challenge of approximating their policy using a finite number of possible ranks. LAS (Pol. 6.4), SERPT (Pol. 6.11), Gittins (Pol. 6.12), and other SOAP policies suffer from the same problem.

Given an "ideal" SOAP policy, such as SRPT, a common approach to scheduling in the LPL setting is to categorize jobs into levels based on their rank under the ideal policy [54, 57, 86, 96]. However, it is not clear how best to do this. Continuing with the SRPT example, if one has only two priority levels, we can assign rank 1 to jobs smaller than a size cutoff c and rank 2 to jobs larger than c, but this begs the question: how do we choose c?

[1] A notable case of queueing theory accounting for some kind of switching overhead is the literature on polling systems, for which Boon et al. [19] and Borst and Boxma [21] provide recent surveys. However, the type of switching overhead is different than preemption overhead



(a) Rank function of generic SOAP policy π.



(b) Rank function of LPL-π, with rank of π shown for reference (dashed orange line).

Figure 9.1. Given a SOAP policy π and a set of rank cutoffs $\{c_1, c_2, c_3\}$, we construct LPL-π, a policy that uses only four ranks, and can thus be implemented with just four priority levels.

In this section, we give guidelines for designing scheduling policies in LPL settings. We tackle the following system design questions:

- How many priority levels do we need to rival the performance of ideal policies?
- How should we choose the rank cutoffs between levels?
- Which ideal policy works best when adapted to the LPL setting?

We spend most of this section addressing the above questions in LPL settings with known job sizes (§ 9.1.2) and unknown job sizes (§ 9.1.3). But before diving in, we briefly review how one adapts scheduling policies to the LPL setting in general (§ 9.1.1).

9.1.1 Adapting SOAP Policies to the LPL Setting

Policy 6.14 gives a general way of constructing an LPL policy from a generic SOAP policy π and set of rank thresholds R. The resulting policy is called *LPL-π*. Below, we briefly review the construction from Policy 6.14, which is also illustrated in Figure 9.1.

Consider a SOAP policy π which uses a continuum of priority levels. We create LPL-π by restricting π's rank function to one of finitely many values in the following way. Suppose our system allows n priority levels. We choose a number of *rank cutoffs* $c_1, c_2, \ldots, c_{n-1}$, defining $c_0 = 0$ and $c_n = \infty$ as edge cases. Whenever π would assign a job a rank between

Table 9.1. Highly variable job size distributions.

Name	Density Function	Coefficient of Variation
Bounded Pareto	$f_S(x) \approx 1/x^2$ for $1 \leq x < 100000$	$C_S^2 \approx 753$
Weibull	$f_S(x) = x^{-3/4}\exp(-x^{1/4})/4$ for $x \geq 0$	$C_S^2 = 69$

c_{i-1} and c_i, LPL-π assigns the job rank c_i. That is,[2]

$$\mathsf{rank}_{\text{LPL-}\pi}(\ell, a) \coloneqq \begin{cases} 0 & \text{if } \mathsf{rank}_\pi(\ell, a) \in [0, c_1) \\ 1 & \text{if } \mathsf{rank}_\pi(\ell, a) \in [c_1, c_2) \\ \vdots & \\ n-2 & \text{if } \mathsf{rank}_\pi(\ell, a) \in [c_{n-2}, c_{n-1}) \\ n-1 & \text{if } \mathsf{rank}_\pi(\ell, a) \in [c_{n-1}, \infty). \end{cases}$$

Figure 9.1 illustrates this rank function. Note that its range includes only n ranks, meaning it can be implemented in a system that has only n priroity levels.

9.1.2 LPL Scheduling with Known Job Sizes: LPL-SRPT

We now address the question of how to minimize mean response time in the LPL setting when job sizes are known to the scheduler. Inspired by SRPT's optimality, we investigate LPL-SRPT. We can think of LPL-SRPT as assigning rank i to jobs with remaining work between cutoffs c_{i-1} and c_i.

Figure 9.2 compares LPL-SRPT to SRPT as a function of the number of priority levels for two different job size distributions, which are described in Table 9.1. For brevity, we show only moderate load $\rho = 0.8$, but the trends are virtually identical at other loads. Optimizing LPL-SRPT's mean response time amounts to tuning the cutoffs $c_1, c_2, \ldots, c_{n-1}$. We consider two strategies for tuning the cutoffs:

- *Heuristic cutoffs:* we set the rank cutoffs to split load evenly between the ranks, a simple heuristic which has been used in practice [57, 96]. More formally, this means equalizing $\mathbf{E}[S\mathbb{1}(S \in [c_i, c_{i+1}))]$ for $i \in \{0, \ldots, n-1\}$. The resulting cutoffs thus depend on the size distribution S but *not* on the load ρ.
- *Optimal cutoffs:* we numerically optimize the rank cutoffs to yield minimal mean response time. The resulting cutoffs depend on both the job size distribution S and the load ρ.

The job size distributions in Table 9.1 are both high-variance job size distributions. We expect our conclusions generalize to other high-variance job size distributions. The low-variance case turns out to be less interesting, because FCFS, which use only a single priority level, already offers reasonable performance.

[2]Policy 6.14 uses c_i instead of i in its definition of rank_π, but this does not affect the scheduling policy, because only the relative ordering of ranks matters.

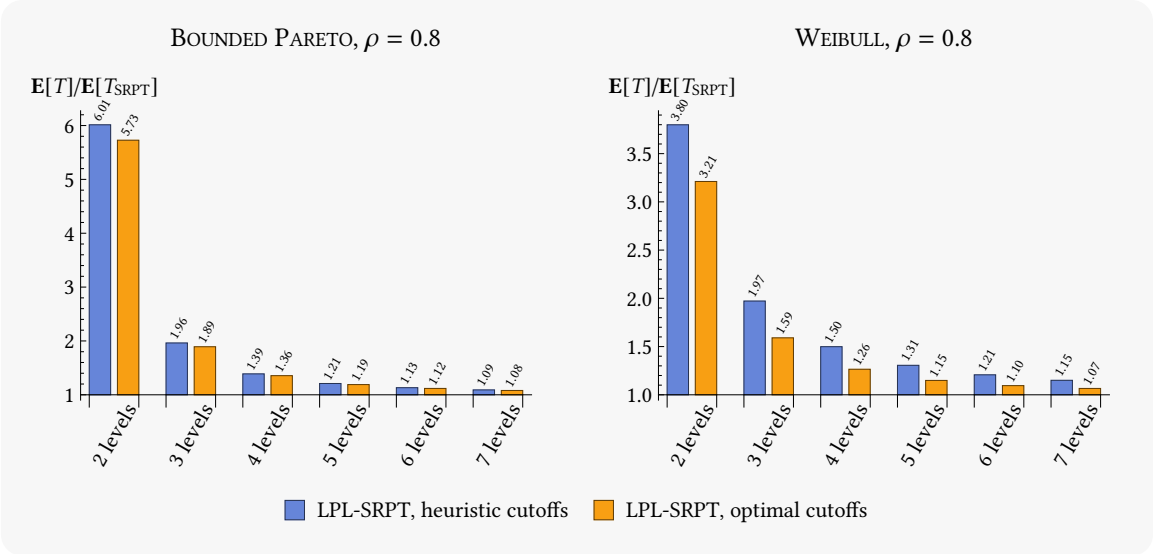


Figure 9.2. Mean response time of LPL-SRPT as a function of number of levels. Heuristic cutoffs are chosen so an equal load of jobs has size between each pair of thresholds. Optimal cutoffs are numerically optimized to minimizes mean response time.

How Many Priority Levels Do We Need?

We see that *roughly 6 priority levels is enough* to approach the mean response time of SRPT, the optimal ideal policy in the size-aware setting. Even with the heuristic cutoffs, 6 levels gives mean response time within 21% of SRPT's at the moderate load $\rho = 0.8$. This supports the empirical findings of LPL system designers: Harchol-Balter et al. [57] use 6 levels, and Montazeri et al. [96] use up to 7 levels.

With this said, we note that using just 2 priority levels is still an order-of-magnitude improvement over FCFS. This is because FCFS has very poor mean response time for the high-variance job size distributions, because small and medium jobs can get stuck behind very long jobs [55].

How Should We Choose Rank Cutoffs?

We see that the *load-balancing heuristic cutoffs serve as a good rule of thumb*. In additional numerical experiments, omitted for brevity, we tried several other simple heuristics, such as evenly splitting the number of arrivals in each bucket or using a geometric sequence for the rank cutoffs c_i. None of these alternatives performed well as consistently as the load-balancing heuristic used in Figure 9.2.

Additional examples at other loads, omitted for brevity, show that the gap between the heuristic and optimal cutoffs becomes more important under higher load, especially for small numbers of priority levels. However, using optimal cutoffs presents a practical difficulty: the optimal cutoffs depend on the load ρ. Fortunately, if we optimize the cutoffs for $\rho = 0.8$, additional numerical experiments, omitted for brevity, show that we achieve

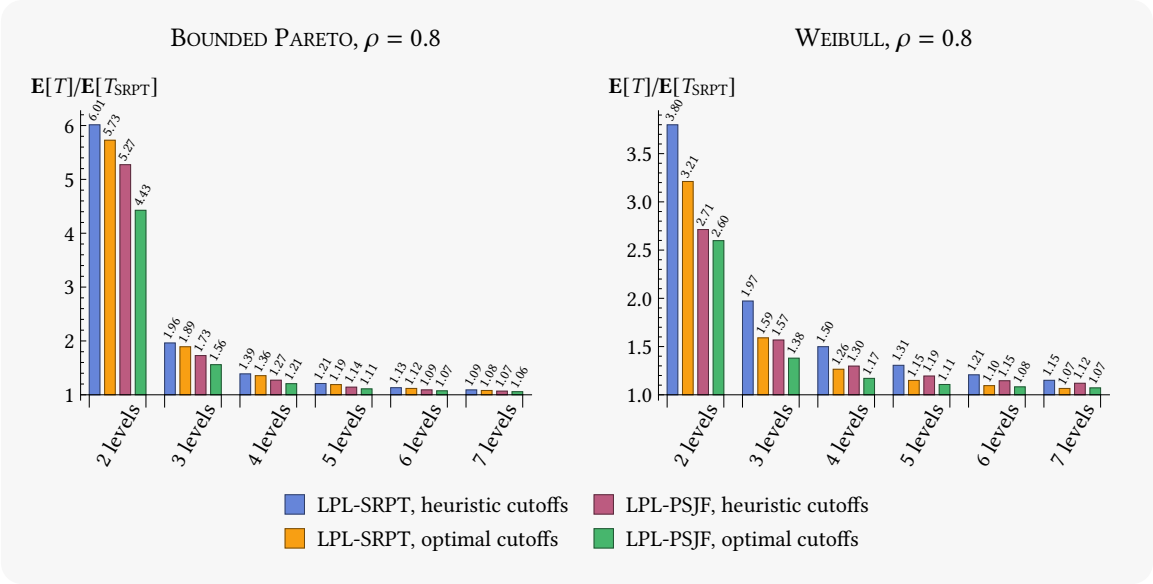


Figure 9.3. Mean response time of LPL-PSJF as a function of number of levels. Note that whether cutoffs are heuristic or optimized, LPL-PSJF outperforms LPL-SRPT. This is surprising given that (ideal) SRPT minimizes mean response time and thus outperforms (ideal) PSJF.

near-optimal performance for all but the highest loads.

Which Ideal Policy Should We Start With?

We began this section by focusing on LPL-SRPT. However, even though SRPT is the optimal policy with infinite priority levels, LPL-SRPT turns out *not* to be the best policy in the LPL setting.

Recall that SRPT assigns ranks using *remaining* work Policy 6.10. This means that LPL-SRPT "upgrades" jobs to the next priority level when they become small enough. Unfortunately, because of the limited number of priority levels, this can cause smaller (new) jobs to have to wait behind larger (recently upgraded) jobs. Fortunately, it turns out that a simple alteration to LPL-SRPT can avoid this issue: never change a job's rank. This results in a policy we call *LPL-PSJF*.[3] The name comes from the *Preemptive Shortest Job First* (PSJF) policy (Pol. 6.9), in which a job's rank is its *original* size, as opposed to its remaining work. Figure 9.3 shows that, counter to intuition, LPL-PSJF outperforms LPL-SRPT.

It is also possible to improve upon LPL-PSJF. However, LPL-PSJF is already an appealing choice for practice [96], so we leave exploring this to future work.

[3]One can view LPL-PSJF as a version of the P-Prio policy (Pol. 6.8).

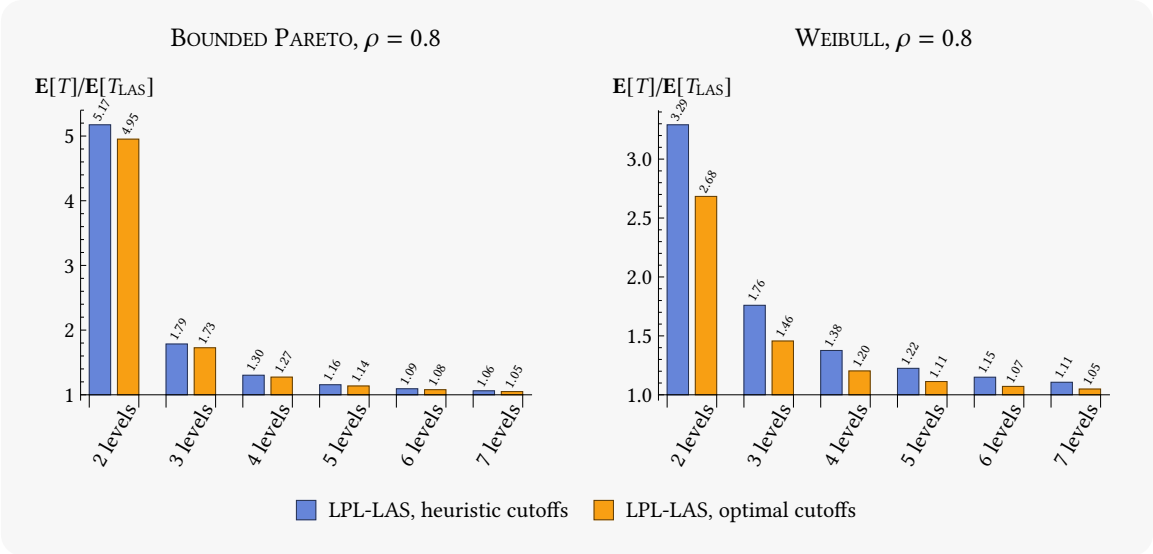


Figure 9.4. Mean response time of LPL-LAS as a function of number of levels. As in Figure 9.2, heuristic cutoffs equalize load of jobs with sizes between each pair of thresholds, and optimal cutoffs numerically optimize to minimize mean response time.

9.1.3 LPL Scheduling with Unknown Job Sizes

We now turn to LPL settings with unknown job sizes. For low-variance job size distributions, FCFS already performs well (§ 5.5.1), so we once again focus on the high-variance job size distributions from Table 9.1. For these distributions, LAS has either optimal or near-optimal mean response time, so LPL-LAS is a promising policy for the LPL setting. We can think of LPL-LAS as assigning rank i to a job with age between cutoffs c_{i-1} and c_i.

Figure 9.4 compares LPL-LAS to LAS as a function of the number of priority levels for two different job size distributions, which are described in Table 9.1. We consider the same two strategies for tuning the cutoffs as in Section 9.1.2: load-balancing *heuristic cutoffs* and numerically-solved *optimal cutoffs*.

How Many Priority Levels Do We Need?

We reach a conclusion similar to the setting of known job sizes: *roughly 5 priority levels is enough* to approach the mean response time of LAS, which is optimal or near-optimal for the job size distributions in Table 9.1. Even with the heuristic cutoffs, 5 levels gives mean response time within 23% of LAS's at the moderate load $\rho = 0.8$.

With this said, as with known job sizes, using just 2 priority levels is still an order-of-magnitude improvement over using FCFS.

How Should We Choose Rank Cutoffs?

We see again that *load-balancing heuristic cutoffs serve as a good rule of thumb*, as in the known-size setting. In particular, this heuristic outperformed the other simple heuristics we tried. Using optimal cutoffs again becomes more important at higher load.

Which Ideal Policy Should We Start With?

We have not found an LPL policy that significantly improves upon LPL-LAS when job sizes are unknown, at least for the job size distributions in Table 9.1. This is not too surprising: both job size distributions have "mostly" decreasing hazard rate,[4]

9.1.4 Summary of LPL Scheduling: 5 or 6 Levels with Load-Balancing Cutoffs Is Good Enough

We have seen that when job sizes are highly variable (Tab. 9.1), one can obtain good mean response time in the LPL setting with 5 or 6 priority levels, using LPL-SRPT or LPL-PSJF with known sizes and LPL-LAS with unknown sizes. Even just 2 priority levels gives an order-of-magnitude improvement over FCFS. A simple load-balancing heuristic suffices for choosing the rank cutoffs between priority levels, though it is possible to improve upon this already-good heuristic.

Can We Theoretically Support Our LPL Scheduling Conclusions?

While we do not have any direct theoretical analysis of the LPL setting, we prove theorems elsewhere in this thesis that support some of the same themes we see in this chapter, particularly with regards to choosing LAS and PSJF as underlying ideal policies. Specifically, both of these policies have the property that a job's rank never improves (i.e. never decreases), an idea which we see in two theoretical chapters later on.

- In Chapter 11, we propose a policy for unknown job sizes in which a job's rank never improves, and we show its mean response time is always within a factor of 5 of optimal.
- In Chapter 12, we study scheduling using noisily estimated job sizes instead of exact job sizes. One can naively adapt both SRPT and PSJF to this setting by using the same rank function, but plugging in the noisy estimates in place of the true sizes. While the noisy version of SRPT can perform poorly with even a small amount of noise, the noisy version of PSJF is much more robust.

[4]The Weibull distribution has decreasing hazard rate. The Bounded Pareto distribution has decreasing hazard rate up to some age threshold, but jobs that reach this threshold are very rare.

9.2 Preemption Checkpoints

Queueing theorists, very much including the author, often assume that jobs may be freely preempted, but this is far from the case in practical computer systems. Programs can have temporary state at all levels of the memory hierarchy, from registers to RAM. For cloud-based jobs, even the disk may be temporary state. Before preempting a job, we must either save or discard its state.

Checkpointing is one solution used to combat lost state. Every job occasionally saves its transient state, and we allow preemptions only immediately after saving. We refer to ages when a job saves its work as *checkpoints*. Because saving work takes time, each checkpoint adds to a job's size.

A very similar situation occurs when scheduling packet flows in networks, e.g. at a network switch. In this setting, each packet flow is a job, and serving a job corresponds to sending its packets. Packets are indivisible, and each packet incurs overhead in the form of its header. We can think of packet boundaries as analogous to checkpoints.

When scheduling in a system with checkpointing, the key question is: *how much service should occur between checkpoints?* Or, in the context of packet flows: *how large should packets be?* Answering this question requires balancing a delicate tradeoff.

- On one hand, less service between checkpoints allows for quicker preemptions. This decreases mean response time because small jobs are less likely to get stuck during long uninterruptible periods between checkpoints.
- On the other hand, checkpointing takes time, so the less service between checkpoints add more to the system load. This in turn can increase mean response time or even cause instability.

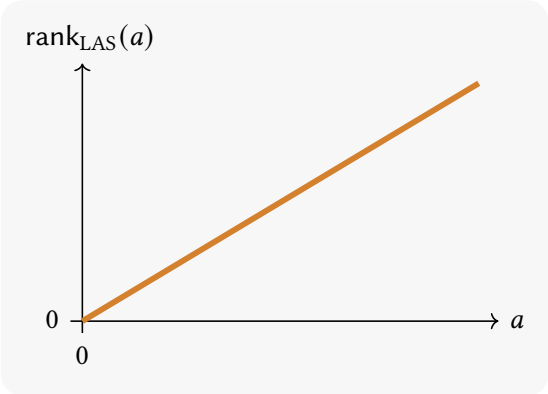
We give a rule of thumb for balancing this tradeoff. After discussing in more detail how we model checkpoints (§ 9.2.1), we determine how frequently checkpoints should occur to minimize mean response time (§ 9.2.2). Throughout, we focus on the case of unknown job sizes. In additional numerical experiments, omitted for brevity, we have observed the same results in the setting with known job sizes.

9.2.1 Jobs with Preemption Checkpoints

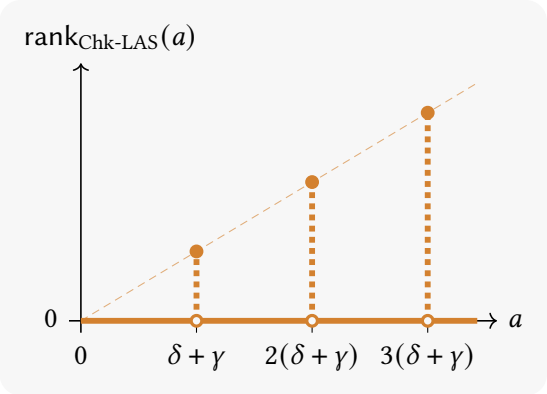
We consider a system where jobs cannot be preempted unless their work is saved. Jobs save their work at specific ages called *checkpoints*. Upon reaching a checkpoint, a job takes a deterministic amount of *overhead time* γ to save its work, after which the job may be preempted.

For simplicity, we study the case where checkpoint ages are distributed evenly with *service quantum* δ. That is, checkpoints occur at ages $0, \delta, 2\delta, \ldots$. Our main task is to optimize the service quantum δ given the job size distribution S, load ρ, and overhead γ.

Overhead time does not represent real progress towards completing the job. To model this, we consider a system with an *overhead-altered job size distribution*. Given original



(a) Rank function of LAS.



(b) Rank function of Chk-LAS, with rank of LAS shown for reference (dashed orange line).

Figure 9.5. Using LAS (Pol. 6.4) but preempting only at age checkpoints yields Chk-LAS (Pol. 6.4). In our setting, there is a gap of $\delta + \gamma$ between checkpoints: after a service quantum δ, an overhead γ occurs, and only after that can the job be preempted.

size distribution S, we define the overhead-altered distribution by

$$\check{S} \coloneqq S + \gamma \left\lfloor \frac{S}{\Delta} \right\rfloor. \tag{9.1}$$

For consistency with our definitions of other SOAP policies, we define scheduling policies in terms of overhead-altered ages, too. In particular, after accounting for overheads, checkpoints actually occur ate ages $0, \delta+\gamma, 2(\delta+\gamma), \ldots$.

A job can only be preempted at checkpoint ages. Given a scheduling policy, we can express this constraint by modifying the scheduling policy's rank function. Specifically, we leave the rank function as-is for checkpoint ages, but for intermediate ages, we set the rank to a low value. This gives a job between checkpoints priority over jobs at checkpoints. We can in principle do this to any SOAP policy π, resulting in a policy we call *Chk-π* (Pol. 6.13). In this section, we focus specifically on Chk-LAS, which is illustrated in Figure 9.5.[5]

9.2.2 Choosing the Right Service Quantum

We focus on a setting with unknown job sizes and high-variance job size distributions, specifically those from Table 9.1.[6] For these job size distributions, LAS has near-optimal mean response time, so we schedule using Chk-LAS (Fig. 9.5).

[5]If we were adapting a generic SOAP policy π, we would want to account for the fact that a job's age includes overheads, but this consideration is not important for LAS. Specifically, if a job's age with overheads is a, then without overheads, the job has received service for $a - \lfloor \frac{a}{\delta+\gamma} \rfloor$ time, resulting in rank function $\text{rank}(\ell, a) = \text{rank}_\pi(\ell, a - \gamma \lfloor \frac{a}{\delta+\gamma} \rfloor)\, \mathbb{1}(a/(\delta+\gamma) \in \mathbb{N})$.

[6]To make the analysis numerically tractable, we actually truncate the distributions at size 5000 and discretize them into increments of size 0.125. The trends we observe are not sensitive to these values.

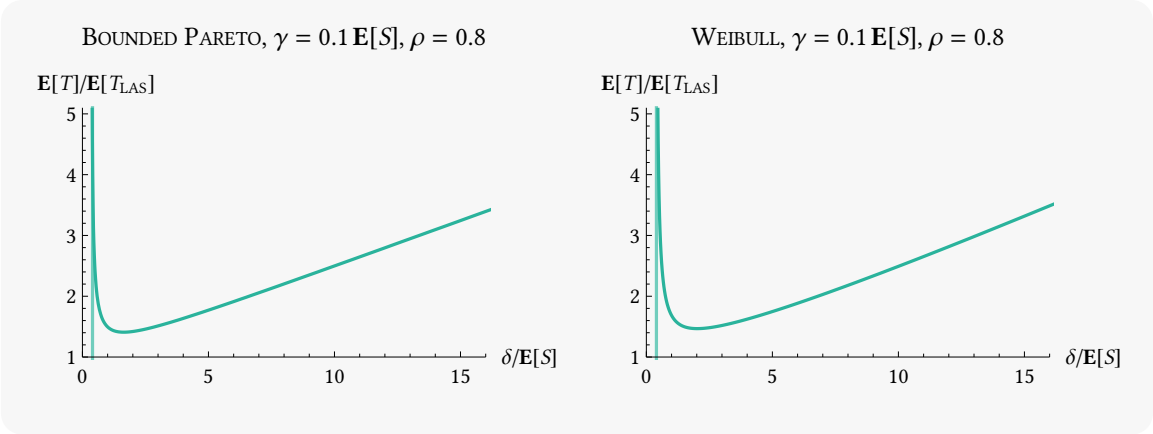


Figure 9.6. Mean response time of Chk-LAS as a function of the normalized service quantum $\delta/\mathbf{E}[S]$ for a system with large checkpoint overhead, namely $\gamma = 0.1\,\mathbf{E}[S]$. We use LAS in a system *without* checkpoints or overhead as a baseline for comparison. We consider Bounded Pareto and Weibull job size distributions (Tab. 9.1) and load $\rho = 0.8$. As we decrease δ, mean response time decreases approximately linearly until a critical value "left wall" value of δ, at which point the system becomes unstable and mean response time approaches infinity. The vertical lines show a conservative estimate, given by (9.2), of where the left wall is.

We show in Figure 9.6 how mean response time varies as a function of the service quantum δ. The figure shows the case of relatively large overhead $\gamma = 0.1\,\mathbf{E}[S]$ at load $\rho = 0.8$. As we will show later, the trends are similar for small overhead and other loads (Fig. 9.7).

For large enough δ, Figure 9.6 shows that mean response time is a roughly linear function of δ on at least a portion of the domain, where smaller δ is better. However, there is a vertical asymptote as δ approaches a small value, which we call the "left wall". This leaves us in a precarious situation: smaller values of δ generally yield lower mean response time, but if δ becomes too small, mean response time suddenly becomes infinite. In the remainder of this section, we explain how to choose a value of δ that is small enough but not too small.

Ensuring Stability with a Safe Service Quantum

As a first step towards optimizing δ, we must figure out the value $\delta_{\text{left wall}}$ at which the left wall asymptote occurs. The asymptote at the left wall is caused by checkpoints becoming so frequent that the overheads make the system unstable, which sends mean response time to infinity.

It turns out that $\delta_{\text{left wall}}$ has a complicated formula, but there is a simpler formula that bounds it. Let

$$\delta_{\text{safe}} = \frac{\gamma\rho}{1-\rho}. \tag{9.2}$$

Plugging $\delta = \delta_{\text{safe}}$ into (9.1) and computing the *overhead-altered load* $\check{\rho} := \lambda\mathbf{E}[\check{S}]$, we see

that $\delta = \delta_{\text{safe}}$ ensures $\check{\rho} < 1$ and thus stability. This holds not just for this example but under any scheduling policy and any job size distribution.

Optimizing the Service Quantum

We want to choose a service quantum δ that is not just stable but also optimizes mean response time. We see from Figure 9.6 that the optimal value of δ is slightly larger than δ_{safe}, but it is not clear how much larger is optimal. Through a number of examples, we have found the following rule of thumb to give near-optimal performance in the typical case when the overhead γ is at most the mean job size $\mathbf{E}[S]$:[7]

$$\delta_{\text{rule of thumb}} = \frac{1}{1-\rho}\sqrt{\frac{\gamma \mathbf{E}[S]}{\rho}}. \tag{9.3}$$

Figure 9.7 show that this rule-of-thumb service quantum yields near-optimal performance for a range of loads ρ and overheads γ.

The intuition behind (9.3) is as follows. We see in Figure 9.7 that mean response time has a "bathtub" shape when plotted as a function of δ on a log scale. Each bathtub is approximately symmetrical near its minimum. This suggests that if we can find the values of δ corresponding to the "walls" of the bathtub, then their geometric mean would be a good rule of thumb. This is exactly the approach we take.

- *Left wall:* we use δ_{safe} as an approximation for the left wall.
- *Right wall:* through additional numerical experiments, omitted for brevity, we have found that the formula $\mathbf{E}[S]/(\rho^2(1-\rho))$ is a good approximation for the right wall. Our search for a good approximation was guided by the observation that the right wall is virtually unaffected by the overhead γ, as we see by comparing Figure 9.7. This makes sense: when the service quantum δ is large, there are very few preemptions, so the overhead γ has little impact of mean response time.

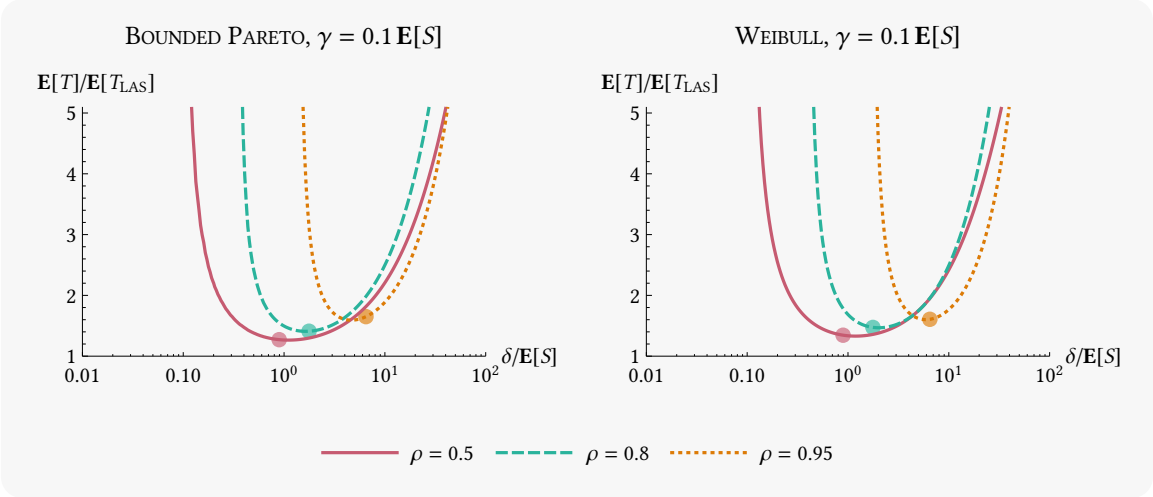
9.2.3 Summary: Rule-Of-Thumb Formula

When scheduling in systems with checkpointing, there is a tradeoff between making checkpoints less frequent, which avoids checkpoint-associated overhead, and more frequent, which enables smarter scheduling. In (9.3), we propose a rule-of-thumb formula that yields a near-optimal service quantum. Although we focus throughout on unknown job sizes, we have observed that this rule of thumb also works well when using Chk-SRPT with known job sizes.

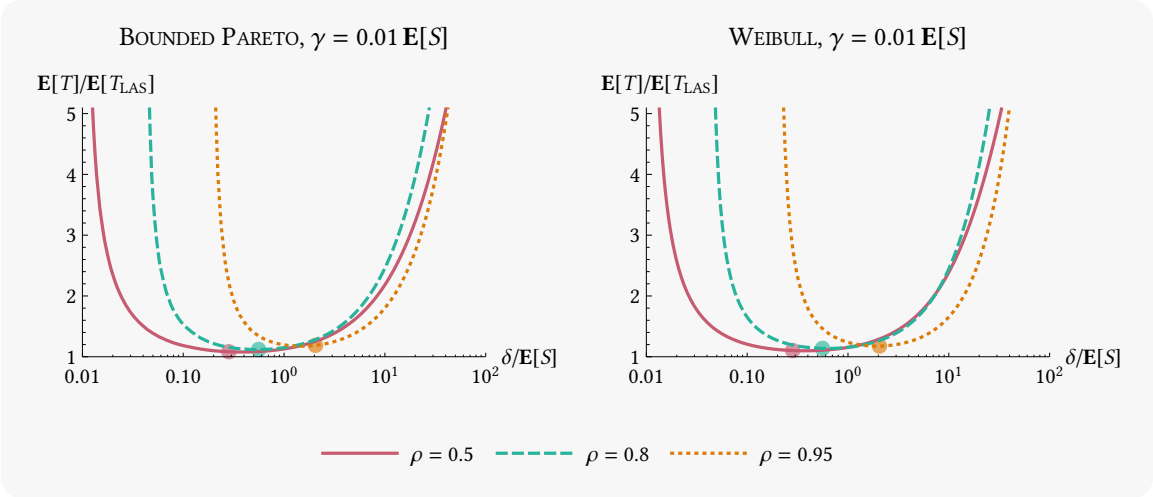
What About Non-Constant Service Quanta?

One can use SOAP to analyze checkpointing with different amounts of service between each pair of checkpoints. In fact, Policy 6.13 is written in a general way that allows for this

[7] One could replace γ by $\max\{\gamma, \mathbf{E}[S]$ in (9.3) to obtain a formula that works even with $\gamma > \mathbf{E}[S]$.



(a) Large checkpoint overhead, namely $\gamma = 0.1\,\mathbf{E}[S]$.



(b) Small checkpoint overhead, namely $\gamma = 0.01\,\mathbf{E}[S]$.

Figure 9.7. Mean response time of Chk-LAS as a function of the normalized service quantum $\delta/\mathbf{E}[S]$ (log scale) for two systems with different checkpoint overheads. We consider Bounded Pareto and Weibull job size distributions (Tab. 9.1) for several values of load ρ. We highlight the rule-of-thumb value of δ given by (9.3).

possibility, and exploring this is a possible direction for future work. With that said, there are two reasons we focus on the case of constant service quanta.

- The first reason is practical: having a constant service quantum, or perhaps a small number of different possible service quanta, is likely the most feasible option in many systems. One example is scheduling packet flows in networks, where packet sizes are generally standardized.
- The second reason is theoretical: in Chapter 13, we show that if the service quanta grow to large as a job's age increases, it can cause poor tail of response time for heavy-tailed job size distributions.

Chapter 10

Gittins vs. Simpler Substitutes

The Gittins policy (Pol. 6.12) minimizes mean response time in the M/G/1 [44]. However, the practicality of actually implementing Gittins is often questioned.

Why might Gittins be impractical? Let us first recall how Gittins is defined. In the unlabeled case, meaning when the scheduler knows no information about any specific job's size (§ 5.2.3), Gittins has rank function

$$\text{rank}_{\text{Gittins}}(a) = \inf_{b>a} \frac{\mathbf{E}[\min\{S, b\} - a \mid S > a]}{\mathbf{P}[S \leq b \mid S \geq a]}. \tag{10.1}$$

Notice that Gittins's rank function involves solving an optimization problem at every age a, where the optimization depends on the job size distribution S.

This brings us to our first reason Gittins may be impractical: solving the optimization problem in (10.1) can be difficult. While it is known to be solvable in polynomial time for discrete size distributions [24], to the best of our knowledge, it is an open question whether Gittins's rank function can be computed efficiently for continuous size distributions.

A second reason Gittins may be impractical is its dependence on the size distribution S. In fact, when jobs have informative labels, then Gittins depends on the entire joint label-size distribution (L, S). But this joint distribution may not always be known exactly in practice. For example, suppose jobs are labeled with noisy size estimates. Even if we have a good idea of what the size distribution is, we may not know have enough data to build a precise model of the noise.

In light of these obstacles, the goal of this chapter is to determine whether simpler policies than Gittins can serve as a reasonable substitute, achieving near-optimal mean response time. We study three different settings:

- (§ 10.1) *Unlabeled:* The scheduler knows no specific information about any particular job, and in particular does not know job sizes.
- (§ 10.2) *Class-labeled:* Jobs come from one of a small number of classes, and each job is labeled with a class.
- (§ 10.3) *Estimate-labeled:* We have some way of estimating a job's size when it arrives, and each job is labeled with its estimate.

In each setting, we take a computational approach, numerically applying the SOAP analysis (Ch. 7) to a variety of examples. Our findings inspire theoretical investigations that appear in Chapters 11 and 12.

Finally, we note that another reason Gittins may be impractical to implement is that it is a preemptive policy. Practical systems can have a variety of preemption limitations, making it impossible to implement the theoretically ideal Gittins. We study this issue in Chapter 9.

Section 10.1 is based on material from Scully and Harchol-Balter [123], but Section 10.3 is new material.

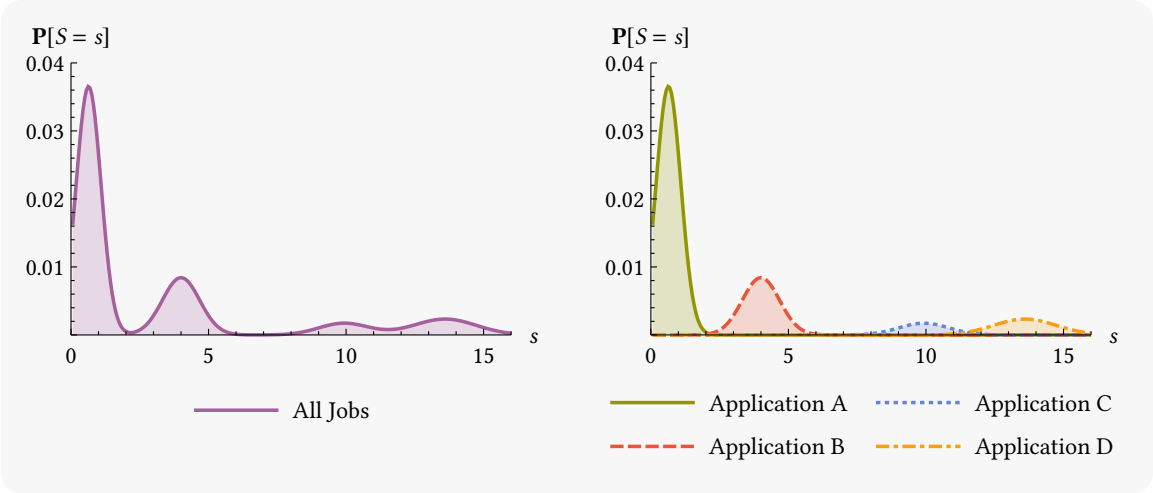


Figure 10.1. Example job size distribution S. For each possible size s, we show the probability that an arriving job's size is s. We can imagine that this size distribution arises from serving jobs from four different applications, each of which has an approximately Gaussian size distribution. For simplicity, in this example, the possible sizes are discretized in increments of 1/16.

10.1 Unknown Sizes: Use SERPT

10.1.1 Imitating Gittins's Rank Function

Looking at (10.1), we see that the Gittins rank function trades off between two ideas. The first is that we want to favor jobs which have small expected remaining work, which is captured by the numerator of. The second is that we want to favor jobs that are likely to complete soon, which is captured by the denominator. Figure 10.1 shows an example of a job size distribution, and Figure 10.2(a) shows what the Gittins rank function looks like for it in the unlabeled case.

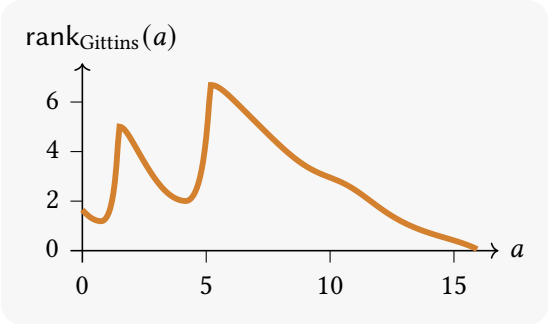
How might we simplify Gittins? One idea is to worry just about a job's expected remaining work, without worrying about the probability it will complete soon. After all, if a job will likely complete soon, that affects its expected remaining work. The policy that always serves the job of least expected remaining work to schedule is SERPT (Pol. 6.11):

$$\text{rank}_{\text{SERPT}}(a) = \mathbf{E}[S - a \mid S > a].$$

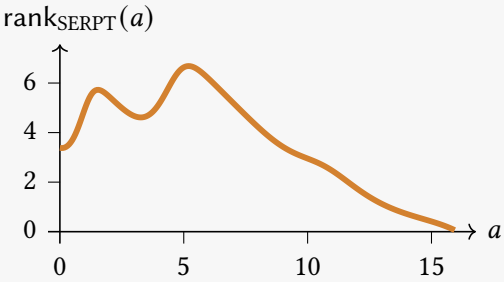
Put another way, $\text{rank}_{\text{SERPT}}$ is like $\text{rank}_{\text{Gittins}}$, but instead of optimizing the parameter b that appears in $\text{rank}_{\text{Gittins}}$, we simply set $b = \infty$.

We show an example of SERPT's rank function in Figure 10.2(b). Comparing it to Gittins's rank function for the same distribution in Figure 10.2(a), we see that the Gittins and SERPT rank functions are rather similar. Both, roughly speaking, have two "hills", and the rank functions are exactly the same after the second "hill".[1]

[1]This is not a coincidence but rather an instance of a more general result [3, Lem. 8].



(a) Rank function of Gittins.



(b) Rank function of SERPT.

Figure 10.2. Rank functions of (a) Gittins and (b) SERPT in the unlabeled case for the job size distribution shown in Figure 10.1. Similarly to the distribution, both rank functions are discretized in age increments of 1/16.

10.1.2 How Does SERPT Compare to Gittins?

Given that SERPT's rank function seems so similar to Gittins's, it is natural to ask: does SERPT also perform similarly to Gittins? This is certainly true for the job size distribution from Figure 10.1. We show in Figure 10.3 the mean response times of several policies for this distribution. We observe that SERPT is nearly optimal, with mean response time within 3% of Gittins's at all loads. Other common policies that work with unknown job sizes, such as FCFS (Pol. 6.5) and LAS (Pol. 6.4), have much worse performance relative to Gittins. It thus seems that SERPT has near-optimal mean response time.

Are These Observations Robust to Parameter Changes?

One might worry that our observations are specific to the size distribution from Figure 10.1. To test whether SERPT is near-optimal under broader conditions, we repeated this section's analysis for 100 different randomly generated scenarios with the goal of finding the worst case scenario for SERPT. Each scenario is similar to Figure 10.1 in that the overall job size distribution is a mixture of jobs from four applications, each with a discretized Gaussian distribution, but we randomly generated the parameters of each Gaussian.

Figure 10.4 summarizes the results by showing the *worst observed mean response time ratio* between several policies and Gittins across all 100 scenarios. We focus on high load $\rho = 0.95$ to emphasize the differences between the scheduling policies.[2] To clarify what Figure 10.4 represents, consider the SERPT bar, which has value 1.072. This means that for the size-oblivious setting at load $\rho = 0.95$, across all 100 generated scenarios, the maximum ratio $\mathbf{E}[T_{\text{SERPT}}]/\mathbf{E}[T_{\text{Gittins}}]$ was 1.072. The bars for FCFS and LAS are computed similarly.

[2] We actually computed the ratio at all loads, but Figure 10.4 shows only $\rho = 0.95$ for simplicity. Accounting for other loads increased SERPT's worst observed mean response time ratio increases by no more than 0.01.

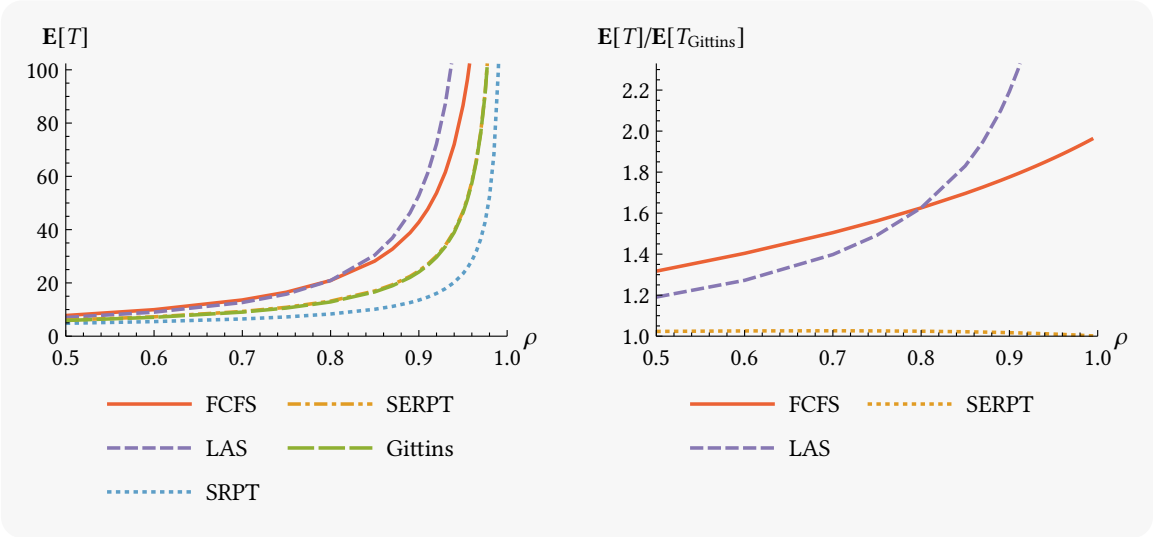


Figure 10.3. Mean response times of several scheduling policies for the job size distribution from Figure 10.1. We show both mean response times (left) and the mean response time ratios relative to Gittins (right), which minimizes mean response time when job sizes are unknown. SERPT performs nearly as well as Gittins, and in particular much better than either FCFS or LAS. We show SRPT as a point of comparison, even though it uses job size information.

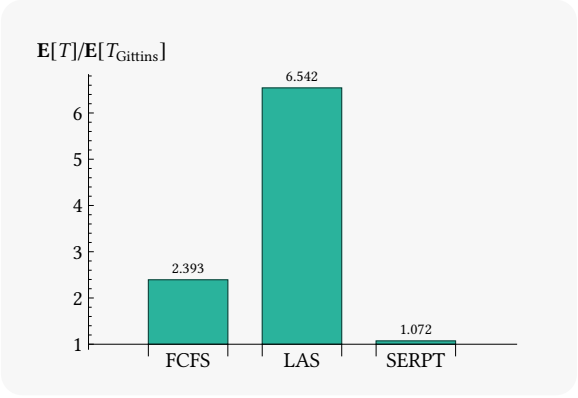


Figure 10.4. Worst observed mean response time ratios relative to Gittins at load $\rho = 0.95$. Each bar is the maximum ratio out of 100 randomly generated scenarios, each of which is a variation of Figure 10.1 with different parameters. Specifically, the job size distribution is a mixture of four applications' distributions, each of which is a Gaussian with uniformly distributed mean and standard deviation, discretized and restricted to the interval $[0, 16]$. We ensure that in each scenario, one application's mean is in each of the intervals $[0, 4]$, $[4, 8]$, $[8, 12]$, and $[12, 16]$.

10.1.3 Can We Theoretically Bound SERPT's Performance?

One might hope to use the SOAP analysis (Ch. 7) to give a theoretical bound on SERPT's mean response time that holds for *all* job size distributions, not just those we have tested. We have not yet been able to prove such a result for SERPT. However, there is hope that such a bound may be provable, because we prove such a bound in Chapter 11 not for SERPT, but for a simple modification of it we call *monotonic SERPT (M-SERPT)*. We show in Chapter 11 that M-SERPT's mean response time ratio compared to Gittins is at most 5 for all job size distributions at all loads: $\mathbf{E}[T_{\text{M-SERPT}}]/\mathbf{E}[T_{\text{Gittins}}] \leq 5$.

We conjecture that a similar result holds for (unmodified) SERPT. The worst-case scenario we have found so far is a particular pathological job size distribution which is unlikely to occur in practice (Ch. 11). Even in this worst-case scenario, SERPT's mean response time ratio compared to Gittins is only 2, whereas FCFS and LAS both have unbounded mean response time ratio compared to Gittins.

10.2 Multiclass Systems: Again, Use SERPT

We have seen in Section 10.2 that when job sizes are unknown, SERPT can serve as a simple substitute for Gittins that performs nearly as well. But it is often the case that we have some amount of information about jobs' sizes. Can we use simpler substitutes for Gittins in these cases? The rest of this chapter studies this question. This section considers a multiclass setting, where jobs come from one of two classes and are labeled with their class, which indirectly gives us some information about the job's size. Section 10.3 below considers the case where each job's size is noisily estimated. This can be thought of as a limiting case of the multiclass setting considered here, with each size estimate constitutes a class of jobs, but the specifics are different enough that we give it is own section.

We can use both Gittins and SERPT in multiclass settings. The only difference is that instead of using the overall size distribution S to compute a job's rank, we use the appropriate label-conditional size distribution S_ℓ (Pols. 6.11 and 6.12), where a job's label is its class. After describing the specific multiclass scenarios we study (§ 10.2.1), we ask two questions:

- (§ 10.2.2) How does SERPT compare to Gittins in the multiclass setting?
- (§ 10.2.3) Can a policy even simpler than SERPT still be near-optimal in the multiclass setting?

10.2.1 Three Multiclass Systems

To compare different multiclass scheduling policies, we consider the following running example. Consider a system with jobs as illustrated in Figure 10.1. Jobs from four applications, which we call A, B, C, and D. If we have one job from each application, it is likely that their size ordering from smallest to largest is A < B < C < D.

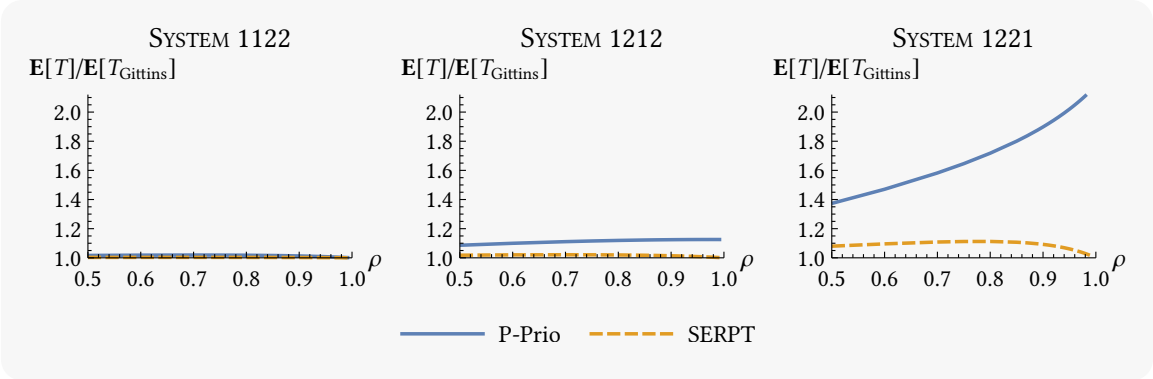


Figure 10.5. Mean response time ratios relative to Gittins for each of SERPT and P-Prio (§ 10.2.3). We consider each of the multiclass systems described in Section 10.2.1. We see that SERPT is consistently near-optimal in all three systems. However, P-Prio is less consistent. It performs well in System 1122, where class 1 jobs are consistently smaller than class 2 jobs. But in System 1221, where class 1 jobs "sandwich" class 2 jobs, P-Prio performs poorly. System 1212 is between these extremes.

In the rest of this section, we consider three ways of splitting the four applications into two classes.

- *System 1122:* class 1 is A and B, and class 2 is C and D.
- *System 1212:* class 1 is A and C, and class 2 is B and D.
- *System 1221:* class 1 is A and D, and class 2 is B and C.

For example, in System 1212, the scheduler knows that a class 2 job comes from application B or D.

10.2.2 Comparing Gittins and SERPT in the Class-Labeled Setting

Figure 10.5 compares the mean response times of Gittins and SERPT in all three multiclass systems described in Section 10.2.1. We see that SERPT is a good substitute for Gittins, as it maintains mean response time within 12% of Gittins's throughout.

That SERPT is near-optimal in this setting is unsurprising if we compare its rank function to that of Gittins. As in the size-oblivious setting, the two rank functions are rather similar. Figure 10.6 demonstrates this for System 1221. Both policies initially assign class 1 jobs low rank, because most class 1 jobs in System 1221 are from application A. However, once a class 1 job has run for long enough, it becomes most likely that it is from application D, meaning it is likely to be large, so both policies' rank functions increase accordingly.

10.2.3 Even Simpler than SERPT: P-Prio

Perhaps the simplest scheduling policy that uses class information is the *Preemptive Priority (P-Prio)* policy (Pol. 6.8). Given an ordering of the classes from best to worst priority, P-Prio

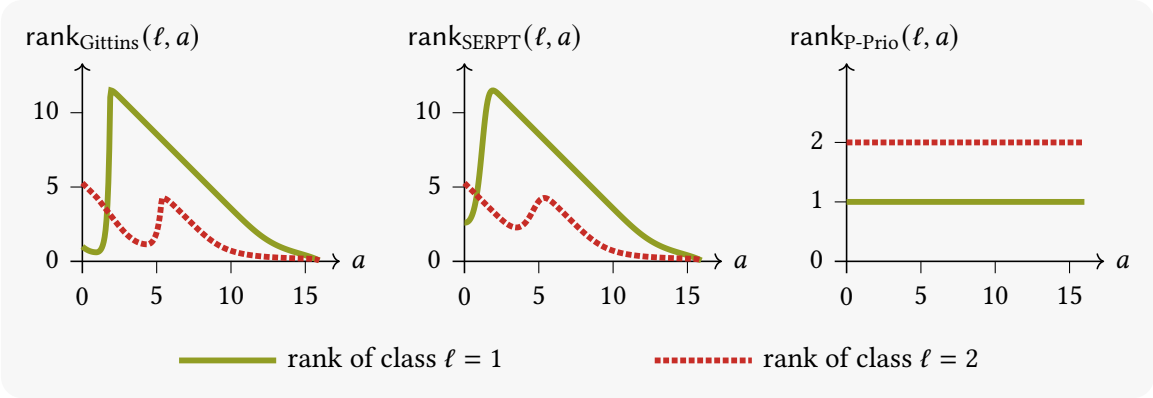


Figure 10.6. Comparison of the rank functions of Gittins, SERPT, and P-Prio (§ 10.2.3) in System 1221 (§ 10.2.1), in which class 1 jobs come from application A or D, and class 2 jobs come from application B or C (Fig. 10.1).

always serves the job in the best possible priority class, serving jobs within each class in FCFS order. One can make P-Prio favor short jobs by ordering the classes in order of expected size.

Can we use P-Prio as a Gittins substitute? To answer this question, we compare P-Prio to SERPT and Gittins in the three multiclass systems described in Section 10.2.1. In all cases, P-Prio prioritizes class 1 over class 2, as class 1 jobs have lower expected size.

Section 10.2.1 shows that P-Prio sometimes performs comparably to SERPT and Gittins, but sometimes SERPT and Gittins are clearly superior. We discuss each of the three systems in more detail below. In additional numerical experiments, we also evaluated NP-Prio (Pol. 6.8), the nonpreemptive cousin of P-Prio. But it turns out P-Prio outperforms NP-Prio in all three systems at all loads, so we omit the NP-Prio results for brevity.

System 1122: P-Prio Is Nearly Optimal

In System 1122, all three of P-Prio, SERPT, and Gittins have very similar mean response time. This makes sense because in this system, the scheduler knows a class 1 job is smaller than a class 2 job with high probability, so strictly prioritizing class 1 jobs is an excellent heuristic.

System 1212: P-Prio Is Okay

In System 1212, P-Prio has worse mean response time than SERPT and Gittins, but only by 10–15%. This makes sense because in this system, the scheduler knows a class 1 job is more likely than not to be smaller than a class 2 job. However, exceptions occur frequently enough that strictly prioritizing class 1 jobs is not as effective as in System 1122.

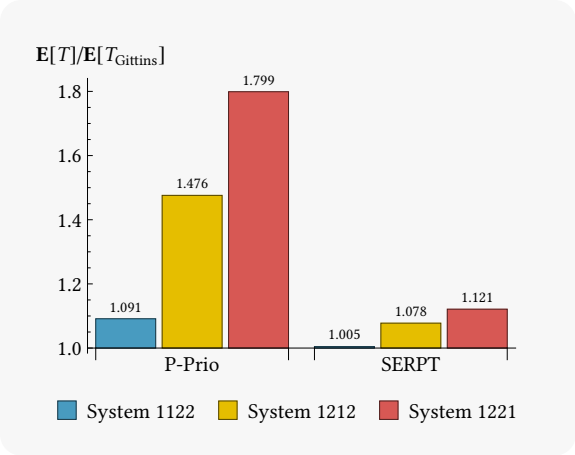


Figure 10.7. Worst observed mean response time ratios relative to Gittins at load $\rho = 0.95$. Each bar represents the maximum ratio out of 100 randomly generated scenarios, each of which is a variation of Figure 10.1 with different parameters. See Figure 10.4 for details of how we generate the parameters. We label the applications in order of increasing mean as A, B, C, and D, from which we define Systems 1122, 1212, and 1221, as described in Section 10.2.

System 1221: P-Prio Is Poor

In System 1221, SERPT and Gittins significantly outperform P-Prio. In fact, using P-Prio turns out to be even worse than using the size-oblivious version of SERPT or Gittins. This makes sense because in this system, given a class 1 job and a class 2 job, it is not clear to the scheduler which is larger. P-Prio prioritizes class 1 over class 2 because class 1 jobs are smaller on average. However, once a class 1 job reaches a large enough age, its remaining work is likely to be large. As shown in Figure 10.6, SERPT and Gittins deal with this by updating a class 1 job's rank during service, but P-Prio is limited by its static priorities.

Are These Observations Robust to Parameter Changes?

As in Section 10.1.2, we want to check that these trends are not too sensitive to the exact parameters of Figure 10.1. We therefore again repeat our analysis with 100 randomly generated variations of the same setup. And once again, as we show in Figure 10.7, our conclusions appear to be robust to changes in the parameters.

10.2.4 Summary: Stick to SERPT

We have seen that SERPT consistently achieves mean response time close to that of Gittins, despite having a significantly simpler rank function. We therefore recommend SERPT as a substitute for Gittins for most situations. In some specific cases, an even simpler policy such as P-Prio may suffice, but there are plenty of cases where Gittins and SERPT outperform these even simpler heuristics.

10.3 Size Estimates: Use PSJF-E for Low Noise, Ignore Estimates for High Noise

We now consider a setting in which the scheduler, rather than being given each job's exact size, is given a noisy estimate of the job's size. Specifically, letting $\sigma \geq 0$ be a *noise level*, we suppose that a job of size s is labeled by an estimate

$$(L \mid S = s) \coloneqq s \exp(\text{Normal}(0, \sigma)).$$

That is, we assume unbiased log-normal multiplicative noise that is independent of the job size.

Throughout this section, we consider job size distributions S that are Weibull with varying shape parameter k. This family of size distributions and noise models has commonly been used to study scheduling with size estimates in the literature (§ 2.4.2). What distinguishes our work from these past works is that, thanks to the SOAP analysis, we can evaluate SERPT and Gittins in addition to simpler heuristics.

One can define SERPT and Gittins for any label-size distribution (L, S) (Pols. 6.11 and 6.12), including the noisy size estimate model described above. It is natural to ask whether SERPT is again a good substitute for Gittins. Numerical experiments, which we omit for brevity, confirm that the answer is again yes.

However, both SERPT and Gittins require knowledge of the exact label-size distribution (L, S), namely an exact description of the noise model. However, in practice, we may not know the exact noise model. Can we achieve near-optimal mean response time using size estimates, but without relying on the specifics of the noise model?

10.3.1 Simple Heuristics for Scheduling with Size Estimates

When we know job sizes exactly, SRPT minimizes mean response time, and PSJF is often nearly as good [55]. One would expect that, at least for low noise σ, naive analogues SRPT and PSJF to work with noisy size estimates would yield good performance. We formally define these analogues below.

Policy 10.1. The *SRPT with Estimates (SRPT-E)* policy is the policy one obtains by plugging possibly noisy size estimates into SRPT's rank function (Pol. 6.10):[3]

$$\text{rank}_{\text{SRPT-E}}(\ell, a) = (\ell - a)^+.$$

Policy 10.2. The *PSJF with Estimates (PSJF-E)* policy is the policy one obtains by plugging possibly noisy size estimates into PSJF's rank function (Pol. 6.9):

$$\text{rank}_{\text{PSJF-E}}(\ell, a) = \ell.$$

That is, PSJF-E is simply P-Prio (Pol. 6.8) where a job's class is its size estimate.

[3] We take the positive part because Definition 6.1 defines ranks to be nonnegative. This choice is not essential, but it makes for simpler notation, especially in Chapter 6.

One advantage of SRPT-E and PSJF-E is that they do not depend on the noise model, or even on the job size distribution. However, we cannot expect good performance from them when noise is very high. In particular, in the $\sigma \to \infty$ limit, the size estimates become useless. In this case, the best performance we can hope for is that of Gittins in the unlabeled case, namely unknown job sizes. To prevent confusion, we call this *Unlabeled Gittins*, to distinguish it from the size-estimate-aware version of Gittins. For the Weibull distributions we consider, Unlabeled Gittins takes a particularly simple form: it is LAS for $k < 1$, and it is FCFS for $k \geq 1$.

10.3.2 Evaluating SRPT-E, PSJF-E, and Unlabeled Gittins

Figure 10.8 compares the mean response times of SRPT-E, PSJF-E, and Unlabeled Gittins to size-estimate-aware Gittins, with SRPT's performance as a baseline. For brevity, we show only representative noise levels $\sigma \in \{0.5, 1.0, 1.5\}$ and shape parameters $k \in \{0.25, 1.0\}$, but the discussion below is supported by additional numerical experiments with other values of σ and k, which we omit for brevity.

What Performance Is Achievable for a Given Noise Level?

Gittins (with access to size estimates) represents the minimum possible mean response time we can hope to achieve. When $\sigma = 0$, meaning job sizes are exactly known, Gittins reduces to SRPT. As σ increases, Gittins's mean response time increases, but never exceeding that of Unlabeled Gittins. In the examples we have observed, including those in Figure 10.8, Gittins's mean response time is very close to that of Unlabeled Gittins when $\sigma \approx 1$. That means when $\sigma \approx 1$, the size estimates are effectively useless, because ignoring them, as Unlabeled Gittins does, does not hurt performance.

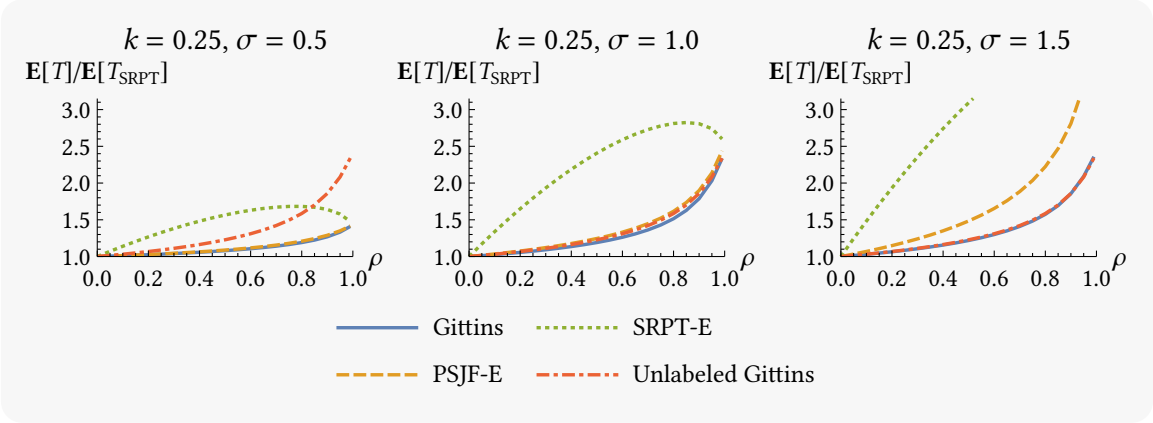
How Close to Optimal Are SRPT-E and PSJF-E?

We see in Figure 10.8 that for shape $k = 1$, SRPT-E has near-optimal performance for a range of noise levels. But for shape $k = 0.25$, SRPT-E performs poorly, even at low noise. We can explain why this is using the SOAP analysis, specifically the mean response time formula (Thm. 7.15(c)). Because the estimation noise is multiplicative, there is a positive probability that a job spends a constant fraction of its service time at rank 0. Using the SOAP analysis (Thm. 7.15(c)), one can show that this means there is a constant c_σ, which depends on σ but not on the size distribution S, such that

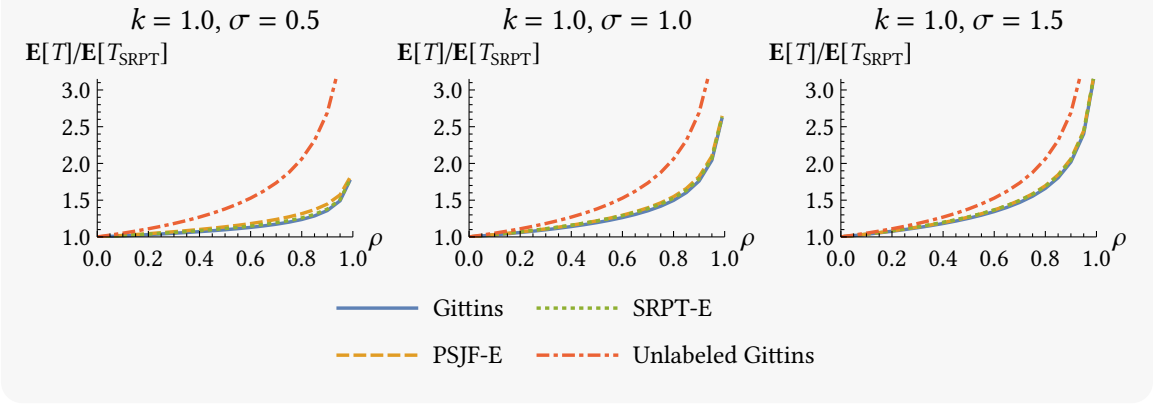
$$\mathbf{E}[T_{\text{SRPT-E}}] \geq c_\sigma \rho \mathbf{E}[S^2]$$

for all loads ρ and size distributions S. We formalize a version of this statement in Chapter 12. The takeaway is that for high-variance job size distributions, SRPT-E is a bad choice.

The story is much better for PSJF-E. Like SRPT-E, it is near-optimal for shape $k = 1$. But unlike SRPT-E, it is also near-optimal for $k = 0.25$, at least for low-to-medium noise



(a) High-variance job size distribution, namely Weibull with small shape parameter $k = 0.25$. Even under low noise, SRPT-E performs poorly. PSJF-E is near-optimal for low noise, and Unlabeled Gittins (which in this case is simply LAS) is near-optimal for high noise. The switch between "low" and "high" noise happens at $\sigma \approx 1.0$.



(b) Low-variance job size distribution, namely Weibull with small shape parameter $k = 1.0$ (i.e. exponential). Here the story is less interesting than the high-variance case: SRPT-E and PSJF-E are both near-optimal for a large range of noise levels σ.

Figure 10.8. Mean response time ratios relative to SRPT (without noise) for several noise levels σ. The job size distribution is a (discretized) Weibull distribution with shape parameter k. We consider both (a) high-variance $k = 0.25$ and (b) low-variance $k = 1.0$. To clarify, (ordinary) Gittins uses the size estimates, while Unlabeled Gittins ignores the size estimates.

$\sigma \leq 1$. This is good news, because as discussed above, this is exactly the range of noise levels for which we have something to gain from using size estimates. The takeaway is that when noise is low enough, we can use the very simple PSJF-E, and when noise is high enough, we can simply ignore the estimates and use Unlabeled Gittins.

10.3.3 Can We Theoretically Bound PSJF-E's Performance?

One might hope to use the SOAP analysis (Ch. 7) to give a theoretical bound on PSJF-E's mean response time that holds for a large class of label-size distributions (L, S) that goes beyond what we have tested here. We address this question in Chapter 12. Specifically, the result we show for PSJF-E is that if the noise is multiplicatively bounded above and below by constants, meaning $L \in [\beta S, \alpha S]$ with probability 1, then PSJF-E's mean response time is within a constant factor of PSJF's when given exact size information: $\mathbf{E}[T_{\text{PSJF-E}}] \leq \frac{\alpha}{\beta}\mathbf{E}[T_{\text{PSJF}}]$. Whether a similar bound holds for unbounded multiplicative noise, such as the log-normal model considered in this section, remains an open question.

Chapter 11

Monotonic SERPT (M-SERPT)

We have seen through several numerical examples in Chapter 10 that in the M/G/1, SERPT has near-optimal mean response time, namely close to that of Gittins, in a wide variety of situations. This raises the question: can we *prove* that SERPT satisfies a mean response time guarantee? This would be a useful result because SERPT's rank function is substantially simpler than Gittins's (Pols. 6.11 and 6.12). Unfortunately, comparing SERPT to Gittins for all job size distributions is difficult, even with the SOAP analysis (§ 11.1). We have not yet been able to prove a mean response time guarantee for SERPT.

What we have been able prove is a guarantee for a new variant of SERPT, which we call *Monotonic SERPT (M-SERPT)*. Specifically, we show that for an M/G/1 with unknown job sizes (i.e. the unlabeled case), M-SERPT is a 5-approximation for minimizing mean response time, meaning its mean response time is a most 5 times that of Gittins (§ 11.2). This is the first result showing a constant approximation ratio for any scheduling policy other than Gittins itself.[1]

We suspect that these results are not the end of the story for SERPT and M-SERPT. We have observed in numerical experiments similar to those in Chapter 10, omitted for brevity, that M-SERPT generally has mean response time similar to SERPT, with each sometimes outperforming the other. Moreover, both policies generally perform much better than 5 times Gittins's mean response time. We therefore conjecture that both policies are actually 2-approximations for mean response time, and we provide a lower bound showing such a result would be tight (§ 11.3).

This chapter is a summary of Scully et al. [125], to which we refer the reader for proofs.

11.1 Problem: Bounding SERPT's Mean Response Time

Our goal is to compare $\mathbf{E}[T_{\text{SERPT}}]$ to $\mathbf{E}[T_{\text{Gittins}}]$. Specifically, we conjecture that the ratio $\mathbf{E}[T_{\text{SERPT}}]/\mathbf{E}[T_{\text{Gittins}}]$ is bounded by a constant for all job size distributions and loads. In principle, the SOAP analysis gives a formula for the mean response time under any SOAP policy (Thm. 7.15(c)). Why is bounding $\mathbf{E}[T_{\text{SERPT}}]/\mathbf{E}[T_{\text{Gittins}}]$ still challenging?

The short answer is that while the SOAP analysis gives us a way to compute $\mathbf{E}[T_{\text{SERPT}}]$ and $\mathbf{E}[T_{\text{Gittins}}]$ for any *particular* size distribution and load, it does not immediately yield a bound that holds for *all* size distributions and loads. Moreover, SERPT and Gittins have rank functions that change depending on the job size distribution, which further complicates

[1] We show another approximation ratio result in Chapter 16. But that result is for policies whose rank functions are close to Gittins's, whereas M-SERPT's rank function can have significant differences from Gittins's.

matters (§ 11.1.1). To work around this, we introduce M-SERPT, a variant of SERPT that ends up having a slightly simpler mean response time formula (§ 11.1.2). As evidenced by the existence of this chapter, this slight simplification makes all the difference.

11.1.1 Why Comparing SERPT and Gittins Is Hard

Examining the mean response time formula as given by combining Theorem 7.15(c) and Lemma 7.17, we see that the mean response time of an M/G/1 using SOAP policy π depends on two things:[2]

- The arrival process, specifically via the arrival rate λ and the size distribution's tail $\mathbf{P}[S > t]$.
- The rank function rank_π of the SOAP policy π, specifically via the worst future rank function (Def. 7.9) and $\leq r$-intervals (Def. 7.16(a)). Recall that an $\leq r$-interval is a maximal interval of ages a during which $\mathsf{rank}_\pi(a) \leq r$.

While the $\mathbf{E}[T]$ formula depends on both the arrival process and the rank function, the dependence on the rank function is much more complicated. For instance, if the job size distribution has support on a finite set, then for a fixed rank function, $\mathbf{E}[T]$ is a rational function of λ and the probabilities of each of the finitely many possible sizes. A similar statement holds for general size distributions, though with a continuous analogue of a rational function.

What makes SERPT and Gittins difficult to analyze for all size distributions is that their rank functions depend on the size distributions. That is, to be precise, SERPT and Gittins are not really individual SOAP policies but rather SOAP policy *constructions*: given a size distribution, they yield a rank function tuned to that distribution.

11.1.2 M-SERPT Simplifies the Story

We have seen that comparing SERPT to Gittins for all job size distributions is difficult because both policies' rank functions depend on the size distribution. To overcome this obstacle, we define a new policy that, while still having a rank function that depends on the job size distribution,

Policy 11.1. The *Monotonic SERPT (M-SERPT)* policy is the SOAP policy with rank function

$$\mathsf{rank}_{\text{M-SERPT}}(a) \coloneqq \max_{0 \leq b \leq a} \mathsf{rank}_{\text{SERPT}}(b) = \max_{0 \leq b \leq a} \mathbf{E}[S - b \mid S > b].$$

As suggested by its name, M-SERPT is indeed monotonic, as a job's rank only increases with age.

One could easily generalize Policy 11.1 to define M-SERPT in the labeled case, or more generally to define a monotonic version of any SOAP policy, but our focus here is on M-SERPT in the unlabeled case.

[2] The discussion below focuses on the unlabeled case, as does the rest of this chapter.

Why is there hope of comparing M-SERPT to Gittins? After all, SERPT and M-SERPT depend on the size distribution S in similar ways. What makes M-SERPT simpler? The key is M-SERPT's eponymous monotonicity, which simplifies the way its mean response time depends on the rank function in two important ways:

- For any size s, the worst future rank of a job of size s is the same at all ages. Specifically, for all $s > a \geq 0$, we have[3]

$$\mathsf{worst}_{\text{M-SERPT}}(s, a) = \mathsf{rank}_{\text{M-SERPT}}(s).$$

- Because a job's rank never decreases, jobs are never recycled (Def. 7.5). This means that for any rank r, there is at most one $\leq r$-interval, which always begins at age 0 and ends at the earliest age with rank greater than r.

Of course, these properties of M-SERPT do not immediately solve the problem of bounding its mean response time relative to Gittins, but they represent a helpful first step.

11.2 Main Result: M-SERPT Is a 5-Approximation for Mean Response Time

Theorem 11.2. *Consider an M/G/1 with any job size distribution and load ρ in the unlabeled case. The mean response time ratio between M-SERPT and Gittins is bounded by*[4]

$$\frac{\mathbf{E}[T_{\text{M-SERPT}}]}{\mathbf{E}[T_{\text{Gittins}}]} \leq \begin{cases} \dfrac{4}{1+\sqrt{1-\rho}} & \text{if } 0 \leq \rho < 0.9587 \\ \dfrac{1}{\rho}\log\dfrac{1}{1-\rho} & \text{if } 0.9587 \leq \rho < 0.9898 \\ 1 + \dfrac{4}{1+\sqrt{1-\rho}} & \text{if } 0.9898 \leq \rho < 1. \end{cases}$$

In particular, the ratio is at most 5, so M-SERPT is a 5-approximation for the problem of minimizing mean response time with unknown job sizes.

See Scully et al. [125, Thm. 5.1] for the proof of Theorem 11.2. The rest of this section discusses two corollaries of Theorem 11.2, each of which resolves an open problem in queueing theory.

11.2.1 LAS for Increasing Mean Residual Lifetime

Job size distributions S where the expected remaining size of a job $\mathbf{E}[S - a \mid S > a]$ is (strictly) increasing as a function of age a are said to have the (strictly) *Increasing Mean*

[3] For simplicity of exposition, we neglect details to do with limit ranks (Def. 7.8) throughout this chapter.

[4] The numbers 0.9587 and 0.9898 are approximations accurate to four decimal places.

Residual Lifetime (IMRL) property. For strictly IMRL size distributions, SERPT and M-SERPT both reduce to a simple, familiar policy: LAS (Pol. 6.4).

LAS was thought for some time to minimize mean response time for IMRL size distributions [111], though it turned out there was an error in the proof [2]. Nevertheless, it has been observed that LAS generally performs well for IMRL size distributions, leaving open the question: can we prove a response time guarantee on LAS for IMRL size distributions? Thus far, it has only been shown that LAS outperforms PS (§ 5.3.1) in this case.

Theorem 11.2 resolves a significant special case of this question, namely the performance of LAS for *strictly* IMRL size distributions. This is because M-SERPT reduces to LAS in this case.

Corollary 11.3. *Consider an M/G/1 with any strictly IMRL job size distribution and any load in the unlabeled case. LAS is a* 5*-approximation for minimizing mean response time.*

11.2.2 Performance Achievable by MLPS Policies

Aside from SOAP, there are few M/G/1 analyses that apply to an entire classes of policies. One of these is the analysis of the *Multi-Level Processor Sharing (MLPS)* policy class [74]. MLPS policies are specified by a list of threshold ages $a_1, a_2, \ldots$, with $a_0 = 0$, where interval $[a_i, a_{i+1})$ is called the *ith level*. Roughly speaking, an MLPS policy prioritizes jobs by the level, with lower levels having priority, and uses one of FCFS, LAS, or PS used within each level.

An MLPS policy is parameterized by the threshold ages a_i and the choice of which policy to use within each level. While there has been significant work on MLPS policies [1, 4, 9], a question remains open: how should one optimize the parameters of an MLPS policy to minimize mean response time? This is an important question because in some applications, implementing MLPS policies may be simpler than implementing general SOAP policies like Gittins.

Theorem 11.2 takes a significant step towards resolving this question, because it turns out M-SERPT is an MLPS policy: age intervals where M-SERPT's rank is strictly increasing are levels that use LAS, and age intervals where M-SERPT's rank is constant are levels that use FCFS. This means that MLPS policies can achieve within a constant factor of optimal mean resposne time.

Corollary 11.4. *Consider an M/G/1 with any job size distribution in the unlabeled case. There exists an MLPS policy which for all loads is a* 5*-approximation for minimizing mean response time.*

11.3 Approximation Ratio Lower Bounds for SERPT and M-SERPT

We have shown that M-SERPT is a 5-approximation for minimizing mean response time. Is this bound tight? That is, is there a size distribution such that $\mathbf{E}[T_{\text{M-SERPT}}]/\mathbf{E}[T_{\text{Gittins}}] = 5$,

or at least a sequence of distributions such that $\mathbf{E}[T_{\text{M-SERPT}}]/\mathbf{E}[T_{\text{Gittins}}]$ approaches 5? Or is the true approximation ratio lower?

We conjecture that M-SERPT is actually a 2-approximation for mean response time, and we conjecture the same for SERPT. This is based on exploring the performance of SERPT and M-SERPT on a wide range of numerical examples, including the use of numerical optimization to find the worst-case ratio relative to Gittins within a class of size distributions. Throughout all this exploration, the maximum ratios $\mathbf{E}[T_{\text{SERPT}}]/\mathbf{E}[T_{\text{Gittins}}]$ and $\mathbf{E}[T_{\text{M-SERPT}}]/\mathbf{E}[T_{\text{Gittins}}]$ I have observed are no more than 2. We describe the scenario that achieves this below. It implies that SERPT and M-SERPT both have approximation ratio at least 2.

The worst mean response time ratios I have observed come from distributions of the form

$$S = \begin{cases} 1-\delta & \text{w.p. } 1-\delta \\ 1 & \text{w.p. } \delta-\delta^2 \\ \delta^{-1}+1 & \text{w.p. } \delta^2, \end{cases}$$

where $\delta \approx 0$ is a small positive number. One can compute using the SOAP analysis [125, § 7] that at high load $\rho \approx 1$,

$$\frac{\mathbf{E}[T_{\text{SERPT}}]}{\mathbf{E}[T_{\text{Gittins}}]} \approx \frac{\mathbf{E}[T_{\text{M-SERPT}}]}{\mathbf{E}[T_{\text{Gittins}}]} \approx \frac{2\delta^3 + 2\delta(1-\rho) + (1-\rho)^2}{\delta^3 + \delta(1-\rho) + (1-\rho)^2}.$$

If we set $\delta = (1-\rho)^{2/3}$, the right-hand side approaches 2 in the $\rho \to 1$ limit.

Curiously, for any *fixed* value of δ in the example above, the right-hand side approaches 1 in the heavy-traffic $\rho \to 1$ limit. This leaves open the possibility that SERPT and M-SERPT could be heavy-traffic optimal for mean response time. However, the example does show that even if $\lim_{\rho\to 1} \mathbf{E}[T_{\text{SERPT}}]/\mathbf{E}[T_{\text{Gittins}}] = 1$ (or similarly for M-SERPT) for any size distribution, the convergence cannot be uniform in the size distribution.

Chapter 12

Adapting SRPT to Noisy Job Size Estimates

This chapter studies the problem of scheduling in an M/G/1 where job sizes are uncertain, but we learn an estimate for each job's size when it arrives. Scheduling in this setting has been the subject of several simulation and numerical studies, including our own in Chapter 10, but there are no strong theoretical bounds on scheduling with size estimates in the M/G/1 (§ 2.4.2).

When scheduling with size estimates, a natural idea is to naively use the rank function of SRPT or PSJF to schedule, but plugging the estimated size into the rank function instead of the true size. This results in policies we call *SRPT with Estimates (SRPT-E)* and *PSJF with Estimates (PSJF-E)*, respectively (Pols. 10.1 and 10.2).

Do SRPT-E and PSJF-E have good mean response time? We have seen in Chapter 10 that while PSJF-E seems to perform nearly optimally under low-to-moderate noise, SRPT-E can perform poorly even under low noise (§ 12.1). This prompts two questions:

- Can we prove a mean response time bound for PSJF-E that explains its robustness to noise?
- Is there a variant of SRPT other than SRPT-E that is as robust to noise as PSJF-E?

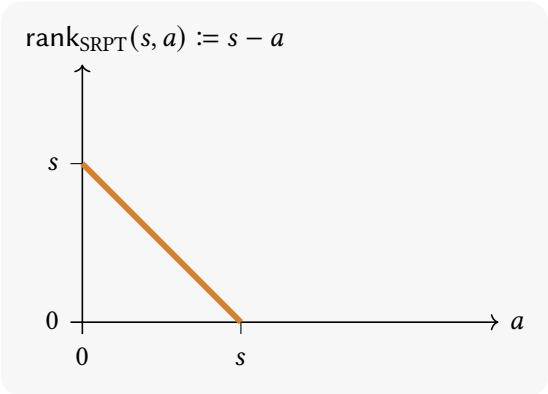
This chapter provides positive answers to both of these questions (§ 12.2). In particular, we introduce a new variant of SRPT, which we call *SRPT with Bounce (SRPT-B)*, whose performance matches SRPT's when noise is zero but gracefully degrades as noise increases. We also show that PSJF-E has a similar relationship with PSJF (§ 12.3).

We have already mentioned five scheduling policies in this introduction, and a sixth policy plays a crucial role in our analysis. To aid the reader in keeping track of the main policies discussed in this chapter, Figure 12.1 illustrates all of their rank functions.

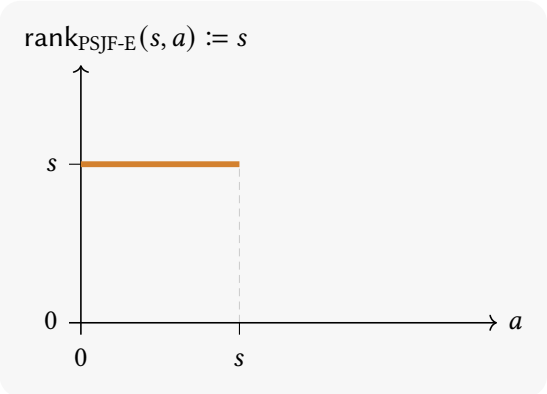
This chapter is a summary of Scully et al. [120], to which we refer the reader for proofs.

12.1 Problem: SRPT-E Can Perform Poorly Even under Low Noise

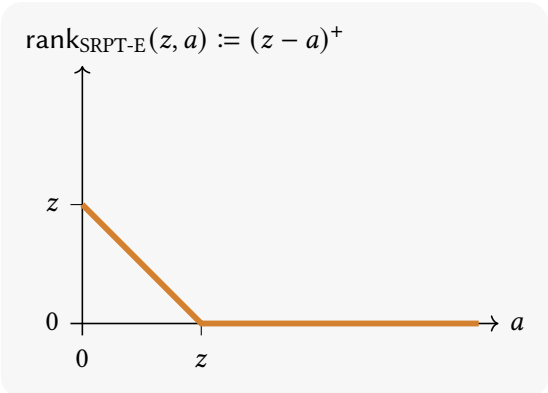
We have seen in Chapter 10 that SRPT-E can have poor mean response time, even when size estimate noise is relatively low. After introducing the noise model we consider in this chapter (§ 12.1.1), we show that SRPT-E can have very performance for arbitrarily low noise levels (§ 12.1.2). This motivates us to come up with criteria we might want a policy for size estimates to satisfy (§ 12.1.3). We use the resulting criteria as benchmarks for evaluating policies in the remainder of the chapter.



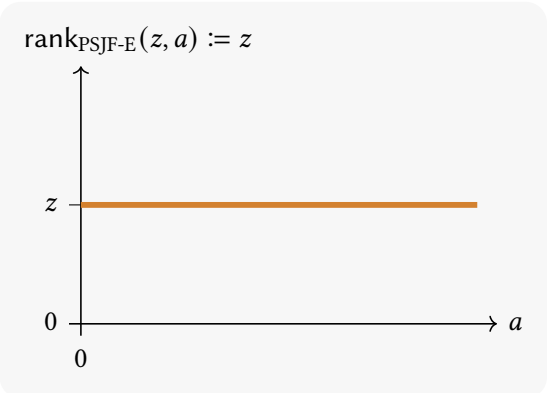
(a) SRPT.



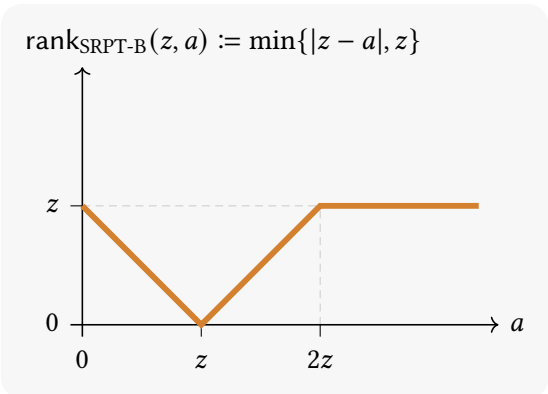
(b) PSJF.



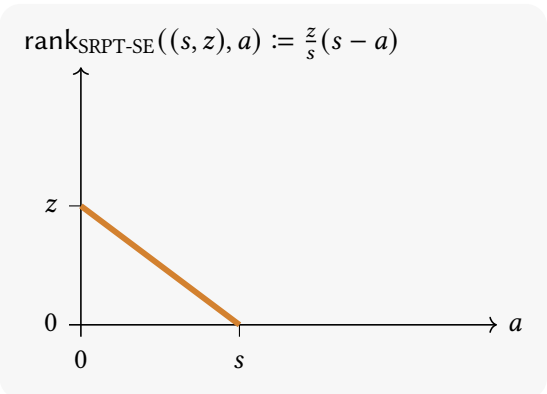
(c) SRPT with Estimates (SRPT-E).



(d) PSJF with Estimates (PSJF-E).



(e) SRPT with Bounce (SRPT-B).



(f) SRPT with Scaling Estimates (SRPT-SE).

Figure 12.1. Rank functions of policies discussed in this chapter. Throughout, s denotes a job's true size, and z denotes a job's estimated size.

12.1.1 The Bounded Noise Model

We study a model we call (α, β)-*bounded size estimate noise*, or simply *bounded noise*. Here $\alpha \geq \beta > 0$ are constants measuring maximum possible estimation error. In this model, a job's true size and estimated size are drawn i.i.d. from a joint *truth-estimate distribution* (S, Z), where S denotes true size as usual, and Z denotes estimated size. A job's true and estimated sizes are guaranteed obey[1]

$$\beta S \leq Z \leq \alpha S. \tag{12.1}$$

Aside from (12.1), we make no additional assumptions on the truth-estimate distribution. Note that we do not require $\alpha \geq 1$ or $\beta \leq 1$.

The bounded noise model serves as a good starting point for theoretical analysis, but it is admittedly not always realistic. Studying models with unbounded noise, such as the log-normal noise model used in Chapter 10, is a potential direction for future research.

How Size Estimates Relate to Labels

For the purposes of formally specifying a label-size distribution and defining SOAP policies, we label jobs with either their true size, estimated size, or possibly both, depending on the policy in question. Of course, if only job size estimates are known, we can only implement policies where a job's label is its estimated size, such as SRPT-E. But policies that use a job's true size still play an important role in our analysis (§ 12.2.1).

12.1.2 Lower Bound for SRPT-E in the Bounded Noise Model

Theorem 12.1. *Consider an M/G/1 with (α, β)-bounded size estimate noise, and suppose $\beta < 1$. For every job size distribution S, there exists a truth-estimate distribution (S, Z), namely $Z = \beta S$, such that the mean response time of SRPT-E is at least*

$$\mathbf{E}[T_{\text{SRPT-E}}] \geq (1-\beta)^2 \rho \mathbf{E}[\mathcal{E}S] + \mathbf{E}[S] = (1-\beta)^2 \tfrac{\lambda}{2} \mathbf{E}[S^2] + \mathbf{E}[S].$$

The key observation behind Theorem 12.1 is that under SRPT-E, a job can have rank 0, and thus be nonpreemptible, for a $1-\beta$ fraction of its time in service. From this observation, the result follows quickly from the SOAP analysis (Thm. 7.15(c)) [120, Thm. 6.1].

Theorem 12.1 implies that SRPT-E's mean response time is sensitive to the variance of the job size distribution. This means that SRPT-E's mean response time is not even necessarily finite. In contrast, SRPT's mean response time is always finite, and it can be bounded by expressions that depend on only the mean of the job size distribution [146].

[1] The intended mnemonic is that α stands for "above", and β stands for "below".

12.1.3 What Does Good Performance Look Like under Bounded Noise?

We have seen that SRPT-E can perform poorly for even arbitrarily small noise, namely for α and β arbitrarily close to 1. We would like to find a policy with better performance, but this prompts a question: what does good performance look like in the bounded noise model?

One possibility is to compare to the Gittins policy (Pol. 6.12). Gittins uses the truth-estimate distribution (S, Z) to design a rank function that minimizes mean response time for that specific truth-estimate distribution. We might look at the approximation ratio of a policy π relative to Gittins, namely an upper bound on $\mathbf{E}[T_\pi]/\mathbf{E}[T_{\text{Gittins}}]$. However, there are two concerns that make this approach unappealing.

- The fact that Gittins's rank function depends on the truth-estimate distribution makes it difficult to use the SOAP analysis to prove theorems about its mean response time (§ 11.1).
- In practice, we may not have a very clear idea of what the truth-estimate distribution actually is. Without knowing the exact distribution (S, Z), we cannot actually implement Gittins, so it is not clear that Gittins is a reasonable benchmark.

Instead of comparing to Gittins, we take inspiration from the growing literature on algorithms with predictions [10, 11, 92, 110] and compare to SRPT, the policy that would minimize mean response time if size estimates were perfect. The key idea is that rather than looking for constant bounds on the approximation ratio relative to SRPT, namely $\mathbf{E}[T_\pi]/\mathbf{E}[T_{\text{SRPT}}]$, we look for bounds that depend on the noise level, which in our case means depending on α and β.

Definition 12.2. Consider any scheduling policy π in an M/G/1 with (α, β)-bounded size estimate noise. Suppose there exists a function f such that for all $\alpha \geq \beta > 0$, π's approximation ratio relative to SRPT is bounded by

$$\frac{\mathbf{E}[T_\pi]}{\mathbf{E}[T_{\text{SRPT}}]} \leq f(\alpha, \beta).$$

We say that π is

(a) *c-consistent* if $\limsup_{\alpha,\beta\to 1} f(\alpha, \beta) = c$,
(b) *c-graceful* if $f(\alpha, \beta) \leq c\frac{\alpha}{\beta}$ for all $\alpha \geq \beta > 0$, and
(c) *c-robust* if $f(\alpha, \beta) \leq c$ for all $\alpha \geq \beta > 0$.

Note that c-robustness implies c-gracefulness, which in turn implies c-consistency. We sometimes omit the "c-" prefix.

We can define the same concepts *relative to π'* for another policy π' taking SRPT's place, but we compare to SRPT by default.

Ideally, we would like to find a policy that is c-consistent, g-graceful, and r-robust for some small constants $c < g < r$. Even better would be if the policy works with *no tuning*,

meaning that, unlike Gittins, it works the same way for any truth-estimate distribution and any values of α and β.

We have seen in Theorem 12.1 that SRPT-E does not satisfy any of the above desiderata, not even consistency. It also turns out that constant-factor robustness is unachievable [120, Appx. B]. Fortunately, we are able to achieve all the other criteria, namely with PSJF-E and SRPT-B.

12.2 Main Result: Adding a "Bounce" to SRPT Ensures Graceful Degradation

Our main result concerns the SRPT-B policy, defined formally below.

Policy 12.3. The *SRPT with Bounce (SRPT-B)* policy is the SOAP policy with rank function

$$\mathsf{rank}_{\text{SRPT-B}}(z, a) = \min\{|z - a|, z\}.$$

See Figure 12.1(e) for an illustration.

Theorem 12.4. *Consider an M/G/1 with (α, β)-bounded size estimate noise. The approximation ratio of SRPT-B is bounded by*

$$\begin{aligned}\frac{\mathbf{E}[T_{\text{SRPT-B}}]}{\mathbf{E}[T_{\text{SRPT}}]} &\le \frac{\alpha}{\beta} + \left(\frac{3}{2}\alpha\mathbb{1}(\beta < 1) + 1\right)\min\left\{1, \max\left\{1 - \frac{1}{\alpha}, \frac{1}{\beta} - 1\right\}\right\} \\ &\le 3.5\frac{\alpha}{\beta}.\end{aligned}$$

so SRPT-B is 1-consistent and 3.5-graceful.

We give a high-level sketch of the proof of Theorem 12.4 later in this section (§§ 12.2.1–12.2.3). See [120, Thm. 8.1] for the full proof.

There are a few essential characteristics of SRPT-B's rank function that are essential for Theorem 12.4, but there are other characteristics that could likely be altered without affecting consistency and gracefulness. Consider SRPT-B's rank function for a job with estimated size z.

- The rank being $z - a$ prior to the bounce helps achieve 1-consistency, because it ensures agreement with SRPT when $\alpha = \beta = 1$. However, it is likely that the rank need not be precisely $z - a$ prior to the bounce, provided it still approaches SRPT's rank as α and β approach 1.
- The bounce is necessary to avoid jobs being nonpreemptible for a fraction of their service time, which is the issue SRPT-E has. However, it is likely that the bounce could occur at an age other than z, provided the bounce location approaches the true size as α and β approach 1.

- The rank function having finite upward slope after the bounce is necessary to ensure 1-consistency. Otherwise, jobs with slightly underestimated sizes could be preempted at age z when they are close to completing. However, it is likely that we could use a slope other than 1.
- Limiting the bounce to stay below the initial rank z is important for gracefulness. Otherwise, jobs become likely to be preempted at age $2z$, when the job reaches ranks worse than any it had previously.[2] However, it is likely that we could instead limit the bounce to γz for some constant $\gamma \in (0, 1)$.

12.2.1 Main Idea: Use SRPT-SE as an Intermediate

We might think to prove Theorem 12.4 by applying the SOAP analysis to each of SRPT-B and SRPT and comparing the resulting mean response time formulas. This may in principle be possible, but the rank functions of SRPT-B and SRPT are different enough that a direct comparison is difficult. We have found it simpler to take a different approach: we compare both SRPT-B and SRPT to a policy whose rank function shares some characteristics with each.

Policy 12.5. The *SRPT with Scaling Estimates (SRPT-SE)* policy is the SOAP policy with rank function

$$\text{rank}_{\text{SRPT-B}}((s, z), a) = \frac{z}{s}(s - a).$$

See Figure 12.1(e) for an illustration. Note that SRPT-SE uses both a job's true size s and estimated size z, so it cannot be implemented using size estimates alone.

SRPT-SE is a useful intermediate because it has some properties in common with SRPT-B and others in common with SRPT.

- SRPT-SE is similar to SRPT-B in that both initially assign a job rank equal to its estimated size, and both never increase the job's rank beyond this initial rank (Figs. 12.1(e) and 12.1(f)).
- SRPT-SE is similar to SRPT in that both assign jobs rank proportional to their remaining work (Figs. 12.1(a) and 12.1(f)).

The rest of the proof thus proceeds in two steps: compare SRPT-B to SRPT-SE (§ 12.2.2), then compare SRPT-SE to SRPT (§ 12.2.2).

12.2.2 Comparing SRPT-B to SRPT-SE: Use SOAP

We compare SRPT-B to SRPT-SE using the SOAP analysis. This involves separate computations for each of waiting time and residence time (§ 7.1.1).

Lemma 12.6. *Consider an M/G/1 with (α, β)-bounded size estimate noise.*

[2] This is an instance of the Pessimism Principle (Prop. 7.7): without limiting the bounce, a job with $S > 2Z$ has a greater worst ever rank than one with $S \leq 2Z$.

(a) The mean waiting time of SRPT-B is bounded above by

$$\mathbf{E}[T^{\text{wait}}_{\text{SRPT-B}}] \leq \mathbf{E}[T^{\text{wait}}_{\text{SRPT-SE}}] + \frac{3}{2}\alpha(1-\beta)^+\left(\frac{1}{\rho}\log\frac{1}{1-\rho}\right)\mathbf{E}[S].$$

(b) The mean residence time of SRPT-B is bounded above by

$$\mathbf{E}[T^{\text{resd}}_{\text{SRPT-B}}] \leq \mathbf{E}[T^{\text{resd}}_{\text{SRPT-SE}}] + \min\left\{1, \max\left\{1-\frac{1}{\alpha}, \frac{1}{\beta}-1\right\}\right\}\left(\frac{1}{\rho}\log\frac{1}{1-\rho}\right)\mathbf{E}[S].$$

The quantity $\left(\frac{1}{\rho}\log\frac{1}{1-\rho}\right)\mathbf{E}[S]$ shows up in both bounds. A result of Wierman et al. [146, Thm. 5.8] implies that this is less than $\mathbf{E}[T_{\text{SRPT}}]$.

12.2.3 Comparing SRPT-SE to SRPT: Use WINE

To compare SRPT-SE to SRPT, we actually need to peek ahead into Part III. One of the main results in Part III concerns what we call (α, β)*-approximate Gittins policies* (Def. 16.3), which are policies whose rank functions are within a constant factor of Gittins's rank function. Specifically, we show that such a policy is a $\frac{\alpha}{\beta}$-approximation for mean response time relative to Gittins (Thm. 16.5). This result is exactly what we need to compare SRPT-SE to SRPT, because SRPT-SE's rank function is always within a constant factor of SRPT's, and SRPT can be seen as a special case of Gittins (Pol. 6.12).

Lemma 12.7. *Consider an M/G/1 with (α, β)-bounded size estimate noise. The approximation ratio of SRPT-SE is bounded by*

$$\frac{\mathbf{E}[T_{\text{SRPT-SE}}]}{\mathbf{E}[T_{\text{SRPT}}]} \leq \frac{\alpha}{\beta}.$$

Proof. Under (α, β)-bounded size estimate noise, if SRPT would assign a job rank r, SRPT-SE would assign the job a rank in $[\beta r, \alpha r]$. This makes SRPT-SE an (α, β)-approximate Gittins policies, so the result follows from Theorem 16.5. □

12.3 PSJF Has Natural Graceful Degradation

We have seen in Section 12.2 that with the right tweak, SRPT can be turned into SRPT-B, a policy that is consistent and graceful. But we saw in Chapter 10 that the natural analogue of PSJF for estimated sizes, namely PSJF-E, has near-optimal mean response time. Can we show that PSJF-E is consistent and graceful like SRPT-B? The following result answers this affirmatively.

Theorem 12.8. *Consider an M/G/1 with (α, β)-bounded size estimate noise. The approximation ratio of PSJF-E relative to PSJF is bounded by*

$$\frac{\mathbf{E}[T_{\text{PSJF-E}}]}{\mathbf{E}[T_{\text{PSJF}}]} \leq \frac{\alpha}{\beta}.$$

This implies PSJF-E is

(a) 1-*consistent and* 1-*graceful relative to PSJF, and*
(b) 1.5-*consistent and* 1.5-*graceful relative to SRPT.*

The comparison to SRPT follows from the comparison to PSJF and a result of Wierman et al. [146, Thm.].See Scully et al. [120, Thm. 9.1] for the rest of the proof of Theorem 12.8. While the analysis of PSJF-E is quite different from the analysis of SRPT-B, it also involves a cocktail of SOAP and WINE.

Chapter 13

Response Time Tail of SOAP Policies

The last several chapters have all focused on the metric of mean response time $\mathbf{E}[T]$. But this is far from the only metric a system designer might hope to optimize. For example, it may be important to ensure that jobs do not have especially long response times. The most relevant metric for this situation is the *response time tail* $\mathbf{P}[T > t]$, which is the subject of this chapter.

In the interest of theoretical tractability, we focus on the *asymptotic tail*, meaning we characterize the behavior of $\mathbf{P}[T > t]$ in the $t \to \infty$ limit. The asymptotic tails of a handful of scheduling policies have been analyzed in the past (§ 2.1.3), and there has also been work on deriving general conditions under which a scheduling policy has certain asymptotic tail characteristics [101, 149]. However, we lack simple conditions for determining the tail asymptotics of a scheduling policy.

This chapter studies the tail asymptotics of SOAP policies. Focusing on SOAP policies prompts the following question: can we easily determine a policy's tail asymptotics from its rank function (§ 13.1)? We obtain results that take promising first steps in this direction: we give conditions under which a SOAP policy is *asymptotically tail-optimal*, meaning $\mathbf{P}[T > t]$ decays in some sense as quickly as possible in the $t \to \infty$ limit (§ 13.2). We apply our results to answer two questions:

- (§ 13.3) Can we simultaneously achieve optimal or near-optimal mean response time and asymptotic tail optimality? In particular, when is Gittins (Pol. 6.12), the optimal policy for mean response time, also asymptotically tail-optimal?
- (§ 13.4) When scheduling with preemption checkpoints (Pol. 6.13), how frequent do checkpoints need to be to ensure asymptotic tail optimality?

We consider both heavy-tailed and light-tailed job size distributions throughout.

All of our results focus on an M/G/1 in the unlabeled case. All SOAP policies mentioned in this chapter should be understood as having rank functions that depend only on age.

This chapter is a summary of Scully et al. [127] and Scully and van Kreveld [126], to which we refer the reader for proofs.

13.1 Problem: Analyzing the Asymptotic Response Time Tail

Our goal is to characterize the asymptotic response time tail of a SOAP policy in terms of its rank function. The step we take towards this goal is giving conditions on a policy's rank function that imply asymptotic tail optimality. Before we can state our results in Section 13.2, we need to define what asymptotic tail optimality means. We do this below for heavy-tailed (§ 13.1.1) and light-tailed (§ 13.1.2) job size distributions.

13.1.1 Definitions for Heavy-Tailed Size Distributions

We begin by defining the class of heavy-tailed job size distributions we consider. We emphasize that these are not the only distributions that are generally referred to as "heavy-tailed", but we still call them simply "heavy-tailed" to reduce clutter.

Definition 13.1. We call a job size distribution S *heavy-tailed* if

$$\liminf_{\varepsilon \downarrow 0} \liminf_{s \to \infty} \frac{\mathbf{P}[S > (1+\varepsilon)a]}{\mathbf{P}[S > s]} = 1$$

and there exist $\beta \geq \alpha > 1$ such that for sufficiently large $t \geq s$,[1]

$$\Omega\left(\left(\frac{t}{s}\right)^{-\beta}\right) \leq \frac{\mathbf{P}[S > t]}{\mathbf{P}[S > s]} \leq O\left(\left(\frac{t}{s}\right)^{-\alpha}\right).$$

Definition 13.2. A scheduling policy π is *asymptotically tail-optimal for heavy-tailed job sizes* (or simply *tail-optimal*) if for all heavy-tailed size distributions S and loads ρ, its M/G/1 response time distribution T_π satisfies

$$\lim_{t \to \infty} \frac{\mathbf{P}[T_\pi > t]}{\mathbf{P}[S > (1-\rho)t]} = 1.$$

Roughly speaking, for a policy π to be tail-optimal for heavy-tailed sizes, it needs to ensure that very large jobs have response time roughly proportional to their size [149]. Some examples of policies that are tail-optimal for heavy-tailed sizes are LAS, SRPT, and PS [101].

13.1.2 Definitions for Light-Tailed Size Distributions

Definition 13.3. The *decay rate* of a random variable V, denoted $d(V)$, is

$$d(V) \coloneqq \lim_{t \to \infty} \frac{-\log \mathbf{P}[V > t]}{t}.$$

That is, if the decay rate $d(V)$ is finite, then $\mathbf{P}[V > t] = \exp(-d(V)\, t \pm o(t))$. Roughly speaking, higher decay rates correspond to lighter tails.

The class of light-tailed distributions we consider are a subclass of those with positive decay rate.

[1] We formally define the multivariable $O(\cdot)$ and $\Omega(\cdot)$ notations as follows. Suppose $x_1, \ldots, x_n$ are non-negative variables. The notation $O(f(x_1, \ldots, x_n))$ stands for an unspecified expression $g(x_1, \ldots, x_n) \geq 0$ for which there exist constants $C, y_0, \ldots, y_n \geq 0$ such that for all $x_1 \geq y_1, \ldots, x_n \geq y_n$, we have $g(x_1, \ldots, x_n) \leq C f(x_1, \ldots, x_n)$. The $\Omega(\cdot)$ notation is the same but with the inequality reversed.

Definition 13.4. Given a job size distribution S, let

$$\theta^* := \inf\{\theta \in \mathbb{R} \mid \mathcal{L}[S](\theta) < \infty\}.$$

We call S *light-tailed* if either of the following holds:

- $\theta^* = -\infty$, or
- $-\infty < \theta^* < 0$ and $\lim_{\theta \downarrow \theta^*} \mathcal{L}[S](\theta) = \infty$.

In either case, one can show $\theta^* = -d(S)$ [90, 97, 98].

Definition 13.5. We classify scheduling policies π as follows based on their M/G/1 response time distribution T_π.

(a) A policy π is *log-tail-optimal for light-tailed job sizes* (or simply *tail-optimal*) if it maximizes $d(T_\pi)$ among all scheduling policies.
(b) A policy π is *log-tail-pessimal for light-tailed job sizes* (or simply *tail-pessimal*) if it minimizes $d(T_\pi)$ among all scheduling policies.
(c) A policy π is *log-tail-intermediate for light-tailed job sizes* (or simply *tail-intermediate*) if it is neither tail-optimal nor tail-pessimal.

It is known that FCFS is tail-optimal in the light-tailed case [23, 133]. In contrast, policies like LAS, SRPT, and PS that are tail-optimal in the heavy-tailed case are tail-pessimal in the light-tailed case [149].

13.2 Main Results: Conditions on a Rank Function that Ensure Tail Optimality

13.2.1 Results for Heavy-Tailed Size Distributions

We define two conditions on a rank function that suffice for tail optimality in the heavy-tailed case. The first condition is a simpler special case of the second. Both conditions give a "asymptotic sketch" of the rank function of a SOAP policy π in terms of a small number of parameters.

Condition 13.6. There exist $\delta \geq \gamma > 0$ such that the rank function of SOAP policy π obeys

$$\Theta(a^\gamma) \leq \text{rank}_\pi(a) \leq O(a^\delta).$$

Condition 13.7. There exist $\zeta, \xi \in [0, \infty)$ and $\chi \in [\max\{1, \zeta + \xi\}, \infty]$ such that the rank function of SOAP policy π obeys the following properties for any size s and interval of ages (b, c) where $\text{rank}_\pi(a) \leq \text{worst}_\pi(s)$ for all $a \in (b, c)$:[2]

(a) $c \leq O(s^\chi)$; and
(b) if $b \geq s$, then $c - b \leq O(b^\zeta s^\xi)$.

[2]Recall from Definition 7.9 that $\text{worst}_\pi(s)$ is the worst ever rank attained by a job of size s.

Our main result for the heavy-tailed case specifies ranges of the parameters such that Conditions 13.6 and 13.7 imply tail-optimality. The ranges depend on the tail exponents α and β of the job size distribution (Def. 13.1).

Theorem 13.8. *Consider an M/G/1 with heavy-tailed size distribution in the unlabeled case.*
(a) A SOAP policy π is tail-optimal if it satisfies Condition 13.6 with

$$\frac{\delta}{\gamma} - \frac{\gamma}{\delta} < \frac{\alpha - 1}{\beta}.$$

(b) A SOAP policy π is tail-optimal if it satisfies Condition 13.7 with

$$\zeta + (\xi - 1)^+ - \frac{(1 - \xi)^+}{\chi} < \frac{\alpha - 1}{\beta}.$$

See Scully et al. [127, Thm. 3.1] for the proof of Theorem 13.8(a), and see Scully and van Kreveld [126, Thm. 4.4] for the proof of Theorem 13.8(b).

13.2.2 Result for Light-Tailed Size Distributions

In the light-tailed case, the response time tail asymptotics of a SOAP policy (in the unlabeled case) turn out to be determined by a single quantity: the age at which a job attains the maximum possible rank, and how that compares to the maximum possible job size.

Definition 13.9. The *maximum job size* for size distribution S, denoted $s_{\max}$, is

$$s_{\max} := \inf\{s \geq 0 \mid \mathbf{P}[S > s] = 0\}.$$

Note that $s_{\max} = \infty$ for many size distributions.

Definition 13.10. The *worst age* under SOAP policy π, denoted a^*_π, is the least age at which the rank function has a global maximum:

$$a^*_\pi := \inf\{a \geq 0 \mid \mathsf{rank}_\pi(a) \geq \mathsf{rank}_\pi(b) \text{ for all } b \geq 0\}.$$

If the rank function has no global maximum, we let $a^*_\pi := s_{\max}$.

Theorem 13.11. *Consider an M/G/1 with light-tailed size distribution in the unlabeled case.*
(a) A SOAP policy π is tail-optimal if and only if $a^*_\pi = 0$.
(b) A SOAP policy π is tail-intermediate if and only if $0 < a^*_\pi < s_{\max}$.
(c) A SOAP policy π is tail-pessimal if and only if $a^*_\pi = s_{\max}$.

See Scully and van Kreveld [126, Thm. 4.7] for the proof of Theorem 13.11.

Improving the Leading Constant

Note that any policy π with $a^*_\pi = 0$ must actually be FCFS, meaning FCFS is the only tail-optimal SOAP policy in the unlabeled case. In fact, it is likely that the proof of Theorem 13.11 can be generalized beyond the unlabeled case. However, even though FCFS is tail-optimal, Grosof et al. [53] show that its response time tail can be asymptotically improved by a constant factor. The policy they use to do so, called *Nudge*, is not a SOAP policy. This suggests that in the light-tailed case, SOAP policies cannot provide the best possible leading constant for the asymptotic response time tail.

13.3 Simultaneously Optimizing the Mean and Tail of Response Time

When scheduling with unknown job sizes, Gittins is known to minimize mean response time [44]. Therefore, simultaneously optimizing the mean and asymptotic tail of response time essentially boils down to determining when Gittins is tail-optimal.

The results of Section 13.2 are a big step towards answering this question, but they fall slightly short of fully answering it. This is because they apply to specific rank functions, whereas Gittins's rank function varies depending on the size distribution. However, they do substantially reduce the problem: to determine whether Gittins is tail optimal, we just need to determine for which size distributions its rank function satisfies certain conditions.

By applying Theorems 13.8 and 13.11 and carefully analyzing Gittins's rank function, we show the following results:

- (§ 13.3.1) In the heavy-tailed case, Gittins is always tail-optimal.
- (§ 13.3.2) In the light-tailed case, Gittins can be any of tail-optimal, tail-intermediate, or tail-pessimal. However, if Gittins is tail-pessimal, we can often slightly adjust Gittins's rank function to obtain a new SOAP policy that is tail-intermediate while still having near-optimal mean response time.

Many of these results also apply to SERPT and M-SERPT (Pols. 11.1 and 6.11), two simpler policies which also have good mean response time.

13.3.1 Gittins in the Heavy-Tailed Case

In the heavy-tailed case, both SERPT and M-SERPT satisfy the simple Condition 13.6.

Theorem 13.12. *Consider an M/G/1 with heavy-tailed size distribution in the unlabeled case. SERPT and M-SERPT both satisfy Condition 13.6 with $\gamma = \delta = 1$, so both are tail-optimal.*

See Scully et al. [127, Cor. 3.6] for the proof of Theorem 13.12.

It turns out that Gittins does not satisfy Condition 13.6 for general heavy-tailed size distributions. This is because Gittins's rank function can drop to near 0 at large ages, which SERPT's and M-SERPT's do not do in the heavy-tailed case. Fortunately, Gittins does

satisfy the somewhat more complicated Condition 13.7. Roughly speaking, even though its rank function can drop to near 0, it never does so for too long, meaning jobs at high ages do not delay jobs at low ages for too long.

Theorem 13.13. *Consider an M/G/1 with heavy-tailed size distribution in the unlabeled case. Gittins satisfies Condition 13.7 with $\zeta = 0$, $\xi = 1$, and $\chi = \infty$, so it is tail-optimal.*

See Scully and van Kreveld [126, Thm. 4.5] for the proof of Theorem 13.13.

13.3.2 Gittins in the Light-Tailed Case

In the light-tailed case, Theorem 13.11 tells us that asymptotic tail performance is determined by the worst age (Def. 13.10). One common way of classifying size distributions has to do with the worst age of SERPT.

Definition 13.14.

(a) We say a size distribution S is *New Better than Used in Expectation (NBUE)* if $a^*_{\text{SERPT}} = 0$, meaning a job's expected remaining work is maximized at age 0. That is, for all $a \in [0, s_{\max})$,
$$\mathbf{E}[S] \geq \mathbf{E}[S_a] = \mathbf{E}[S - a \mid S > a].$$

(b) We say a size distribution S is *Eventually New Better than Used in Expectation (ENBUE)* if $S_a \coloneqq (S - a \mid S > a)$ is NBUE for some age $a \geq 0$, or equivalently $a^*_{\text{SERPT}} < s_{\max}$.

Whether the size distribution is NBUE, ENBUE but not NBUE, or not ENBUE immediately characterizes the asymptotic tail performance of SERPT. It also does so for M-SERPT, which clearly has the same worst age as SERPT (Pol. 11.1). Less obviously, results of Aalto et al. [3, 4] imply that Gittins also has the same worst age as SERPT, so it has the same asymptotic tail performance.

Theorem 13.15. *Consider an M/G/1 with light-tailed size distribution S in the unlabeled case.*

(a) Gittins, SERPT, and M-SERPT are tail-optimal if S is NBUE.
(b) Gittins, SERPT, and M-SERPT are tail-intermediate if S is ENBUE but not NBUE.
(c) Gittins, SERPT, and M-SERPT are tail-pessimal if S is not ENBUE.

See Scully and van Kreveld [126, Thm. 4.10] for a proof of the Gittins case of Theorem 13.15. The SERPT and M-SERPT cases follow from the above discussion.

13.4 Ensuring Tail Optimality when Scheduling with Preemption Checkpoints

One can see from either prior work [101] or Theorem 13.8 that LAS is tail-optimal in the heavy-tailed case. However, LAS assumes that we can preempt jobs whenever we like,

which is not possible in some systems. For example, in Chapter 9 we considered scheduling with *preemption checkpoints*, where jobs are only preemptible at certain checkpoint ages. Can LAS still be tail-optimal in the presence of preemption checkpoints? If so, how often do checkpoints need to occur to maintain tail optimality? For the heavy-tailed case, we can answer the above questions using Theorem 13.8.

To model preemption checkpoints using rank functions, one sets the rank to 0 at all non-checkpoint ages. This transforms LAS into a policy we call *Checkpointed LAS (Chk-LAS)* (Pol. 6.13), whose asymptotic tail performance we characterize below.

Theorem 13.16. *Consider an M/G/1 with heavy-tailed size distribution in the unlabeled case, and consider the Chk-LAS policy with checkpoint ages* $0, a_1, a_2, \ldots$. *If* $a_{n+1} = O(a_n^{\zeta})$, *then Chk-LAS is tail-optimal if*

$$\zeta \leq \frac{\alpha - 1}{\beta}.$$

In particular, for constant or uniformly bounded checkpoint gaps, we have $\zeta = 0$, *so Chk-LAS is always tail-optimal.*

Proof. One can check that Chk-LAS satisfies Condition 13.7 with $\xi = 0$, $\chi = \infty$, and the given value of ζ, so the result follows from Theorem 13.8(b). □

Part III

WINE

Chapter 14

The Markov-Process Job Model

There is a great deal of work in queueing theory on scheduling under uncertainty. Much of this work is on the Gittins policy, which optimizes mean response time and similar metrics for a variety of stochastic uncertainty models. However, as discussed in Section 2.2, this prior work on Gittins is fragmented, with different uncertainty models being considered one-by-one.

The aim of this chapter is to present a very flexible job model that can model many types of uncertainty, and in particular the types of stochastic uncertainty for which we might hope to define a version of Gittins. We call our model the *Markov-process job model* (§ 14.1). The basic idea is that each job is a continuous-time Markov process whose state evolves during service. By varying the state space and dynamics of the Markov process, we obtain different types of uncertainty.

In addition to modeling different types of uncertainty, we will see that Markov-process jobs also allow us to model different types of preemption constraints and holding cost metrics. For example, by carefully choosing the holding cost function, we can model the problem of minimizing mean slowdown with unknown or partially known job sizes (§ 14.2), a problem previously solved only under certain restrictive technical assumptions [122, 136].

We conclude the chapter by defining a general version of Gittins for Markov-process jobs (§ 14.3). This new version of Gittins subsumes most, if not all, other versions of Gittins introduced in the M/G/1 scheduling literature.

This chapter is based on material from Scully and Harchol-Balter [122].

14.1 Markov-Process Jobs

We model jobs as *absorbing continuous-time strong Markov processes*. The state of a job encodes all information that the scheduler knows about the job. Without loss of generality, we assume all jobs share a common state space $\mathbb{X}$ and follow the same stochastic Markovian dynamics. However, the realization of the dynamics may be different for each job. In particular, the *initial state* of each job is drawn from a distribution X_{new}, so different jobs may start in different states.

While a job is in service, its state stochastically advances according to the Markovian dynamics. This evolution is independent of the arrival process and the evolution of other jobs. The rate of evolution is scaled by the job's service rate, so the states of two jobs sharing the server equally will each evolve half as quickly as either being served alone. A job's state does not change while waiting in the queue.

In addition to the main job state space $\mathbb{X}$, there is one additional *final state*, denoted x_{done}. When a job enters state x_{done}, it completes and exits the system. One can think of a size S

as the stochastic amount of time it takes for a job to go from its initial state, which is drawn from X_{new}, to the final state x_{done}. Because we assume $\mathbf{E}[S] < \infty$, every job eventually reaches x_{done} with probability 1. For ease of notation, we follow the convention that $x_{\text{done}} \notin \mathbb{X}$.

14.1.1 Preemptible and Nonpreemptible States

Every job state is either preemptible or nonpreemptible. The job in service can only be preempted if it is in a preemptible state. We write $\mathbb{X}^{\text{P}}$ for the set of preemptible states and $\mathbb{X}^{\text{NP}} = \mathbb{X} \setminus \mathbb{X}^{\text{P}}$ for the set of nonpreemptible states. Naturally, we assume the scheduler knows which states are preemptible.

We assume all jobs start in a preemptible state, meaning $X_{\text{new}} \in \mathbb{X}^{\text{P}}$ with probability 1. This means that all jobs in the queue are in preemptible states, and only the job in service can be in a nonpreemptible state.

We assume preemption occurs with no cost or delay. Because a job's state only changes during service, our model is preempt-resume, meaning that preemption does not cause loss of work.

During service, a job alternates between preemptible and nonpreemptible. We call an amount of service during which a job is nonpreemptible a *nonpreemptible segment*. We assume that the length distribution of a job's nonpreemptible segments has finite variance,[1] because otherwise, the mean holding cost is infinite [55, § 31.4].

14.1.2 System State and Scheduling

The state of the system can be described by a list $(x_1, \ldots, x_n)$, where n is the number of jobs in the system, and $x_i \in \mathbb{X}$ is the state of the ith job. We denote the equilibrium distribution of the system state as $(X_1, \ldots, X_N)$, where N is the equilibrium distribution of the number of jobs.

We can still use rank functions to represent scheduling policies in much the same way as explained in Chapter 6. The main difference is that the domain of a rank function is the set of preemptible states:

$$\mathsf{rank} : \mathbb{X}^{\text{P}} \to \mathbb{R}_{\geq 0}.$$

A rank function represents the scheduling policy that

- prioritizes jobs in nonpreemptible states over those in preemptible states, and
- prioritizes among jobs in preemptible states in rank order, where lower rank is better.[2]

It may be that a job is served at a rate less than the full service capacity of 1. This could be because of multiple jobs sharing a single server (§ 5.3.2) or because of our convention

[1] Adapting the notation of Chapters 7 and 8 to Markov-process jobs, we can write this more formally as $\mathbf{E}_{\text{rcy}}(\mathbb{X}^{\text{NP}})[(S_{\text{rcy}}(\mathbb{X}^{\text{NP}}))^2] < \infty$.

[2] Unlike the SOAP-based results in Part II, the WINE-based results in Part III do not rely on any particular tiebreaking rule.

that the multiserver M/G/k has servers of speed $1/k$ (§ 5.2.4). In either case, while a job is being served at rate u, its state evolves at rate u, analogous to a job's age increasing at rate u in label-age job model of Chapter 5.

14.1.3 Holding Costs

The main metric we study in the Markov-process job model is *mean holding cost*. A job's *holding cost* is a cost we pay for each unit of time it is not complete. We allow a job's holding cost to depend on its state, so it may change during service. A job's holding cost is thus determined by a function

$$h : \mathbb{X} \to \mathbb{R}_{>0}.$$

For ease of notation, we also define $h(x_{\text{done}}) = 0$. We assume that holding costs are deterministic, positive, and known to the scheduler. We can think of holding costs as having dimensions COST RATE := COST/TIME.

The *system holding cost*, or simply "holding cost" when unambiguous, is the total holding cost of jobs in the system, namely

$$H := \sum_{i=1}^{N} h(X_i).$$

We also define the *preemptible* and *nonpreemptible* system holding costs, which only count jobs in preemptible or nonpreemptible states:

$$H^{\text{P}} := \sum_{i=1}^{N} h(X_i)\, \mathbb{1}(X_i \in \mathbb{X}^{\text{P}}),$$

$$H^{\text{P}} := \sum_{i=1}^{N} h(X_i)\, \mathbb{1}(X_i \in \mathbb{X}^{\text{NP}}).$$

Our objective is generally to schedule to *minimize mean system holding cost* $\mathbf{E}[H]$. It turns out that for the systems we consider, the mean nonpreemptible holding cost $\mathbf{E}[H^{\text{NP}}]$ is independent of the scheduling policy (Ch. 16), in which case this reduces to minimizing the mean preemptible holding cost $\mathbf{E}[H^{\text{P}}]$.

14.1.4 What Does the Scheduler Know?

The scheduler also knows, at every moment in time, the current state of all jobs in the system. This assumption is natural because the intuition of our model is that a job's state encodes everything the scheduler knows about the job.

We assume the scheduler knows a description of the job model: the state space $\mathbb{X}$, the subset of preemptible states $\mathbb{X}^{\text{P}} \subseteq \mathbb{X}$, and the Markovian dynamics that govern how a job's state evolves. This assumption is necessary for the Gittins policy, as the policy's definition depends on the job model.

We assume that the scheduler knows the holding cost $h(x)$ of each state $x \in \mathbb{X}$. However, it is often possible to transform some problems with unknown holding costs into problems with known holding costs. We discuss how to do this in Section 14.2.1. A notable example is minimizing mean slowdown when sizes are unknown to the scheduler (Ex. 14.3).

14.1.5 Technical Foundations

We have thus far avoided discussing technical measurability conditions that the job model must satisfy. For example, if the job Markov process has uncountable state space $\mathbb{X}$, one should make some topological assumptions on $\mathbb{X}$, $\mathbb{X}^{\mathrm{P}}$, and $\mathbb{X}^{\mathrm{NP}}$, as well as some continuity assumptions on holding costs. As another example, when discussing sets of states $\mathbb{Y} \subseteq \mathbb{X}$, we should always restrict our attention to measurable subsets.

As discussed in Section 5.6.5, we consider measure theoretic technicalities outside the scope of this thesis. Below, we briefly describe what would be necessary to rigorize the foundations. See Scully et al. [118, Appendix D] for additional discussion.

Our results in Part III are predicated on being able to apply basic optimal stopping theory to solve a single-job optimal stopping problem called the *Gittins game* (Ch. 15). Optimal stopping of general Markov processes is a broad field, and the theory has been developed under many different types of assumptions [106, 129]. Rather than choosing a specific set of assumptions under which the optimal stopping theory for the Gittins game can be developed, we treat our results on Gittins as *reductions*, applying to any job model for which some intuitive but technical properties of the Gittins game can be verified.

14.2 Examples of Markov-Process Jobs and Holding Costs

Example 14.1. To model known sizes, let a job's state be its remaining size. The state space is $\mathbb{X} = \mathbb{R}_{>0}$, the initial state distribution X_{new} is the size distribution S, and the final state is $x_{\text{done}} = 0$. During service, a job's state decreases at rate 1.

Example 14.2. To model unknown sizes, let a job's state be its age, meaning the amount of time it has been served so far. The state space is $\mathbb{X} = \mathbb{R}_{\geq 0}$, all jobs start in initial state $X_{\text{new}} = 0$, and the final state x_{done} is an isolated point. During service, a job's state increases at rate 1, but it also has a chance to jump to x_{done}. The jump probability depends on the size distribution S: the probability a job jumps while being served from state x to state $y > x$ is $\mathbf{P}[S \leq y \mid S > x]$.

14.2.1 Mean Slowdown and Unknown Holding Costs

Recall from Section 14.1.3 that we assume that the holding cost of every job state is known to the scheduler. However, some scheduling problems involve unknown holding costs.

An important example is minimizing mean slowdown, in which a job's holding cost is the reciprocal of its service time. Unless all service times are known to the scheduler, this involves unknown holding costs.

Fortunately, we can transform many problems with unknown holding costs into problems with known holding costs. Suppose a job's current unknown holding cost depends only on its current and future states. Then for all job states $x \in \mathbb{X}$, let

$$h(x) = \mathbf{E}[\text{unknown holding cost of a job in state } x \mid \text{job reached state } x], \tag{14.1}$$

where the expectation is taken over a random realization of a job's path through the state space. The mean holding cost of nonclairvoyant policies is unaffected by this transformation.

Example 14.3. Consider the system from Example 14.2. It has unknown service times, and a job's state x is its age. Suppose all states are preemptible. To minimize mean slowdown, we give a job with service time s holding cost s^{-1}. This turns (14.1) into

$$h(x) = \mathbf{E}[S^{-1} \mid S > x].$$

More generally, if the holding cost of a job of size s is $\hat{h}(s)$, then (14.1) becomes

$$h(x) = \mathbf{E}[\hat{h}(S) \mid S > x].$$

One can generalize Example 14.3 to any scenario where a job's holding cost depends on only its current and future states, namely by taking an expectation over the future states.

14.3 The Gittins Policy with Markov-Process Jobs

14.3.1 Gittins Rank Ingredients: Relevant Work, Holding Cost Change

All of the relevant system definitions in Section 7.2 apply virtually verbatim to the Markov-process job model. The only difference is that instead of states being label-size pairs, states inhabit a general space $\mathbb{X}$, implying the following changes:

- We consider sets of general states $\mathbb{Y} \subseteq \mathbb{X}$ instead of sets of label-size pairs $\mathbb{L} \times \mathbb{R}_{\geq 0}$.
- We consider general states $x \in \mathbb{X}$ instead of label-size pairs $(\ell, a) \in \mathbb{L} \times \mathbb{R}_{\geq 0}$.

We can similarly generalize Relevant Work Decomposition and its associated definitions from Section 8.3.

The relevant system definition that is most important for defining Gittins is a job's relevant remaining work (Def. 7.3). We denote by $S_x(\mathbb{Y})$ the remaining $\mathbb{Y}$-work of a job in state x.

Definition 14.4. Let $\mathbb{Y} \subseteq \mathbb{X}$ be a set of job states and $x \in \mathbb{X}$ be a job state. The $\mathbb{Y}$-*exit state* of x, denoted $Z_x(\mathbb{Y})$, is the random state of a job when it first exits $\mathbb{Y}$, given that it starts in state x.[3] In particular, if the job completes before exiting $\mathbb{Y}$, then $Z_x(\mathbb{Y}) = x_{\text{done}}$.

14.3.2 The Gittins Rank Function

Policy 14.5. The *Gittins* policy is the policy represented by the rank function $\text{rank}_{\text{Gittins}} : \mathbb{X}^{\text{P}} \to \mathbb{R}_{>0}$ defined by

$$\text{rank}_{\text{Gittins}}(x) = \inf_{\mathbb{Y} \supseteq \mathbb{X}^{\text{NP}}} \frac{\mathbf{E}[S_x(\mathbb{Y})]}{h(x) - \mathbf{E}[h(Z_x(\mathbb{Y}))]},$$

where the infimum is taken over all sets of states $\mathbb{Y}$ such that $\mathbb{X}^{\text{NP}} \subseteq \mathbb{Y} \subseteq \mathbb{X}$ and the denominator is nonzero.

Because nonpreemptible states have priority over preemptible states, we only need to define the rank function for preemptible states. But for convenience of notation, we let $\text{rank}_{\text{Gittins}}(x) = 0$ for $x \in \mathbb{X}^{\text{NP}}$.

A job's rank under Gittins has dimensions TIME/COST RATE = TIME^2/COST. Intuitively, we can think of a state's Gittins rank as a measure of how long on average it takes to decrease the holding cost of a job starting at that state. We can think of the special case where we aim to minimize mean number in system as being the case of dimensionless holding costs, meaning COST RATE = 1. A job's rank then has dimensions TIME, which should feel familiar from (Part II)'s presentation of Gittins (Pol. 6.12).

14.3.3 Examples of the Gittins Rank Function

Example 14.6. Consider scheduling jobs with known sizes, as in Example 14.1, to minimize mean response time. As discussed in Policy 6.12, Gittins reduces to SRPT in this case, meaning a job with remaining work x has rank

$$\text{rank}_{\text{Gittins}}(x) = x.$$

More generally, suppose that a job's state consists of its remaining work w and a constant holding cost c. That is, we have $h(w, c) = c$. In this case, Gittins reduces to *Weighted SRPT*, which has rank function

$$\text{rank}_{\text{Gittins}}(w, c) = \frac{w}{c}.$$

That is, jobs with less remaining work and greater holding cost are prioritized. One can view this as an instance of the $c\mu$ rule (Ex. 14.8).

[3] It may be that $Z_x(\mathbb{Y}) \in \mathbb{Y}$ with positive probability, such as if the job Markov process has continuous trajectories and $\mathbb{Y}$ is a closed set.

Example 14.7. Consider scheduling jobs with unknown sizes, as in Example 14.2, to minimize mean slowdown, as in Example 14.3. A job's state x is its age, and a job's holding cost at age x is $h(x) = \mathbf{E}[S^{-1} \mid S > x]$, so the Gittins rank function is

$$\mathsf{rank}_{\text{Gittins}}(x) = \inf_{y>x} \frac{\mathbf{E}[\min\{S, y\} - x \mid S > x]}{\mathbf{E}[S^{-1}\mathbb{1}(S \le y) \mid S > x]}.$$

Example 14.8. The celebrated *$c\mu$ rule* [28], where we prioritize jobs in order of increasing holding cost times expected size, can be viewed as a special case of Gittins. The $c\mu$ rule is known to minimize mean holding cost in two cases: preemptive multiclass systems with exponential job size distributions for each class, and nonpreemptive multiclass systems with general job size distributions for each class, where in both cases a job's holding cost depends on its class and does not change during service. Both of these systems can be modeled using the Markov-process job model. Moreover, we can model combinations of them. Our Gittins optimality result (Ch. 16) thus generalizes the $c\mu$ rule's optimality to systems where each class can be either preemptible with exponential size distribution, or nonpreemptible with general size distribution. One can even mix in jobs with known sizes, as in Example 14.6.

Chapter 15

WINE: Relating Gittins-Flavored Relevant Work to Holding Cost

This chapter introduces a new queueing identity, called *WINE: Work Integral Number Equality*, that gives a new way of characterizing the number-in-system N, or more generally system holding cost H, in essentially any queueing system. In combination with Little's law [84], WINE gives a new way to analyze a system's mean (weighted) response time.

The primary motivation for WINE is to be able to analyze policies like SRPT and Gittins in multiserver systems, such as the M/G/k. As discussed in Chapter 4, the SOAP analysis strategy that is successful for the M/G/1 does not generally work in the M/G/k. Because WINE applies even in multiserver systems, it opens up a new avenue for analyzing SRPT and Gittins in multiserver systems. We carry out such analyses in Chapter 17.

With that said, WINE is also useful in the simpler setting of single-server scheduling. One specific application has to do with proving Gittins's optimality. In Chapter 14, we introduced a highly general job model and defined a generalization of the Gittins policy for it. In light of prior work on the Gittins policy [44], it is a foregone conclusion that our new generalization of it minimizes mean holding cost in the M/G/1. However, proving this rigorously for such a general job model is challenging, as evidenced by the fact that the numerous prior proofs of Gittins's optimality [18, 28, 29, 43, 76, 79, 116, 128, 128, 136, 141] consider only special cases and require restrictive technical assumptions.

WINE turns out to be the missing ingredient for proving Gittins's optimality in the M/G/1. WINE reduces proving Gittins's optimality to a relatively simple question about relevant work, thus dodging the technical obstacles that prior proofs had to contend with. More generally, as we show in Chapter 16, WINE allows us to easily show that approximate Gittins policies are approximately optimal, where here "approximate" means within a constant multiplicative factor.

We begin the chapter by presenting the simplest version of WINE, which we call "SRPT-flavored" due to its connection to SRPT's rank function (§ 15.1). Just as Gittins can be viewed as a generalization of SRPT, the full version of WINE, which we call "Gittins-flavored", generalizes SRPT-flavored WINE. Proving Gittins-flavored WINE amounts to proving properties of the Gittins rank function, which is significantly more complicated than the SRPT rank function. To do so, we present an alternative definition for the Gittins rank function based on an optimization problem called the *Gittins game* (§ 15.2), from which Gittins-flavored wine follows relatively easily (§ 15.3).

This chapter is based on material from Scully and Harchol-Balter [122], though the name "WINE" is new as of this thesis.

15.1 SRPT-Flavored WINE

As a warmup to proving the full "Gittins-flavored" version WINE, this section proves a special case called "SRPT-flavored" WINE. We note that this special case was independently derived by Banerjee et al. [12] (§ 2.2.3).

Throughout this section, we work with perhaps the simplest possible Markov-process job model: a job's state is its remaining work, and jobs have constant holding cost $h(x) = 1$. In this setting, Gittins reduces to SRPT (Pol. 6.12), which has rank function $\mathsf{rank}_{\mathrm{SRPT}}(x) = x$: a job's rank is its remaining work.

15.1.1 Clearer Notation for Relevant Work

SRPT-flavored WINE is defined using relevant work, where the set of "relevant" states are those whose rank under SRPT is at most some threshold r. In Chapter 7, we denoted this set of states by simply "$\leq r$", which is unambiguous as long as only one scheduling policy is being discussed. However, it turns out that SRPT-flavored WINE holds under *any* scheduling policy, not just SRPT. As such, we introduce some more verbose but less ambiguous notation for specifying relevant state sets.

Definition 15.1. Let $\mathsf{func} \colon \mathbb{X} \to \mathbb{R}_{\geq 0}$ be a function on states, typically a rank function of some policy, and $c \geq 0$ be a constant. The notation $(\mathsf{func} \leq c)$ denotes the set

$$(\mathsf{func} \leq c) \coloneqq \{x \in \mathbb{X} \mid \mathsf{func}(x) \leq c\}.$$

Some notes about this definition:

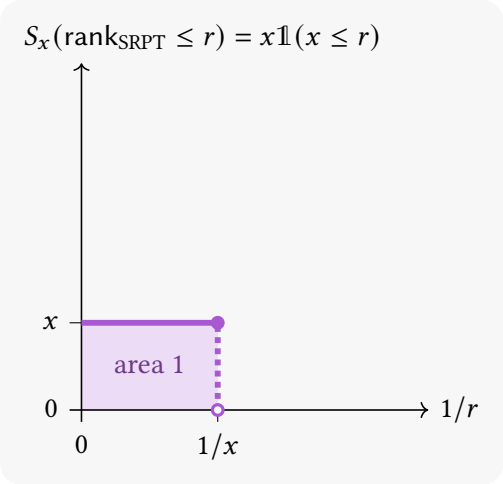
- We drop the parentheses when the grouping is unambiguous, as in $W(\mathsf{func} \leq c)$.
- We define other function inequality notation similarly, such as $(\mathsf{func}_1 < c \leq \mathsf{func}_2)$.
- To prevent ambiguity between functions and constants, we always use sans-serif font to denote functions in the context of this notation.
- In informal discussion, we use the phrase *π-flavored relevant* as a synonym for $(\mathsf{rank}_\pi \leq r)$-relevant or $(\mathsf{rank}_\pi < r)$-relevant, as in "Gittins-flavored relevant remaining work".

15.1.2 Proof of SRPT-Flavored WINE

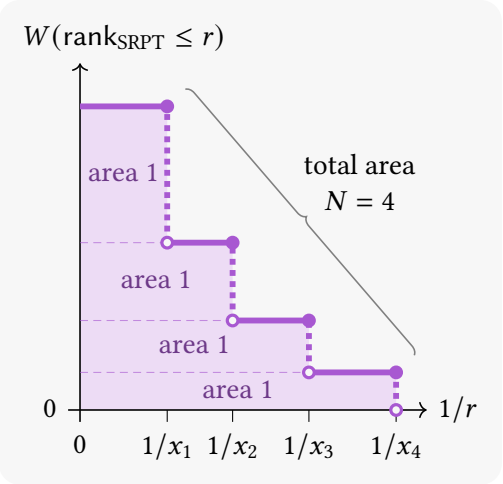
Recall that WINE stands for Work Integral Number Equality. For SRPT-flavored WINE, this means we somehow integrate SRPT-flavored relevant system work and obtain the number-in-system. How can we count the number of jobs via relevant work? The key step is to "count" a single job via its relevant remaining work.

Proposition 15.2. *For any amount of remaining work $x > 0$,*

$$1 = \int_0^\infty \frac{S_x(\mathsf{rank}_{\mathrm{SRPT}} \leq r)}{r^2} \, \mathrm{d}r.$$



(a) Integrating remaining $(\text{rank}_{\text{SRPT}} \le r)$-work of single job of remaining work x (Prop. 15.2).



(b) Integrating system $(\text{rank}_{\text{SRPT}} \le r)$-work with $N = 4$ jobs present (Thm. 15.3).

Figure 15.1. Geometry of SRPT-flavored WINE. Plots (a) and (b) both put $1/r$ on the horizontal axis because $1/r^2\,\mathrm{d}r = \mathrm{d}(1/r)$. We see in (a) that a single job's relevant remaining work integrates to 1, which means that relevant system work integrates to the number-in-system N in (b).

Proof. If $x \le r$, then the job has rank at most r for the rest of its service, so its relevant remaining work is x. If instead $x > r$, then the job is irrelevant, so its relevant remaining work is 0. This means

$$S_x(\text{rank}_{\text{SRPT}} \le r) = x\mathbb{1}(x \le r),$$

from which the result follows. Figure 15.1(a) geometrically interprets this integral. □

From the single-job relevant work integral above, the multi-job relevant work integral below follows easily.

Theorem 15.3 (SRPT-Flavored WINE)**.** *In any queueing system using any scheduling policy π, the number-in-system can be expressed as an integral of SRPT-flavored relevant work:*

$$N_\pi = \int_0^\infty \frac{W_\pi(\text{rank}_{\text{SRPT}} \le r)}{r^2}\,\mathrm{d}r.$$

In particular, provided the steady-state expectations are well defined,

$$\mathbf{E}[N_\pi] = \int_0^\infty \frac{\mathbf{E}[W_\pi(\text{rank}_{\text{SRPT}} \le r)]}{r^2}\,\mathrm{d}r.$$

Proof. We omit the subscript π throughout. Relevant system work is the sum of each job's

relevant remaining work, so[1]

$$\begin{aligned}\int_0^\infty \frac{W(\mathsf{rank}_{\mathrm{SRPT}} \le r)}{r^2}\,\mathrm{d}r &= \int_0^\infty \frac{\sum_{i=1}^N S_{X_i}(\mathsf{rank}_{\mathrm{SRPT}} \le r)}{r^2}\,\mathrm{d}r \\ &= \sum_{i=1}^N \int_0^\infty \frac{S_{X_i}(\mathsf{rank}_{\mathrm{SRPT}} \le r)}{r^2}\,\mathrm{d}r \\ &= N. \qquad \text{[by Prop. 15.2]}\end{aligned}$$

Figure 15.1(b) geometrically interprets this integral. The expected value version follows from Tonelli's theorem. □

Scope of SRPT-Flavored WINE's Validity

The statement of SRPT-flavored WINE (Thm. 15.3) mentions "any" queueing system and "any" scheduling policy. The suspicious reader will be excused for asking: what is the precise scope of "any"?

The best way to understand WINE is to observe that it is not really about queueing at all. Instead, WINE is a statement that holds for any list of numbers: if we have a list of n positive reals $(x_1, \ldots, x_n)$ and define $w(r) \coloneqq \sum_{i=1}^n x_i \mathbb{1}(x_i \le r)$, then we have $\int_0^\infty w(r)/r^2\,\mathrm{d}r = n$. In this sense, WINE really does hold for any queueing system using any scheduling policy, provided that some notion of remaining work can be defined.

We can even use SRPT-flavored WINE in systems where the scheduler does not know job sizes. The scheduler might not know the current value of $W(\mathsf{rank}_{\mathrm{SRPT}} \le r)$. But from the perspective of analysis, we may be able to compute, say, its expectation, in which case WINE yields the mean number-in-system $\mathbf{E}[N]$.

15.1.3 Why Is WINE Useful?

By now, it should be clear that SRPT-flavored WINE (Thm. 15.3) is a strangely simple fact to form the foundation of half a thesis, with Figure 15.1(b) telling essentially the entire story. How can an identity as simple as SRPT-flavored WINE be useful?

The key to WINE's usefulness is that *relevant system work can be much simpler than number-in-system.* In particular, Relevant Work Decomposition Theorem 8.7 gives a formula for relevant system work, which is useful in both the M/G/1 and M/G/k.

In the M/G/1, WINE and Relevant Work Decomposition give a very brief proof that SRPT minimizes mean response time.

- Because SRPT is $(\mathsf{rank}_{\mathrm{SRPT}} \le r)$-prioritizing for all $r \ge 0$, by Corollary 8.9, SRPT minimizes mean system $(\mathsf{rank}_{\mathrm{SRPT}} \le r)$-work in the M/G/1 for all $r \ge 0$.
- Therefore, by SRPT-flavored WINE (Thm. 15.3) and Little's law [84], SRPT minimizes mean response time in the M/G/1.

[1] Recall from Section 14.1.2 that X_i is the state of the ith job in the system.

SRPT's optimality in the M/G/1 is, of course, a well known result [116]. But essentially the same argument with Gittins-flavored WINE proves Gittins's optimality in the M/G/1, which is a novel result in a setting as general as the Markov-process job model (Ch. 16).

In the M/G/k, WINE and Relevant Work Decomposition are again a potent combination: Relevant Work Decomposition relates SRPT- or Gittins-flavored relevant work in the M/G/k to that in the M/G/1, and WINE turns this into a relationship between SRPT's or Gittins's mean response time in the M/G/k to that in the M/G/1. Relevant Work Decomposition introduces a term that is intractable to analyze exactly, but we are able to bound it well enough to give meaningful bounds on mean response time (Ch. 17).

15.2 The Gittins Game

Having gotten a taste of WINE's utility in Section 15.1.3, we would like to generalize SRPT-flavored WINE, which is useful for analyzing SRPT, to Gittins-flavored WINE, which is useful for analyzing Gittins. The main difficulty in doing so is that Gittins's rank function is defined in terms of a potentially difficult optimization problem (§ 14.3). The goal of this section is therefore to better understand Gittins's rank function. From our new understanding, Gittins-flavored WINE quickly follows (§ 15.3).

Our tool for understanding Gittins's rank function is an optimization problem that we call the *Gittins game* (§ 15.2.1). The Gittins game can be seen as an alternative way of defining the Gittins rank function [35, 44, 140]. Most importantly for our purposes, the Gittins game exposes a new connection between the Gittins rank function and Gittins-flavored relevant work (§ 15.2.2).

For this section, we return to the general setting of the Markov-process job model (Ch. 14), with a general state space $\mathbb{X}$, general Markovian dynamics, and a general holding cost function $h : \mathbb{X} \to \mathbb{R}_{>0}$.

15.2.1 Defining the Gittins Game

The *Gittins game* is an optimization problem[2] concerning serving a single job. It has two parameters:

- the *initial state* $x \in \mathbb{X}$ of the job, which is usually a preemptible state; and
- a *penalty* $r \geq 0$, which has dimensions $\text{TIME}^2/\text{COST}$.

The goal of the game is to end the game in as little time as possible in expectation. There are two possible actions:

- By default, we *serve the job*. During service, the job's state changes according to its usual Markovian dynamics. If the job completes by entering state x_{done}, then the game ends immediately.

[2]The optimization problem has a single decision maker, so the Gittins game is not a game in the game-theoretic sense. Caught in conflict between being completely correct and alliterating, alas, alliteration allured this thesaurus-attached author.

- If the job's current state is preemptible, then we may choose to *give up*. If we decide to give up when the job is in some state z, then we immediately stop serving the job, but the game ends anyway after a delay of $rh(z)$ additional time.

That is, the Gittins game involves serving the job until it either completes or we decide to give up, with the penalty r determining how bad giving up is.

Because the job's state evolution is Markovian, the Gittins game is a Markovian optimal stopping problem. This means there is an optimal policy of the following form: for some *give-up set* $\mathbb{Z} \subseteq \mathbb{X}^{\mathrm{P}}$, we give up when the job's state first enters $\mathbb{Z}$. Moreover, this set need not depend on the starting state. We use this observation to formally define the Gittins game. To match our existing notation, we phrase the definition in terms of the *service set* $\mathbb{Y} = \mathbb{X} \setminus \mathbb{Z}$ instead of the give-up set $\mathbb{Z}$, but the idea is the same.

Definition 15.4. The *Gittins game* is an optimization problem. The parameters are a starting state $x \in \mathbb{X}$ and penalty $r \geq 0$, and the control is a service set $\mathbb{Y} \supseteq \mathbb{X}^{\mathrm{NP}}$.

(a) The *disutility of service set* $\mathbb{Y}$ for state x and penalty r is[3]

$$\mathsf{game}_x(r, \mathbb{Y}) \coloneqq \mathbf{E}[S_x(\mathbb{Y})] + r\mathbf{E}[h(Z_x(\mathbb{Y}))].$$

The objective of the Gittins game is to choose $\mathbb{Y}$ to minimize $\mathsf{game}_x(r, \mathbb{Y})$.

(b) The *minimal disutility* for state x and penalty r is

$$\mathsf{game}_x(r) \coloneqq \inf_{\mathbb{Y} \supseteq \mathbb{X}^{\mathrm{NP}}} \mathsf{game}_x(r, \mathbb{Y}).$$

(c) The *optimal service set* for penalty r is[4]

$$\mathbb{Y}^*(r) \coloneqq \mathbb{X}^{\mathrm{NP}} \cup \{y \in \mathbb{X}^{\mathrm{P}} \mid \mathsf{game}_y(r) < rh(y)\}.$$

Some notes about this definition:

- When we discuss a penalty r, it is understood that $r \geq 0$.
- When we discuss a service set $\mathbb{Y}$, it is understood that $\mathbb{X}^{\mathrm{NP}} \supseteq \mathbb{Y} \supseteq \mathbb{X}$.

As suggested by its name, for any state x and penalty r, the optimal service set does indeed solve the Gittins game:

$$\mathsf{game}_x(r) = \mathsf{game}_x(r, \mathbb{Y}^*(r)). \tag{15.1}$$

Intuitively, (15.1) is almost trivial, because Definition 15.4(c) defines $\mathbb{Y}^*(r)$ to be exactly the set of states for which giving up is either disallowed or strictly suboptimal. To formalize (15.1), and moreover to formalize the Definition 15.4, one needs the job Markov process to satisfy some reasonable topological conditions, so we implicitly assume these (§ 14.1.5).

[3] See Section 14.3.1 for the definitions of $S_x(\mathbb{Y})$ and $Z_x(\mathbb{Y})$.

[4] There may actually be multiple service sets that are all optimal, but for simplicity of language, we still refer to this one as "the" optimal service set.

Bounding Minimal Disutility

Lemma 15.5. *For all $x \in \mathbb{X}^{\mathrm{P}}$ and $r \geq 0$,*[5]

$$\mathbf{E}[S_x(\mathbb{Y}^*(r))] \leq \mathsf{game}_x(r) \leq \min\{rh(x), \mathbf{E}[S_x]\}.$$

Proof. Definition 15.4 and (15.1) implies $S_x(\mathbb{Y}^*(r)) \leq \mathsf{game}_x(r, \mathbb{Y}^*(r)) = \mathsf{game}_x(r)$, which is the lower bound. The upper bound follows from, roughly speaking, giving up as early or late as possible in the Gittins game.

- Giving as early as possible means giving up immediately, which is possible because x is preemptible. It takes time $rh(x)$ to give up from x, so $\mathsf{game}_x(r) \leq rh(x)$.
- Giving up as late as possible means never giving up. It takes expected time $\mathbf{E}[S_x]$ to complete the job from x, so $\mathsf{game}_x(r) \leq \mathbf{E}[S_x]$. □

15.2.2 Relating the Gittins Game to Relevant Work

To relate the Gittins game to Gittins-flavored relevant work, we first need to relate the Gittins game to the Gittins rank function.

Lemma 15.6.

(a) For all $r \geq 0$,

$$\mathbb{Y}^*(r) = (\mathsf{rank}_{\mathrm{Gittins}} < r) \coloneqq \{x \in \mathbb{X} \mid \mathsf{rank}_{\mathrm{Gittins}}(x) < r\}.$$

(b) For all $x \in \mathbb{X}^{\mathrm{P}}$,

$$\begin{aligned}\mathsf{rank}_{\mathrm{Gittins}}(x) &= \sup\{r \geq 0 \mid x \in \mathbb{Y}^*(r)\} = \sup\{r \geq 0 \mid \mathsf{game}_x(r) < rh(x)\} \\ &= \inf\{r \geq 0 \mid x \notin \mathbb{Y}^*(r)\} \ = \inf\{r \geq 0 \mid \mathsf{game}_x(r) = rh(x)\}.\end{aligned}$$

Proof. By Definition 15.4(c), it suffices to show that for all $x \in \mathbb{X}^{\mathrm{P}}$ and $r \geq 0$,

$$\mathsf{rank}_{\mathrm{Gittins}}(x) < r \quad \text{if and only if} \quad \mathsf{game}_x(r) < rh(x).$$

But by Policy 14.5 and Definition 15.4, both of these are equivalent to the existence of a service set $\mathbb{Y} \supseteq \mathbb{X}^{\mathrm{NP}}$ such that

$$\mathbf{E}[S_x(\mathbb{Y})] < r(h(x) - \mathbf{E}[h(Z_x(\mathbb{Y}))]).$$ □

Lemma 15.7. *For all $x \in \mathbb{X}^{\mathrm{P}}$, the minimal disutility $\mathsf{game}_x : \mathbb{R}_{\geq 0} \to \mathbb{R}_{\geq 0}$ is continuous, concave, and almost everywhere differentiable with*

$$\frac{\mathrm{d}}{\mathrm{d}r}\mathsf{game}_x(r) = \mathbf{E}[h(Z_x(\mathsf{rank}_{\mathrm{Gittins}} < r))].$$

[5] The result generalizes to all $x \in \mathbb{X}$ if we replace $h(x)$ with $\mathbf{E}[h(Z_x(\mathbb{X}^{\mathrm{NP}}))]$, but we do not need this generality.

Proof. For any service set $\mathbb{Y}$, Definition 15.4(a) tells us the disutility of service set $\mathbb{Y}$ is linear in the penalty r. Because a minimum of linear functions is concave, Definition 15.4(b) implies game_x is concave. Concavity implies continuity for $r > 0$, and Lemma 15.5 implies continuity at $r = 0$. Concavity also implies differentiability almost everywhere, and an envelope theorem [88] gives the derivative where it exists:

$$\begin{aligned}
\frac{\mathrm{d}}{\mathrm{d}r}\,\mathsf{game}_x(r;\mathbb{Y}^*(r)) &= \frac{\mathrm{d}}{\mathrm{d}r}\,\mathsf{game}_x(r;\mathbb{Y}^*(r)) && \text{[by (15.1)]}\\
&= \frac{\mathrm{d}}{\mathrm{d}r}\,\mathsf{game}_x(r;\mathbb{Y}^*(q))\bigg|_{q=r} && \text{[by Milgrom and Segal [88, Theorem 1]]}\\
&= \mathbf{E}\big[h\big(Z_x(\mathbb{Y}^*(q))\big)\big]\Big|_{q=r} && \text{[by Def. 15.4(a)]}\\
&= \mathbf{E}\big[h\big(Z_x(\mathsf{rank}_{\text{Gittins}} < r)\big)\big]. && \text{[by Thm. 15.6(a)]}
\end{aligned}$$

□

15.3 Gittins-Flavored WINE

15.3.1 Relevant Work of Preemptible Jobs

Definition 15.8. Let $\mathbb{Y} \subseteq \mathbb{X}$. The $\mathbb{Y}$-*relevant work of preemptible jobs* is

$$W^{\mathrm{P}}(\mathbb{Y}) = \sum_{i=1}^{N} S_{X_i}(\mathbb{Y})\,\mathbb{1}(X_i \in \mathbb{X}^{\mathrm{P}}).$$

Similarly, the $\mathbb{Y}$-*relevant work of nonpreemptible jobs* is

$$W^{\mathrm{NP}}(\mathbb{Y}) = \sum_{i=1}^{N} S_{X_i}(\mathbb{Y})\,\mathbb{1}(X_i \in \mathbb{X}^{\mathrm{NP}}).$$

15.3.2 Proof of Gittins-Flavored WINE

Proposition 15.9. *For all* $x \in \mathbb{X}^{\mathrm{P}}$,

$$h(x) = \int_0^\infty \frac{\mathbf{E}[S_x(\mathsf{rank}_{\text{Gittins}} < r)]}{r^2}\,\mathrm{d}r.$$

Proof. By Lemma 15.7,

$$\frac{\mathrm{d}}{\mathrm{d}r}\frac{\mathsf{game}_x(r)}{r} = \frac{r\,\frac{\mathrm{d}}{\mathrm{d}r}\,\mathsf{game}_x(r) - \mathsf{game}_x(r)}{r^2} = \frac{-\mathbf{E}[S_x(\mathsf{rank}_{\text{Gittins}} < r)]}{r^2},$$

so the integral is

$$\int_0^\infty \frac{\mathbf{E}[S_x(\mathsf{rank}_{\text{Gittins}} < r)]}{r^2}\,\mathrm{d}r = \lim_{r\to 0}\frac{\mathsf{game}_x(r)}{r} - \lim_{r\to\infty}\frac{\mathsf{game}_x(r)}{r}.$$

The $r \to \infty$ limit is 0 by Lemma 15.5, and the $r \to 0$ limit is $h(x)$ by Theorem 15.6(b) and the fact that ranks of preemptible states are positive (Pol. 14.5). □

Theorem 15.10 (Gittins-Flavored WINE)**.** *In any queueing system using a nonclairvoyant scheduling policy π, meaning one that does not depend on the future states of any jobs, the preemptible holding cost can be expressed as an integral of expected Gittins-flavored relevant work:*[6]

$$H_\pi^{\mathrm{P}} = \int_0^\infty \frac{\mathbf{E}[W_\pi^{\mathrm{P}}(\mathsf{rank}_{\mathrm{Gittins}} < r) \mid X_1, \ldots, X_N]}{r^2} \, \mathrm{d}r.$$

In particular, provided the steady-state expectations are well defined,

$$\mathbf{E}[H_\pi^{\mathrm{P}}] = \int_0^\infty \frac{\mathbf{E}[W_\pi^{\mathrm{P}}(\mathsf{rank}_{\mathrm{Gittins}} < r)]}{r^2} \, \mathrm{d}r.$$

Proof. We omit the subscript π throughout. Gittins-flavored relevant system work is the sum of each job's relevant remaining work, so

$$\begin{aligned}
&\int_0^\infty \frac{\mathbf{E}[W^{\mathrm{P}}(\mathsf{rank}_{\mathrm{Gittins}} < r) \mid X_1, \ldots, X_N]}{r^2} \, \mathrm{d}r \\
&\quad = \int_0^\infty \frac{\mathbf{E}[\sum_{i=1}^N S_{X_i}(\mathsf{rank}_{\mathrm{Gittins}} < r) \, \mathbb{1}(X_i \in \mathbb{X}^{\mathrm{P}}) \mid X_1, \ldots, X_N]}{r^2} \, \mathrm{d}r \\
&\quad = \sum_{i=1}^N \mathbb{1}(X_i \in \mathbb{X}^{\mathrm{P}}) \int_0^\infty \frac{\mathbf{E}[S_{X_i}(\mathsf{rank}_{\mathrm{Gittins}} < r) \mid X_1, \ldots, X_N]}{r^2} \, \mathrm{d}r \\
&\quad = \sum_{i=1}^N \mathbb{1}(X_i \in \mathbb{X}^{\mathrm{P}}) \int_0^\infty \frac{\mathbf{E}[S_{X_i}(\mathsf{rank}_{\mathrm{Gittins}} < r) \mid X_i]}{r^2} \, \mathrm{d}r && \text{[by nonclairvoyance]} \\
&\quad = \sum_{i=1}^N \mathbb{1}(X_i \in \mathbb{X}^{\mathrm{P}}) h(x) && \text{[by Prop. 15.9]} \\
&\quad = H^{\mathrm{P}}. && \text{[by Def. 15.8]}
\end{aligned}$$

The expected value version follows from Tonelli's theorem. □

There is one subtle difference between Theorems 15.3 and 15.10: the former uses a non-strict inequality, while the latter uses a strict inequality. One can easily verify that Theorem 15.3 still holds if we instead use a strict inequality. One can also show that Theorem 15.3 still holds if we instead use a non-strict inequality, though this takes some more effort.

[6] Recall from Section 14.1.2 that X_i is the state of the ith job in the system. In particular, $\mathbf{E}[W(\mathsf{rank}_{\mathrm{Gittins}} < r) \mid X_1, \ldots, X_N]$ takes expectation only over the future evolution of each job, because we condition on the entire system state.

Chapter 16

(Approximate) Gittins's (Approximate) Optimality in the M/G/1

The goal of this chapter is to prove that the Gittins policy minimizes mean holding cost in the M/G/1 under essentially any Markov-process job model (Ch. 14). In light of substantial prior work on Gittins [18, 28, 29, 43, 76, 79, 116, 128, 128, 136, 141], this result is not surprising. However, technical obstacles have a general result difficult to prove in the past. See Scully and Harchol-Balter [122] for a detailed review of prior proofs and their limitations.

Combining WINE (Ch. 15) and Relevant Work Decomposition (Ch. 8) makes proving Gittins's optimality easy, overcoming these past technical limitations (§ 16.1). In fact, we can prove something even stronger: if a policy assigns ranks within a constant factor of Gittins's, then it is in some sense within a constant factor of optimal (§ 16.2).

Section 16.1 is based on material from Scully and Harchol-Balter [122], but Section 16.2 is new material.

16.1 Optimality of Gittins

This section proves the following result.

Theorem 16.1. *Consider an M/G/1 under any Markov-process job model with any holding cost function. The Gittins policy minimizes mean preemptible holding cost* $\mathbf{E}[H^{\mathrm{P}}]$ *and mean system holding cost* $\mathbf{E}[H]$ *among all nonclairvoyant scheduling policies.*

In the case where all states are preemptible, Theorem 16.1 follows almost immediately from WINE (Thm. 15.10), which relates holding cost to Gittins-flavored relevant work, and Corollary 8.9, which implies that Gittins itself minimizes Gittins-flavored relevant work. However, this is not the complete story when some states are nonpreemptible, because WINE (Thm. 15.10) accounts only for jobs in preemptible states. Fortunately, it turns out that both the holding cost and relevant work caused by jobs in nonpreemptible states is the same across all scheduling policies.

Lemma 16.2. *Consider an M/G/1 under any Markov-process job model with any holding cost function.*

(a) The mean nonpreemptible holding cost $\mathbf{E}[H^{\mathrm{NP}}]$ *is the same under all scheduling policies.*

(b) Let $\mathbb{Y} \subseteq \mathbb{X}$ *be a set of states. The mean nonpreemptible relevant work* $\mathbf{E}[W^{\mathrm{NP}}(\mathbb{Y})]$ *is the same under all scheduling policies.*

In both cases, we restrict attention to scheduling policies that are work-conserving, or more generally those that do not destabilize the system.

Proof. We first observe that (b) is a special case of (a): simply let the holding cost of a nonpreemptible state x be its expected relevant remaining work, namely $h(x) \coloneqq \mathbf{E}[S_x(\mathbb{Y})]$.[1]

It thus suffices to prove (a). The important observation is that there is that if a job is in a nonpreemptible state, it must be in service, because all jobs start in a preemptible state (§ 14.1.1). Because the system is stable, a job is in service with probability ρ, and conditional on a job being in service, its steady-state distribution X_{eq} is the same under any scheduling policy.[2] We therefore have $\mathbf{E}[H^{\text{NP}}] = \mathbf{E}[h(X_{\text{eq}})\,\mathbb{1}(X_{\text{eq}} \in \mathbb{X}^{\text{NP}})]$ under any scheduling policy. □

Now that we know that scheduling does not impact how nonpreemptible states contribute to holding cost or relevant work, we are ready to prove Gittins's optimality using WINE.

Proof of Theorem 16.1. Let π be any nonclairvoyant scheduling policy. In light of Theorem 16.2(a), it suffices to prove $\mathbf{E}[H^{\text{P}}_{\text{Gittins}}] \le \mathbf{E}[H^{\text{P}}_{\pi}]$. We compute[3]

$$\begin{aligned}
\mathbf{E}[H^{\text{P}}_{\text{Gittins}}] &= \int_0^\infty \frac{\mathbf{E}[W^{\text{P}}_{\text{Gittins}}(\mathsf{rank}_{\text{Gittins}} < r)]}{r^2}\,\mathrm{d}r && \text{[by Thm. 15.10]}\\
&= \int_0^\infty \frac{\mathbf{E}[W_{\text{Gittins}}(\mathsf{rank}_{\text{Gittins}} < r)] - \mathbf{E}[W^{\text{NP}}_{\text{Gittins}}(\mathsf{rank}_{\text{Gittins}} < r)]}{r^2}\,\mathrm{d}r\\
&\le \int_0^\infty \frac{\mathbf{E}[W_{\pi}(\mathsf{rank}_{\text{Gittins}} < r)] - \mathbf{E}[W^{\text{NP}}_{\pi}(\mathsf{rank}_{\text{Gittins}} < r)]}{r^2}\,\mathrm{d}r\\
&\qquad \text{[by Cor. 8.9 and Thm. 16.2(b)]}\\
&= \int_0^\infty \frac{\mathbf{E}[W^{\text{P}}_{\pi}(\mathsf{rank}_{\text{Gittins}} < r)]}{r^2}\,\mathrm{d}r\\
&= \mathbf{E}[H^{\text{P}}_{\pi}]. && \text{[by Thm. 15.10]}
\end{aligned}$$

□

16.2 Approximate Optimality of Approximate Gittins

16.2.1 Approximate WINE

Definition 16.3. A scheduling policy is a (β, α)-*Gittins policy* for $\alpha > \beta > 0$ if it is represented by a rank function rank that satisfies

$$\beta\,\mathsf{rank}_{\text{Gittins}}(x) \le \mathsf{rank}(x) \le \alpha\,\mathsf{rank}_{\text{Gittins}}(x).$$

[1] This works for the same reason that we can reduce some unknown holding cost problems to known holding cost problems by looking at conditional expected holding cost (§ 14.2.1). Specifically, we would like to set "$h(x) \coloneqq S_x(\mathbb{Y})$", but we require holding costs to be deterministic, so we take an expectation over the job's remaining $\mathbb{Y}$-work.

[2] One can view X_{eq} as the stationary distribution of the Markov process that "loops" serving jobs forever. This is the same as the job Markov process, except if the job's state would become x_{done}, we instead "restart" the job by having it jump to a state distributed as X_{new}.

[3] We implicitly assume here that $\mathbf{E}[W^{\text{NP}}_{\pi}(\mathsf{rank}_{\text{Gittins}} < r)]$ is finite. It is possible to show that if it were infinite, then by our assumption that all states have positive holding cost (§ 14.1.3), all policies, including Gittins, would have infinite mean holding cost.

We also call such a policy a *γ-Gittins* policy for $\gamma = \alpha/\beta$, because scaling all ranks by the same factor does not affect the scheduling policy. We typically denote a γ-approximate Gittins policies by γ-G.

Theorem 16.4 (Approximate WINE)**.** *Let $\alpha > \beta > 0$ and $\gamma = \alpha/\beta$. Consider any (β, α)-approximately Gittins policy γ-G. In any queueing system using a nonclairvoyant scheduling policy, the preemptible holding cost can be bounded using an integral of expected γ-G-flavored relevant work:*

$$\frac{H_\pi^{\mathrm{P}}}{\alpha} \le \int_0^\infty \frac{\mathbf{E}[W_\pi^{\mathrm{P}}(\mathsf{rank}_{\gamma\text{-G}} < r) \mid X_1, \ldots, X_N]}{r^2}\,\mathrm{d}r \le \frac{H_\pi^{\mathrm{P}}}{\beta}.$$

In particular, provided the steady-state expectations are well defined,

$$\frac{\mathbf{E}[H_\pi^{\mathrm{P}}]}{\alpha} \le \int_0^\infty \frac{\mathbf{E}[W_\pi^{\mathrm{P}}(\mathsf{rank}_{\gamma\text{-G}} < r)]}{r^2}\,\mathrm{d}r \le \frac{\mathbf{E}[H_\pi^{\mathrm{P}}]}{\beta}.$$

Proof. We omit the subscript π throughout. By Definition 16.3, for all $r \ge 0$,

$$(\mathsf{rank}_{\mathrm{Gittins}} < r/\alpha) \subseteq (\mathsf{rank}_{\gamma\text{-G}} < r) \subseteq (\mathsf{rank}_{\mathrm{Gittins}} < r/\beta),$$

so Definition 7.3 implies that in any system state,

$$W^{\mathrm{P}}(\mathsf{rank}_{\mathrm{Gittins}} < r/\alpha) \le W^{\mathrm{P}}(\mathsf{rank}_{\gamma\text{-G}} < r) \le W^{\mathrm{P}}(\mathsf{rank}_{\mathrm{Gittins}} < r/\beta). \tag{16.1}$$

The lower and upper bounds then follow from two very similar computations, so we show just the lower bound computation:

$$\begin{aligned}
&\int_0^\infty \frac{\mathbf{E}[W^{\mathrm{P}}(\mathsf{rank}_{\gamma\text{-G}} < r) \mid X_1, \ldots, X_N]}{r^2}\,\mathrm{d}r \\
&\quad\ge \int_0^\infty \frac{\mathbf{E}[W^{\mathrm{P}}(\mathsf{rank}_{\mathrm{Gittins}} < r/\alpha) \mid X_1, \ldots, X_N]}{r^2}\,\mathrm{d}r && \text{[by (16.1)]} \\
&\quad= \frac{1}{\alpha}\int_0^\infty \frac{\mathbf{E}[W^{\mathrm{P}}(\mathsf{rank}_{\mathrm{Gittins}} < q) \mid X_1, \ldots, X_N]}{q^2}\,\mathrm{d}q && \text{[by letting } q = r/\alpha\text{]} \\
&\quad= \frac{H^{\mathrm{P}}}{\alpha}. && \text{[by Thm. 15.10]}
\end{aligned}$$

The expected value version follows from Tonelli's theorem. □

16.2.2 Using Approximate WINE to Prove Approximate Optimality

Theorem 16.5. *Consider an M/G/1 under any fully preemptible Markov-process job model, meaning $\mathbb{X} = \mathbb{X}^{\mathrm{P}}$, with any holding cost function. Any γ-Gittins policy γ-G has mean preemptible holding cost within a factor of γ of optimal:*

$$\mathbf{E}[H_{\gamma\text{-G}}^{\mathrm{P}}] \le \gamma \mathbf{E}[H_{\mathrm{Gittins}}^{\mathrm{P}}].$$

Proof. The result follows from Approximate WINE (Thm. 16.4) and the fact that γ-G minimizes γ-G-flavored relevant work (Cor. 8.9). Specifically, letting $\alpha > \beta > 0$ be such that γ-G is (β, α)-approximately Gittins (i.e. $\gamma = \alpha/\beta$), we compute[4]

$$\begin{aligned}
\mathbf{E}[H^{\mathrm{P}}_{\gamma\text{-G}}] &\leq \alpha \int_0^\infty \frac{\mathbf{E}[W^{\mathrm{P}}_{\gamma\text{-G}}(\mathsf{rank}_{\gamma\text{-G}} < r)]}{r^2}\,\mathrm{d}r && \text{[by Thm. 16.4]} \\
&\leq \alpha \int_0^\infty \frac{\mathbf{E}[W^{\mathrm{P}}_{\mathrm{Gittins}}(\mathsf{rank}_{\gamma\text{-G}} < r)]}{r^2}\,\mathrm{d}r && \text{[by Cor. 8.9 and Thm. 16.2(b)]} \\
&\leq \frac{\alpha}{\beta}\mathbf{E}[H^{\mathrm{P}}_{\gamma\text{-G}}] = \gamma\mathbf{E}[H_{\gamma\text{-G}}]. && \text{[by Thm. 16.4]}
\end{aligned}$$

□

[4]The step where we apply Corollary 8.9 and Theorem 16.2(b) uses the same trick as the proof of Theorem 16.1

Chapter 17

Response Time of Gittins in the M/G/k

This chapter bounds the mean response time of the Gittins policy in the M/G/k, namely Gittins-k. We consider an arbitrary *fully preemptible Markov-process job model with constant holding cost* (Ch. 14). That is, every state is preemptible, meaning $\mathbb{X}^{\mathrm{P}} = \mathbb{X}$, and has the holding cost, which we arbitrarily set to $h(x) = 1$. The state space and dynamics are unrestricted, so our results apply to settings with known job sizes (in which Gittins-k reduces to SRPT-k), unknown job sizes, and a wide range of scenarios with partial job size information.

17.1 Main Result: Mean Response Time Bound for Gittins-k

We follow our usual convention that each of the k servers in the M/G/k has speed $1/k$, giving the system maximum service rate 1 (§ 5.2.4). Under this convention, we prove the following result.

Theorem 17.1. *Consider an M/G/k with any fully preemptible Markov-process job model with constant holding cost. The mean response time of Gittins-k is bounded by*

$$\mathbf{E}[T_{\text{Gittins-}k}] \leq \mathbf{E}[T_{\text{Gittins-1}}] + \overbrace{\left(\frac{9}{8\log\frac{3}{2}} + 1\right)}^{\approx 3.7746}(k-1)\frac{\mathbf{E}[S]}{\rho}\log\frac{1}{1-\rho}.$$

Proof. The result follows from Propositions 17.3, 17.6, and 17.9, which appear later in this section. □

The bound in Theorem 17.1 implies that for (roughly speaking) finite-variance job size distributions, Gittins-k has the best possible mean response time in the heavy-traffic limit, because in that setting, the $\mathbf{E}[T_{\text{Gittins-1}}]$ term dominates.

Corollary 17.2. *Consider an M/G/k with any fully preemptible Markov-process job model with constant holding cost. If* $\mathbf{E}[S^2(\log S)^+] < \infty$*, then*

$$\lim_{\rho\to 1}\frac{\mathbf{E}[T_{\text{Gittins-}k}]}{\mathbf{E}[T_{\text{Gittins-1}}]} = 1,$$

and thus Gittins-k is optimal for mean response time in the heavy-traffic limit.

Proof. The limit of mean response time ratios follows from Theorem 17.1 and prior results on the heavy-traffic scaling of mean response time under SRPT-1, which is a lower bound on the scaling of Gittins-1. Specifically, Scully et al. [118, Appx. B.2] show using results from Lin et al. [83] that when $\mathbf{E}[S^2(\log S)^+] < \infty$,

$$\mathbf{E}[T_{\text{SRPT-1}}] > \omega\left(\log \frac{1}{1-\rho}\right).$$

That the limit of mean response time ratios implies heavy-traffic optimality follows from the fact that Gittins-1 minimizes mean response time in the M/G/1 (Thm. 16.1). Specifically our convention of scaling M/G/k server speeds to $1/k$ each, one can view the M/G/k as a restricted version of the M/G/1, so the best possible mean response time in the M/G/k is at least that of Gittins-1 in the M/G/1. □

17.1.1 Comparison with Previous Publication

This chapter is based on work published in Scully et al. [118]. However, the bound in Theorem 17.1 differs slightly from the main result of Scully et al. [118, Thm. 1.1 and 1.2]. The version in Theorem 17.1 has a major advantage over the prior work: the bound on the gap $\mathbf{E}[T_{\text{Gittins-}k}] - \mathbf{E}[T_{\text{Gittins-1}}]$ is insensitive to the job size distribution beyond its mean. In contrast, the bound shown by Scully et al. [118] depends on higher moments of the job size distribution. In particular, they require $\mathbf{E}[S^\alpha] < \infty$ for some $\alpha > 1$, and their bound has a $\frac{1}{\alpha-1} \log \mathbf{E}[S^\alpha]$ term, meaning it diverges as $\alpha \to 1$.

With that said, the bound shown by Scully et al. [118] does have the advantage that it may be tighter for some size distributions. In particular, it implies that for size distributions S with $\mathbf{E}[S^\alpha] < \infty$ for all $\alpha \geq 1$, the mean response time gap between Gittins-k and Gittins-1 grows as $(k-1)\, o\big(\log \frac{1}{1-\rho}\big)$. We leave the question of tightening the bound to future work.

17.2 Proof: Combining WINE and Relevant Work Decomposition

We illustrate our approach in Figure 17.1. The main idea is to use Relevant Work Decomposition (Ch. 8) to compare the mean relevant system work of Gittins-k to that of Gittins-1. WINE (Ch. 15) and Little's law [84] combine to convert this into a comparison of mean response time, yielding an expression for $\mathbf{E}[T_{\text{Gittins-}k}] - \mathbf{E}[T_{\text{Gittins-1}}]$.

In Figure 17.1, and more generally in the rest of this section, we write $\prec r$ as shorthand for $(\mathsf{rank}_{\text{Gittins}} < r)$ throughout, as the only rank function under consideration is that of Gittins. When we omit the subscript on a quantity, we are referring by default to that quantity in the M/G/k under Gittins-k.

Proposition 17.3. *Consider an M/G/k with any fully preemptible Markov-process job model with constant holding cost. The mean response time gap between Gittins-k and an M/G/1*

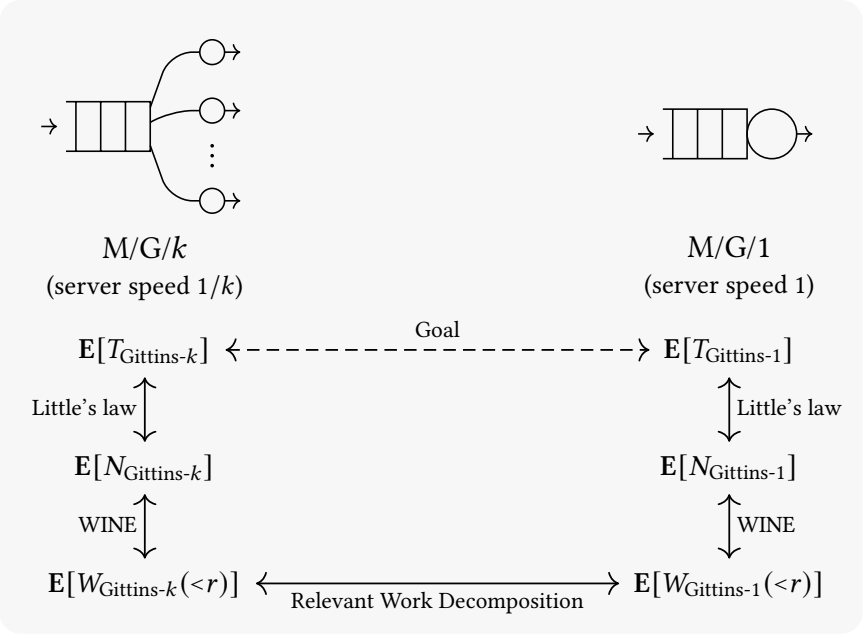


Figure 17.1. Our strategy for analyzing the mean response time of Gittins-k, of which SRPT-k is a special case, in the M/G/k. We use WINE to relate mean response time to relevant work, then use Relevant Work Decomposition to bound relevant work in the M/G/k.

using Gittins-1 is[1]

$$\mathbf{E}[T_{\text{Gittins-}k}] - \mathbf{E}[T_{\text{Gittins-1}}] = \frac{1}{\lambda}\int_0^\infty \frac{\mathbf{E}[(1-J(<r))\,W(<r)] + \lambda_{\text{rcy}}(<r)\,\mathbf{E}_{\text{rcy}}(<r)[S_{\text{rcy}}(<r)\,W(<r)]}{r^2(1-\rho(<r))}\,\mathrm{d}r,$$

where quantities on the right-hand side refer to Gittins-k.

Proof. The result is immediate from combining Theorems 8.7(b) and 15.10 then applying Little's law [84], namely $\mathbf{E}[N] = \lambda\mathbf{E}[T]$. □

One can actually generalize Proposition 17.3 beyond the Gittins-k. The proof only assumes that we are comparing a system with M/G arrivals, of which an M/G/k using Gittins-k is a special case, to an M/G/1 using Gittins-1. Nevertheless, we hereafter work exclusively with Gittins-k, and thus usually omit the subscript.

The main remaining challenge is that the right-hand side of Proposition 17.3 is intractable to analyze exactly. Bounding this expression is the main technical accomplishment of this chapter. The expression can be separated into two terms.

- (§ 17.2.1) *Relevant idleness:* the term containing $\mathbf{E}[(1-J(<r))\,W(<r)]$ measures $<r$-work while the system has $<r$-idle servers.

[1] See Section 8.3 for the definitions of $J(<r)$, $\lambda_{\text{rcy}}(<r)$, and $\mathbf{E}_{\text{rcy}}(<r)[\cdot]$. More generally, see Appendix A for an index of notation.

- (§ 17.2.2) *Recycling:* the term containing $\lambda_{\text{rcy}}(<r)\,\mathbf{E}_{\text{rcy}}(<r)[S_{\text{rcy}}(<r)\,W(<r)]$ measures $<r$-work at the moments immediately after $<r$-recyclings occur.

17.2.1 Bounding the Relevant Idleness Term

Lemma 17.4. *Consider an M/G/k with any fully preemptible Markov-process job model with constant holding cost. Under Gittins-k,*

$$\int_0^\infty \frac{\mathbf{E}[(1-J(<r))\,W(<r)]}{r^2(1-\rho(<r))}\,\mathrm{d}r \le (k-1)\inf_{n\in\mathbb{Z}_{\ge 1}} n\left(\frac{1}{1-\rho}\right)^{1/n}.$$

Proof. At a high level, our approach is as follows. In the M/G/k, $1-J(<r)$ is the fraction of servers that are $<r$-idle. Because Gittins-k prioritizes $<r$-jobs, whenever $1-J(<r)>0$, there are at most $k-1$ $<r$-jobs in the system, or else all servers would be $<r$-busy. Roughly speaking, we would like to use WINE inside the expectation, because the $W(<r)$ represents the $<r$-work of at most $k-1$ jobs. However, it is not so straightforward to do this because of the other factors which depend on r.

Observe that the sets (Def. 7.2(a))

$$<r \coloneqq \{x \in \mathbb{X} \mid \mathsf{rank}_{\text{Gittins}}(x) < r\},$$
$$\le r \coloneqq \{x \in \mathbb{X} \mid \mathsf{rank}_{\text{Gittins}}(x) \le r\}$$

are monotonic, namely growing, as functions of r. This means $<r$-fresh load $\rho(<r)$ (Def. 7.4(c)) is increasing in r, with $\rho(<0)=0$ and $\rho(<\infty)=\rho$. Our idea is to break the integral into $n \ge 1$ "chunks" with boundaries at

$$0 =: r_0 < r_1 < \ldots < r_{n-1} < r_n \coloneqq \infty.$$

Using the monotonicity of $<r$, we obtain

$$\begin{aligned}\int_0^\infty \frac{\mathbf{E}[(1-J(<r))\,W(<r)]}{r^2(1-\rho(<r))}\,\mathrm{d}r &= \sum_{i=1}^{n}\int_{r_{m-1}}^{r_m} \frac{\mathbf{E}[(1-J(<r))\,W(<r)]}{r^2(1-\rho(<r))}\,\mathrm{d}r \\ &\le \sum_{i=1}^{n}\int_{r_{m-1}}^{r_m} \frac{\mathbf{E}[(1-J(\le r_{m-1}))\,W(<r)]}{r^2(1-\rho(<r_m))}\,\mathrm{d}r.\end{aligned}$$

Let

$$\gamma \coloneqq \left(\frac{1}{1-\rho}\right)^{1/n}.$$

It suffices to show that each of the n chunks, namely each term in the sum, is at most $(k-1)\gamma$. To facilitate doing so, we choose the values of $r_1,\ldots,r_{n-1}$ such that for all $m \in \{1,\ldots,n\}$,[2]

$$\frac{1-\rho(\le r_{m-1})}{1-\rho(<r_m)} \le \gamma, \tag{17.1}$$

[2]The numerator uses $\le r_{m-1}$ instead of $<r_{m-1}$ to account for the fact that $\rho(<r)$ may have discontinuities. One can show that $\le r_{m-1}$ is equivalent to the right limit $<(r_{m-1}+)$.

which is possible by the definition of γ and monotonicity of $\rho(<r)$.

In the M/G/k, $1 - J(<r)$ is the fraction of servers that are $<r$-idle. If this is nonzero, then because Gittins-k prioritizes $<r$-jobs, there can be at most $k-1$ $<r$-jobs. Therefore, without loss of generality, we can index jobs such that whenever $1 - J(<r) > 0$, all $<r$-jobs are among jobs $1, \ldots, k-1$. This means, by Tonelli's theorem,

$$\begin{aligned}\int_{r_{m-1}}^{r_m} \frac{\mathbf{E}[(1 - J(\leq r_{m-1}))\, W(<r)]}{r^2(1-\rho(<r_m))}\,\mathrm{d}r &= \sum_{i=1}^{k-1} \int_{r_{m-1}}^{r_m} \frac{\mathbf{E}[(1 - J(\leq r_{m-1}))\, S_{X_i}(<r)]}{r^2(1-\rho(<r_m))}\,\mathrm{d}r \\ &= \sum_{i=1}^{k-1} \mathbf{E}\left[\frac{1 - J(\leq r_{m-1})}{1-\rho(<r_m)} \int_{r_{m-1}}^{r_m} \frac{S_{X_i}(<r)}{r^2}\,\mathrm{d}r\right].\end{aligned}$$

It suffices to show each term of the sum is at most γ. Because Gittins-k is nonclairvoyant, the distribution of $S_{X_i}(<r)$ depends only on X_i, so

$$\begin{aligned}&\mathbf{E}\left[\frac{1 - J(\leq r_{m-1})}{1-\rho(<r_m)} \int_{r_{m-1}}^{r_m} \frac{S_{X_i}(<r)}{r^2}\,\mathrm{d}r\right] \\ &\quad= \mathbf{E}\left[\frac{1 - J(\leq r_{m-1})}{1-\rho(<r_m)} \int_{r_{m-1}}^{r_m} \frac{\mathbf{E}[S_{X_i}(<r) \mid X_i]}{r^2}\,\mathrm{d}r\right] \\ &\quad= \mathbf{E}\left[\frac{1 - J(\leq r_{m-1})}{1-\rho(<r_m)}\right]. && \text{[by Prop. 15.9 and } h(x) = 1\text{]} \\ &\quad= \frac{1 - \rho(<r_{m-1}) - \rho_{\mathrm{rcy}}(<r_{m-1})}{1-\rho(<r_m)} && \text{[by Cor. 8.5]} \\ &\quad\leq \gamma. && \text{[by (17.1) and } \rho_{\mathrm{rcy}}(<r) \geq 0\text{]}\end{aligned}$$

□

Lemma 17.5. *Consider an M/G/k with any fully preemptible Markov-process job model with constant holding cost. Under Gittins-k,*

$$\int_0^\infty \frac{\mathbf{E}[(1 - J(<r))\, W(<r)]}{r^2(1-\rho(<r))}\,\mathrm{d}r \leq (k-1)\frac{\rho}{1-\rho}.$$

Proof. Let A and B be two random servers, each independently and uniformly sampled (with replacement) from the k servers in the M/G/k. Let X_A and X_B be the states of jobs at servers A and B, respectively, setting the state to x_{done} if the server is idle. By reasoning about $(1 - J(<r))\, W(<r)$ similarly to the proof of Lemma 17.5, we can write its expectation in terms of X_A and X_B:

$$\mathbf{E}[(1 - J(<r))\, W(<r)] = k\mathbf{E}[S_{X_A}(<r)\, \mathbb{1}(\mathsf{rank}_{\mathrm{Gittins}}(X_B) \geq r)].$$

For any state x, we have

$$S_x(<r)\, \mathbb{1}(\mathsf{rank}_{\mathrm{Gittins}}(x) \geq r) = 0,$$

which means

$$\begin{aligned}\mathbf{E}[(1-J(<r))\,W(<r)] &= k\mathbf{E}[S_{X_A}(<r)\,\mathbb{1}(\mathsf{rank}_{\text{Gittins}}(X_B)\geq r, A\neq B)]\\ &= (k-1)\mathbf{E}[S_{X_A}(<r)\,\mathbb{1}(\mathsf{rank}_{\text{Gittins}}(X_B)\geq r)\mid A\neq B]\\ &\leq (k-1)\mathbf{E}[S_{X_A}(<r)\mid A\neq B]\\ &= (k-1)\mathbf{E}[S_{X_A}(<r)]\\ &= (k-1)\rho\mathbf{E}[S_{X_A}(<r)\mid X_A\neq x_{\text{done}}].\end{aligned}$$

Because Gittins-k is nonclairvoyant, the distribution of $S_{X_A}(<r)$ depends only on X_A. Bounding $\rho(<r)\leq\rho$ and applying the single-job version of WINE (Prop. 15.9) yields the desired result. □

Proposition 17.6. *Consider an M/G/k with any fully preemptible Markov-process job model with constant holding cost. Under Gittins-k,*

$$\int_0^\infty \frac{\mathbf{E}[(1-J(<r))\,W(<r)]}{r^2(1-\rho(<r))}\,\mathrm{d}r \leq \frac{9}{8\log\frac{3}{2}}(k-1)\log\frac{1}{1-\rho}.$$

Proof. After combining the bounds from Lemmas 17.4 and 17.5 by taking their minimum, all that remains is to find a multiple of $\log\frac{1}{1-\rho}$ that upper bounds this minimum. One can easily verify that

$$\begin{aligned}\rho\in[0,0.83] \quad &\Rightarrow \quad \frac{\rho}{1-\rho}\leq\frac{9}{8\log\frac{3}{2}}\log\frac{1}{1-\rho},\\ \rho\in\left[1-\left(\frac{4}{9}\right)^n, 1-\left(\frac{8}{27}\right)^n\right] \quad &\Rightarrow \quad n\left(\frac{1}{1-\rho}\right)^{1/n}\leq\frac{9}{8\log\frac{3}{2}}\log\frac{1}{1-\rho}.\end{aligned}$$

The union of the load intervals above is $[0,1)$, so the desired bound holds at all loads. □

17.2.2 Bounding the Recycling Term

Lemma 17.7. *Consider an M/G/k with any fully preemptible Markov-process job model with constant holding cost. Under Gittins-k,*

$$\lambda_{\text{rcy}}(<r)\,\mathbf{E}_{\text{rcy}}(<r)[S_{\text{rcy}}(<r)\,W(<r)]\leq(k-1)r\rho_{\text{rcy}}(<r).$$

Proof. In the instant before an $<r$-recycling (Def. 7.5) occurs, the job undergoing $<r$-recycling is $<r$-irrelevant but in service, so at least one server is $<r$-idle. Because Gittins-k prioritizes $<r$-jobs, it must be that at immediately before the $<r$-recycling, there are at most $k-1$ $<r$-jobs. Without loss of generality, we can index jobs such that those $<r$-jobs are among jobs $1,\ldots,k-1$. This lets us write

$$\lambda_{\text{rcy}}(<r)\,\mathbf{E}_{\text{rcy}}(<r)[S_{\text{rcy}}(<r)\,W(<r)]=\sum_{i=1}^{k-1}\lambda_{\text{rcy}}(<r)\,\mathbf{E}_{\text{rcy}}(<r)[S_{\text{rcy}}(<r)\,S_{X_i}(<r)].$$

Because Gittins-k is nonclairvoyant, the distribution of $S_{X_i}(<r)$ depends only on X_i. Using this, we obtain

$$\begin{aligned}
&\lambda_{\text{rcy}}(<r)\,\mathbf{E}_{\text{rcy}}(<r)[S_{\text{rcy}}(<r)\,S_{X_i}(<r)] \\
&\quad= \lambda_{\text{rcy}}(<r)\,\mathbf{E}_{\text{rcy}}(<r)\big[S_{\text{rcy}}(<r)\,\mathbf{E}[S_{X_i}(<r)\mid X_i]\big] \\
&\quad\le r\lambda_{\text{rcy}}(<r)\,\mathbf{E}_{\text{rcy}}(<r)[S_{\text{rcy}}(<r)] && \text{[by Lem. 15.5]} \\
&\quad= r\rho_{\text{rcy}}(<r). && \text{[by Def. 7.5]}
\end{aligned}$$

□

Lemma 17.8. *Consider an M/G/k with any fully preemptible Markov-process job model with constant holding cost. For all* $r \ge 0$,[3]

$$\begin{aligned}
\rho_{\text{rcy}}(<r) &\le \lambda r\mathbf{E}[h(Z_{X_{\text{new}}}(<r))], \\
\rho(<r)+\rho_{\text{rcy}}(<r) &\le \lambda\mathbf{E}[\mathsf{game}_{X_{\text{new}}}(<r)] \coloneqq \lambda\mathbf{E}[S_{X_{\text{new}}}(<r)] + \lambda r\mathbf{E}[h(Z_{X_{\text{new}}}(<r))].
\end{aligned}$$

Proof. By Definition 7.4(c), it suffices to prove the second inequality. Our approach is to bound the expected amount of service a job receives while it is $<r$-fresh or $<r$-recycled. Specifically, by Corollary 8.5 and Little's law [84], it suffices to show that while serving a job to completion starting at any initial state x, the expected amount of service during which the job is $<r$-relevant is at most $\mathsf{game}_x(<r)$.

The key idea is to consider the following "bank account" variant of the Gittins game. We start the game with a bank account storing value u. Whenever the job is *irrelevant*, we may spend the bank account value at rate 1 to serve the job without incurring the usual cost. We cannot spend from the bank account while the job is relevant. If we stop serving the job before it completes, we still pay the penalty r, and any leftover value in the bank account is worthless.

Let $\mathsf{game}_x(r;u)$ be the minimal disutility in the bank account Gittins game starting with value u in the bank account. We have $\mathsf{game}_x(r;u) = \mathsf{game}_x(r)$, and clearly $\mathsf{game}_x(r;u)$ is (weakly) decreasing in u, because we can always choose not to spend from the bank account. It thus suffices to show that

$$\mathsf{game}_x(r;\infty) \coloneqq \lim_{u\to\infty} \mathsf{game}_x(r;u)$$

is the expected amount of service during which the job is $<r$-relevant as it is served from initial state x to completion. When $u=\infty$, we will never give up on the job while it is $<r$-irrelevant. It thus suffices to show that when $u=\infty$, it is always optimal to serve the job whenever it is $<r$-relevant.[4]

Theorem 15.6(a) implies that when $u=0$, namely in the ordinary Gittins game, it is optimal to serve the job while it is $<r$-relevant. But increasing u only makes serving the

[3] Recall from Section 14.1 that X_{new} is the random state of a new job, and recall from Definition 14.4 that $Z_x(\mathbb{Y})$ is the random state a job starting in state x is in when it first leaves set of states $\mathbb{Y}$. In our constant-holding cost setting, where $h(x)=1$ for all $x\in\mathbb{X}$ but $h(x_{\text{done}})=0$, we can think of $\mathbf{E}[h(Z_{X_{\text{new}}}(<r))]$ as the probability a job completes before it reaches or exceeds rank r.

[4] Here we are somewhat informal in that we reason about $u=\infty$ directly. Strictly speaking, we should show that the probability we give up approaches zero as $u\to\infty$, but this is not hard to do.

job more appealing, so serving the job while it is $<r$-relevant is optimal for any value of u, as desired. □

Proposition 17.9. *Consider an M/G/k with any fully preemptible Markov-process job model with constant holding cost. Under Gittins-k,*

$$\int_0^\infty \frac{\lambda_{\text{rcy}}(<r)\,\mathbf{E}_{\text{rcy}}(<r)[S_{\text{rcy}}(<r)\,W(<r)]}{r^2(1-\rho(<r))} \leq (k-1)\log\frac{1}{1-\rho}.$$

Proof. We compute

$$\begin{aligned}
&\int_0^\infty \frac{\lambda_{\text{rcy}}(<r)\,\mathbf{E}_{\text{rcy}}(<r)[S_{\text{rcy}}(<r)\,W(<r)]}{r^2(1-\rho(<r))} \\
&\quad\leq (k-1)\int_0^\infty \frac{\mathbf{E}[h(Z_{X_{\text{new}}}(<r))]}{1-\rho(<r)}\,\mathrm{d}r && \text{[by Lems. 17.7 and 17.8]} \\
&\quad\leq (k-1)\int_0^\infty \frac{\mathbf{E}[h(Z_{X_{\text{new}}}(<r))]}{1-\lambda\mathbf{E}[\text{game}_{X_{\text{new}}}(r)]}\,\mathrm{d}r && \text{[by Lem. 17.8 and } \rho_{\text{rcy}}(<r)\geq 0\text{]} \\
&\quad= (k-1)\log\frac{1}{1-\rho}. && \text{[by Lem. 15.7]}
\end{aligned}$$

□

Part IV

Conclusion

Chapter 18

Conclusion

We have presented two new queueing-theoretic tools, SOAP (Part II) and WINE (Part III), and applied them to solve numerous problems in scheduling theory (§ 18.1). We end the thesis by discussing potential avenues for future work (§ 18.2).

18.1 Open Problems Solved

18.1.1 First Response Time Analysis for Many Policies in the M/G/1

The core problem SOAP solves is analyzing the response time distribution of an M/G/1 under any policy from a broad policy class. The class, which we call *SOAP policies*, consists, roughly speaking, of policies where a job's priority depends on its age, namely how long the job has been served so far (Ch. 6). We give a *universal analysis* of all SOAP policies, deriving formulas for the mean and LST of response time in an M/G/1 (Ch. 7).

There are infinitely many SOAP policies, the vast majority of them have never been analyzed before, so SOAP gives the first analysis of such policies. This includes, with a single exception [48], all policies where a job's priority varies nonmonotonically as a function of age. Some specific examples of nonmonotonic policies are the following:

- (Pol. 6.11) SERPT, a policy that aims to achieve low mean response time when job sizes are uncertain.
- (Pol. 6.12) Gittins, the policy that minimizes mean response time when job sizes are uncertain. Despite being known to minimize mean response time, the actual mean response time achieved by Gittins was previously unknown.
- (Pol. 6.13) Policies where jobs can only be preempted at certain age checkpoints.

18.1.2 First Comparison of (Monotonic) SERPT and Gittins

While Gittins is known to minimize mean response time, SERPT is a much simpler policy. It is thus natural to wonder whether SERPT's mean response time is close enough to Gittins's to be able to serve as a substitute. This appears to be the case in several numerical applications of SOAP (Ch. 10). While we do not manage to prove a formal response time guarantee for SERPT itself, we propose a new variant of SERPT, called *Monotonic SERPT (M-SERPT)*, which we prove has mean response time within a factor of 5 of Gittins's (Ch. 11).

Consequences of M-SERPT Bounds for LAS and the MLPS Class

It turns out that M-SERPT is a member of the MLPS class of policies [74], so our result for M-SERPT implies that MLPS policies can achieve within a constant factor of the optimal mean response time.

For some size distributions, namely those with strictly increasing mean residual lifetime, M-SERPT reduces to LAS, so LAS is within a constant factor of optimal for such distributions.

18.1.3 First Asymptotic Tail Analysis for Many Policies in the M/G/1

While SOAP gives us formulas for the mean and LST of response time, it is non trivial to extract results about the response time tail $\mathbf{P}[T > t]$ from the LST. We take a step in this direction by giving conditions under which a SOAP policy's response time tail is asymptotically optimal, meaning it decays as fast as possible as $t \to \infty$ (Ch. 13). Our results cover both heavy-tailed and light-tailed job size distributions.

Consequences for Gittins's Asymptotic Response Time Tail

We use the above result to investigate the asymptotic response time tail of Gittins. While Gittins is known to have optimal mean response time, nothing was previously known about its response time tail. We show that for heavy-tailed job size distributions, Gittins is (asymptotically) tail-optimal, but that for light-tailed job size distributions, Gittins can be anywhere between tail-optimal and tail-pessimal (Ch. 13). Fortunately, we also show that in cases where Gittins is tail-pessimal, we can slightly adjust Gittins to obtain a policy that has near-optimal mean response time and avoids tail-pessimality.

18.1.4 First Policies with Provable Robustness to Size Estimation Error

We consider the problem of scheduling to minimzie mean response time when given noisy size estimates. We begin studying this problem by numerically applying SOAP (Ch. 10). We observe that naively translating SRPT (Pol. 6.10) to this setting results in poor performance, even though the seemingly similar PSJF policy (Pol. 6.9) is much more robust to size estimation noise. Guided by these numerical results, we again use SOAP, this time theoretically, to prove that PSJF does indeed satisfy some formal robustness properties (Ch. 12). We also show how to modify SRPT to make it similarly robust.

18.1.5 First Response Time Bounds for Two Policies in the M/G/k

Very few scheduling policies have been analyzed in the M/G/k. Even SRPT, which was analyzed in the M/G/1 decades ago [117], has not been analyzed in the M/G/k. We obtain

the first mean response time bounds on SRPT in the M/G/k (Ch. 17). Our bounds are tight enough to imply that when the job size distribution has (roughly speaking) finite variance, SRPT is optimal for mean response time in heavy traffic. We also show analogous results for Gittins. We prove the bounds by combining WINE (Ch. 15) with Relevant Work Decomposition (Ch. 8).

18.1.6 New Generalization and Variants of the Gittins Policy

Several versions of the Gittins policy exist in the queueing literature (§ 2.2), with each version defined for a specific model of job size uncertainty. We present a new highly general job model that includes many other models of job size uncertainty as special cases, resulting in a highly general version of Gittins that subsumes most previous versions (Ch. 14). We use WINE to prove that this new general version of Gittins is indeed optimal for mean response time, or more generally for mean holding cost (Ch. 16).

In addition to studying Gittins itself, we also study policies that assign priorities in approximately the same way as Gittins. We show that such policies have approximately optimal performance (Ch. 16).

18.2 Future Work

18.2.1 Next Steps

Multiserver Systems Beyond the M/G/k

We prove our mean response time bounds for SRPT and Gittins in the M/G/k (Ch. 17) by combining WINE (Ch. 15) and Relevant Work Decomposition (Ch. 8). However, WINE holds in any queueing system, and Relevant Work Decomposition holds in any system with M/G arrivals. It thus seems likely that we can use a similar technique to analyze other multiserver systems, particularly if their scheduling policy is an appropriate analogue of SRPT or Gittins. An example might be scheduling in input-queued switches [37, 59, 60, 85, 132] that use SRPT or Gittins instead of FCFS within each of its queues. Of course, determining exactly what constitutes an "appropriate analogue" of SRPT or Gittins may well be challenging.

Heavy-Traffic Analysis

SOAP gives an exact M/G/1 mean response time formula (Ch. 7), so it should in principle be possible to determine how mean response time scales in the heavy-traffic limit, namely as $\rho \to 1$. We have already analyzed Gittins, SERPT, and M-SERPT in the unlabeled case this way [119], and it may be possible to extend the technique to other SOAP policies. For a heavy-traffic analysis of Gittins in the Markov-process job model, WINE (Ch. 15) may provide an easier path than SOAP.

Fairness

The analyses of SRPT and other policies enables work studying their fairness properties, namely whether they (roughly speaking) give adequate attention to jobs of all sizes [15, 143–145]. With the SOAP analysis in hand, can we do the same for all SOAP policies? In addition to considering fairness based on size, it may also be important to consider fairness based on label.

Batch-Poisson Arrival Processes

This thesis studies the M/G/1 and M/G/k, both of which have Poisson arrivals. However, we believe that most or all of our results can be generalized to batch-Poisson arrivals without too much trouble.

Unifying Work Decomposition Law

The work decomposition laws we show (Ch. 8) are similar to, but not quite the same as, similar work decomposition laws from the literature (§ 2.4.1). Can we find a single work decomposition law that has all the others as special cases?

Provable Robustness under Unbounded Size Estimation Error

Our results on proving robustness to noisy job size estimates (Ch. 12) assume that multiplicative estimation error is bounded. Can we generalize the robustness results to cases where the multiplicative estimation error is unbounded, such as log-normal noise models?

18.2.2 Long-Term Directions

Scheduling in Network Switches

SOAP can analyze scheduling policies that have access to only a limited number of priority levels, and it can also analyze scheduling policies where a job can only be preempted at certain checkpoint ages (Ch. 9). These are two preemption limitations that show up when scheduling packet flows in network switches [96]: the switch has a limited number of priority levels built into its hardware, and packet flows consist of packets that we would prefer not to preempt. Does SOAP in the M/G/1 provide an accurate enough model to tune real-world scheduling policies for network switch scheduling? If not, what other essential features of network switch scheduling can we integrate into our model?

Modeling Dynamic Preemption Overhead

We study scheduling with preemption limited to checkpoint ages in a setting where each checkpoint incurs an overhead (Ch. 9). However, we assume that the overhead is incurred even if the job is not preempted. This is the right model for some systems, such

as scheduling packet flows, because each packet carries a header. But other systems do not incur significant scheduling overhead until a job is actually preempted. Can we apply ideas similar to SOAP to analyze scheduling with dynamic preemption overhead?

Scheduling with Under-Specified Uncertainty Model

We consider scheduling with unknown job sizes throughout this thesis (Chs. 11 and 13). However, we always assume knowledge of the job size distribution. But in practice, we may only approximately know the job size distribution, or we may have to infer it from data. How should we schedule when even the size distribution itself is uncertain, in addition to the uncertainty of individual jobs' sizes? One idea is to try to design a distributionally robust version of Gittins.

Appendix

Appendix A

Index of Notation

A.1 General

Notation	Description	References
$\mathbb{N}$	natural numbers, i.e. $\{0, 1, 2, \ldots\}$	
$\mathbb{Z}_{\geq 1}$	positive integers, i.e. $\{1, 2, \ldots\}$	
$\mathbb{R}$	real numbers	
$\mathbb{R}_{\geq 0}$	nonnegative reals, i.e. $[0, \infty)$	
$\mathbb{R}_{>0}$	positive reals, i.e. $(0, \infty)$	
$\mathbf{P}[\cdot]$	probability	
$\mathbf{E}[\cdot]$	expectation	
$:=$	defined to be equal to	
$f(x-)$	left limit of f at x, i.e. $\lim_{y \uparrow x} f(y)$	§ 5.6.6
$f(x+)$	right limit of f at x, i.e. $\lim_{y \downarrow x} f(y)$	§ 5.6.6
$(x)^+$	positive part of x, i.e. $\max\{x, 0\}$	§ 5.6.6

A.2 Distributions

Notation	Description	References
Bernoulli(p)	Bernoulli distribution	§ 5.6.2
Geo(p)	geometric distribution with support at 0	§ 5.6.2
$\mathcal{E}V$	excess distribution of V	Def. 5.1
$(\mathcal{E}V)_i$	ith of many i.i.d. samples from $\mathcal{E}V$	Def. 5.1
$\mathcal{L}[V](\theta)$	Laplace-Stieltjes transform (LST) of V	Def. 5.1
$=_{\text{st}}$	equality in distribution	§ 5.6.2
$\leq_{\text{st}}$	stochastic ordering	§ 5.6.2

We typically abuse notation by not distinguishing between random variables and their distributions (§ 5.6.2).

A.3 M/G Arrivals

Notation	Description	References
λ	job arrival rate	§ 5.2.2
S	job size distribution	§ 5.2.2

Continued on next page

Continued from previous page

Notation	Description	References
ρ	load	§ 5.2.2
$\mathbb{L}$	set of possible labels	§ 5.2.2
L	job label distribution	§ 5.2.2
(L, S)	label-size pair distribution	§ 5.2.2
S_ℓ	label-conditional size distribution	§ 5.2.3
$S_{\ell,a}$	state-conditional remaining work distribution	§ 5.2.3
(L_i, A_i)	state, i.e. label-age pair, of ith job in the system	§ 5.6.1

A.4 Queueing Metrics

Notation	Description	References
T	response time	§ 5.4
$T(\ell, s)$	label-size conditional response time	§ 5.4
N	number of jobs in system	§ 5.4.3
W	system work, i.e. total remaining work of all jobs in the system	§ 5.5.1
B	busy period	§ 5.5.2
$B(v)$	busy period started by initial work v	§ 5.5.2
J	rate at which system is serving jobs	§ 8.1.1
$h(x)$	holding cost of a job in state x	§ 14.1.3
H	system holding cost, i.e. total holding cost of all jobs in the system	§ 14.1.3

We often specify the scheduling policy π in a subscript, as in T_π.

A.5 Scheduling Policies

Notation	Description	References
π	variable denoting a scheduling policy	
π-k	k-server version of policy π	§ 6.1.5
FCFS	First-Come, First-Served (a.k.a. first-in, first-out)	§ 5.3.1 and Pol. 6.5
LCFS	Last-Come, First-Served (a.k.a. last-in, first-out)	§ 5.3.1 and Pol. 6.6
ROS	Random Order of Service	§ 5.3.1
PLCFS	Last-Come, First-Served (a.k.a. last-in, first-out)	§ 5.3.1 and Pol. 6.7
PS	Processor Sharing	§ 5.3.1
LAS	Least Attained Service (a.k.a. foreground-background)	§ 5.3.1 and Pol. 6.4
NP-Prio	Nonpreemptive Priority	§ 5.3.1 and Pol. 6.8
P-Prio	Preemptive Priority	§ 5.3.1 and Pol. 6.8

Continued on next page

Continued from previous page

Notation	Description	References
SJF	Shortest Job First	§ 5.3.1 and Pol. 6.9
PSJF	Preemptive Shortest Job First	§ 5.3.1 and Pol. 6.9
SRPT	Shortest Remaining Processing Time	§ 5.3.1 and Pol. 6.10
SERPT	Shortest Expected Remaining Processing Time	Pol. 6.11
M-SERPT	Monotonic SERPT	Pol. 11.1
Gittins	the Gittins policy (a.k.a. the Gittins index policy)	Pols. 14.5 and 6.12
Chk-π	Checkpointed π	Pol. 6.13
LPL-π	Limited-Priority-Level π	Pol. 6.14
PSJF-E	PSJF with Estimates	Pol. 10.2
SRPT-E	SRPT with Estimates	Pol. 10.1
SRPT-B	SRPT with Bounce	Pol. 12.3
SRPT-SE	SRPT with Scaling Estimates	Pol. 12.5

A.6 SOAP and Rank Functions

Notation	Description	References
$\mathsf{rank}_\pi(\ell, a)$	rank of a job with label ℓ and age a under π	Def. 6.1
$\mathsf{worst}_\pi(\ell, s, a)$	worst future rank of a job with label ℓ, size s, and age a under π	Def. 7.9
$\mathsf{worst}_\pi(\ell, s)$	worst ever rank of a job with label ℓ and size s under π	Def. 7.9
T^{wait}	waiting time, i.e. time between arrival and first service	§ 7.1.1
$T^{\text{wait}}(\ell, s)$	label-size conditional waiting time	§ 7.1.1
T^{resd}	residence time, i.e. time between first service and departure	§ 7.1.1
$T^{\text{resd}}(\ell, s)$	label-size conditional residence time	§ 7.1.1

In the SOAP analysis, we use T^{wait} and T^{resd} to denote *pessimism-adjusted* waiting and residence times (§ 7.3.3).

A.7 Relevant Work and Related Concepts

Notation	Description	References
$\mathbb{Y}$	variable denoting a set of "relevant" job states	
$\preceq r, \prec r$	sets of states with rank at most or strictly less than r	Def. 7.2(a)
$(\mathsf{rank}_\pi \leq r)$	alternative notation for $\preceq r$ that makes the policy π explicit	Def. 15.1
$S_x(\mathbb{Y})$	$\mathbb{Y}$-relevant remaining work of a job in state x	Def. 7.3
$W(\mathbb{Y})$	$\mathbb{Y}$-relevant system work	Def. 7.3
$S(\mathbb{Y})$	$\mathbb{Y}$-relevant size of a generic job	Def. 7.4

Continued on next page

Continued from previous page

Notation	Description	References
$\rho(\mathbb{Y})$	$\mathbb{Y}$-fresh load, i.e. $\lambda \mathbf{E}[S(\mathbb{Y})]$	Def. 7.4
$\lambda_{\text{rcy}}(\mathbb{Y})$	average rate of $\mathbb{Y}$-recyclings	Def. 7.5
$S_{\text{rcy}}(\mathbb{Y})$	$\mathbb{Y}$-relevant remaining work of a job immediately after its $\mathbb{Y}$-recycling	Def. 7.5
$\rho_{\text{rcy}}(\mathbb{Y})$	$\mathbb{Y}$-recycled load, i.e. $\lambda_{\text{rcy}}(\mathbb{Y})\, \mathbf{E}[S_{\text{rcy}}(\mathbb{Y})]$	Def. 7.5
$B(\mathbb{Y}, v)$	$\mathbb{Y}$-relevant busy period started by initial work v	Def. 7.6
$J(\mathbb{Y})$	rate at which $\mathbb{Y}$-relevant jobs are being served	§ 8.3.1
$\mathbf{E}_{\text{rcy}}(\mathbb{Y})[\cdot]$	expectation sampled immediately after a $\mathbb{Y}$-recycling	§ 8.3.1
$W_{\text{min}}(\mathbb{Y})$	minimum possible steady-state $\mathbb{Y}$-relevant system work in an M/G/1 under any policy	Def. 8.6
$Z_x(\mathbb{Y})$	$\mathbb{Y}$-exit state of x, i.e. the random state of a job when it first exits $\mathbb{Y}$, given that it starts at x	Def. 14.4

We use a number of shorthands when discussing the above concepts (Defs. 15.1 and 7.2). While we formally define each of the above for just one of the label-age job model (Ch. 5) or the Markov-process job model (Ch. 14), all of the definitions can be easily adapted to either model.

Bibliography

[1] Samuli Aalto and Urtzi Ayesta. 2006. Mean Delay Analysis of Multi Level Processor Sharing Disciplines. In *Proceedings of IEEE INFOCOM 2006. 25nd Annual Joint Conference of the IEEE Computer and Communications Societies*. IEEE, Barcelona, Spain, 11 pages. doi:10.1109/INFOCOM.2006.262.

[2] Samuli Aalto and Urtzi Ayesta. 2006. On the Nonoptimality of the Foreground-Background Discipline for IMRL Service Times. *Journal of Applied Probability* 43, 2 (June 2006), 523–534. doi:10.1239/jap/1152413739.

[3] Samuli Aalto, Urtzi Ayesta, and Rhonda Righter. 2009. On the Gittins Index in the M/G/1 Queue. *Queueing Systems* 63, 1-4 (Dec. 2009), 437–458. doi:10.1007/s11134-009-9141-x.

[4] Samuli Aalto, Urtzi Ayesta, and Rhonda Righter. 2011. Properties of the Gittins Index with Application to Optimal Scheduling. *Probability in the Engineering and Informational Sciences* 25, 3 (July 2011), 269–288. doi:10.1017/S0269964811000015.

[5] Maryam Akbari-Moghaddam and Douglas G. Down. 2021. SEH: Size Estimate Hedging for Single-Server Queues. In *Quantitative Evaluation of Systems (QEST 2021)*, Alessandro Abate and Andrea Marin (Eds.). Springer, Cham, Switzerland, 168–185. arXiv:2101.00007. doi:10.1007/978-3-030-85172-9_9.

[6] Elene Anton, Urtzi Ayesta, Matthieu Jonckheere, and Ina Maria Verloop. 2021. A Survey of Stability Results for Redundancy Systems. In *Modern Trends in Controlled Stochastic Processes: Theory and Applications, V.III.*, Alexey B. Piunovskiy and Yi Zhang (Eds.). Number 41 in Emergence, Complexity and Computation. Springer, Cham, Switzerland, 266–283. doi:10.1007/978-3-030-76928-4.

[7] Elene Anton, Rhonda Righter, and Ina Maria Verloop. 2022. Efficient Scheduling in Redundancy Systems with General Service Times. arXiv:2206.10164.

[8] Guillaume Aupy, Anne Benoit, Henri Casanova, and Yves Robert. 2016. Checkpointing Strategies for Scheduling Computational Workflows. *International Journal of Networking and Computing* 6, 1 (2016), 2–26. doi:10.15803/ijnc.6.1_2.

[9] Konstantin Avrachenkov, Patrick Brown, and Natalia Osipova. 2009. Optimal Choice of Threshold in Two Level Processor Sharing. *Annals of Operations Research* 170, 1 (Sept. 2009), 21–39. doi:10.1007/s10479-008-0430-2.

[10] Yossi Azar, Stefano Leonardi, and Noam Touitou. 2021. Flow Time Scheduling with Uncertain Processing Time. In *Proceedings of the 53rd Annual ACM SIGACT Symposium on Theory of Computing (STOC 2021)*. ACM, Rome, Italy, 1070–1080. doi:10.1145/3406325.3451023.

[11] Yossi Azar, Stefano Leonardi, and Noam Touitou. 2022. Distortion-Oblivious Algorithms for Minimizing Flow Time. In *Proceedings of the 2022 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2022)*. ACM, Alexandria, VA, 252–274. doi:10.1137/1.9781611977073.13.

[12] Sayan Banerjee, Amarjit Budhiraja, and Amber L. Puha. 2021. Heavy Traffic Scaling Limits for Shortest Remaining Processing Time Queues with Heavy Tailed Processing Time Distributions. arXiv:2003.03655.

[13] Nikhil Bansal. 2005. On the Average Sojourn Time under M/M/1/SRPT. *Operations Research Letters* 33, 2 (March 2005), 195–200. doi:10.1016/j.orl.2004.04.006.

[14] Nikhil Bansal and David Gamarnik. 2006. Handling Load with Less Stress. *Queueing Systems* 54, 1 (Sept. 2006), 45–54. doi:10.1007/s11134-006-8218-z.

[15] Nikhil Bansal and Mor Harchol-Balter. 2001. Analysis of SRPT Scheduling: Investigating Unfairness. *ACM SIGMETRICS Performance Evaluation Review* 29, 1 (June 2001), 279–290. doi:10.1145/384268.378792.

[16] Nikhil Bansal, Bart Kamphorst, and Bert Zwart. 2018. Achievable Performance of Blind Policies in Heavy Traffic. *Mathematics of Operations Research* 43, 3 (Aug. 2018), 949–964. doi:10.1287/moor.2017.0890.

[17] Luca Becchetti and Stefano Leonardi. 2004. Nonclairvoyant Scheduling to Minimize the Total Flow Time on Single and Parallel Machines. *J. ACM* 51, 4 (July 2004), 517–539. doi:10.1145/1008731.1008732.

[18] Dimitris Bertsimas. 1995. The Achievable Region Method in the Optimal Control of Queueing Systems; Formulations, Bounds and Policies. *Queueing Systems* 21, 3 (Sept. 1995), 337–389. doi:10.1007/BF01149167.

[19] Marko A. A. Boon, Rob D. van der Mei, and Erik M. M. Winands. 2011. Applications of Polling Systems. *Surveys in Operations Research and Management Science* 16, 2 (July 2011), 67–82. doi:10.1016/j.sorms.2011.01.001.

[20] Aleksandr A. Borovkov. 1970. Factorization Identities and Properties of the Distribution of the Supremum of Sequential Sums. *Theory of Probability and Its Applications* 15, 3 (Jan. 1970), 359–402. doi:10.1137/1115046.

[21] Sem C. Borst and Onno J. Boxma. 2018. Polling: Past, Present, and Perspective. *TOP* 26, 3 (Oct. 2018), 335–369. doi:10.1007/s11750-018-0484-5.

[22] Onno J. Boxma and Wim P. Groenendijk. 1987. Pseudo-Conservation Laws in Cyclic-Service Systems. *Journal of Applied Probability* 24, 4 (1987), 949–964. doi:10.2307/3214218.

[23] Onno J. Boxma and Bert Zwart. 2007. Tails in Scheduling. *ACM SIGMETRICS Performance Evaluation Review* 34, 4 (March 2007), 13–20. doi:10.1145/1243401.1243406.

[24] Jhelum Chakravorty and Aditya Mahajan. 2014. Multi-Armed Bandits, Gittins Index, and Its Calculation. In *Methods and Applications of Statistics in Clinical Trials*, N. Balakrishnan (Ed.). Wiley, Hoboken, NJ, 416–435. doi:10.1002/9781118596333.ch24.

[25] Jacob W. Cohen. 1973. Some Results on Regular Variation for Distributions in Queueing and Fluctuation Theory. *Journal of Applied Probability* 10, 2 (June 1973), 343–353. doi:10.2307/3212351.

[26] Jacob W. Cohen. 1982. *The Single Server Queue* (second ed.). Number 8 in North-Holland Series in Applied Mathematics and Mechanics. North-Holland, Amsterdam, The Netherlands.

[27] Richard W. Conway, William L. Maxwell, and Louis W. Miller. 1967. *Theory of Scheduling*. Addison-Wesley, Reading, MA.

[28] David R. Cox and Walter L. Smith. 1961. *Queues*. Methuen, London, UK.

[29] Marcus Dacre, Kevin D. Glazebrook, and José Niño-Mora. 1999. The Achievable Region Approach to the Optimal Control of Stochastic Systems. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)* 61, 4 (1999), 747–791.

[30] Matteo Dell'Amico. 2019. Scheduling with Inexact Job Sizes: The Merits of Shortest Processing Time First. arXiv:1907.04824.

[31] Matteo Dell'Amico, Damiano Carra, and Pietro Michiardi. 2016. PSBS: Practical Size-Based Scheduling. *IEEE Trans. Comput.* 65, 7 (July 2016), 2199–2212. doi:10.1109/TC.2015.2468225.

[32] Matteo Dell'Amico, Damiano Carra, Mario Pastorelli, and Pietro Michiardi. 2014. Revisiting Size-Based Scheduling with Estimated Job Sizes. In *22nd IEEE International Symposium on Modelling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS 2014)*. IEEE, Paris, France, 411–420. doi:10.1109/MASCOTS.2014.57.

[33] Tadashi Dohi, Naoto Kaio, and Shunji Osaki. 2000. The Optimal Age-Dependent Checkpoint Strategy for a Stochastic System Subject to General Failure Mode. *J. Math. Anal. Appl.* 249, 1 (Sept. 2000), 80–94. doi:10.1006/jmaa.2000.6939.

[34] Jing Dong and Rouba Ibrahim. 2021. SRPT Scheduling Discipline in Many-Server Queues with Impatient Customers. *Management Science* 67, 12 (Dec. 2021), 7708–7718. doi:10.1287/mnsc.2021.4110.

[35] Ioana Dumitriu, Prasad Tetali, and Peter Winkler. 2003. On Playing Golf with Two Balls. *SIAM Journal on Discrete Mathematics* 16, 4 (Jan. 2003), 604–615. doi:10.1137/S0895480102408341.

[36] Seyed Emadi, Rouba Ibrahim, and Saravanan Kesavan. 2019. Can "Very Noisy" Information Go a Long Way? An Exploratory Analysis of Personalized Scheduling in Service Systems. *Working paper* (Jan. 2019), 40 pages.

[37] Atilla Eryilmaz and R. Srikant. 2012. Asymptotically Tight Steady-State Queue Length Bounds Implied by Drift Conditions. *Queueing Systems* 72, 3 (Dec. 2012), 311–359. doi:10.1007/s11134-012-9305-y.

[38] Val Andrei Fajardo and Steve Drekic. 2017. Waiting Time Distributions in the Preemptive Accumulating Priority Queue. *Methodology and Computing in Applied Probability* 19, 1 (March 2017), 255–284. doi:10.1007/s11009-015-9476-1.

[39] S. W. Fuhrmann. 1984. A Note on the M/G/1 Queue with Server Vacations. *Operations Research* 32, 6 (1984), 1368–1373.

[40] S. W. Fuhrmann and Robert B. Cooper. 1985. Stochastic Decompositions in the M/G/1 Queue with Generalized Vacations. *Operations Research* 33, 5 (Oct. 1985), 1117–1129. doi:10.1287/opre.33.5.1117.

[41] Kristen Gardner, Mor Harchol-Balter, Alan Scheller-Wolf, and Benny Van Houdt. 2017. A Better Model for Job Redundancy: Decoupling Server Slowdown and Job Size. *IEEE/ACM Transactions on Networking* 25, 6 (Dec. 2017), 3353–3367. doi:10.1109/TNET.2017.2744607.

[42] Kristen Gardner, Mor Harchol-Balter, Alan Scheller-Wolf, Mark Velednitsky, and Samuel Zbarsky. 2017. Redundancy-d: The Power of d Choices for Redundancy. *Operations Research* 65, 4 (Aug. 2017), 1078–1094. doi:10.1287/opre.2016.1582.

[43] John C. Gittins. 1989. *Multi-Armed Bandit Allocation Indices* (first ed.). Wiley, Chichester, UK.

[44] John C. Gittins, Kevin D. Glazebrook, and Richard Weber. 2011. *Multi-Armed Bandit Allocation Indices* (second ed.). Wiley, Chichester, UK.

[45] John C. Gittins and David M. Jones. 1974. A Dynamic Allocation Index for the Sequential Design of Experiments. In *Progress in Statistics*, Joseph M. Gani, Károly Sarkadi, and István Vincze (Eds.). Number 9 in Colloquia Mathematica Societatis János Bolyai. North-Holland, Amsterdam, The Netherlands, 241–266.

[46] Kevin D. Glazebrook. 2003. An Analysis of Klimov's Problem with Parallel Servers. *Mathematical Methods of Operations Research* 58, 1 (Sept. 2003), 1–28. doi:10.1007/s001860300278.

[47] Kevin D. Glazebrook and José Niño-Mora. 2001. Parallel Scheduling of Multiclass M/M/m Queues: Approximate and Heavy-Traffic Optimization of Achievable Performance. *Operations Research* 49, 4 (Aug. 2001), 609–623. doi:10.1287/opre.49.4.609.11225.

[48] Carmelita Goerg. 1990. Further Results on a New Combined Strategy Based on the SRPT-principle. *IEEE Transactions on Communications* 38, 5 (May 1990), 568–570. doi:10.1109/26.54967.

[49] Carmelita Goerg and Xuan Huy Pham. 1985. Improving Mean Delay in Data Communication Networks by New Combined Strategies Based on the SRPT-principle. In *Teletraffic Issues in an Advanced Information Society: Proceedings of the Eleventh International Teletraffic Congress (ITC 11)*, Vol. 5. North-Holland, Kyoto, Japan, 931–937.

[50] Isaac Grosof, Ziv Scully, and Mor Harchol-Balter. 2018. SRPT for Multiserver Systems. *Performance Evaluation* 127–128 (Nov. 2018), 154–175. arXiv:1805.07686. doi:10.1016/j.peva.2018.10.001.

[51] Isaac Grosof, Ziv Scully, and Mor Harchol-Balter. 2019. Load Balancing Guardrails: Keeping Your Heavy Traffic on the Road to Low Response Times. *Proceedings of the ACM on Measurement and Analysis of Computing Systems* 3, 2 (June 2019), 42:1–42:31. arXiv:1905.03439. doi:10.1145/3341617.3326157.

[52] Isaac Grosof, Ziv Scully, and Mor Harchol-Balter. 2019. SRPT for Multiserver Systems. *ACM SIGMETRICS Performance Evaluation Review* 46, 2 (Jan. 2019), 9–11. doi:10.1145/3305218.3305223.

[53] Isaac Grosof, Kunhe Yang, Ziv Scully, and Mor Harchol-Balter. 2021. Nudge: Stochastically Improving upon FCFS. *Proceedings of the ACM on Measurement and Analysis of Computing Systems* 5, 2 (June 2021), 21:1–21:29. doi:10.1145/3460088.

[54] Juncheng Gu, Mosharaf Chowdhury, Kang G. Shin, Yibo Zhu, Myeongjae Jeon, Qian Junjie, Hongqiang Liu, and Chuanxiong Guo. 2019. Tiresias: A GPU Cluster Manager for Distributed Deep Learning. In *Proceedings of the 16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 2019)*. USENIX, Boston, MA, 486–500.

[55] Mor Harchol-Balter. 2013. *Performance Modeling and Design of Computer Systems: Queueing Theory in Action*. Cambridge University Press, Cambridge, UK.

[56] Mor Harchol-Balter, Takayuki Osogami, Alan Scheller-Wolf, and Adam Wierman. 2005. Multi-Server Queueing Systems with Multiple Priority Classes. *Queueing Systems* 51, 3 (Dec. 2005), 331–360. doi:10.1007/s11134-005-2898-7.

[57] Mor Harchol-Balter, Bianca Schroeder, Nikhil Bansal, and Mukesh Agrawal. 2003. Size-Based Scheduling to Improve Web Performance. *ACM Transactions on Computer Systems* 21, 2 (May 2003), 207–233. doi:10.1145/762483.762486.

[58] Yige Hong and Weina Wang. 2022. Sharp Waiting-Time Bounds for Multiserver Jobs. arXiv:2109.05343.

[59] Daniela Hurtado-Lange and Siva Theja Maguluri. 2020. Transform Methods for Heavy-Traffic Analysis. *Stochastic Systems* 10, 4 (Dec. 2020), 275–309. doi:10.1287/stsy.2019.0056.

[60] Daniela Hurtado-Lange, Sushil Mahavir Varma, and Siva Theja Maguluri. 2022. Logarithmic Heavy Traffic Error Bounds in Generalized Switch and Load Balancing Systems. *Journal of Applied Probability* (June 2022), 18. doi:10.1017/jpr.2021.82.

[61] Esa Hyytiä, Samuli Aalto, and Aleksi Penttinen. 2012. Minimizing Slowdown in Heterogeneous Size-Aware Dispatching Systems. *ACM SIGMETRICS Performance Evaluation Review* 40, 1 (June 2012), 29–40. doi:10.1145/2318857.2254763.

[62] Esa Hyytiä, Aleksi Penttinen, and Samuli Aalto. 2012. Size- and State-Aware Dispatching Problem with Queue-Specific Job Sizes. *European Journal of Operational Research* 217, 2 (March 2012), 357–370. doi:10.1016/j.ejor.2011.09.029.

[63] Bala Kalyanasundaram and Kirk R. Pruhs. 2003. Minimizing Flow Time Nonclairvoyantly. *J. ACM* 50, 4 (July 2003), 551–567. doi:10.1145/792538.792545.

[64] Bart Kamphorst and Bert Zwart. 2020. Heavy-Traffic Analysis of Sojourn Time under the Foreground–Background Scheduling Policy. *Stochastic Systems* 10, 1 (March 2020), 1–28. doi:10.1287/stsy.2019.0036.

[65] Frank P. Kelly. 1976. The Departure Process from a Queueing System. *Mathematical Proceedings of the Cambridge Philosophical Society* 80, 2 (Sept. 1976), 283–285. doi:10.1017/S0305004100052919.

[66] Frank P. Kelly. 2011. *Reversibility and Stochastic Networks* (revised ed.). Cambridge University Press, Cambridge, UK.

[67] David G. Kendall. 1953. Stochastic Processes Occurring in the Theory of Queues and Their Analysis by the Method of the Imbedded Markov Chain. *Annals of Mathematical Statistics* 24, 3 (1953), 338–354.

[68] Harry Kesten and Johannes Th. Runnenburg. 1957. Priority in Waiting Line Problems. I. *Indagationes Mathematicae* 60 (1957), 312–324. doi:10.1016/S1385-7258(57)50042-5.

[69] Harry Kesten and Johannes Th. Runnenburg. 1957. Priority in Waiting Line Problems. II. *Indagationes Mathematicae* 60 (1957), 325–336. doi:10.1016/S1385-7258(57)50043-7.

[70] Aleksandr Khintchine. 1932. Mathematische Theorie der stationären Reihe [Mathematical theory of a stationary queue]. *Matematicheskii Sbornik* 39, 4 (1932), 73–84.

[71] Toshikazu Kimura. 1983. Diffusion Approximation for an M/G/m Queue. *Operations Research* 31, 2 (April 1983), 304–321. doi:10.1287/opre.31.2.304.

[72] John F. C. Kingman. 1962. On Queues in Which Customers Are Served in Random Order. *Mathematical Proceedings of the Cambridge Philosophical Society* 58, 1 (Jan. 1962), 79–91. doi:10.1017/S0305004100036239.

[73] John F. C. Kingman. 2009. The First Erlang Century—and the Next. *Queueing Systems* 63, 1 (Nov. 2009), 3. doi:10.1007/s11134-009-9147-4.

[74] Leonard Kleinrock. 1976. *Queueing Systems, Volume 2: Computer Applications*. Wiley, New York, NY.

[75] Leonard Kleinrock and Richard R. Muntz. 1972. Processor Sharing Queueing Models of Mixed Scheduling Disciplines for Time Shared System. *J. ACM* 19, 3 (July 1972), 464–482. doi:10.1145/321707.321717.

[76] Gennadi P. Klimov. 1974. Time-Sharing Service Systems. I. *Theory of Probability & Its Applications* 19, 3 (1974), 532–551. doi:10.1137/1119060.

[77] Julian Köllerström. 1974. Heavy Traffic Theory for Queues with Several Servers. I. *Journal of Applied Probability* 11, 3 (Sept. 1974), 544–552. doi:10.2307/3212698.

[78] Julian Köllerström. 1979. Heavy Traffic Theory for Queues with Several Servers. II. *Journal of Applied Probability* 16, 2 (June 1979), 393–401. doi:10.2307/3212906.

[79] Tze Leung Lai and Zhiliang Ying. 1988. Open Bandit Processes and Optimal Scheduling of Queueing Networks. *Advances in Applied Probability* 20, 2 (1988), 447–472. doi:10.2307/1427399.

[80] Jan Karel Lenstra and David B. Shmoys. 2020. Elements of Scheduling. arXiv:2001.06005.

[81] Stefano Leonardi and Danny Raz. 2007. Approximating Total Flow Time on Parallel Machines. *J. Comput. System Sci.* 73, 6 (Sept. 2007), 875–891. doi:10.1016/j.jcss.2006.10.018.

[82] Yuan Li and David A. Goldberg. 2017. Simple and Explicit Bounds for Multi-Server Queues with Universal $1/(1-\rho)$ Scaling. arXiv:1706.04628.

[83] Minghong Lin, Adam Wierman, and Bert Zwart. 2011. Heavy-Traffic Analysis of Mean Response Time under Shortest Remaining Processing Time. *Performance Evaluation* 68, 10 (Oct. 2011), 955–966. doi:10.1016/j.peva.2011.06.001.

[84] John D. C. Little. 2011. Little's Law as Viewed on Its 50th Anniversary. *Operations Research* 59, 3 (June 2011), 536–549. doi:10.1287/opre.1110.0940.

[85] Siva Theja Maguluri and R. Srikant. 2016. Heavy Traffic Queue Length Behavior in a Switch under the MaxWeight Algorithm. *Stochastic Systems* 6, 1 (June 2016), 211–250. doi:10.1287/15-SSY193.

[86] Andrea Marin, Sabina Rossi, and Carlo Zen. 2020. Size-Based Scheduling for TCP Flows: Implementation and Performance Evaluation. *Computer Networks* 183 (Dec. 2020), 107574:1–107574:15. doi:10.1016/j.comnet.2020.107574.

[87] Nicole Megow and Tjark Vredeveld. 2014. A Tight 2-Approximation for Preemptive Stochastic Scheduling. *Mathematics of Operations Research* 39, 4 (Nov. 2014), 1297–1310. doi:10.1287/moor.2014.0653.

[88] Paul Milgrom and Ilya Segal. 2002. Envelope Theorems for Arbitrary Choice Sets. *Econometrica* 70, 2 (March 2002), 583–601. doi:10.1111/1468-0262.00296.

[89] Rupert G. Miller, Jr. 1960. Priority Queues. *Annals of Mathematical Statistics* 31, 1 (March 1960), 86–103. doi:10.1214/aoms/1177705990.

[90] Ante Mimica. 2016. Exponential Decay of Measures and Tauberian Theorems. *J. Math. Anal. Appl.* 440, 1 (Aug. 2016), 266–285. doi:10.1016/j.jmaa.2016.03.042.

[91] Isi Mitrani and Peter J. B. King. 1981. Multiprocessor Systems with Preemptive Priorities. *Performance Evaluation* 1, 2 (May 1981), 118–125. doi:10.1016/0166-5316(81)90014-6.

[92] Michael Mitzenmacher. 2020. Scheduling with Predictions and the Price of Misprediction. In *11th Innovations in Theoretical Computer Science Conference (ITCS 2020) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 151)*, Thomas Vidick (Ed.). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Seattle, WA, 14:1–14:18. doi:10.4230/LIPICS.ITCS.2020.14.

[93] Michael Mitzenmacher and Matteo Dell'Amico. 2020. The Supermarket Model with Known and Predicted Service Times. arXiv:1905.12155.

[94] Masakiyo Miyazawa. 1994. Decomposition Formulas for Single Server Queues with Vacations : A Unified Approach by the Rate Conservation Law. *Communications in Statistics. Stochastic Models* 10, 2 (Jan. 1994), 389–413. doi:10.1080/15326349408807301.

[95] Masakiyo Miyazawa. 1994. Rate Conservation Laws: A Survey. *Queueing Systems* 15, 1 (March 1994), 1–58. doi:10.1007/BF01189231.

[96] Behnam Montazeri, Yilong Li, Mohammad Alizadeh, and John Ousterhout. 2018. Homa: A Receiver-Driven Low-Latency Transport Protocol Using Network Priorities. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication (SIGCOMM 2018)*. ACM, Budapest, Hungary, 221–235. doi:10.1145/3230543.3230564.

[97] Kenji Nakagawa. 2005. Tail Probability of Random Variable and Laplace Transform. *Applicable Analysis* 84, 5 (May 2005), 499–522. doi:10.1080/00036810410001724436.

[98] Kenji Nakagawa. 2007. Application of Tauberian Theorem to the Exponential Decay of the Tail Probability of a Random Variable. *IEEE Transactions on Information Theory* 53, 9 (Sept. 2007), 3239–3249. doi:10.1109/TIT.2007.903114.

[99] Victor F. Nicola and J. M. van Spanje. 1990. Comparative Analysis of Different Models of Checkpointing and Recovery. *IEEE Transactions on Software Engineering* 16, 8 (Aug. 1990), 807–821. doi:10.1109/32.57620.

[100] Rudesindo Núñez-Queija. 2001. Note on the GI/GI/1 Queue with LCFS-PR Observed at Arbitrary Times. *Probability in the Engineering and Informational Sciences* 15, 2 (April 2001), 179–187. doi:10.1017/S0269964801152034.

[101] Rudesindo Núñez-Queija. 2002. Queues with Equally Heavy Sojourn Time and Service Requirement Distributions. *Annals of Operations Research* 113, 1/4 (July 2002), 101–117. doi:10.1023/A:1020905810996.

[102] Misja Nuyens and Adam Wierman. 2008. The Foreground–Background Queue: A Survey. *Performance Evaluation* 65, 3-4 (March 2008), 286–307. doi:10.1016/j.peva.2007.06.028.

[103] Misja Nuyens, Adam Wierman, and Bert Zwart. 2008. Preventing Large Sojourn Times Using SMART Scheduling. *Operations Research* 56, 1 (Feb. 2008), 88–101. doi:10.1287/opre.1070.0504.

[104] Misja Nuyens and Bert Zwart. 2006. A Large-Deviations Analysis of the GI/GI/1 SRPT Queue. *Queueing Systems* 54, 2 (Oct. 2006), 85–97. doi:10.1007/s11134-006-8767-1.

[105] Natalia Osipova, Urtzi Ayesta, and Konstantin Avrachenkov. 2009. Optimal Policy for Multi-Class Scheduling in a Single Server Queue. In *21st International Teletraffic Congress (ITC 21)*. IEEE, Paris, France, 1–8.

[106] Goran Peskir and Albert N. Shiryaev. 2006. *Optimal Stopping and Free-Boundary Problems*. Birkhäuser Verlag, Basel. doi:10.1007/978-3-7643-7390-0.

[107] Michael Pinedo. 2016. *Scheduling: Theory, Algorithms, and Systems* (fifth ed.). Springer, Cham, Switzerland.

[108] Felix Pollaczek. 1930. Über eine Aufgabe der Wahrscheinlichkeitstheorie. II [On a problem in probability theory. II]. *Mathematische Zeitschrift* 32, 1 (Dec. 1930), 729–750. doi:10.1007/BF01194663.

[109] Felix Pollaczek. 1930. Über eine Aufgabe der Wahrscheinlichkeitstheorie. I [On a problem in probability theory. I]. *Mathematische Zeitschrift* 32, 1 (Dec. 1930), 64–100. doi:10.1007/BF01194620.

[110] Manish Purohit, Zoya Svitkina, and Ravi Kumar. 2018. Improving Online Algorithms via ML Predictions. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems (NeurIPS 2018)*, S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett (Eds.), Vol. 31. Curran Associates, Inc., Montréal, Canada, 9684–9693.

[111] Rhonda Righter, J. George Shanthikumar, and Genji Yamazaki. 1990. On Extremal Service Disciplines in Single-Stage Queueing Systems. *Journal of Applied Probability* 27, 2 (June 1990), 409–416. doi:10.2307/3214660.

[112] Sheldon M. Ross. 1996. *Stochastic Processes* (second ed.). Wiley, New York.

[113] Alan Scheller-Wolf. 2003. Necessary and Sufficient Conditions for Delay Moments in FIFO Multiserver Queues with an Application Comparing s Slow Servers with One Fast One. *Operations Research* 51, 5 (Oct. 2003), 748–758. doi:10.1287/opre.51.5.748.16759.

[114] Alan Scheller-Wolf and Rein Vesilo. 2006. Structural Interpretation and Derivation of Necessary and Sufficient Conditions for Delay Moments in FIFO Multiserver Queues. *Queueing Systems* 54, 3 (Nov. 2006), 221–232. doi:10.1007/s11134-006-0068-1.

[115] Linus E. Schrage. 1967. The Queue M/G/1 with Feedback to Lower Priority Queues. *Management Science* 13, 7 (March 1967), 466–474. doi:10.1287/mnsc.13.7.466.

[116] Linus E. Schrage. 1968. A Proof of the Optimality of the Shortest Remaining Processing Time Discipline. *Operations Research* 16, 3 (June 1968), 687–690. doi:10.1287/opre.16.3.687.

[117] Linus E. Schrage and Louis W. Miller. 1966. The Queue M/G/1 with the Shortest Remaining Processing Time Discipline. *Operations Research* 14, 4 (Aug. 1966), 670–684. doi:10.1287/opre.14.4.670.

[118] Ziv Scully, Isaac Grosof, and Mor Harchol-Balter. 2020. The Gittins Policy Is Nearly Optimal in the M/G/k under Extremely General Conditions. *Proceedings of the ACM on Measurement and Analysis of Computing Systems* 4, 3 (Nov. 2020), 43:1–43:29. doi:10.1145/3428328.

[119] Ziv Scully, Isaac Grosof, and Mor Harchol-Balter. 2021. Optimal Multiserver Scheduling with Unknown Job Sizes in Heavy Traffic. *Performance Evaluation* 145 (Jan. 2021), 102150:1–102150:31. arXiv:2003.13232. doi:10.1016/j.peva.2020.102150.

[120] Ziv Scully, Isaac Grosof, and Michael Mitzenmacher. 2022. Uniform Bounds for Scheduling with Job Size Estimates. In *13th Innovations in Theoretical Computer Science Conference (ITCS 2022) (Leibniz International Proceedings in Informatics (LIPIcs))*. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Berkeley, CA, 41:1–41:30. arXiv:2110.00633. doi:10.4230/LIPIcs.ITCS.2022.41.

[121] Ziv Scully and Mor Harchol-Balter. 2018. SOAP Bubbles: Robust Scheduling under Adversarial Noise. In *56th Annual Allerton Conference on Communication, Control, and Computing*. IEEE, Monticello, IL, 144–154. doi:10.1109/ALLERTON.2018.8635963.

[122] Ziv Scully and Mor Harchol-Balter. 2021. The Gittins Policy in the M/G/1 Queue. In *19th International Symposium on Modeling and Optimization in Mobile, Ad Hoc, and Wireless Networks (WiOpt 2021)*. IFIP, Philadelphia, PA, 248–255. doi:10.23919/WiOpt52861.2021.9589051.

[123] Ziv Scully and Mor Harchol-Balter. 2021. How to Schedule Near-Optimally under Real-World Constraints. arXiv:2110.11579.

[124] Ziv Scully, Mor Harchol-Balter, and Alan Scheller-Wolf. 2018. SOAP: One Clean Analysis of All Age-Based Scheduling Policies. *Proceedings of the ACM on Measurement and Analysis of Computing Systems* 2, 1 (April 2018), 16:1–16:30. arXiv:1712.00790. doi:10.1145/3179419.

[125] Ziv Scully, Mor Harchol-Balter, and Alan Scheller-Wolf. 2020. Simple Near-Optimal Scheduling for the M/G/1. *Proceedings of the ACM on Measurement and Analysis of Computing Systems* 4, 1 (May 2020), 11:1–11:29. arXiv:1907.10792. doi:10.1145/3379477.

[126] Ziv Scully and Lucas van Kreveld. 2021. When Does the Gittins Policy Have Asymptotically Optimal Response Time Tail? arXiv:2110.06326.

[127] Ziv Scully, Lucas van Kreveld, Onno J. Boxma, Jan-Pieter Dorsman, and Adam Wierman. 2020. Characterizing Policies with Optimal Response Time Tails under Heavy-Tailed Job Sizes. *Proceedings of the ACM on Measurement and Analysis of Computing Systems* 4, 2 (June 2020), 30:1–30:33. doi:10.1145/3392148.

[128] Kenneth C. Sevcik. 1971. *The Use of Service Time Distributions in Scheduling*. Ph. D. Dissertation. University of Chicago, Chicago, IL. doi:10.2172/4710384.

[129] Albert N. Shiryaev. 2008. *Optimal Stopping Rules* (second ed.). Number 8 in Applications of Mathematics. Springer, Berlin, Germany.

[130] Andrei Sleptchenko, Aart van Harten, and Matthieu van der Heijden. 2005. An Exact Solution for the State Probabilities of the Multi-Class, Multi-Server Queue with Preemptive Priorities. *Queueing Systems* 50, 1 (May 2005), 81–107. doi:10.1007/s11134-005-0359-y.

[131] David A. Stanford, Peter Taylor, and Ilze Ziedins. 2014. Waiting Time Distributions in the Accumulating Priority Queue. *Queueing Systems* 77, 3 (July 2014), 297–330. doi:10.1007/s11134-013-9382-6.

[132] Alexander L. Stolyar. 2004. Maxweight Scheduling in a Generalized Switch: State Space Collapse and Workload Minimization in Heavy Traffic. *Annals of Applied Probability* 14, 1 (2004), 1–53.

[133] Alexander L. Stolyar and Kavita Ramanan. 2001. Largest Weighted Delay First Scheduling: Large Deviations and Optimality. *Annals of Applied Probability* 11, 1 (2001), 1–48.

[134] Lajos Takács. 1963. Delay Distributions for One Line with Poisson Input, General Holding Times, and Various Orders of Service. *Bell System Technical Journal* 42, 2 (March 1963), 487–503. doi:10.1002/j.1538-7305.1963.tb00509.x.

[135] Lajos Takács. 1964. Priority Queues. *Operations Research* 12, 1 (Feb. 1964), 63–74. doi:10.1287/opre.12.1.63.

[136] G. von Olivier. 1972. Kostenminimale Prioritäten in Wartesystemen vom Typ M/G/1 [Cost-minimum priorities in queueing systems of type M/G/1]. *Elektronische Rechenanlagen* 14, 6 (Dec. 1972), 262–271. doi:10.1524/itit.1972.14.16.262.

[137] Weina Wang, Mor Harchol-Balter, Haotian Jiang, Alan Scheller-Wolf, and R. Srikant. 2019. Delay Asymptotics and Bounds for Multitask Parallel Jobs. *Queueing Systems* 91, 3 (April 2019), 207–239. doi:10.1007/s11134-018-09597-5.

[138] Ward Whitt. 2000. The Impact of a Heavy-Tailed Service-Time Distribution upon the M/GI/s Waiting-Time Distribution. *Queueing Systems* 36, 1 (Nov. 2000), 71–87. doi:10.1023/A:1019143505968.

[139] Ward Whitt. 2004. A Diffusion Approximation for the G/GI/n/m Queue. *Operations Research* 52, 6 (2004), 922–941.

[140] Peter Whittle. 1980. Multi-Armed Bandits and the Gittins Index. *Journal of the Royal Statistical Society: Series B (Methodological)* 42, 2 (1980), 143–149.

[141] Peter Whittle. 2005. Tax Problems in the Undiscounted Case. *Journal of Applied Probability* 42, 3 (Sept. 2005), 754–765. doi:10.1239/jap/1127322025.

[142] Adam Wierman. 2007. Fairness and Classifications. *ACM SIGMETRICS Performance Evaluation Review* 34, 4 (March 2007), 4–12. doi:10.1145/1243401.1243405.

[143] Adam Wierman. 2007. *Scheduling for Today's Computer Systems: Bridging Theory and Practice.* Ph. D. Dissertation. Carnegie Mellon University, Pittsburgh, PA.

[144] Adam Wierman and Mor Harchol-Balter. 2003. Classifying Scheduling Policies with Respect to Unfairness in an M/GI/1. *ACM SIGMETRICS Performance Evaluation Review* 31, 1 (June 2003), 238–249. doi:10.1145/885651.781057.

[145] Adam Wierman and Mor Harchol-Balter. 2005. Classifying Scheduling Policies with Respect to Higher Moments of Conditional Response Time. *ACM SIGMETRICS Performance Evaluation Review* 33, 1 (June 2005), 229–240. doi:10.1145/1071690.1064238.

[146] Adam Wierman, Mor Harchol-Balter, and Takayuki Osogami. 2005. Nearly Insensitive Bounds on SMART Scheduling. *ACM SIGMETRICS Performance Evaluation Review* 33, 1 (June 2005), 205–216. doi:10.1145/1071690.1064236.

[147] Adam Wierman and Misja Nuyens. 2008. Scheduling despite Inexact Job-Size Information. *ACM SIGMETRICS Performance Evaluation Review* 36, 1 (June 2008), 25–36. doi:10.1145/1384529.1375461.

[148] Adam Wierman, Erik M. M. Winands, and Onno J. Boxma. 2007. Scheduling in Polling Systems. *Performance Evaluation* 64, 9-12 (Oct. 2007), 1009–1028. doi:10.1016/j.peva.2007.06.015.

[149] Adam Wierman and Bert Zwart. 2012. Is Tail-Optimal Scheduling Possible? *Operations Research* 60, 5 (Oct. 2012), 1249–1257. doi:10.1287/opre.1120.1086.

[150] Ronald W. Wolff. 1982. Poisson Arrivals See Time Averages. *Operations Research* 30, 2 (1982), 223–231.

[151] Qiaomin Xie and Yi Lu. 2015. Priority Algorithm for Near-Data Scheduling: Throughput and Heavy-Traffic Optimality. In *2015 IEEE Conference on Computer Communications (INFOCOM 2015)*. IEEE, Kowloon, Hong Kong, 963–972. doi:10.1109/INFOCOM.2015.7218468.

[152] David D. Yao. 1985. Refining the Diffusion Approximation for the M/G/m Queue. *Operations Research* 33, 6 (Dec. 1985), 1266–1277. doi:10.1287/opre.33.6.1266.

[153] Sergey F. Yashkov. 1987. Processor-Sharing Queues: Some Progress in Analysis. *Queueing Systems* 2, 1 (March 1987), 1–17. doi:10.1007/BF01182931.

[154] Sergey F. Yashkov and A. S. Yashkova. 2007. Processor Sharing: A Survey of the Mathematical Theory. *Automation and Remote Control* 68, 9 (Sept. 2007), 1662–1731. doi:10.1134/S0005117907090202.