THESIS

A Principled Approach to Parallel Job Scheduling

Benjamin Berg

July 27th, 2022

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
Mor Harchol-Balter, CMU (Chair)
Gregory R. Ganger, CMU
Christos Kozyrakis, Stanford
Benjamin Moseley, CMU
Weina Wang, CMU

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

Keywords: Scheduling, Performance Modeling

Dedication

Abstract

A wide range of modern computer systems process workloads composed of *parallelizable jobs*. Data centers, supercomputers, machine learning clusters, distributed computing frameworks, and databases all process jobs which are designed to be parallelized across multiple servers or cores. A job will receive some *speedup* from being parallelized across additional servers or cores, allowing the job to complete more quickly. However, jobs generally cannot be perfectly parallelized, and receive diminishing returns from being allocated additional cores. Hence, given a fixed number of cores, it is not obvious how to allocate cores across a set of jobs in order to reduce the response times of the jobs — the times from when each job arrives to the system until it is completed. While this question has been considered by the worst-case scheduling community, existing theoretical results are hampered by strong lower bounds on worst-case performance and tend to suggest policies which do not work well in practice. Meanwhile, state-of-the-art systems employ simple heuristic-based policies that leave significant room for improvement. The goal of this thesis is to develop and analyze policies for scheduling parallelizable jobs which perform well both in theory *and* in practice.

Our approach in developing new scheduling policies for parallelizable jobs is threefold. First, we develop new stochastic models of parallelizable jobs running in a multicore system. Second, we analyze these new models using the tools of stochastic performance modeling, showing that a stochastic style of analysis emits scheduling policies which provably outperform the policies suggested by the worst-case literature. Finally, we validate our theoretical models through simulation to show that the scheduling policies we derive work well in practice. We consider the case where job sizes are completely unknown to the system, the case where job sizes are perfectly known to the system, and the case where some jobs are known to be larger than others on average. Similarly, we consider the case where all jobs are assumed to have the same level of parallelizability, the case where some jobs are more parallelizable than others, and the case where a job's parallelizability can change over time.

Acknowledgments

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Many problems in the design of modern computer systems are, fundamentally, scheduling problems where a system must decide how to effectively share a set of hardware resources between many simultaneous users. Since the early days of computing [86], system designers have been concerned with mechanisms to enable users to take turn accessing scarce computational resources such as CPU time, memory, and network bandwidth. The goal of enabling this resource sharing spawned entire subfields of systems research from operating systems to networking to computer architecture.

Along with the development of new systems came a host of interesting theoretical questions about what *scheduling policies* should be used to decide how resources were divided among users. However, for many years, the need to implement improved scheduling policies in computer systems was masked by a rapid improvement in the underlying hardware. As has been well-documented, we are reaching the physical limits of the hardware improvements that have driven progress over the last 50 years [97]. As a result, the development of algorithms derived from new theoretical models is poised to become a major source of performance improvement in the coming years [59]. As users continue to demand access to faster and cheaper computer systems, the need to develop and deploy performant scheduling policies is becoming more acute.

Developing new scheduling policies for computer systems is complicated by the fact that, in order to continue to offer performance improvements, modern systems have become quite complex. In particular, as the ability to make a single CPU core faster has diminished, system designers have increasingly turned to multicore and multi-server architectures which are built to exploit parallelism in order to reduce job processing times. While systems which run multiple jobs simultaneously in parallel have been well-studied, jobs in classical models such as the M/G/k have assumed that each job runs on a single core. In modern computer systems, however, an individual job can often be completed more quickly if it is parallelized across multiple cores. As we will see, designing policies for scheduling these *parallelizable jobs* is non-trivial, but can dramatically improve the performance of modern computer systems.

1.1 The Importance of Scheduling Parallelizable Jobs

Parallelizable workloads are ubiquitous and appear across a diverse array of modern computer systems. Data centers, supercomputers, machine learning clusters, distributed computing frameworks, and databases all process jobs designed to be parallelized across many servers or cores. Systems which process parallelizable jobs must employ some type of scheduling policy which decides how servers or cores are allocated among the jobs in the system at every moment in time. As it turns out, the choice of which scheduling policy to use has the potential to impact system performance dramatically. In particular, if we measure the *response time* of each job to be the time from when a job arrives to the system until the job is completed, the choice of scheduling policy can change the mean response time across jobs by orders of magnitude. In fact, choosing the wrong policy for scheduling parallelizable jobs can even push the system to become overloaded and unstable, causing the mean response time across jobs to be infinite.

The reason that choosing a good policy for scheduling parallelizable jobs is both particularly important *and* particularly difficult is because jobs are not typically perfectly parallelizable. Ideally, allocating additional cores to a parallelizable job would proportionally reduce the time required to complete the job. That is, ideally a job would run twice as fast on two cores as it does on one core. However, parallelism is rarely available for free. Parallelizing a computation can incur overheads in hardware, such as the overhead of maintaining a cache-coherent shared memory system [65]. Parallelism can also incur overheads in software, such as the overhead of creating and multiplexing between threads of execution. Aside from these overheads, there are algorithmic limits to how some computations can be parallelized, even under ideal circumstances [19].

As a result of these limitations, jobs typically receive a *diminishing marginal benefit* from being allocated additional cores. While a job may run faster on two cores than it does on one core, it generally will not run twice as fast. Furthermore, many jobs reach a point where they no longer benefit appreciably from being allocated additional cores, and allocating cores to a job past this point is equivalent to leaving those cores idle. Hence, the wrong choice of scheduling policy can potentially result in a gross misuse of system resources, and can cause a dramatic increase in job response times.

The goal of this thesis is to develop and analyze scheduling policies which explicitly account for the limited parallelizability of the jobs they schedule. In particular, given a set of n cores, we aim to design scheduling policies that allocate cores to jobs in order to reduce job response times.

1.2 The State-of-the-Art In Scheduling Parallelizable Jobs

Many real-world systems have faced the challenge of scheduling parallelizable jobs (see Section 3.1). These systems have historically relied on a variety of greedy heuristic policies to try and reduce job response times[23, 67, 82, 102]. While these heuristic-based approaches are often simple to implement, the policies used provide no provable performance guarantees, often require significant parameter tuning, and can be far from optimal. As we will show in this thesis, these policies can be significantly improved by more formally understanding the problem of scheduling parallelizable jobs.

On the theoretical side, the worst-case theoretical computer science (TCS) community has been working on the problem of scheduling parallelizable jobs for the past 20 years (see Section 3.2). In the online setting where jobs arrive over time, strong lower bounds have been obtained on the achievable worst-case mean response time. In particular, the TCS community has shown that it is impossible to design a scheduling policy which performs within a constant factor of the optimal policy in the worst case [60]. Because it is so difficult to perform anywhere close to optimally in the worst case, simple policies such as dividing cores equally across jobs can be shown to meet the lower bounds on achievable worst-case performance. However, these simple policies often perform poorly in practice.

This thesis aims to address this disconnect between state-of-the-art systems and the theoretical community when it comes to the problem of scheduling parallelizable jobs.

1.3 Thesis Statement

Modern systems which process parallelizable jobs employ heuristic-based scheduling policies that are far from optimal, and stand to be significantly improved by a deeper theoretical understanding of scheduling and resource allocation problems. However, the prior worst-case theoretical research on scheduling parallelizable jobs has produced scheduling policies that are theoretically interesting, but can perform poorly in practice compared to state-of-the-art systems. Using a stochastic performance modeling approach, we can develop new policies for scheduling parallelizable jobs which are both practical *and* grounded in a formal understanding of the problem. In other words, we can develop scheduling policies that provably perform well in practice.

1.4 Tradeoffs in Scheduling Parallelizable Jobs

By employing a stochastic performance modeling approach, this thesis will show that the problem of scheduling parallelizable jobs comes down to a set of *tradeoffs* that must be balanced differently depending on the exact setting being considered. Understanding these tradeoffs at a high level is helpful in understanding the contributions of this thesis, and we will refer back to these tradeoffs throughout the subsequent chapters in sections marked **Scheduling Tradeoffs**. Figure 1.1 illustrates the tradeoffs that our scheduling policies will have to balance.

The tradeoffs in Figure 1.1 arise from the fact that jobs generally are not perfectly parallelizable. As described above, when jobs receive a diminishing marginal benefit from being allocated additional cores, allocating too many cores to an individual job can severely impact system performance. To avoid these damaging scenarios, we can monitor the *system efficiency* of a particular allocation, a measurement of how much benefit the jobs in the system are receiving from their current allocations. Intuitively, it seems that we should aim for scheduling policies which *maximize system efficiency*. These policies are getting the most "bang for their buck", and are successfully avoiding the potential sources of overhead associated with parallelism. We will see that policies generally maximize system efficiency by sharing the available system resources amongst all the

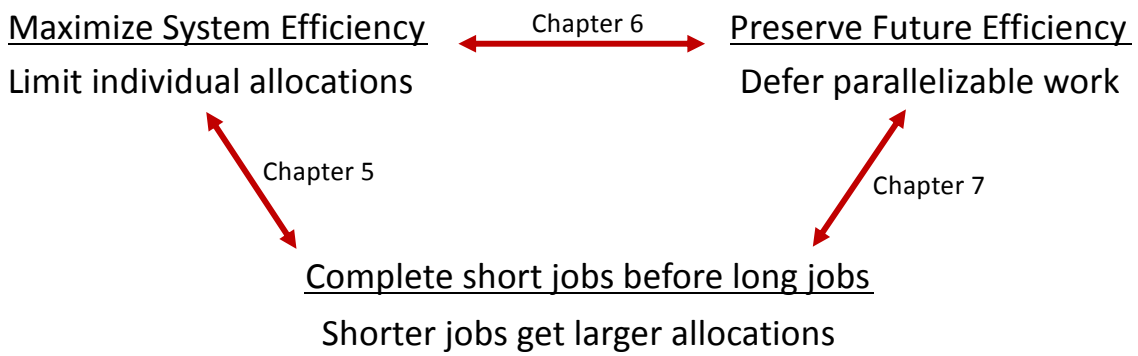


Figure 1.1: A map of the goals and tradeoffs that arise when scheduling parallelizable jobs. A scheduling policy should limit allocations to individual jobs in order to maximize system efficiency (Chapter 4). Scheduling policies must balance a tradeoff between maximizing system efficiency and favoring short jobs (Chapter 5). When some jobs are more parallelizable than others, a scheduling policy must balance the present system efficiency with the future system efficiency (Chapter 6). In some cases (Chapter 7), a scheduling policy may have to simultaneously consider a job's size *and* its level of parallelizability.

jobs in the system, and ensuring that no individual job receives too many cores. While maximizing system efficiency is one intuitive goal, it turns out than an optimal scheduling policy must balance *multiple* competing goals.

For instance, some jobs may only need to run for *short* periods of time, while other jobs may need to run for *long* periods of time in order to complete. A classic result in scheduling theory which is noted by both the worst-case community and the performance modeling community is that overall mean response time can be reduced by *completing short jobs before long jobs*. One might therefore think to employ a scheduling policy which gives larger allocations to short jobs than to long jobs. Note, however, that this plan stands in stark opposition to the goal of limiting the number of cores allocated to an individual job in order maximize system efficiency. As a result, a good scheduling policy must balance a tradeoff between the benefits of favoring short jobs and the cost of the decreasing system efficiency by giving larger allocations to shorter jobs.

We must also account for cases where different jobs have different levels of parallelizability. For example, some jobs might be highly parallelizable, making nearly perfect use of additional cores, while other jobs may receive little or no benefit from being parallelized. It seems natural that a scheduling policy should favor the more parallelizable jobs, since these jobs will benefit more from their larger allocations. However, this type of policy has a hidden cost. A policy which favors the more parallelizable jobs will quickly complete the more parallelizable jobs. The system will then be left only with jobs which do not benefit from being parallelized. In these future states, even if we choose an allocation that maximizes system efficiency, we would have preferred to have more parallelizable jobs in the system in order to more efficiently use the available cores. Hence, it can be beneficial for a scheduling policy to *defer parallelizable work* in order to avoid running out of the more parallelizable jobs in the future. A good scheduling policy must balance a tradeoff between the maximizing the *present* system efficiency and maximizing the *future* system efficiency.

Finally, we are sometimes forced to consider cases where all of these goals compete simultaneously. For example, should a policy favor short jobs, even if they are less parallelizable? And how does the benefit of deferring parallelizable work compare with benefit of favoring a short job? These are the types of questions addressed by the contributions of this thesis.

1.5 Contributions

This thesis address the problem of scheduling parallelizable jobs via a three-pronged approach:

1. The theoretical models and style of analysis employed by the worst-case scheduling community has advocated the use of scheduling policies which do not perform well in real-world systems. We therefore begin by developing new models of parallelizable jobs in multicore systems using the tools of stochastic performance modeling.
2. Using our newly developed models, we derive and analyze new scheduling policies which provably result in reduced job response times.
3. Finally, we validate our theoretical models through simulation and, when necessary, we design new heuristic policies based on our theoretical results. Our goal here is to demonstrate that we can create policies inspired by our theoretical models that represent an improvement over the heuristic policies currently used in the systems community.

The contributions of this thesis are as follows:

We develop the first stochastic model of parallelizable jobs running in a multicore machine (Chapter 2).

- To describe the running time of each parallelizable job, we define a job's *inherent size*, the time it takes the job to run on a single core.
- We also define a job's *speedup function*, $s(k)$, to be a sublinear, increasing, concave function which tells us how many times faster a job would complete if run on k cores as opposed to a single core.

Using our new model, we begin by considering the case where the scheduler has very little information about the jobs it is running (Chapter 4). Specifically, we assume that job sizes are unknown to the system and exponentially distributed, and that all jobs follow the same speedup function. Hence, the scheduler cannot differentiate between jobs on the basis of their remaining sizes or how parallelizable they are.

- In this case we show that `EQUI`, a policy which dynamically allocates an equal number of cores to each job in the system at every moment in time, is optimal with respect to mean response time.
- We analyze fixed-width policies, a class of policies that allocate the same number of cores to every job and do not require dynamically reallocating cores. We show how to determine the best fixed-width policy as a function of the system load and number of cores in the system.
- We prove that the best fixed-width policy performs almost as well as `EQUI` as the number of

cores in the system increases. Hence, we say that the best fixed-width policy is near-optimal in the large-system limit.

In many real-world scenarios, the system will have accurate estimates of the sizes of each job. Hence, we next consider the case where job sizes are known to the system (Chapter 5). Here, we find that an optimal policy should prioritize short jobs ahead of long jobs.

- When job sizes are known exactly to the system, and all jobs are present at time 0, we derive a policy called high-efficiency SRPT (heSRPT) which minimizes mean response time. The heSRPT policy balances a tradeoff between biasing towards short jobs and maintaining overall system efficiency.
- We prove that heSRPT can be generalized to minimize a whole class of weighted response time metrics, including mean slowdown (see Section 5.2)
- We show how heSRPT can be used to create a new policy for the online case where jobs arrive to the system over time. The online policy we create outperforms existing policies by an order of magnitude.

It is often the case that the system processes jobs with vastly different levels of parallelizability. We therefore consider the case where there are multiple classes of jobs, and each class of jobs follows a different speedup function (Chapter 6).

- We analyze a proportionally fair class of policies called GREEDY, and derive the best proportionally fair policy, GREEDY*.
- We show how to numerically compute an optimal policy. This allows us to see that GREEDY* performs well in a wide range of scenarios.
- In the special case where jobs are either fully parallelizable (elastic) or non-parallelizable (inelastic), we prove that a special case of the GREEDY* policy called Inelastic-First is optimal.
- We provide the first analysis of the response time under the Inelastic-First policy.

While parallelizable jobs exhibit the kind of concave speedup functions we describe in our model, these speedup functions are measuring the job's average level of parallelizability over time. Many systems, such as databases, process jobs consisting of multiple distinct phases where each phase has a different speedup function. Hence, we derive policies for scheduling jobs consisting of multiple phases (Chapter 7).

- We develop a new model of parallelizable jobs consisting of phases, where each job stochastically moves between highly-parallelizable elastic phases and non-parallelizable inelastic phases.
- Under this new model, we prove that a generalization of the Inelastic-First policy from Chapter 6 is optimal with respect to mean response time.
- Using queries from the Star Schema Database Benchmark [76], we perform simulations to show that we can dramatically improve mean query latency when processing real-world database workloads. Using our theoretical results, we construct a new policy called `IF-SRPT` which performs even better in practice, outperforming current database schedulers by a factor of 2.

1.6 Outline

The rest of this thesis is organized as follows:

- **Model** We begin by describing the basic model used in this thesis in Chapter 2.
- **Prior Work** We then provide an in-depth discussion of prior work in Chapter 3.
- **Scheduling Without Job Sizes.** In Chapter 4 we consider the case where a job's size is unknown to the system, and cannot be learned by the scheduler. We prove that `EQUI` is the optimal policy in this case, and show that the optimal fixed-width policy is near optimal in the large-system limit. These results first appeared in [8].
- **Scheduling With Job Sizes.** In Chapter 5 we consider the case where job sizes are known exactly to the system. We prove the optimality of heSRPT in this case for a range of weighted response time metrics including mean response time and mean slowdown. These results first appeared in [10]
- **Scheduling With Multiple Speedup Functions.** In Chapter 6 we discuss the case where jobs are allowed to follow different speedup functions. We analyze a proportionally fair class of policies called GREEDY, and consider a special case where jobs are either fully parallelizable (elastic) or non-parallelizable (inelastic). These results first appeared in [8] and [9].
- **Scheduling Jobs with Phases.** In Chapter 7 we develop a model for jobs composed of multiple phases, and provide optimality results in this model. These results first appeared in [11].

Chapter 2

Our Model

We now provide a description of the basic underlying model that will be used in our theoretical results. Throughout this thesis, we will continually tweak the specific assumptions or parameters of the model we are considering to derive results in different scenarios. However, this chapter will outline the basic assumptions of our model and the accompanying notation we will use throughout the thesis.

System Model

We assume that our system consists of n homogeneous cores. While we will use the terminology of parallelizing jobs across *multiple cores* in our theoretical results, the policies we derive could apply equally to scheduling jobs across *multiple servers* in a datacenter or distributed computing framework.

Arrivals

We are interested in the case where a stream of parallelizable jobs arrive to the system over time. We will therefore assume that jobs arrive to the system according to a stationary stochastic process with some fixed average arrival rate. We will assume this arrival process is Poisson unless otherwise noted.

Inherent Job Size

Each arriving job has an associated *inherent job size*, X, which is defined as the job's running time when run on a single core. A job's size represents the amount of work associated with a job. For instance, a database query may have some number of table rows it must examine, or an ML training job may be required to complete some number of iterations of gradient descent. To standardize the definition of job size across these applications, we use the job's single-core running time, which we assume is directly proportional to the amount of work the job must perform.

We will generally consider each job size, X, to be a random variable whose value and distribution may be known or unknown to the system. Chapter 4 considers the case where each job size is unknown and exponentially distributed, meaning the system has no ability to tell which jobs have

shorter or longer remaining sizes. Chapter 5 considers the case where job sizes are drawn from an arbitrary distribution and are exactly known to the system, so the system can precisely differentiate shorter jobs from longer jobs. Chapter 6 and Chapter 7 both consider cases where the system has partial size information in that some jobs are known to have smaller remaining sizes *on average* than other jobs, but the precise job sizes are not known to the system.

Speedup Functions

We are often interested in how long a job will run if allocated $k > 1$ cores. We assume that any job can be run on any subset of cores. We therefore define a *speedup function*, $s(k) : \mathbb{R}^+ \to \mathbb{R}^+$ which tells us how many times faster a job will complete if run on k cores instead of a single core. We assume that $s(k)$ is a sublinear, non-decreasing, concave function, a phenomenon that has been observed across a wide array of systems and applications [62, 79, 109]. Given a job's size, X, and speedup function, $s(k)$, the running time of a job on k cores, X_k is defined to be

$$X_k = \frac{X}{s(k)} \quad \forall k > 0. \tag{2.1}$$

One popular speedup function described in the literature is *Amdahl's law*[45], which considers every job as being composed of some fraction, f, of work which is perfectly parallelizable while the remaining $(1-f)$ fraction of the job receives no speedup from parallelization. Hence, Amdahl's law defines $s(k)$ to be

$$s(k) = \frac{1}{\frac{f}{k} + (1-f)}.$$

Other theoretical work has used a simple *power law speedup function* to approximate a variety of empirically derived speedup curves[49]. A power law speedup function is defined to be of the form

$$s(k) = k^p \quad 0 < p \leq 1.$$

Because a job's core allocation may change over time, we must define the running time of a job whose core allocation is not fixed for its entire lifetime. We define a job's *remaining size* at time t, $X(t)$, to be the remaining time required to complete the job using a single server. If a job receives an allocation of k servers from time t until time t', the job's remaining size at time t' is defined to be

$$X(t) - (t' - t)s(k).$$

Here, $s(k)$ is effectively denoting how many times faster a job's work is completed on k servers than on a single server. This notion is completely compatible with our prior definition.

In general, it could be the case that every job represents the execution of a totally unique computation, and therefore any particular job, job i, will have its own unique corresponding speedup function, $s_i(k)$. However, it is frequently the case that a workload consists a small set of programs which are executed repeatedly, perhaps with different inputs. In Chapter 4 and Chapter 5, we consider the case where all jobs are instances of a single program and thus all follow the same, single speedup function, $s(k)$.

Next, in Chapter 6, we consider the case where jobs may belong to one of multiple *classes*, where jobs within a class represent instances of the same program, and therefore receive the same speedup. Specifically, class c jobs all follow the same speedup function $s_c(k)$. Chapter 6 also considers an important special case where jobs belong to one of two particular classes. We call the first class of jobs *elastic* because they are perfectly parallelizable across any number of cores. We call the second class of jobs *inelastic* because they are not parallelizable and receive no benefit from running on more than one core. More formally, elastic jobs follow a speedup function $s_E(k)$ and inelastic jobs follow a speedup function $s_I(k)$ where

$$s_E(k) = k \quad \forall k \geq 0 \qquad s_I(k) = \begin{cases} k & 0 \leq k \leq 1 \\ 1 & k > 1 \end{cases}.$$

Finally, in Chapter 7, we consider the case where a job's speedup function may change over time. It is common for parallelizable jobs to consist of multiple *phases* of computation, where each phase is either highly parallelizable or highly non-parallelizable. Hence, we model each job as being composed of a sequence of elastic and inelastic phases. Each phase has a corresponding inherent size, and a job is considered complete once all of its phases have been completed in order. Elastic phases follow the speedup function $s_E(k)$ and inelastic phases follow the speedup function $s_I(k)$. Because each job may follow a different sequence of phases, the speedup a job receives from being allocated k servers for its entire lifetime may be different for each job. For a detail description of how we model the sequence of phases for each job, see Section 7.2.

Scheduling Policies

We are generally interested in maintaining low *response times* across the stream of incoming jobs. A job's response time, which we denote with the random variable T, is defined to be the time between when a job arrives to the system and when it is completed. We will be interested in scenarios where the limiting distribution of T exists, in which case we will often seek to minimize the *mean response time*, $\mathbb{E}[T]$, across jobs. It will also often be useful to consider a related quantity, the random variable N, which denotes the number of jobs in the system.

In order for the limiting distribution of response time to exist, it suffices to show that the system is *stable*. We define the *system load*, ρ, of a system to be

$$\rho = \frac{\lambda \mathbb{E}[X]}{n}.$$

Following standard queueing-theoretic techniques, one can usually show that a system is stable if

$$\rho < 1 \quad \text{and} \quad \mathbb{E}[X^2] < \infty.$$

We will explicitly note when these conditions do not suffice for stability.

Our goal is to derive and analyze *scheduling policies* which define the number of servers allocated to each job in the system at every moment in time. We will assume that jobs can receive

fractional server allocations. Hence, for a scheduling policy P, we will define an allocation function, $\boldsymbol{\theta}^P(t)$, which defines the fraction of the n servers allocated to each job in the system at time t. Specifically, at time t, when there are $N(t)$ jobs in the system,

$$\boldsymbol{\theta}^P(t) = \{\theta_1^P(t), \theta_2^P(t), \ldots, \theta_N^P(t)\}$$

where

$$0 \leq \theta_i^P(t) \leq 1 \quad \forall 1 \leq i \leq N(t) \quad \text{and} \quad \sum_{i=1}^{N(t)} \theta_i^P = 1.$$

We define the *instantaneous system efficiency* of the system under a policy P at time t to be the sum of the speedups received by the jobs currently in the system. That is,

$$\text{System efficiency at time } t = \sum_{i=1}^{N(t)} s(\theta_i^P(t) \cdot n).$$

As long as speedups are not super-linear, the maximum instantaneous system efficiency is n. System efficiency can be thought of as a measure of how much overhead due to parallelism is being incurred by the current server allocation.

Chapter 3

Prior Work

Despite the prevalence of parallelizable data center workloads, it is not known, in general, how to optimally allocate cores to a set of parallelizable jobs. The work on scheduling parallelizable jobs has generally been conducted across three different research communities — the systems community, the worst-case theoretical community, and the performance modeling community. In this section, we review the contributions from each community and explain how the work of this thesis, which takes a performance modeling approach, aims to address gaps in the literature from the other two communities.

3.1 The Systems Community

The need to schedule parallelizable jobs with has been identified across a wide range of systems. Specifically, this problem has been discussed in the context of cluster scheduling [22, 23, 98, 101], high-performance computing (HPC) [87, 95], distributed machine learning [62], and databases [43, 58, 102].

The state-of-the-art in cluster scheduling and HPC systems is to let the *user* decide their job's resource allocations by reserving the resources they desire [63, 98, 100], and then to allow the system to pack jobs onto the underlying hardware [69, 84, 99]. Users reserve resources greedily, leading to low system efficiency. While there has been work which shows the benefits of allowing the system to determine resource allocations in data centers [23, 95], the allocation policies described here are based on heuristics and make no provable performance guarantees.

Machine learning frameworks are also faced with the challenge of allocating resources to model training and serving jobs which are notoriously parallelizable. These jobs are known to follow the same kind of sublinear speedup functions we describe in this thesis [62]. However, state-of-the-art frameworks for scheduling machine learning jobs are still based on heuristics such as trying to balance load [67] or optimize for *goodput* [82] (which we refer to as instantaneous system efficiency in this thesis).

The database community has tackled the problem of how to effectively parallelize queries [58]. However, [58] also claims that, in the absence of exogenously defined query priorities, there is not benefit to prioritizing one query over another. We will see in this thesis that the priorities given to

different parallelizable jobs can have massive impacts on job response times. More recently, [102] has noted that prioritizing short queries over long queries can reduce mean query latency, however they do so through the use of a heuristic policy. We will see in Section 7.7.1 that solely prioritizing short queries can have unintended effects on mean query latency.

In summary, while many systems have described practical approaches to resource allocation for parallelizable jobs, these solutions are generally based on heuristics. As a results, these solutions often require significant parameter tuning, and do not provide formal guarantees about performance.

Instead of focusing on developing better scheduling policies, major advances from the systems community in the area of scheduling parallelizable jobs have focused on building scalable schedulers which are capable of checkpointing, preempting, and resuming jobs while maintaining low overhead and mitigating straggler effects. The Hopper system [85] demonstrates techniques for greatly reducing straggler effects by which the completion of a parallelizable computation is delayed by a small number of tasks which take longer to complete. We therefore assume that the scalability of a job can be measured by a speedup curve rather than modeling a parallelizable job at the task level. Meanwhile, advances in the Operating Systems community have demonstrated the ability to implement centralized scheduling policies which can preempt running jobs with mere microseconds of overhead in a multicore machine [52] or even a full rack of servers [111]. Similar results have been observed in the ability to share and dynamically re-allocate hardware accelerators such as GPUs for use in the training of machine learning models [105]. We will therefore consider scheduling policies which have the ability to preempt jobs and change their resource allocations with no overhead.

Our goal is to improve upon the heuristic policies used in these state-of-the-art systems by providing practical policies with provably optimal or near-optimal performance.

3.2 The Worst-Case Theoretical Community

A significant portion of the prior theoretical work on scheduling parallelizable comes from the worst-case theoretical community. This work uses *worst-case* analysis, in that that job sizes, arrival times, and speedup functions are all assumed to be adversarially chosen. Given these pessimistic assumptions, there is a fundamental result [60] showing that any policy for scheduling jobs onto a set of n identical servers can be arbitrarily far from the optimal policy. Specifically, no policy can achieve a competitive ratio better than $\Theta(\log \min(p, \frac{M}{n}))$ where M is the number of jobs and p is the ratio of the maximum job size to the minimum job size. Hence, the worst-case community is either looks for policies that achieve a competitive ratio close to this fundamental lower bound, or uses resource augmentation arguments [1].

It is easiest to understand the prior worst-case theoretical work on scheduling parallelizable jobs in terms of the model of parallelism considered. We will therefore discuss several theoretical models of parallelism before considering prior work from the systems community on scheduling parallelizable jobs.

[1]Resource augmentation analysis is a relaxation of competitive analysis that, for some $s > 1$, compares an algorithm using speed s processors against the optimal policy using speed 1 processors.

Jobs with Speedup Curves

The basic model used in this thesis considers a *speedup function*, $s(k)$, that describes the speedup a job receives from running on k servers. Here, $s(k)$ is some positive concave non-decreasing function. Work using this model from the worst-case scheduling literature finds that, when job sizes are known, a generalization of EQUI is $\Omega(\log p)$-competitive with the optimal policy [49]. Moreover, EQUI is again shown to be constant competitive with constant resource augmentation [25, 27].

Overall, the general consensus from both the worst-case scheduling community is that EQUI should be used to achieve good or possibly optimal mean response time. This thesis will begin by showing conditions for the optimality of EQUI using our stochastic model using average-case analysis. However, we will see that these conditions are fairly restrictive (see Chapter 4), and that the performance of EQUI is often far from optimal both in theory and in practice. This discrepancy is largely due to the overly pessimistic nature of the adversarial assumptions made by the worst-case theoretical community.

Jobs with Parallelizable Phases

Another body of work from the worst-case scheduling community [24, 25, 26, 27] considers jobs whose speedup functions change over time. This work considers the problem of scheduling parallelizable jobs composed of phases of differing parallelizability. However, due to the worst-case nature of the analysis, this work is forced to either consider an offline problem where all jobs arrive at time 0 [26], or to rely on resource augmentation [24, 25, 27] to provide an algorithm which is within a (potentially large) constant factor of the optimal policy. This work concludes that the EQUI policy, as well as a generalization of it, is constant competitive given a small constant resource augmentation. We will see that the gap between EQUI and the optimal policy grows even wider using our stochastic models of jobs which consist of multiple phases. This difference has been corroborated by real-world systems experiments (see Chapter 7).

DAG Jobs

A separate branch of theoretical work on scheduling parallel jobs that developed concurrently with the above models considers every parallel job as consisting of a set of tasks with precedence constraints specified by a Directed-Acyclic-Graph (DAG). In this model, introduced in [14], a task can only run on a single server, but any two tasks that do not share a precedence relationship can be run in parallel. Much of the work in this area is concerned with how to efficiently schedule a *single* DAG job onto a set of servers [13, 14, 15].

Recently, [4] considered the online problem of scheduling a stream of DAG jobs to minimize the worst case mean response time. Using a resource augmentation argument, they show that EQUI and its generalization are constant competitive with constant resource augmentation.

3.3 The Performance Modeling Community

The results of the worst-case theory suggest that, without resource augmentation, there is little room to improve the worst-case performance of scheduling policies for parallelizable jobs because simple policies like EQUI are capable of meeting known lower bounds on achievable performance. However, we have also seen that the problem of scheduling parallelizable jobs remains open from the point of view of system designers. In particular, while the worst-case theory suggests policies which may perform reasonably in the worst-case, system designers are often interested in *optimizing for the common case*. This suggests that we should be using the kind of average-case analysis favored by the stochastic performance modeling community. Here, job sizes and arrival times are assumed to be governed by underlying stochastic processes, and the goal is generally to measure and optimize the *average, steady-state* performance of the system with respect to metrics like mean response time.

Much of the prior work from the performance modeling community has been limited to scheduling jobs on a single server [20]. While there has been work on scheduling in stochastic multiserver systems (e.g [3, 5, 28, 32, 36, 42]), this work assumes that jobs run on at most one server at a time (that is, all jobs are not parallelizable).

The closest the performance modeling community has come to considering parallelizable jobs is the related problem of *fork-join parallel* queueing. The fork-join model considers a special case of DAG scheduling where a system of n servers processes a stream of jobs consisting of n tasks with no precedence constraints. When a job arrives to the system, one task is immediately dispatched to each of the n servers. Results in this model have been hard to come by – the exact mean response time for this system is known only when task sizes are exponentially distributed and $n = 2$ [71]. Recent work has considered the idea of *limited fork-join* queueing, where each job consists of $k < n$ tasks [103], but this work provides only upper bounds on response time. In all of the work on fork-join queueing, all jobs have the same, fixed level of parallelism which cannot be changed by the system to improve response times.

More recently, the performance modeling community has considered the scheduling of *multiserver jobs*[38]. Here, each job requires a particular number of servers in order to run. If a job's demand cannot be satisfied, the job either remains in the queue until it can be run, or in some circumstances the job is dropped. Unlike the fork-join model, the job will complete at the same time on each server it occupies. In this model, there is once again no flexibility to choose the level of parallelism for a particular job. Work in this model has thus far focused on stability results [29] and response time bounds [47].

The problem of scheduling parallelizable jobs also shares some similarities with co-flow scheduling [18, 51, 53, 83, 91] where one allocates a continuously divisible resource, link bandwidth, to a set of flows to minimize mean response time. However, here there is usually no explicit notion of a flow's speedup function. The most applicable work here is [1], which explores the tradeoff between efficiency and opportunistic scheduling in wireless networks. Section 5.5.1 generalizes the results of [1] to metrics beyond mean response time, including mean slowdown.

Hence, although the tools of the performance modeling community are well-suited to address the problem of scheduling parallelizable jobs, the work of this thesis represents a novel research contribution.

Chapter 4

Scheduling Without Job Sizes

4.1 Introduction

Perhaps the most difficult demand that today's parallel systems must meet is that users often expect their jobs to be served quickly, but without having to provide the system with any information about what the job is doing, how long they expect the job to run, or how well the job can be parallelized. These expectations likely come from the way users have been trained to interact with modern operating systems (OSs). Most modern OSs do not introspect into the programs they run, and by default do not make predictions about how long a program will run or how effectively it will make use of additional cores. The user, however, expects the OS to provide prompt execution on the underlying hardware, even if the system is also being asked to run several other programs simultaneously. This execution model has also permeated to the cloud computing context, where users ask the system to execute their virtual machines and containers on demand, and for indefinite periods of time. Hence, it is crucial to have the ability to make good scheduling decisions, even when the system has no information about the sizes or speedup functions of the jobs it is running.

The problem of scheduling parallelizable jobs without size information also stands to improve reservation-based systems [63, 98, 100], where users reserve the number of cores or servers on which they want to run their jobs. In these systems, users are known to dramatically overstate their resource needs, and then underutilize the resources they reserve. That is, not even the users know how to describe the sizes and speedup functions of their own jobs in these systems. The policies described in this chapter could improve these systems by eschewing user reservations and instead allowing a scheduling policy to decide how resources are allocated to each job.

4.1.1 Scheduling Tradeoffs

To model the scenarios where the system has little information about the jobs it runs, we will assume that all jobs look identical to the scheduler in terms of job size and speedup function. Specifically, the scheduler has no ability to tell if one job has a smaller remaining size than another, and the scheduler has no ability to tell whether one job is more parallelizable than another.

We can now consider how the particular scenarios modeled in this chapter affect the tradeoffs

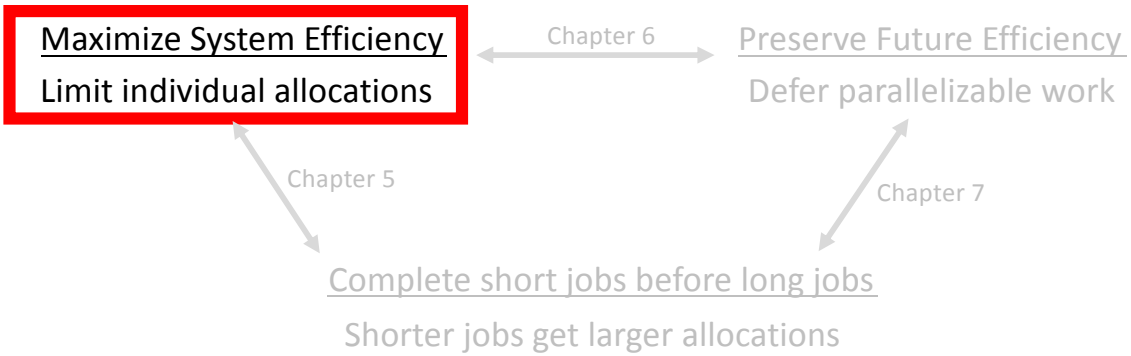


Figure 4.1: When jobs look identical with respect to their sizes and speedup functions, a scheduling policy should maximize instantaneous system efficiency.

outlined in Section 1.4. Figure 4.1 shows an illustration of our tradeoffs, updated to match the scenario where jobs look identical with respect to their remaining sizes and speedup functions. Clearly, the setting considered in this chapter simplifies the tradeoffs that a scheduling policy must balance. Without the ability to differentiate short jobs from long jobs, or less parallelizable jobs from more parallelizable jobs, a scheduling policy's sole concern is maximizing the instantaneous efficiency of the system. While we have intuitively argued that this involves limiting the number of cores allocated to any one individual jobs, this chapter will formalize that intuition.

Key Insight

When scheduling parallelizable jobs, increasing the number of cores allocated to an individual job will generally reduce the response time of that particular job. However, jobs receive a diminishing marginal benefit from being allocated additional cores. Hence, increasing the number of cores allocated to a job which already has a large allocation may barely decrease the job's response time, while increasing the number of cores allocated to a job with a small allocation may greatly decrease the job's response time. Similarly, jobs with large allocations will suffer mildly from having their allocations decreased as compared to jobs with small allocations. It therefore seems natural to increase the allocations of cores to jobs with smaller allocations which will benefit greatly from the additional cores, while incurring a mild cost from reducing the allocations of cores to jobs with larger allocations. The result is that policy called `EQUI`, which splits cores *equally* across jobs, is optimal with respect to mean response time.

We formalize this argument by reasoning about the instantaneous system efficiency of an allocation, a metric which tells us how much total benefit the jobs in the system are receiving from their current allocations. We show that, by equalizing the marginal benefit each job would receive from being allocated an additional core, `EQUI` *maximizes instantaneous system efficiency*. As Figure 4.1 suggests, when jobs cannot be differentiated on the basis of their size or speedup function, maximizing system efficiency is sufficient to minimize mean response time.

4.1.2 Contributions

This chapter considers the case where the size (inherent work) associated with each job is unknown to the system, and that the system is not able to develop a better estimate of a job's size by running the job. Furthermore, we assume that all jobs follow the same, single speedup function. The result is that the system cannot differentiate between jobs on the basis of their sizes or speedup functions. Our goal is to allocate the n identical cores in the system across a stream of arriving jobs in order to minimize the mean response time across jobs.

While the problem of optimally scheduling parallelizable jobs has known lower bounds in the worst case, this problem had never been studied in the average case. In this chapter, we provide the first average-case analysis of policies for scheduling parallelizable jobs. By making stochastic assumptions about the arrival times of the jobs, and the about the distribution of job sizes, we provide the first average-case optimality results for this problem. The contributions of this chapter are as follows:

- We begin in Section 4.3 by analyzing the performance of the EQUI policy. We provide a proof that EQUI is optimal with respect to mean response time when job sizes are exponentially distributed and all jobs follow the same speedup function. EQUI is a policy which first appeared in [24]. Under EQUI, at all times, the n cores are equally divided among the jobs in the system. Specifically, whenever there are ℓ jobs in the system, then each job is parallelized across n/ℓ cores. EQUI is an idealized policy in that (i) EQUI assumes that when a job runs on n/ℓ cores its run time will be distributed as $\frac{X}{s(n/\ell)}$, even if n/ℓ is not an integer, and (ii) EQUI requires jobs to be *malleable* – every job can change its level of parallelization while it runs.
- Next, Section 4.4 introduces the concept of *fixed-width* scheduling policies. In practice, jobs are not malleable but are often *moldable* – a job can run on any number of cores, k, but it must run on the same k cores for the duration of its lifetime. In theory, a scheduling policy could choose a different number of cores on which to run each incoming job. We define an extremely simple class of scheduling policies, called *fixed-width* policies, whereby every job is run on the *same fixed number* of cores.

  We take this a step further and impose the constraint that the n cores of the machine are partitioned into statically-sized "chunks", each with k cores, where k divides n. For example, Figure 4.4 shows some potential chunkings of a 16-core machine. Every arriving job is dispatched to a single chunk and runs across all the cores in that chunk.

  We introduce a policy called *JSQ-Chunk* which uses a fixed width, k, and assigns jobs to chunks according to the Join-Shortest-Queue dispatching policy. We provide an approximate response time analysis for JSQ-Chunk, and, more significantly, we show how to choose the *optimal fixed-width*, k^*, for JSQ-Chunk as a function of the system load, ρ.
- Finally, Section 4.5 shows that a fixed-width scheduling policy can perform near-optimally in the large-system limit. Fixed-width scheduling policies seem heavily restricted in their ability to effectively exploit parallelism. We prove, however, that certain fixed-width policies, like JSQ-Chunk, can achieve mean response time converging to that of EQUI as the number of cores, n, becomes large.

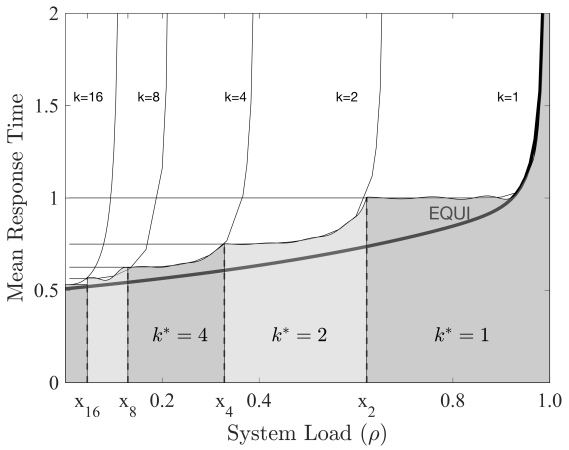


Figure 4.2: Mean response times under JSQ-Chunk (thin line) and EQUI (thick line) when $n = 64$. We assume a speedup curve of Amdahl's law with parameter $p = 0.5$ and a shifted Pareto job size distribution with $\alpha = 2$ and mean $\mathbb{E}[X] = 1$.

We provide a preview of contributions 2 and 3 in Figure 4.2. Here we see the mean response times under JSQ-Chunk, as a function of system load ρ, for various choices of k. For a given ρ, this figure shows how to choose the optimal k^*. The figure shows a separate curve for mean response time under each possible choice of k when using JSQ-Chunk. As expected, increasing k can reduce mean response time, but also reduces the system's stability region. The shaded regions under these curves denote the values of ρ for which a particular choice of k is optimal. For example, when $x_8 \leq \rho \leq x_4$, $k^* = 4$. Surprisingly, the performance of JSQ-Chunk, when using the optimal chunk size k^*, is not far from that of EQUI– we will prove that this difference vanishes in systems with many cores.

4.2 Our Model

We assume that jobs arrive into a n-core machine according to a Poisson Process with rate Λ jobs/second where

$$\Lambda := \lambda n$$

for some λ.

In this chapter, we assume that job sizes are unknown to the system. Furthermore, we assume that the system does not learn more about a job's size as the job runs. That is, we assume each job size X is exponentially distributed with rate μ, meaning every job in the system has the same distribution of remaining size, regardless of how long it has been running. The optimality proofs in this chapter will require this exponential assumption, but all of the performance analysis presented admits general job size distributions.

We assume that the each job's speedup function, $s(k)$, is *non-decreasing* and *concave*, in agreement with functions described in [46]. Additionally, we will focus on instances where jobs receive

an imperfect speedup and thus $s(k)$ is assumed to be *sublinear*: $s(k) < k$ for all $k > 1$ and there exists some constant $c > 0$ such that $s(k) < c$ for all $k \geq 0$.

An example of a well-known speedup function is Amdahl's law [66], which models every job as having a fraction of work, p, which is parallelizable. The speedup factor $s(k)$ is then a function of the parameter p as follows:

$$s(k) = \frac{1}{\frac{p}{k} + 1 - p}.$$

Figure 4.3a shows Amdahl's law under various values of p. Although Amdahl's law ignores aspects of job behavior, we see in 4.3b that several workloads from the PARSEC-3 benchmark [109] follow speedup curves which can be accurately modeled by Amdahl's law. Hence, although our analysis will not rely on the specifics of the speedup function, we will use Amdahl's law in numerical examples.

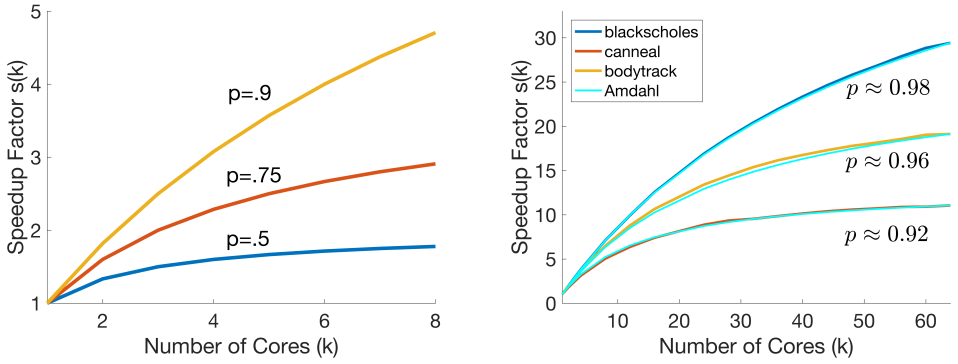


Figure 4.3: Various speedup curves under (a) Amdahl's law and (b) the PARSEC-3 Benchmark. We see that the PARSEC-3 speedup curves are accurately approximated by Amdahl's law; these approximations are shown in (b) in light blue.

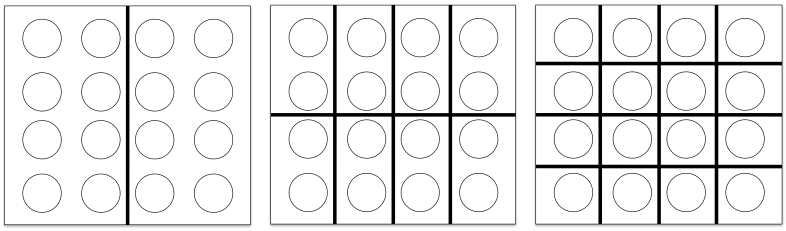
We assume that both the system load, ρ, and s are constant over time.

In this chapter, we assume that jobs are homogeneous with respect to s – all jobs receive the same speedup due to parallelization.

At any moment in time, there may be several jobs (job pieces) running on a particular core. We assume each core time-shares between its jobs (i.e. *Processor Sharing*), which is typical for processors in multicore machines. For example, suppose there are only 2 jobs in the system, each of size x, and each runs on the *same* 5 cores. Each job receives a speedup of $s(5)$ since it runs on 5 cores, but its run time is $2 \cdot \frac{x}{s(5)}$ because each job only gets half of each core.

The Problem

Our goal is to find and analyze scheduling policies that aim to minimize the mean response time, $\mathbb{E}[T]$, across all jobs. Clearly, $\mathbb{E}[T]$ depends on the scheduling policy, the speedup function, s, the arrival rate of jobs into the system, Λ, the mean job size, $\mathbb{E}[X]$, and the number of cores, n.



(a) $k = 8$ (b) $k = 2$ (c) $k = 1$

Figure 4.4: A machine with $n = 16$ cores under various chunk sizes, k. Each job is dispatched to a single chunk.

4.3 EQUI: An Optimal Policy

In order to effectively parallelize jobs, one imagines that using a *dynamic* level of parallelization might greatly improve mean response time. Specifically, it makes sense to consider policies where the level of parallelization of an incoming job is determined based on the state of the system at the time when the job arrives. In addition, if jobs are malleable, active jobs could change their levels of parallelization during their lifetimes as the state of the system changes.

EQUI [24] is a generalization of Processor Sharing to systems with multiple cores. Under EQUI, whenever there are ℓ jobs in the system, each job runs on n/ℓ of the cores. We have defined $\mu = 1/\mathbb{E}[X]$ to be the rate at which jobs complete when run on a single core. EQUI is an idealized policy in that it assumes that when a job runs on n/ℓ cores its run time will be distributed as $\frac{X}{s(n/\ell)}$, even if n/ℓ is not an integer. This corresponds to a service rate of $s(n/\ell)\mu$ for each job when jobs are exponentially distributed. Additionally, we assume that $s(k) = k$ when $k \leq 1$ to account for the effects of Processor Sharing. Hence, when there are ℓ jobs in the system, the total rate at which jobs complete is $\ell\mu s(n/\ell)$ which is equal to $n\mu$ when $\ell \geq n$. Importantly, EQUI always processes every job in the system at the same rate, even when there are more than n jobs in the system.

Figure 4.5 shows a Markov chain representing the total number of jobs in the system under EQUI. The Markov chain for EQUI allows us to formalize the intuition that EQUI is optimal because it maximizes instantaneous system efficiency. Our argument consists of three steps. First, we note that the departure rates we calculated for the Markov chain are proportional to the instantaneous system efficiency under EQUI in each state. Second, Theorem 4.3 will begin by showing that EQUI maximizes the total rate of departures in every state. Finally, Theorem 4.3 will also show that no policy can do better than EQUI by using a lower total rate of departures in any state. That is, we will show that EQUI is optimal *because* it maximizes the instantaneous system efficiency in every state.

The Markov chain for EQUI assumes that job sizes are exponentially distributed. However, Lemma 4.1 proves that EQUI's performance is actually insensitive to the job size distribution. We begin by proving Lemma 4.1 before proceeding with the proof of Theorem 4.3.

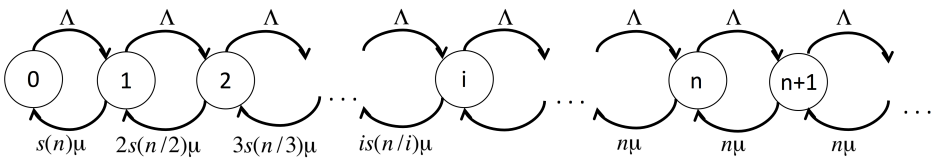


Figure 4.5: Markov chain representing the total number of jobs under EQUI.

4.3.1 Insensitivity of EQUI

Lemma 4.1. *Mean response time under* EQUI *is insensitive to the job size distribution.*

Proof. A system is considered to practice processor sharing if, regardless of how many jobs are in the system, each job receives the same service rate. Note that under EQUI, when m jobs are in the system, each job receives the same service rate by running on n/m of the cores. Hence, EQUI fits the definition of a processor sharing system, albeit one with state dependent service rates because the total rate of departures depends on the number of jobs in the system. It is known from [7, 16] that the mean response time in processor sharing systems with state dependent service rates is insensitive to the job size distribution, thus our performance analysis of EQUI holds for the case where job sizes follow a general distribution. □

4.3.2 Proving that EQUI is Optimal

Theorem 4.3 relies on Lemma 4.2.

Lemma 4.2. *For any concave, sublinear function, s, the function $i \cdot s\left(\frac{n}{i}\right)$ is increasing in i for all i for all $i < n$, and is non-decreasing in i for all $i \geq n$.*

Proof. To see that $i \cdot s\left(\frac{n}{i}\right)$ is increasing in i when $i < n$, we can consider the following difference for any $\delta > 0$:

$$i \cdot (1+\delta)s\left(\frac{n}{i \cdot (1+\delta)}\right) - i \cdot s\left(\frac{n}{i}\right) = i\left((1+\delta)s\left(\frac{n}{i \cdot (1+\delta)}\right) - s\left(\frac{n}{i}\right)\right).$$

Since $\frac{n}{i} > 1$, s increases sublinearly, and $(1+\delta)s\left(\frac{n}{i \cdot (1+\delta)}\right) > s\left(\frac{n}{i}\right)$. Thus

$$i \cdot (1+\delta)s\left(\frac{n}{i \cdot (1+\delta)}\right) - i \cdot s\left(\frac{n}{i}\right) > 0$$

and $i \cdot s\left(\frac{n}{i}\right)$ is increasing in i.

For any $i \geq n$, we have assumed $s\left(\frac{n}{i}\right) = \frac{n}{i}$. Thus,

$$i \cdot s\left(\frac{n}{i}\right) = n \qquad \forall i \geq n,$$

which is non-decreasing in i. □

Theorem 4.3. *Within our model, assuming jobs are malleable, have exponentially distributed sizes, and all follow the same speedup function, s, $\mathbb{E}[T]^{EQUI} \leq \mathbb{E}[T]^{P}$ for any scheduling policy P.*

Proof. Let P be a scheduling policy which processes malleable jobs and currently has i active jobs (the system is in state i). In every state, i, P must decide (i) how many jobs, j, to run and (ii) how to allocate the n cores amongst the j jobs. Hence, for some $\vec{\theta}$, the rate of departures from state i under P is at most

$$\mu \sum_{k=1}^{j} s(n\theta_k) \tag{4.1}$$

where $0 < j \leq \min(i, n)$, $\theta_k > 0$ for all $1 \leq k \leq j$, and $\sum_{k=1}^{j} \theta_k = 1$. Note that (4.1) is an upper bound on P's departure rate, since it assumes that the j jobs run on disjoint partitions of cores; given that s is concave the total rate of departures will not increase when a single core (or fraction of a core) is allocated to more than one job.

We also know that

$$\frac{1}{j} \sum_{k=1}^{j} s\left(n\theta_k\right) \leq s\left(\frac{n}{j}\right)$$

again, by the concavity of s. Hence,

$$\sum_{k=1}^{j} s(n\theta_k) \leq j \cdot s\left(\frac{n}{j}\right) \qquad \forall\, 0 < j \leq i,\ \forall\, \vec{\theta} \in (0,1]^j$$

and thus an upper bound on P's total rate of departures from any state is of the form

$$j \cdot s\left(\frac{n}{j}\right) \mu.$$

By Lemma 4.2, $j \cdot s\left(\frac{n}{j}\right)$ is non-decreasing in j. Thus, an upper bound on P's rate of departures from any state i is

$$i \cdot s\left(\frac{n}{i}\right) \mu. \tag{4.2}$$

Furthermore, we can see that EQUI achieves this departure rate in every state, i. Hence, (4.2) is the maximal rate of departures from any state, i.

We can now compare the Markov chain corresponding to P with the Markov chain for EQUI (Figure 4.5) and relate the number of jobs in the system, N, under both policies. Both chains have the same arrival rates, but the rate of departures under EQUI is greater than or equal to the rate of departures under P in any state, i, since the departure rate chosen by EQUI is maximal. Thus,

$$\mathbb{E}[N]^{\text{EQUI}} \leq \mathbb{E}[N]^{\text{P}}$$

and, by Little's Law,

$$\mathbb{E}[T]^{\text{EQUI}} \leq \mathbb{E}[T]^{\text{P}}.$$

□

4.3.3 EQUI with General Size Distributions

Theorem 4.3 requires exponentially distributed job sizes. One might think that EQUI's insensitivity (Lemma 4.1) might imply that EQUI is optimal under all job size distributions. However, this is false. Consider for example the case where jobs follow a Pareto distribution with decreasing hazard rate. Independent of the speedup function, the optimal policy should devote more cores to jobs which have lower ages, and are thus more likely to finish in the immediate future. This differs from EQUI's equal division of resources.

In general, optimal scheduling of jobs with general job size distributions is an extremely difficult problem. Even optimally scheduling generally distributed jobs on a *single core* requires using a policy derived from the Gittins index [89]. In our case, where the optimal scheduling policy must allocate many cores, finding the optimal policy is at least as difficult. In fact, it is not known how to even numerically compute the optimal policy in this case.

4.4 Fixed-Width Policies

While EQUI performs very well, it requires jobs to change their levels of parallelization while running. Fixed-width policies require that jobs be moldable, but not necessarily malleable, which is more realistic for certain workloads. In general, a fixed-width policy is any policy which chooses a fixed level of parallelization, k, to use for each arriving job. Every arrival is parallelized across k cores, and is run on the same k cores for its entire lifetime. In this section, we consider several natural fixed-width policies.

We begin by defining the Random dispatching policy, which parallelizes each arriving job across k cores chosen uniformly at random. It turns out (see Theorem 4.6) that Random is dominated by the Random-Chunk policy; hence we devote Section 4.4.1 to Random-Chunk. In Section 4.4.2, we consider an improved fixed-width policy called JSQ-Chunk, which dispatches arrivals to the chunk with the shortest queues.

Like EQUI, Random-Chunk and JSQ-Chunk are insensitive to the job size distribution. Thus, our analysis of these policies admits general job size distributions, X.

4.4.1 Random-Chunk

The motivation behind the Random-Chunk policy is that we would like to use random assignment while ensuring that the different pieces of a single job complete at the same time. For a level of parallelization, k, we begin by partitioning the cores into $c = n/k$ chunks of size k. We will only consider values of k which divide n, creating a uniform partition of the cores. When a job arrives, a chunk is selected uniformly at random, and the job is parallelized across all k cores in this chunk.

It is easy to see that under Random-Chunk each piece of a job will at all times experience the same state, and hence each of the k job pieces will complete at the same time. Consequently, the response time for a job is equal to the response time of any of its k pieces, which in turn is simply the mean response time at a single core. This allows us to easily derive an expression for the overall mean response time under Random-Chunk with a level of parallelization k in Theorem 4.4 below.

Theorem 4.4. *The mean response time under Random-Chunk with a level of parallelization, k, is given by*

$$\mathbb{E}[T]^{\textit{Rand-Chunk}} = \frac{\mathbb{E}[X]}{s(k) - k\rho}.$$

Proof. Recall that cores follow the PS scheduling discipline. We know from [55] that the mean response time in a single $M/G/1/PS$ queue with arrival rate Λ, mean job size $\mathbb{E}[X]$ and load $\rho = \frac{\Lambda \mathbb{E}[X]}{n}$ is given by

$$\mathbb{E}[T] = \frac{\mathbb{E}[X]}{1-\rho}.$$

It will suffice to analyze the response time of one core in our system. Under a level of parallelization of k, the arrivals into a single core follow a Poisson process with rate

$$\Lambda_k := \frac{\Lambda}{c}$$

and the mean service requirement of job pieces is $\mathbb{E}[X_k]$ (see (2.1)). Thus, we can define the load on a single core in this system to be

$$\rho_k := \Lambda_k \mathbb{E}[X_k].$$

This allows us to express $\mathbb{E}[T]^{\text{Rand-Chunk}}$ under a level of parallelization of k as:

$$\mathbb{E}[T]^{\text{Rand-Chunk}} = \frac{\mathbb{E}[X_k]}{1-\rho_k} = \frac{\mathbb{E}[X_k]}{1 - \rho\frac{k}{s(k)}} = \frac{\mathbb{E}[X]}{s(k) - k\rho}.$$

□

Corollary 4.5. *The optimal Random-Chunk policy uses chunk size k^*, where*

$$k^* = \underset{k}{\operatorname{argmin}} \frac{\mathbb{E}[X]}{s(k) - k\rho}.$$

If we define the vector $\boldsymbol{k} = (k_1, k_2, \ldots k_m)$ to be the factors of n in increasing order, this implies:

$$k^* = \begin{cases} k_m & 0 \le \rho \le \frac{s(k_m) - s(k_{j-1})}{\mathbb{E}[X]} \\ k_i & \frac{s(k_{i+1}) - s(k_i)}{\mathbb{E}[X]} < \rho \le \frac{s(k_i) - s(k_{i-1})}{\mathbb{E}[X]}, 1 < i < m \\ k_1 & \frac{s(k_2) - s(k_1)}{\mathbb{E}[X]} < \rho. \end{cases}$$

The results of this analysis are shown in Figure 4.6. This figure shows the mean response time under various choices of k, and the shaded regions below the curves denote the values of ρ for which each choice of k is optimal. Under high load, we see that the system is unable to tolerate wasting resources by parallelizing jobs, and thus each job is run on a single core ($k^* = 1$). Conversely, under light load it is beneficial to pay this cost of parallelization in order to reduce the service requirement of individual jobs.

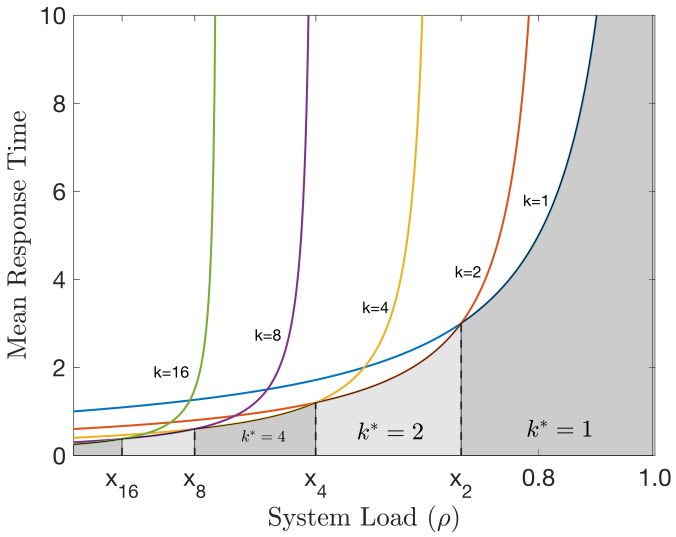


Figure 4.6: Results from analysis. Mean response times under Random-Chunk dispatching, a system with $n = 16$ cores, various choices of the level of parallelization k, and a hyperexponential job size distribution with $\mathbb{E}[X] = 1$ and $C^2 = 10$. The speedup curve used is Amdahl's law with parameter $p = 0.8$.

Random-Chunk vs. Random

As we have seen, Random-Chunk yields much easier analysis than Random. Fortunately, Theorem 4.6 tells us that Random-Chunk results in lower mean response times than Random, rendering the analysis of Random irrelevant.

Theorem 4.6. *The mean response time under Random-Chunk with a fixed level of parallelization, k, is less than under Random with the same k.*

Proof. Regardless of whether we are using Random or Random-Chunk dispatching, the arrival process into a given core is Poisson with rate Λ_k and the service requirement of job pieces is distributed as X_k. Thus, the response time for the ith job *piece* is distributed as some random variable T_i, and the response times of all pieces are identically distributed in both cases.

Under Random-Chunk, the response times of a job's pieces are identical. Hence, under Random-Chunk, the response time of a *job* is simply the response time of one of its pieces. Under Random, the response time of a job is the *maximum* of the k response times of its k pieces. Thus, for a single job, the response time under Random-Chunk has distribution T_1 while under Random the response time of a single job has distribution

$$\max(T_1, T_2, \ldots, T_k)$$

where $T_i \sim T$. Since $T \leq_{st} \max(T_1, T_2, \ldots, T_k)$, we see that, for a given level of parallelization, Random-Chunk's mean response time is less than that of Random. □

Mixed-Random-Chunk

Until now, we have studied Random-Chunk policies where each chunk consists of k cores. It is conceivable, however, that allowing for type 1 chunks of size k_1 and type 2 chunks of size k_2 could improve mean response time.

Having two chunk sizes introduces more dispatching parameters. One must decide how many cores, a, to devote to type 1 chunks. One also must decide what fraction, p, of arrivals to assign to type 1 chunks. Finding the optimal Mixed-Random-Chunk policy then requires simultaneously optimizing over not just k_1 and k_2, but additionally a and p.

We find that while some instances do exist where a Mixed-Random-Chunk policy is better than Random-Chunk, these instances occur very sparsely throughout the parameter space (k_1, k_2, a, p), and the advantage gained is very slight at best. In fact, if we limit ourselves to cases where the arrival rate into each chunk is proportional to the chunk's size, we can prove that Mixed-Random-Chunk is never advantageous in terms of mean response time or variance of response time (see Appendix A.1).

4.4.2 JSQ-Chunk

The Random-Chunk policy can be greatly improved by considering the state of the system when dispatching jobs. Under JSQ-Chunk with a level of parallelization, k, we again partition the system into chunks of size k. When a job arrives to the system, it is parallelized across the k cores in the chunk that is currently serving the fewest jobs. We expect that JSQ-Chunk will have much lower mean response times than Random-Chunk and will admit higher values of k^*. To determine the correct k^* for JSQ-Chunk, it is important that we can accurately analyze mean response time under JSQ-Chunk.

JSQ-Chunk is based on JSQ, an old and well-studied dispatching policy in the traditional queueing literature which assumes no parallelization. Under JSQ, the system consists of c queues and every incoming arrival is immediately dispatched to the queue with the smallest number of jobs. While analyzing response times in a traditional (non-parallel) system with JSQ dispatching is notoriously hard, [70] provides a good closed-form approximation of mean response time. The approximation in [70] assumes FCFS queues with exponentially distributed job sizes. However, as shown in [31], the case of PS queues with generally distributed job sizes has nearly the same mean response time as the case of FCFS queues with exponentially distributed service requirements. Thus, the approximation in [70] still applies to our model of PS queues and generally distributed job sizes. This approximation states:

$$\mathbb{E}[T]^{\text{JSQ}}(\Lambda, c, \mathbb{E}[X]) \approx W_{M/M/c}(\rho)S(\rho)R(\rho) + \mathbb{E}[X],$$

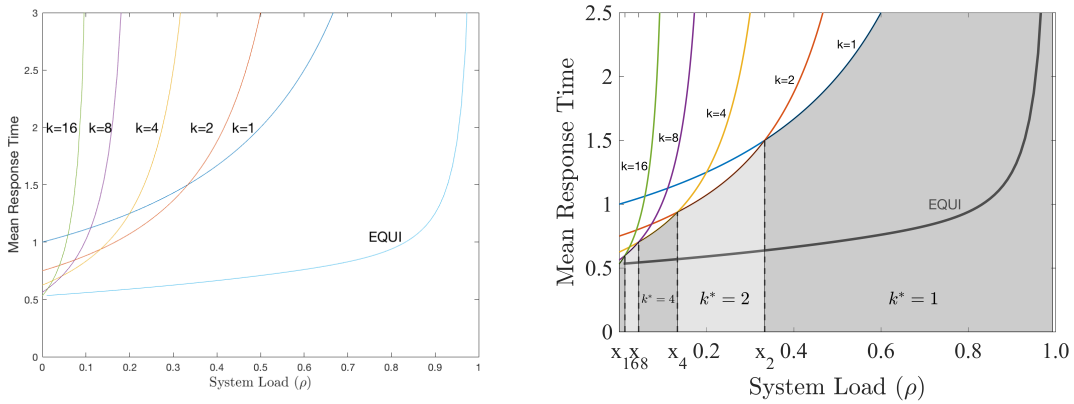
where Λ is the arrival rate, c is the number of queues, X is a random variable representing the size of a job, and $\rho = \Lambda\mathbb{E}[X]/c$. $W_{M/M/c}$ is the mean time in queue in an $M/M/c$ queueing system, and $S(\rho)$ and $R(\rho)$ are experimentally derived correction factors, given in Appendix A.2.

Observation 4.7. *Our parallel system with JSQ-Chunk dispatching with level of parallelization k, n cores, $c = n/k$ chunks, total arrival rate Λ, and job size distribution X has the same mean*

response time as a traditional JSQ queueing system with, c queues, a total arrival rate of Λ, and job size distribution $X_k = \frac{X}{s(k)}$.

Observation 4.7 follows from the fact that, under JSQ-Chunk, all the cores within a chunk have the same queue state and receive the same arrivals. This allows us to directly apply the approximation in [70] to analyze JSQ-Chunk as follows:

$$\mathbb{E}[T]^{\text{JSQ-Chunk}}(\Lambda, n, \mathbb{E}[X], k) = \mathbb{E}[T]^{\text{JSQ}}(\Lambda, n/k, \mathbb{E}[X_k]). \tag{4.3}$$



(a) JSQ-Chunk $n = 16$

(b) Random-Chunk $n = 16$

Figure 4.7: Response times under (a) JSQ-Chunk dispatching and (b) Random-Chunk dispatching. In (a), results are show from both analysis and simulation (the jagged lines), which largely overlap. Both graphs also show the results of analysis of EQUI. We assume a speedup curve of Amdahl's law with a parameter of $p = 0.5$ and exponentially distributed job sizes with mean $\mathbb{E}[X] = 1$.

4.5 JSQ-Chunk Converges to EQUI

4.5.1 The Performance of JSQ-Chunk

Figure 4.7 shows the mean response time under JSQ-Chunk (left graph) as a function of system load ρ, as computed using the approximation in (4.3). In addition, we also show results from simulation which lie almost on top of the approximation results. Fortunately, we see that the approximation for JSQ-Chunk suffices to accurately derive k^* values. Analogous results of the analysis of Random-Chunk are shown in the right graph. We find that, for a given load ρ, k^* is generally lower under Random-Chunk dispatching as compared with JSQ-Chunk due to JSQ-Chunk's superior load balancing. Furthermore, we see that the performance of JSQ-Chunk is much closer to that of EQUI than Random-Chunk. While the response time of Random-Chunk does not depend on the total number of cores, it is not clear how JSQ-Chunk will behave as the number of

cores becomes large. We will now see that mean response time under JSQ-Chunk approaches that of EQUI as the system scales.

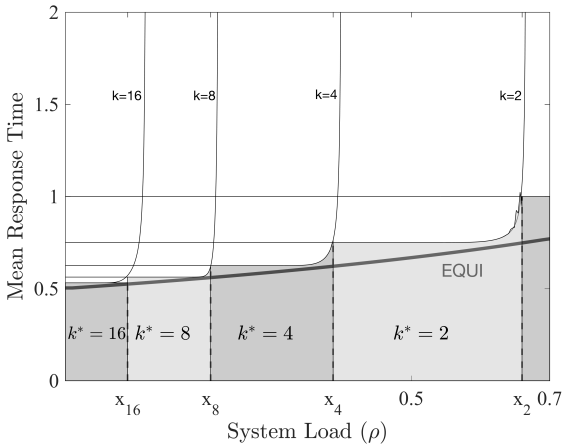


Figure 4.8: Analysis of mean response time under JSQ-Chunk dispatching and EQUI with a large number of cores ($n = 512$). We assume a speedup curve of Amdahl's law with a parameter of $p = 0.5$ mean job size $\mathbb{E}[X] = 1$. Increasing the number of cores narrows the gap between JSQ-Chunk and EQUI, especially at the labeled critical load points.

4.5.2 Why JSQ-Chunk is Close to EQUI

As we saw in Figure 4.7, the mean response time under JSQ-Chunk (with optimal k^*) is close to that under EQUI. Figure 4.8 shows that, as we increase the number of cores to $n = 512$, JSQ-Chunk becomes even closer to EQUI. This is surprising because JSQ-Chunk uses a fixed level of parallelization, while EQUI continuously changes its level of parallelization based on the system state. This phenomenon can be viewed as EQUI choosing an *effective* level of parallelization of k^*. Figure 4.8 also illustrates the *critical load points*, x_{k^*} for values of k^*, at which JSQ-Chunk is indifferent between two choices of k^*. As n increases, these critical load points converge to the instability points for the corresponding choices of k^*, as the curves become more L-shaped. In looking at Figure 4.8, we see that it is just below these critical load points that JSQ-Chunk's performance is closest to the performance of EQUI. The rest of this section is devoted to formally stating and proving this observation (see Theorem 4.9 below).

We first note that, given a fixed value of λ, a fixed mean job size $\mathbb{E}[X]$, and a speedup function s, the choice of the optimal level of parallelization under JSQ-Chunk, k^*, depends only on n, the number of cores in the system. Let $k^*(n)$ denote this optimal level of parallelization given some values of these other system parameters. In particular, note that $\Lambda = \lambda n$ will increase with n, and thus the system load, ρ, remains fixed for all values of n. When n is small, changes in n will have a significant impact on the values of $k^*(n)$, but for sufficiently large values of n, $k^*(n)$ will become constant in n. This is summarized in the following lemma regarding the performance of JSQ-Chunk.

Lemma 4.8. *Given a fixed value of λ, a fixed value of $\mu = \frac{1}{\mathbb{E}[X]}$, and some speedup function s, let $k^*(n)$ denote the optimal level of parallelization under JSQ-Chunk in a system of size n. Let $\mathbb{E}[T]^{\textit{JSQ-Chunk}}$ be the mean response time under JSQ-Chunk with level of parallelization $k^*(n)$. There exists some constant k^*, not dependent on n, such that*

$$\lim_{n\to\infty} k^*(n) = k^*$$

and

$$\lim_{n\to\infty} \mathbb{E}[T]^{\textit{JSQ-Chunk}} = \frac{1}{s(k^*)\mu}.$$

Proof. We can observe that, in this case, the system load $\rho = \frac{\lambda}{\mu}$ is constant (does not depend on n). We know from [35] that for a fixed system load ρ, the probability of queuing under JSQ (and hence also JSQ-Chunk) vanishes as the number of cores becomes large. Thus, the mean response time under JSQ-Chunk with any level of parallelization, k, such that the system is stable, will converge to $\frac{1}{s(k)\mu}$ where $\mu = \frac{1}{\mathbb{E}[X]}$. Since load is fixed, there exists some level of parallelization, k^*, which is the highest value of k for which the JSQ-Chunk system is stable. We know that the mean response time under JSQ-Chunk with level of parallelization k^* converges to $\frac{1}{s(k^*)\mu}$. For any $k > k^*$, the system is unstable. For any $k < k^*$, the mean response time under JSQ-Chunk with level of parallelization k converges to $\frac{1}{s(k)\mu} \geq \frac{1}{s(k^*)\mu}$ since the speedup function is non-decreasing. Thus,

$$\lim_{n\to\infty} \mathbb{E}[T]^{\text{JSQ-Chunk}} = \frac{1}{s(k^*)\mu}$$

as desired. □

We will refer to $\lim_{n\to\infty} k^*(n)$ as k^* for the remainder of the section.

We now want to show that as $n \to \infty$, $\mathbb{E}[T]^{\text{EQUI}} \to \frac{1}{s(k^*)\mu}$ also. That is, *EQUI is behaving as if it was using a fixed level of parallelization of k^*.*

To relate the performance of `EQUI` to k^*, it will be helpful to examine the behavior of `EQUI` at the critical load points. Recall that we can define

$$\Lambda_{k^*} = \frac{\Lambda}{c} = \frac{\lambda n}{c},$$

where $c = \frac{n}{k^*}$ is the number of chunks and

$$\rho_{k^*} = \Lambda_{k^*}\mathbb{E}[X_{k^*}]$$

is the load observed by a chunk of size k^*. We can see from Lemma 4.8 that the critical load points move towards the instability points as n increases. Thus, we define a critical load point under a level of parallelization of k^* to be the point where $\rho_{k^*} = 1$. Observe that

$$\rho_{k^*} = 1 \iff \Lambda = cs(k^*)\mu.$$

We now evaluate `EQUI` at critical load points where $\Lambda = cs(k^*)\mu$. Note that when $k^* = 1$, the mean response time under both `EQUI` and JSQ-Chunk will tend towards infinity. Thus, we only consider cases where $k^* > 1$.

Theorem 4.9. *Given any fixed $k^* > 1$, let $\mathbb{E}[T]^{EQUI}_{\rho_{k^*}=1}$ be the mean response time under* EQUI *when $\rho_{k^*} = 1$. Then,*

$$\lim_{n\to\infty} \mathbb{E}[T]^{EQUI}_{\rho_{k^*}=1} = \frac{1}{s(k^*)\mu} = \lim_{\rho_{k^*}\to 1^-} \lim_{n\to\infty} \mathbb{E}[T]^{JSQ\text{-}Chunk}. \tag{4.4}$$

Proof. In this proof we limit our discussion to exponentially distributed job sizes. However, because EQUI and JSQ-Chunk are insensitive to the job size distribution (see Section 4.3.1 and Section 4.4.2), our proof generalizes to the case where jobs are generally distributed. We can see that the right hand side of this claim follows directly from Lemma 4.8, and we thus proceed to analyzing the mean response time under EQUI.

It will be helpful to start by examining a *threshold chain* as shown in Figure 4.9. Observe that the service rate below the threshold state is μ_{low} and the service rate above the threshold state is μ_{high}. We define

$$\rho_{low} := \frac{\Lambda}{\mu_{low}}, \qquad \rho_{high} := \frac{\Lambda}{\mu_{high}}.$$

We can solve for the mean response time, T, in such a threshold chain, and find that

$$\mathbb{E}[T]^{\text{thresh}} = \frac{1}{\Lambda} \cdot \left(t + \frac{\rho_{low}}{1-\rho_{low}} + \frac{1}{1-\rho_{high}} + \frac{1 + t - t\rho_{high}}{\rho_{high} - 1 + \rho_{low}^t(\rho_{low} - \rho_{high})} \right). \tag{4.5}$$

Constructing an Upper Bound

We now construct a threshold chain which gives an upper bound (UB) on the mean response time under EQUI with arrival rate $\Lambda = cs(k^*)\mu$ (and thus $\rho_{k^*} = 1$) and service rate μ per core. To do this, we set $t = \lceil c(1+\epsilon) \rceil$, $\mu_{low} = s(n)\mu$ and $\mu_{high} = \lceil c(1+\epsilon) \rceil s\left(\frac{n}{\lceil c(1+\epsilon)\rceil}\right)\mu$ for any $\epsilon > 0$. Note that all departure rates in the UB chain are lower than those of EQUI (see Figure 4.5), and thus this chain provides an upper bound on the mean response time under EQUI. Our UB chain has:

$$\rho_{low} = \frac{\Lambda}{s(n)\mu} = \frac{cs(k^*)\mu}{s(n)\mu}$$

and

$$\rho_{high} = \frac{\Lambda}{\lceil c(1+\epsilon) \rceil s\left(\frac{n}{\lceil c(1+\epsilon)\rceil}\right)\mu} = \frac{cs(k^*)\mu}{\lceil c(1+\epsilon) \rceil s\left(\frac{n}{\lceil c(1+\epsilon)\rceil}\right)\mu}.$$

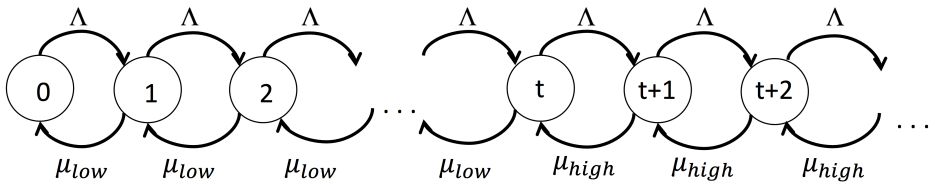


Figure 4.9: A threshold chain with threshold state t and arrival rate Λ. For all states $i \leq t$, the service rate is μ_{low} and for all states $i > t$, the service rate is μ_{high}.

First note that by Lemma 4.2,

$$s(n) < cs(k^*) < \lceil c(1+\epsilon)\rceil s\left(\frac{n}{\lceil c(1+\epsilon)\rceil}\right).$$

Thus $\rho_{high} < 1 < \rho_{low}$. We now apply (4.5) to see that

$$\lim_{n\to\infty} \mathbb{E}[T]^{\text{UB}} = \lim_{n\to\infty} \frac{1}{\Lambda} \cdot \Bigg(\lceil c(1+\epsilon)\rceil + \frac{\rho_{low}}{1-\rho_{low}} + \frac{1}{1-\rho_{high}} + \frac{1+\lceil c(1+\epsilon)\rceil(1-\rho_{high})}{\rho_{high} - 1 + \rho_{low}^{\lceil c(1+\epsilon)\rceil}(\rho_{low}-\rho_{high})}\Bigg).$$

We can see that

$$\lim_{n\to\infty} \frac{1}{\Lambda} \cdot \frac{\rho_{low}}{1-\rho_{low}} = \lim_{n\to\infty} \frac{1}{\Lambda} \cdot \frac{1/\mu_{low}}{1/\Lambda - 1/\mu_{low}} = 0,$$

since $s(n)$ is bounded and μ_{low} therefore converges to a positive constant. Furthermore, since ρ_{high} converges to a constant less than 1,

$$\lim_{n\to\infty} \frac{1}{\Lambda} \cdot \frac{1}{1-\rho_{high}} = 0.$$

Finally, we see that

$$\lim_{n\to\infty} \frac{1+\lceil c(1+\epsilon)\rceil(1-\rho_{high})}{\Lambda\left(\rho_{high} - 1 + \rho_{low}^{\lceil c(1+\epsilon)\rceil}(\rho_{low}-\rho_{high})\right)} = 0,$$

since the numerator grows linearly in n and the denominator grows exponentially in n. Thus,

$$\begin{aligned}\lim_{n\to\infty} \mathbb{E}[T]^{\text{UB}} &= \lim_{n\to\infty} \frac{1}{\Lambda} \cdot \lceil c(1+\epsilon)\rceil \\ &= \lim_{n\to\infty} \frac{1}{\Lambda} \cdot (c(1+\epsilon) + o(n)) \\ &= (1+\epsilon)\frac{1}{s(k^*)\mu}.\end{aligned}$$

Constructing a Lower Bound

Our argument for constructing a lower bound (LB) is largely the same. We again assume $\Lambda = cs(k^*)\mu$ (and thus $\rho_{k^*} = 1$) and we set $t = \lfloor c(1-\epsilon)\rfloor$, $\mu_{low} = \lfloor c(1-\epsilon)\rfloor s\left(\frac{n}{\lfloor c(1-\epsilon)\rfloor}\right)$, and $\mu_{high} = n\mu$. Note that all departure rates in the LB chain are higher than those of EQUI, and thus this chain provides a lower bound on the mean response time under EQUI. We now have

$$\rho_{low} = \frac{\Lambda}{\lfloor c(1-\epsilon)\rfloor s\left(\frac{n}{\lfloor c(1-\epsilon)\rfloor}\right)\mu} = \frac{cs(k^*)\mu}{\lfloor c(1-\epsilon)\rfloor s\left(\frac{n}{\lfloor c(1-\epsilon)\rfloor}\right)\mu}$$

and

$$\rho_{high} = \frac{\Lambda}{n\mu} = \frac{cs(k^*)\mu}{n\mu} = \frac{s(k^*)\mu}{k^*\mu}.$$

By Lemma 4.2 we again see that $\rho_{high} < 1 < \rho_{low}$. We apply (4.5) to see that

$$\lim_{n\to\infty} \mathbb{E}[T]^{\text{LB}} = \frac{1}{\Lambda} \cdot \left(\lim_{n\to\infty} \lfloor c(1-\epsilon) \rfloor + \frac{\rho_{low}}{1-\rho_{low}} + \frac{1}{1-\rho_{high}} + \frac{1 + \lfloor c(1-\epsilon) \rfloor (1+\rho_{high})}{\rho_{high} - 1 + \rho_{low}^{\lfloor c(1-\epsilon) \rfloor}(\rho_{low} - \rho_{high})} \right).$$

Since ρ_{low} converges to a constant greater than 1 and ρ_{high} is a constant,

$$\lim_{n\to\infty} \frac{1}{\Lambda} \cdot \frac{\rho_{low}}{1-\rho_{low}} = 0 \text{ and } \lim_{n\to\infty} \frac{1}{\Lambda} \cdot \frac{1}{1-\rho_{high}} = 0.$$

Finally, we see that

$$\lim_{n\to\infty} \frac{1 + \lfloor c(1-\epsilon) \rfloor (1+\rho_{high})}{\Lambda \left(\rho_{high} - 1 + \rho_{low}^{\lfloor c(1-\epsilon) \rfloor}(\rho_{low} - \rho_{high}) \right)} = 0,$$

since the numerator grows linearly in n and the denominator grows exponentially in n. Thus,

$$\begin{aligned} \lim_{n\to\infty} \mathbb{E}[T]^{\text{UB}} &= \lim_{n\to\infty} \frac{1}{\Lambda} \cdot \lfloor c(1-\epsilon) \rfloor \\ &= \lim_{n\to\infty} \frac{1}{\Lambda} \cdot (c(1-\epsilon) - o(n)) \\ &= (1-\epsilon) \frac{1}{s(k^*)\mu}. \end{aligned}$$

We have therefore shown that

$$(1-\epsilon) \frac{1}{s(k^*)\mu} \leq \lim_{n\to\infty} \mathbb{E}[T]^{\text{EQUI}}_{\rho_{k^*}=1} \leq (1+\epsilon) \frac{1}{s(k^*)\mu}$$

for any $0 < \epsilon \leq 1$ and we have thus shown (4.4) as desired. □

4.6 Conclusion

This chapter introduces the question of how to allocate cores to jobs in a stochastic model of a multicore machine where all jobs have sublinear speedup functions. In particular, we examine the case where job sizes are unknown and exponentially distributed, and all jobs follow the same speedup function, $s(k)$. In the case where all jobs are malleable, we prove that the well-known `EQUI` policy minimizes mean response time by maximizing instantaneous system efficiency. Intuitively, we find that when the scheduler cannot differentiate between jobs on the basis of their remaining sizes nor their speedup functions, the optimal policy is to simply complete work at the fastest rate possible.

While the EQUI policy works well when jobs are malleable, this may not always be the case. Hence, we also study how to choose a good scheduling policy when jobs are only moldable. While it seems that the ability to dynamically change the number of cores allocated to each job should provide a massive benefit with respect to mean response time, we find that one can still achieve near-optimal performance by using the optimal *fixed* level of parallelization, k^*. Hence, if jobs are moldable but not malleable, the correct choice of scheduling policy can still result in near-optimal performance. We show how to analytically determine k^* as a function of system load, the speedup curve, the job size distribution, the number of cores, and the dispatching policy.

The results of this chapter show the potential for great improvement in the performance of systems where the user is tasked with reserving the cores they intend to use. For example, consider a multicore server where each user greedily tries to use all the cores on a machine in order to complete their own job as fast as possible. In a vacuum, a job might benefit from running on all of the available cores, but doing so results in a highly inefficient use of resources and is therefore not an effective strategy for minimizing the overall mean response time across *many* jobs. The results of this chapter show that, in fact, giving large allocations to individual jobs is actually the *worst* allocation to use in terms of its overall effect on mean response time. When job sizes are unknown to the system, we are better off allowing the system to make scheduling decisions that maximize system efficiency by sharing cores equally.

The work described in this chapter also created a new line of theory research within the field of parallel scheduling by performing the first average-case analysis of scheduling policies for parallelizable jobs. Using a stochastic model of parallel jobs with sizes and speedup functions, we provided the first description of policies that could perform optimally or near-optimally when scheduling parallelizable jobs. These results provide interesting context to the previously known worst-case results about the EQUI policy. Prior worst-case work tells us that EQUI is far from optimal in the worst case, but that EQUI may still perform as well as possible given the adversarial assumptions made in the worst-case model. In this chapter, we pinpoint exactly why EQUI can be a good policy in some settings, and give sufficient conditions for EQUI to be optimal with respect to mean response time.

The flip side of this observation is that, when job sizes are known to the system, or some jobs are known to be more parallelizable than others, we can see that the arguments made in this chapter will fall apart. However, while the worst-case theory literature continues to advocate for the use of EQUI in these cases, our models will suggest new, more complex policies which will greatly outperform EQUI. In the subsequent chapters, we will see that maximizing the instantaneous system efficiency is no longer sufficient for minimizing mean response time when jobs can be differentiated on the basis of their sizes and speedup functions. We will show how an optimal policy can reduce the mean response time across jobs by sacrificing instantaneous system efficiency in order to prioritize shorter jobs and preserve the future efficiency of the system.

Chapter 5

Scheduling With Job Sizes

5.1 Introduction

In Chapter 4 we found that under certain circumstances, maximizing instantaneous system efficiency results in a good scheduling policy. Specifically, when all jobs follow the same speedup function and have exponentially distributed sizes, the optimal policy, EQUI, maximizes instantaneous system efficiency. However, in many important real-world systems, the size of each job will be known to the system. Hence, in this chapter, we will relinquish the assumption that job sizes are unknown, and show that an optimal policy can greatly outperform an efficiency-maximizing policy with respect to mean response time when job sizes are known to the system.

In a wide variety of systems, there is sufficient information to make highly accurate predictions about the remaining sizes of the jobs in the system. However, modern systems often fail to grasp the importance of this size information when making scheduling decisions. For example, consider analytics databases which process parallelizable queries. These systems often rely on cost-based query optimizers [21] which are designed to estimate the inherent size of each query. Nonetheless, many databases either process jobs in first-come-first-served (FCFS) order [75] or use EQUI [58]. Similarly, machine learning frameworks process jobs which require completing a known number of iterations of stochastic gradient descent. Here, the number of iterations required is proportional to the size of a job. However, both real-world systems [67] and academic schedulers [82] do not use job sizes to make their scheduling decisions. We will see that the scheduling policies used in these state-of-the-art systems can be dramatically improved by properly leveraging the available job size information.

5.1.1 Scheduling Tradeoffs

This chapter considers the case where jobs sizes are exactly known to the system, but jobs still cannot be differentiated on the basis of their speedup function. This gives the scheduler the ability to see, at every moment in time, which jobs have shorter remaining sizes.

In this case, we can consider how the availability of job size information changes the optimality results from the previous chapter. Figure 5.1 shows an illustration of the tradeoffs a policy must

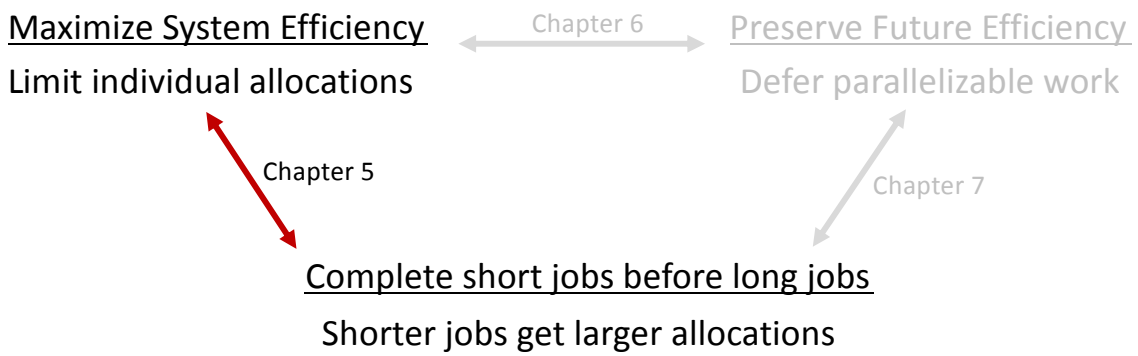


Figure 5.1: When job sizes are known to the system, a scheduling policy must balance a tradeoff between maximizing instantaneous system efficiency and favoring short jobs.

balance, updated to reflect the availability of job size information. We see that, compared to the policies of Chapter 4, policies in this chapter will have to balance a new tradeoff between maximizing instantaneous system efficiency and favoring short jobs over long jobs.

Key Insight

In general, the mean response time across a set of jobs will be reduced when shorter jobs are completed before longer jobs. In fact, in the case where we have a single-core system, the optimal policy for allocating that single core to jobs of known sizes is the Shortest Remaining Processing Time (SRPT) policy, which allocates the entire core to the job with smallest remaining size [93]. Additionally, in a multicore system where jobs are perfectly parallelizable, we note that the system's many cores really function as a single large processor which can be arbitrarily allocated without any loss in instantaneous system efficiency. Hence, if jobs are perfectly parallelizable, SRPT is optimal with respect to mean response time.

However, as we saw in Chapter 4, system efficiency becomes a concern when jobs are only *partially* parallelizable. If we were to employ SRPT when scheduling partially parallelizable jobs, allocating all cores to the shortest job in the system, the loss in system efficiency could be arbitrarily high. The question, then, is how to weigh the benefit of favoring a short job against the loss of instantaneous system efficiency that results from allocating more cores to one individual job. In particular, we have shown that `EQUI` maximizes instantaneous system efficiency, and SRPT gives strict priority to short jobs, but neither of these policies is optimal in general when job sizes are known. We will show in this chapter that an optimal policy must split the difference between `EQUI` and SRPT, figuring out how to favor short jobs while still maintaining the overall efficiency of the system.

5.1.2 Why Core Allocation is Counter-intuitive

Consider a simple system with $n = 10$ cores and $M = 2$ identical jobs of size 1, where $s(k) = k^{.5}$, and where we wish to minimize mean response time (which is equivalent to slowdown in this case). A queueing theorist might look at this problem and say that to minimize response time, we should

use the SRPT policy, first allocating all cores to job one and then all cores to job two. However, this causes the system to be very inefficient. Another intuitive argument would be that, since everything in this system is symmetric, the optimal allocation should be symmetric. Hence, one might think to allocate half the cores to job one and half the cores to job two. While this equal allocation does maximize the efficiency of the system by ensuring that neither job receives too many cores, it does *not* minimize their mean response time. Theorem 5.9 will show that the optimal policy in this case is to allocate 75% of the cores to job one and 25% of the cores to job two. In our simple, symmetric system, the optimal allocation is very asymmetric! Note that this asymmetry is *not* an artifact of the form of the speedup function used. For example, if we had instead assumed that the speedup function s was Amdahl's Law [46] with a parallelizable fraction of $f = .9$, the optimal split is to allocate 63.5% of the system to one of the jobs. Instead, the optimal policy balances the tradeoff between using an efficient equal allocation and using the inefficient SRPT policy which favors small jobs. If we imagine a set of M arbitrarily sized jobs, one suspects that the optimal policy again favors shorter jobs, but it is not obvious how to calculate the exact allocations for this policy.

5.1.3 Why Finding the Optimal Policy is Hard

At first glance, solving for the optimal policy seems amenable to classical optimization techniques. However, naive application of these techniques would require solving $M!$ optimization problems, each consisting of $O(M^2)$ variables and $O(M)$ constraints. Furthermore, although these techniques could produce the optimal policy for a single problem instance, it is unlikely that they would yield a closed form solution. We instead advocate for finding a closed form solution for the optimal policy, which allows us to build intuition about the underlying dynamics of the system.

To understand the source of this complexity, we will first consider the problem of minimizing the total response time of a set of just two jobs of sizes x_1 and x_2 respectively. If we assume that the optimal policy completes job 2 first, we can write an expression for the total response time of the two jobs as follows:

$$T_1 + T_2 = \frac{2 \cdot x_2}{s(\theta_2(0))} + \frac{\left(x_1 - \frac{s(\theta_1(0)) \cdot x_2}{s(\theta_2(0))}\right)}{s(n)}.$$

The first term in this expression describes the total response time accrued between time 0 and time T_2. The duration of this period is $\frac{x_2}{s(\theta_2(0))}$, and because there are two jobs in the system during this period, the total response time accrued during the period is $\frac{2 \cdot x_2}{s(\theta_2(0))}$. The second term encodes the remaining time required to finish job 1 after job 2 completes. This expression assumes that job allocations only change at the time of a departure, which we will prove formally in Theorem 5.1. More interestingly, however, this expression *implicitly encodes* the idea that the policy finishes job 2 before job 1. If, on the other hand, job 1 finishes first, the expression would change to

$$T_1 + T_2 = \frac{2 \cdot x_1}{s(\theta_1(0))} + \frac{\left(x_2 - \frac{s(\theta_2(0)) \cdot x_1}{s(\theta_1(0))}\right)}{s(n)}.$$

This has two main implications. First, it is possible that the allocation policy which minimizes a particular expression for total response time does not complete jobs in the order encoded in the expression. In this case, the value of the expression is meaningless – it does not equal the total response time of the jobs under the computed policy. Hence, to find the optimal policy with respect to a particular completion order, one must minimize the corresponding expression for total response time subject to constraints which maintain the completion order of the allocation policy. Second, because the objective function depends on the completion order of the jobs, there is no single function to try to minimize to recover an optimal policy. Instead, we are interested in the *global* minimum across all of the $M!$ completion orders, each of which has its own *local* minimum that can be obtained by constrained minimization.

When there are only two jobs, it is tractable to directly solve for the optimal policy via known constrained optimization methods. One can use Lagrange multipliers with two constraints – one to ensure a valid allocation policy and one to enforce the completion order. One must solve two such optimization problems corresponding to each of the potential completion orders. However, given a set of M jobs with $M!$ possible completion orders, this naive approach would require solving $M!$ optimization problems, each consisting of $O(M^2)$ variables used to define an allocation policy and $O(M)$ constraints. Even if some of this complexity could be elided by, for instance, proving the completion order of the optimal policy (see Theorem 5.2), Lagrange multipliers are unlikely to emit a closed form given such a complex objective function and large set of constraints.

Additionally, one may think to apply classical techniques from the deterministic scheduling literature. Specifically, one technique is to show that the problem of minimizing weighted response time can be reduced to solving a linear program where the feasible region is a polymatroid [12, 107]. The polymatroid technique can be used to show that a system is *indexable* (the optimal policy is a simple priority policy) or even *decomposable* (job priorities do not depend on the other jobs in the system). This technique can be used to provide a greedy algorithm for obtaining the optimal policy. Unfortunately, the polymatroid technique is not straightforward to apply in our case.

Specifically, one generally proceeds by showing that a problem satisfies the so-called *generalized conservation laws* [12, 107]. However, these conservation laws require scheduling policies which are *work-conserving*, meaning the total rate at which the system completes work remains constant over time. Allocation policies in our setting are not work conserving in general, and the allocations used by the optimal policy do not maintain a constant work rate (see Figure 5.3). Additionally, due to the form of the speedup function $s(k) = k^p$, the region of achievable weighted response times is not a polytope. Furthermore, the optimal policy we derive in Theorem 5.9 will show that our problem is not decomposable for the case of minimizing weighted response time where weights favor small jobs. While there may exist a reduction of our problem that would allow one to establish some form of conservation laws, it is far from obvious what the "conserved quantity" would be for our problem which would allow for application of these techniques.

Prior work has investigated the related problem of scheduling flows in networks where the system capacity evolves over time [88]. However, this work derives an optimal policy only in the case when the work rates of each job are constrained to lie in a series of polymatroids that describe the system capacity at each moment in time. This assumption does not hold in our setting where the work rate of each job is defined by the speedup function $s(k) = k^p$.

Hence, standard techniques do not appear to be compatible with our goal of finding the optimal policy with respect to weighted response time.

5.1.4 Contributions

Given a set of jobs with known sizes, this chapter presents the optimal allocation policy with respect to mean response time. We call this policy, which balances the tradeoff between EQUI and SRPT, *high efficiency SRPT* (heSRPT). We generalize heSRPT to optimize for a broad class of weighted response time metrics which includes the *mean slowdown* metric (see Section 5.2). Our analysis considers the case where all jobs are present in the system at time 0. While this is a common case in practice, it is also interesting to consider an online version of the problem, where jobs arrive over time. Unfortunately, it has been shown that, in general, no optimal policy exists to minimize mean slowdown in the online case [6]. We demonstrate that heSRPT, which is optimal in the offline case, provides an excellent heuristic policy, Adaptive-heSRPT, for minimizing mean slowdown in the online case. Adaptive-heSRPT often performs an order of magnitude better than policies previously suggested in the literature.

In Section 5.3, we provide a complete overview of our results, but here we summarize the main contributions of this chapter.

- To derive the optimal allocation function, we first develop a new technique in Section 5.4 to reduce the dimensionality of the optimization problem. This dimensionality reduction leverages two key properties of the optimal policy. First, in Section 5.4.1, we show that the optimal policy must complete jobs in shortest-job-first order. Then, in Section 5.4.2, we prove the scale-free property of the optimal policy which illustrates the optimal substructure of the optimal policy. These insights reduce the problem of solving $M!$ optimization problems of $O(M^2)$ variables and $O(M)$ constraints to the problem of solving one, unconstrained optimization problem of exactly M variables.
- In Section 5.4.3, we solve our simplified optimization problem to derive the first closed form expression for the optimal allocation of n cores to M jobs which minimizes weighted response time when weights favor small jobs. At any moment in time t we define

  $$\boldsymbol{\theta}^*(t) = (\theta_1^*(t), \theta_2^*(t), \ldots, \theta_M^*),$$

  where $\theta_i^*(t)$ denotes the fraction of the n cores allocated to job i at time t. Note that $\theta_i^*(t)$ does not depend on n. Our optimal allocation balances the size-awareness of SRPT and the high efficiency of EQUI. We thus refer to our optimal policy as *high efficiency SRPT* (heSRPT) (see Theorem 5.9). We also provide a closed form expression for the weighted response time under heSRPT (see Theorem 5.10).
- In Section 5.5.2, we numerically compare the optimal policies with respect to both mean slowdown and mean response time to other heuristic policies proposed in the literature and show that our optimal policies significantly outperform these competitors.
- Finally, Section 5.5.3 turns to the online setting where jobs arrive over time. We propose an online version of heSRPT called *Adaptive-heSRPT* which uses the allocations from heSRPT to

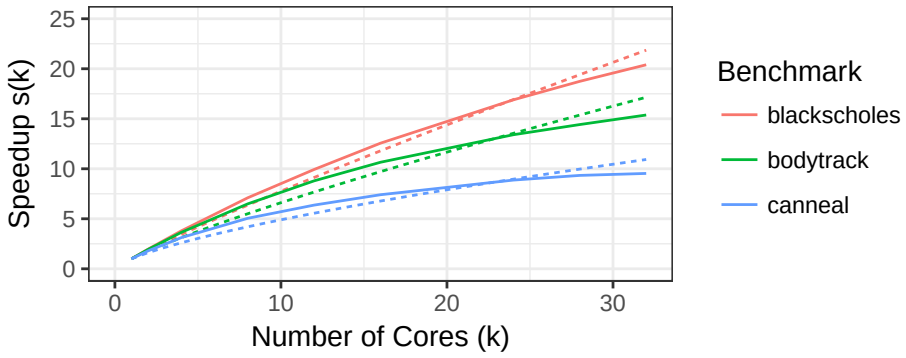


Figure 5.2: Various speedup functions of the form $s(k) = k^p$ (dotted lines) which have been fit to real speedup curves (solid lines) measured from jobs in the PARSEC-3 parallel benchmarks[109]. The three jobs, blackscholes, bodytrack, and canneal, are best fit by the functions where $p = .89$, $p = .82$, and $p = .69$ respectively.

recalculate core allocations on every arrival and departure. Adaptive-heSRPT significantly outperforms competitor policies from the literature in simulation, often by an order of magnitude.

5.2 Our Model

Our model assumes there are n identical cores which must be allocated to M parallelizable jobs. All M jobs are present at time $t = 0$. Job i is assumed to have some inherent size x_i where, without loss of generality (WLOG),

$$x_1 \geq x_2 \geq \ldots \geq x_M.$$

In general we will assume that all jobs follow the *same* speedup function, $s : \mathbb{R}^+ \to \mathbb{R}^+$, which is of the form

$$s(k) = k^p$$

for some $0 < p < 1$.

We choose the family of functions $s(k) = k^p$ because they are (i) sublinear and concave, (ii) can be fit to a variety of empirically measured speedup functions (see Figure 5.2) [109], and (iii) simplify the analysis in this chapter. Note that [46] assumes $s(k) = k^p$ where $p = 0.5$ and explicitly notes that using speedup functions of another form does not significantly impact their results.

In this chapter, we assume that all jobs are malleable, and hence the number of cores allocated to a job can change over the course of the job's lifetime.

In general, we assume that there is some policy, P, which allocates cores to jobs at every time, t. When describing the state of the system, we will use $m^P(t)$ to denote the number of remaining jobs in the system at time t, and $x_i^P(t)$ to denote the remaining size of job i at time t. To describe the state of each job at time t, let $W_i^P(t)$ denote the total amount of work done by policy P on job i by time t. Let $W_i^P(t_1, t_2)$ be the amount of work done by policy P on job i on the interval $[t_1, t_2)$. That is,

$$W_i^P(t_1, t_2) = W_i^P(t_2) - W_i^P(t_1).$$

We denote the completion time of job i under policy P as T_i^P. When the policy P is implied, we will drop the superscript.

We will assume that the number of cores allocated to a job need not be discrete.

For each job, i, we define a corresponding *weight* $w_i > 0$. We then define the *weighted response time* for a set of jobs under policy P, F^P, to be

$$F^P = \sum_{i=1}^{M} w_i \cdot T_i.$$

By manipulating the weights associated with each job, one can derive metrics with different intuitive meanings. We say that weights *favor small jobs* if larger weights are always assigned to smaller jobs. That is, if

$$w_1 \leq w_2 \leq \ldots \leq w_M.$$

The class of weighted response time metrics where weights favor small jobs includes several popular metrics. First, note that the objective of minimizing mean response time is equivalent to minimizing weighted response time when $w_i = 1$ for each job, i. Another important metric that belongs to this class of metrics is *mean slowdown*. The mean slowdown of a policy P, $\overline{S}^P$ is defined as

$$\overline{S}^P = \frac{1}{M} \cdot \sum_{i=1}^{M} \frac{T_i^P}{x_i/s(n)}.$$

The objective of minimizing mean slowdown is equivalent to minimizing weighted response time when $w_i = \frac{1}{x_i/s(n)}$. Clearly, weights favor small jobs in this setting. Slowdown is a measure of how a job was interfered with by other jobs in the system, and is often the metric of interest in the theoretical parallel scheduling literature (where it is also called stretch) [68], as well as the HPC community (where it is called expansion factor) [50].

For the sake of generality, this chapter considers the problem of finding the policy P^* which minimizes the weighted response time for a set of M jobs when weights favor small jobs. Let F^* be the weighted response time under this optimal policy. We will denote the allocation function of the optimal policy as $\boldsymbol{\theta}^*(t)$. Similarly, we let $m^*(t)$, $x_i^*(t)$, $W_i^*(t)$ and T_i^* denote the corresponding quantities under the optimal policy.

5.3 Overview of Our Results

Our goal is to determine the optimal allocation of cores to jobs at every time, t, in order to minimize the weighted response time of a set of M jobs of known size where weights favor small jobs. We derive a closed form for the optimal allocation function

$$\boldsymbol{\theta}^*(t) = (\theta_1^*(t), \theta_2^*(t), \ldots, \theta_M^*(t))$$

which defines the allocation for each job at any moment in time, t, that minimizes weighted response time. We now provide an overview of the main theorems from this chapter.

We begin by showing that the optimal policy completes jobs in *Shortest-Job-First* (SJF) order. The proof of this theorem uses an interchange argument to show that any policy which violates the SJF completion order can be improved. Because jobs follow a concave speedup function, a standard interchange proof fails. Our proof requires careful accounting, and interchanges cores between many jobs simultaneously. This claim is stated in Theorem 5.2.

Theorem 5.2 (Optimal Completion Order)**.** *The optimal policy completes jobs in Shortest-Job-First (SJF) order:*

$$M, M-1, M-2, \ldots, 1.$$

Since jobs are completed in SJF order, we can also conclude that, at time t, the jobs left in the system are specifically jobs $1, 2, \ldots, m(t)$. Theorem 5.2 is proven in Section 5.4.1.

Theorem 5.2 does not rely on the specific form of the speedup function – it holds if $s(k)$ is any increasing, strictly concave function.

Besides completion order, the other key property of the optimal allocation that we exploit is the *scale-free property*. Our scale-free property states that for any job, i, job i's allocation relative to jobs completed after job i (jobs larger than job i) is constant throughout job i's lifetime. This property is stated formally in Theorem 5.5.

Theorem 5.5 (Scale-free Property)**.** *For any job, i, there exists a constant c_i such that, for all $t < T_i^*$*

$$\frac{\theta_i^*(t)}{\sum_{j=1}^{i} \theta_j^*(t)} = c_i.$$

The scale-free property has an intuitive interpretation. One can imagine the optimal policy as starting with some optimal allocation function $\boldsymbol{\theta}^*(0)$, which gives a $\theta_i^*(0)$ fraction of the cores to job i at time 0. When job M completes, it will leave behind a set of $\theta_M^*(0) \cdot n$ newly idle cores. The scale-free property tells us that the new value of the optimal allocation is found by reallocating these idle cores to each job $i < M$ in proportion to $\theta_i^*(0)$. That is, of the $\theta_M^*(0) \cdot n$ newly available cores, job i should receive

$$\frac{\theta_i^*(0)}{\sum_{j=1}^{M-1} \theta_j^*(0)} \cdot \theta_M^*(0) \cdot n$$

additional cores.

An example of the scale-free property in action can be seen in Figure 5.3. Here, the optimal allocation policy gives all 3 jobs a non-zero allocation a time $t = 0$. When job 3 completes, its cores are reallocated to jobs 1 and 2. The allocations of job 1 and job 2 both increase, but the ratio of these jobs' allocations remains constant.

The scale-free property provides an optimal substructure that we exploit to reduce the dimensionality of our optimization problem. For example, consider the case where the optimal allocation function is known for a set of $M-1$ jobs, and we wish to additionally consider adding an Mth job. If we first decide how many cores to allocate to job M, the scale-free property tells us that stealing these cores from the other $M-1$ jobs in proportion to their existing allocations will produce the optimal policy. Hence, knowing the optimal policy for a set of $M-1$ jobs reduces the problem of finding the optimal policy for a set of M jobs to a single variable optimization problem. Theorem 5.5 is proven in Section 5.4.2.

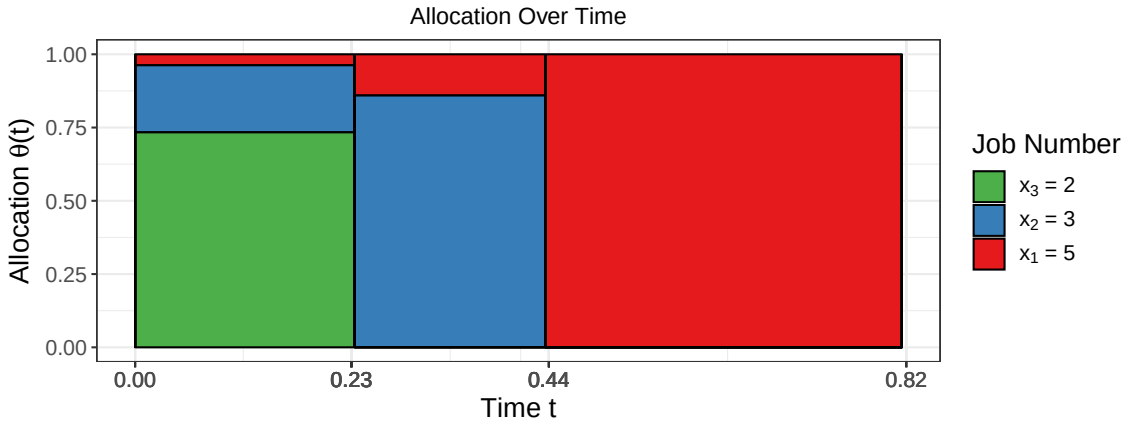


Figure 5.3: Allocations made by the optimal allocation policy with respect to mean slowdown, heSRPT, completing a set of $M = 3$ jobs where the speedup function is $s(k) = k^{.5}$ and $n = 100$. Job 3 completes at time $t = .23$, job 2 completes at time $t = .44$ and job 1 completes at time $t = .82$. Jobs are finished in Shortest Job First order. Rather than allocating the whole system to the shortest job, heSRPT optimally shares the system between all active jobs.

Note that, while we state the scale-free property above for the optimal policy, the scale-free property actually holds for a more a general class of policies. For any given completion order of the jobs, the policy which minimizes weighted response time while completing jobs in the given order will obey the scale-free property.

We are now finally ready to state Theorem 5.9, which provides the allocation function for the optimal allocation policy which minimizes weighted response time when weights favors small jobs.

Theorem 5.9 (Optimal Allocation Function)**.** *At time t, when $m(t)$ jobs remain in the system,*

$$\theta_i^*(t) = \begin{cases} \left(\frac{z(i)}{z(m(t))}\right)^{\frac{1}{1-p}} - \left(\frac{z(i-1)}{z(m(t))}\right)^{\frac{1}{1-p}} & 1 \le i \le m(t) \\ 0 & i > m(t) \end{cases}$$

where

$$z(k) = \sum_{i=1}^{k} w_i.$$

We refer to the optimal allocation policy which uses $\boldsymbol{\theta}^*(t)$ as heSRPT.

Theorem 5.9 is proven in Section 5.4.3. Given the optimal allocation function $\boldsymbol{\theta}^*(t)$, we can also explicitly compute the optimal weighted response time for any set of M jobs. This is stated in Theorem 5.10.

Theorem 5.10 (Optimal Weighted Response Time)**.** *Given a set of M jobs of size $x_1 \ge x_2 \ge \ldots \ge x_M$, the weighted response time, F^*, under the optimal allocation policy $\boldsymbol{\theta}^*(t)$ is given by*

$$F^* = \frac{1}{s(n)} \sum_{k=1}^{M} x_k \cdot \left[z(k)^{\frac{1}{1-p}} - z(k-1)^{\frac{1}{1-p}}\right]^{1-p}$$

where

$$z(k) = \sum_{i=1}^{k} w_i.$$

Note that the optimal allocation policy biases its allocations towards shorter jobs, but does not give strict priority to these jobs in order to maintain the overall efficiency of the system. That is,

$$0 < \boldsymbol{\theta}_1^*(t) < \boldsymbol{\theta}_2^*(t) < \ldots < \boldsymbol{\theta}_{m(t)}^*(t).$$

This is illustrated in Figure 5.3, where the smallest remaining jobs always get the largest allocations under the optimal policy. We refer to the optimal policy derived in Theorem 5.9 as *High Efficiency Shortest-Remaining-Processing-Time* or *heSRPT*.

Section 5.5.1 discusses how to apply Theorems 5.9 and 5.10 in order to optimize both the mean slowdown and mean response time metrics. These applications are summarized in Corollary 5.11.

Corollary 5.11 (Optimal Mean Slowdown and Mean Response Time)**.** *If we define*

$$w_i = \frac{1}{x_i/s(n)} \quad \textit{and thus} \quad z(k) = \sum_{i=1}^{k} \frac{1}{x_i/s(n)},$$

Theorem 5.9 yields the optimal policy with respect to mean slowdown and Theorem 5.10 yields the optimal total slowdown.

If we define

$$w_i = 1 \quad \textit{and thus} \quad z(k) = k,$$

Theorem 5.9 yields the optimal policy with respect to mean response time and Theorem 5.10 yields the optimal total response time.

The remainder of Section 5.5 is devoted to the development of Adaptive-heSRPT, an online version of heSRPT.

5.4 Minimizing Weighted Response Time

5.4.1 The Optimal Completion Order

To determine the optimal completion order of jobs, we first show that the optimal allocation function remains constant between job departures. This implies that the optimal allocation has M decision points where new allocations must be determined.

Theorem 5.1. *Consider any two times t_1 and t_2 where, WLOG, $t_1 < t_2$. Let $m^*(t)$ denote the number of jobs in the system at time t under the optimal policy. If $m^*(t_1) = m^*(t_2)$ then*

$$\boldsymbol{\theta}^*(t_1) = \boldsymbol{\theta}^*(t_2).$$

Proof. Consider any time interval $[t_1, t_2]$ during which no job departs the system, and hence $m^*(t_1) = m^*(t_2)$. Assume for contradiction that under the optimal policy $\boldsymbol{\theta}^*(t_1) \neq \boldsymbol{\theta}^*(t_2)$. To

produce a contradiction, we will show that the weighted response time under the optimal policy can be improved by using a *constant* allocation during the time interval $[t_1, t_2]$. The constant allocation we use is equal to the average value of $\boldsymbol{\theta}^*(t)$ during the interval $[t_1, t_2]$.

Specifically, consider the allocation function $\overline{\boldsymbol{\theta}^*(t)}$ where

$$\overline{\boldsymbol{\theta}^*(t)} = \begin{cases} \frac{1}{t_2 - t_1} \int_{t_1}^{t_2} \boldsymbol{\theta}^*(T) dT & \forall t_1 \leq t \leq t_2 \\ \boldsymbol{\theta}^*(t) & \text{otherwise.} \end{cases}$$

Note that $\overline{\boldsymbol{\theta}^*(t)}$ is constant during the interval $[t_1, t_2]$. Furthermore, because $\sum_{i=1}^{m(t)} \theta_i^*(t) \leq 1$ for any time t, $\sum_{i=1}^{m(t)} \overline{\theta_i^*(t)} \leq 1$ at every time t as well, and $\overline{\boldsymbol{\theta}^*(t)}$ is therefore a feasible allocation function. Because the speedup function, s, is a strictly concave function, Jensen's inequality gives

$$\int_{t_1}^{t_2} s\left(\overline{\boldsymbol{\theta}^*(t)}\right) dt \geq \int_{t_1}^{t_2} s\left(\boldsymbol{\theta}^*(t)\right) dt$$

and for at least one job, i, the above inequality will be strict. Hence, the residual size of each job under the allocation function $\overline{\boldsymbol{\theta}^*(t)}$ at time t_2 is at most the residual size of that job under $\boldsymbol{\theta}^*(t)$ at time t_2, and the residual time of job i is strictly less under the new policy at time t_2. This guarantees that no jobs will have its response time increased by this transformation and at least one job will have its response time decreased by this transformation. We have thus constructed a policy with a lower weighted response time than the optimal policy, a contradiction. □

For the rest of the chapter, we will therefore only consider allocation functions which change exclusively at departure times.

We can now show that the optimal policy will complete jobs in Shortest-Job-First order.

Theorem 5.2 (Optimal Completion Order)**.** *The optimal policy completes jobs in Shortest-Job-First (SJF) order:*

$$M, M-1, M-2, \ldots, 1.$$

Proof. Assume the contrary, that the optimal policy does not follow the SJF completion order. This means there exist two jobs α and β such that $x_\alpha < x_\beta$ but $T_\beta^* < T_\alpha^*$. Note that any number of completions may occur between job α and job β.

If the allocation to job α had always been at least as large as the allocation to job β, it is easy to see that job α would have finished first. Hence, we can identify some intervals of time where: (i) the allocations to job α and job β are constant because there are no departures and (ii) the allocation to job β is higher than the allocation to job α. Let I be the smallest set of disjoint intervals such that for every $[t_1, t_2) = i \in I$

$$\theta_\alpha^*(t) = \theta_\alpha^*(t') \text{ and } \theta_\beta^*(t) = \theta_\beta^*(t') \qquad \forall t, t' \in [t_1, t_2) \tag{5.1}$$

$$\theta_\beta^*(t_1) > \theta_\alpha^*(t_1). \tag{5.2}$$

Note also that on the interval $[T_\beta^*, T_\alpha^*)$, job α receives some positive allocation while job β receives no allocation because it is complete.

On some subset of these intervals we will perform an interchange which will reduce the weighted response time of the jobs, producing a contradiction. Let P be the policy resulting from this interchange. Under P, we will fully exchange β and α's allocations on the interval $[T^*_\beta, T^*_\alpha)$. That is, for any $t \in [T^*_\beta, T^*_\alpha)$, $\theta^P_\beta(t) = \theta^*_\alpha(t)$ and $\theta^P_\alpha(t) = 0$. To offset this interchange we will reallocate cores from β to α on some subset of the intervals in I. We will argue that, after this interchange, $T^P_\beta = T^*_\alpha$ and $T^P_\alpha < T^*_\beta$, leading to a reduction in weighted response time.

Let $W^P_i(t)$ be the total amount of work done by policy P on job i by time t. Let $W^P_i(t_1, t_2)$ be the amount of work done by policy P on job i on the interval $[t_1, t_2)$. That is,

$$W^P_i(t_1, t_2) = W^P_i(t_2) - W^P_i(t_1).$$

Let $\gamma = W^*_\alpha(T^*_\beta, T^*_\alpha)$. Since job α finished at exactly time T^*_α, we know that $W^*_\alpha(T^*_\beta) = x_\alpha - \gamma$. Similarly, we know that

$$W^*_\beta(T^*_\beta) = x_\beta.$$

We can see that policy P has the capacity to do up to γ work on job β on the interval $[T^*_\beta, T^*_\alpha)$. Specifically, if

$$W^P_\beta(T^*_\beta) = x_\beta - \gamma$$

then $T^P_\beta = T^*_\alpha$.

We will now reallocate cores from job β to job α on some of the intervals in I until $W^P_\beta(T^*_\beta) = x_\beta - \gamma$. On every such interval $i = [t_1, t_2) \in I$ we will choose

$$0 \leq \delta_i \leq \theta^*_\beta(t_1) - \theta^*_\alpha(t_1)$$

and let

$$\theta^P_\beta(t_1) = \theta^*_\beta(t_1) - \delta_i \qquad \text{and} \qquad \theta^P_\alpha(t_1) = \theta^*_\alpha(t_1) + \delta_i.$$

Decreasing job β's allocation on this interval has a well defined effect on $W^P_\beta(T^*_\beta)$. Namely, decreasing an allocation of a $\theta^*_\beta(t_1)$ fraction of the cores by δ_i decreases $W^P_\beta(T^*_\beta)$ by

$$(t_2 - t_1)(s(\theta^*_\beta(t_1) \cdot n) - s((\theta^*_\beta(t_1) - \delta_i) \cdot n)).$$

Note that, for any interval $i \in I$, $W^P_\beta(T^*_\beta)$ is a continuous, decreasing function of δ_i.

We will now iteratively apply the following interchange algorithm. Initialize $\delta_i = 0, \forall i \in I$. For each interval $i \in I$, try setting δ_i to be $\theta^*_\beta(i) - \theta^*_\alpha(i)$. If $W^P_\beta(T^*_\beta)$ is greater than $x_\beta - \gamma$ after this interchange, proceed to the next interval. If $W^P_\beta(T^*_\beta) < x_\beta - \gamma$ using this value of δ_i, there must exist some $0 < \delta_i < \theta^*_\beta(i) - \theta^*_\alpha(i)$ such that $W^P_\beta(T^*_\beta) = x_\beta - \gamma$ by the intermediate value theorem. Select this value of δ_i and terminate the algorithm.

To see this algorithm terminates, consider what would happen if

$$\delta_i = \theta^*_\beta(i) - \theta^*_\alpha(i) \qquad \forall i \in I.$$

In this case,

$$\theta^P_\beta(t) \leq \theta^*_\alpha(t) \qquad \forall t \leq T^*_\beta$$

which would imply that

$$W_\beta^P(T_\beta^*) \leq W_\alpha^*(T_\beta^*) = x_\alpha - \gamma < x_\beta - \gamma.$$

Hence the algorithm will terminate before running out of intervals in I. By construction, we know that $T_\beta^P = T_\alpha^*$.

We now show that, after performing the interchange, $T_\alpha^P < T_\beta^*$. To see this, consider the total amount of work done on any interval in $i = [t_1, t_2) \in I$ before and after the interchange. Because only the allocations to α and β have changed, it is clear that

$$\sum_{j=1}^{m(t_1)} W_j^P(t_1, t_2) - W_j^*(t_1, t_2) = (W_\alpha^P(t_1, t_2) + W_\beta^P(t_1, t_2)) - (W_\alpha^*(t_1, t_2) + W_\beta^*(t_1, t_2)).$$

By the strict concavity of s, we know that the second derivative of s is negative. As illustrated in Figure 5.4, and because $\theta_\beta^*(t_1) - \delta_i \geq \theta_\alpha^*(t_1)$ by construction, we know that

$$s((\theta_\alpha^*(t_1) + \delta_i)n) - s(\theta_\alpha^*(t_1)n) > s(\theta_\beta^*(t_1)n) - s((\theta_\beta^*(t_1) - \delta_i)n). \tag{5.3}$$

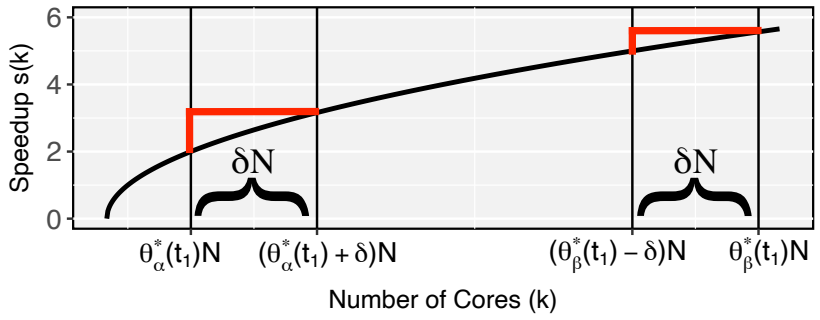


Figure 5.4: An illustration of (5.3). Because the speedup function, s, is concave, its second derivative is negative. Hence, s increases less on the interval $[(\theta_\beta^*(t_1) - \delta)n, \theta_\beta^*(t_1)n]$ than on the interval $[\theta_\alpha^*(t_1)n, (\theta_\alpha^*(t_1) + \delta)n]$.

Factoring out n gives

$$s(\theta_\alpha^*(t_1) + \delta_i) - s(\theta_\alpha^*(t_1)) > s(\theta_\beta^*(t_1)) - s(\theta_\beta^*(t_1) - \delta_i)$$

and thus

$$\big(s(\theta_\alpha^*(t_1) + \delta_i) + s(\theta_\beta^*(t_1) - \delta_i)\big) - \big(s(\theta_\alpha^*(t_1)) + s(\theta_\beta^*(t_1))\big) > 0.$$

Multiplying both sides by $(t_2 - t_1)$, we see that

$$(W_\alpha^P(t_1, t_2) + W_\beta^P(t_1, t_2)) - (W_\alpha^*(t_1, t_2) + W_\beta^*(t_1, t_2)) > 0$$

since both allocations are constant on the interval $[t_1, t_2)$. That is, the total amount of work done on interval i has increased. Since this holds for all $i \in I$ we have that

$$\sum_{[t_1,t_2)\in I} (W_\alpha^P(t_1,t_2) + W_\beta^P(t_1,t_2)) - (W_\alpha^*(t_1,t_2) + W_\beta^*(t_1,t_2)) > 0$$
$$\sum_{[t_1,t_2)\in I} W_\alpha^P(t_1,t_2) - W_\alpha^*(t_1,t_2) - \sum_{[t_1,t_2)\in I} W_\beta^*(t_1,t_2) - W_\beta^P(t_1,t_2) > 0. \tag{5.4}$$

Again, by construction, we have decreased the amount of work done on job β during the intervals in I by exactly γ. That is,

$$\sum_{[t_1,t_2)\in I} W_\beta^*(t_1,t_2) - W_\beta^P(t_1,t_2) = \gamma$$

and hence by (5.4)

$$\sum_{[t_1,t_2)\in I} W_\alpha^P(t_1,t_2) - W_\alpha^*(t_1,t_2) > \gamma.$$

Recall that $W_\alpha^*(T_\beta^*) = x_\alpha - \gamma$. Thus

$$W_\alpha^P(T_\beta^*) - W_\alpha^*(T_\beta^*) = \sum_{[t_1,t_2)\in I} W_\alpha^P(t_1,t_2) - W_\alpha^*(t_1,t_2)$$
$$W_\alpha^P(T_\beta^*) - (x_\alpha - \gamma) > \gamma$$
$$W_\alpha^P(T_\beta^*) > x_\alpha.$$

This implies that, under policy P, job α finishes before time T_β^*.

We can thus conclude that after the interchange, policy P completes job β at exactly T_α^* and policy P completes job α at some time before T_β^*. All other jobs are unchanged by this interchange, and hence their response times remain the same. Hence, it suffices to show that

$$w_\alpha T_\alpha^P + w_\beta T_\beta^P < w_\alpha T_\alpha^* + w_\beta T_\beta^*.$$

Because weights favor small jobs we have that $w_\beta < w_\alpha$. Furthermore, we have assumed that $T_\alpha^* > T_\beta^*$. Hence,

$$w_\beta \left(T_\alpha^* - T_\beta^*\right) < w_\alpha \left(T_\alpha^* - T_\beta^*\right)$$
$$w_\beta T_\alpha^* + w_\alpha T_\beta^* < w_\alpha T_\alpha^* + w_\beta T_\beta^*,$$

and thus by construction

$$w_\beta T_\beta^P + w_\alpha T_\alpha^P < w_\beta T_\alpha^* + w_\alpha T_\beta^* < w_\alpha T_\alpha^* + w_\beta T_\beta^*.$$

This implies that the policy P has a lower weighted response time than the optimal policy, a contradiction. □

Definition 5.3. *It will be useful to consider the rate at which weighted response time is accrued in between departures. Because the optimal completion order is SJF, we know that job k will complete directly after job $k+1$, and that during the interval $[T^*_{k+1}, T^*_k)$, jobs 1 through k will be present in the system. Hence, we define $z(k)$ to be the rate at which total response time is accrued during the interval $[T^*_{k+1}, T^*_k)$, where*

$$z(k) = \sum_{i=1}^{k} w_i.$$

5.4.2 The Scale-Free Property

The goal of this section is to characterize some optimal substructure of the optimal policy that will allow us to reduce the search space for the optimal policy. Hence, we now prove an interesting property of the optimal policy which we call the scale-free property. We will first need a preliminary lemma.

Lemma 5.4. *Consider an allocation function $\boldsymbol{\theta}(t)$ which, on some time interval $[0, T]$ leaves β fraction of the system unused. That is,*

$$\sum_{i=1}^{m(t)} \theta_i(t) = 1 - \beta \qquad \forall t \in [0, T].$$

The total work done on any job i by time T under $\boldsymbol{\theta}(t)$ is equivalent to the total work done on job i by time T under an allocation function $\boldsymbol{\theta}'(t)$ where

$$\boldsymbol{\theta}'(t) = \frac{\boldsymbol{\theta}(t)}{1-\beta} \qquad \forall t \in [0, T]$$

in a system that runs at $(1-\beta)^p$ times the speed of the original system (which runs at rate 1).

Proof. Consider the policy $\boldsymbol{\theta}'(t)$ in a system which runs at $(1-\beta)^p = s(1-\beta)$ times the speed of the original system on $[0, T]$. The service rate of any job in this system on $[0, T]$ is given by

$$s(1-\beta) \cdot s(\boldsymbol{\theta}'(t) \cdot n) = s((1-\beta) \cdot \boldsymbol{\theta}'(t) \cdot n) = s(\boldsymbol{\theta}(t) \cdot n).$$

Since, at any time $t' \in [0, T]$, the service rate for any job i is the same under $\boldsymbol{\theta}(t)$ as it is under $\boldsymbol{\theta}'(t)$ in a system which is $(1-\beta)^p$ times as fast, the same amount of work is done on job i by time T in both systems. □

Using Lemma 5.4 we can characterize the optimal policy. Theorem 5.5 states that a job's allocation relative to the jobs larger than it will remain constant for the job's entire lifetime.

Theorem 5.5 (Scale-free Property)**.** *For any job, i, there exists a constant c_i such that, for all $t < T^*_i$*

$$\frac{\theta^*_i(t)}{\sum_{j=1}^{i} \theta^*_j(t)} = c_i.$$

Proof. We will prove this statement by induction on the overall number of jobs, M. First, note that the statement is trivially true when $M = 1$. It remains to show that if the theorem holds for $M = k$, then it also holds for $M = k + 1$.

Let $M = k+1$ and let T_i^* denote the finishing time of job i under the optimal policy. Recall that the optimal policy finishes jobs according to the SJF completion order, so $T_{i+1}^* \leq T_i^*$. Consider a system which optimally processes k jobs, which WLOG are jobs $1, 2, \ldots, M-1$. We will now ask this system to process an additional job, job M. From the perspective of the original k jobs, there will be some constant portion of the system, θ_M^*, used to process job M on the time interval $[0, T_M^*]$. The remaining $1-\theta_M^*$ fraction of the system will be available during this time period. Just after time T_M^*, there will be at most k jobs in the system, and hence by the inductive hypothesis the optimal policy will obey the scale-free property on the interval $(T_M^*, T_1^*]$.

Consider the problem of minimizing the weighted response time of the M jobs *given any fixed value* of θ_M^* such that the SJF completion order is obeyed. We can write the optimal weighted response time of the M jobs, F^*, as

$$F^* = w_M T_M^* + \sum_{j=1}^{M-1} w_j T_j^*$$

where $w_M T_M^*$ is a constant. Clearly, optimizing weighted response time in this case is equivalent to optimizing the weighted response time for $M - 1 = k$ jobs with the added constraint that θ_M^* is unavailable (and hence "unused" from the perspective of jobs $M-1$ through 1) during the interval $[0, T_M^*]$. By Lemma 5.4, this is equivalent to having a system that runs at a fraction $(1-\theta_M^*)^p$ of the speed of a normal system during the interval $[0, T_M^*]$.

Thus, for some $d > 1$, we will consider the problem of optimizing weighted response time for a set of k jobs in a system that runs at a speed $\frac{1}{d}$ times as fast during the interval $[0, T_M^*]$.

Let $F^*[x_{M-1}, \ldots, x_1]$ be the optimal weighted response time of k jobs of size $x_{M-1} \ldots x_1$. Let $F^s[x_{M-1}, \ldots, x_1]$ be the weighted response time of these jobs in a *slow system* which always runs $\frac{1}{d}$ times as fast as a normal system.

If we let T_i^s be the finishing time of job i in the slow system, it is easy to see that

$$T_i^* = \frac{T_i^s}{d}$$

since we can just factor out a d from the expression for the completion time of every job in the slow system. Hence, we see that

$$F^*[x_{M-1}, \ldots, x_1] = \frac{F^s[x_{M-1}, \ldots, x_1]}{d}$$

by the same reasoning. Clearly, then, the optimal allocation function in the slow system, $\boldsymbol{\theta}^s(t)$, is equal to $\boldsymbol{\theta}^*(t)$ at the respective departure times of each job. That is,

$$\boldsymbol{\theta}^*(T_j^*) = \boldsymbol{\theta}^s(T_j^s) \qquad \forall 1 \leq j \leq M-1.$$

We will now consider a *mixed* system which is "slow" for some interval $[0, T_M^*]$ that ends before T_{M-1}^s, and then runs at normal speed after time T_M^*. Let $F^Z[x_{M-1}, \ldots, x_1]$ denote the weighted

response time in this mixed system and let $\boldsymbol{\theta}^Z(t)$ denote the optimal allocation function in the mixed system. We can write

$$F^Z[x_{M-1}, \ldots, x_1] = z(k) \cdot T_M^* + F^*[x_{M-1} - \frac{s(\theta_{M-1}^Z)T_M^*}{d}, \ldots x_1 - \frac{s(\theta_1^Z)T_M^*}{d}].$$

Similarly we can write

$$F^s[x_{M-1}, \ldots, x_1] = z(k) \cdot T_M^* + F^s[x_{M-1} - \frac{s(\theta_{M-1}^s)T_M^*}{d}, \ldots x_1 - \frac{s(\theta_1^s)T_M^*}{d}].$$

Let T_i^Z be the finishing time of job i in the mixed system under $\boldsymbol{\theta}^Z(t)$. Since $z(k) \cdot T_M^*$ is a constant not dependent on the allocation function, we can see that the optimal allocation function in the mixed system will make the same allocation decisions as the optimal allocation function in the slow system at the corresponding departure times in each system. That is,

$$\boldsymbol{\theta}^*(T_j^*) = \boldsymbol{\theta}^s(T_j^s) = \boldsymbol{\theta}^Z(T_j^Z) \qquad \forall 1 \leq j \leq M-1.$$

By the inductive hypothesis, the optimal allocation function in the slow system obeys the scale-free property. Hence, $\boldsymbol{\theta}^Z(t)$ also obeys the scale-free property for this set of k jobs given any fixed value of θ_M^*.

We now apply Lemma 5.4 again, multiplying $\boldsymbol{\theta}^Z(t)$ by $1-\theta_M^*$ on the interval $[0, T_M^*]$ to recover an allocation policy with θ_M^* unused cores on $[0, T_M^*]$. We call the resulting policy $\boldsymbol{\theta}^P(t)$. By Lemma 5.4, jobs $M-1, \ldots, 1$ have the same residual sizes at time T_M^* in a mixed system under $\boldsymbol{\theta}^Z(t)$ as they do in a constant speed system under $\boldsymbol{\theta}^P(t)$. Hence, because the mixed system under $\boldsymbol{\theta}^Z(t)$ and the constant speed system under $\boldsymbol{\theta}^P(t)$ are identical after time T_M^*, the weighted response time in these two systems is the same. Therefore, if $\boldsymbol{\theta}^Z(t)$ is optimal in the mixed system, $\boldsymbol{\theta}^P(t)$ minimizes the weighted response time of jobs $M-1, \ldots, 1$ for any fixed value of θ_M^* in a normal speed system. Since $\boldsymbol{\theta}^Z(t)$ obeys the scale-free property, $\boldsymbol{\theta}^P(t)$ obeys the scale-free property for jobs $1, \ldots, M-1$. Hence, for a set of $M = k+1$ jobs, given any allocation θ_M^* to job M on the interval $[0, T_M^*]$, the optimal allocations to the remaining $M-1$ jobs obey the scale-free property. Finally, the scale-free property is trivially satisfied for job M by allocating any fixed θ_M^* to job M on the interval $[0, T_M^*]$ and setting $c_i = \theta_M^*$. The optimal allocation function for processing the $M = k+1$ jobs therefore obeys the scale-free property. This completes the proof by induction. □

Definition 5.6. *The scale-free property tells us that, under the optimal policy, job i's allocation relative to the jobs completed after it is a constant, c_i. Note that the jobs completed after job i are precisely the jobs with an initial size of at least x_i, since the optimal policy follows the SJF completion order. It will be useful to define a* **scale-free constant**, *ω_i^*, for every job i, where for any $t < T_i^*$*

$$\omega_i^* = \frac{1}{c_i} - 1 = \frac{\sum_{j=1}^{i-1} \theta_j^*(t)}{\theta_i^*(t)}.$$

Note that we define $\omega_1 = 0$. Let $\boldsymbol{\omega}^* = (\omega_1^*, \omega_2^*, \ldots, \omega_M^*)$ denote the scale-free constants corresponding to each job.

5.4.3 Finding the Optimal Allocation Function

We will now make use of the scale-free property and our knowledge of the optimal completion order to find the optimal allocation function. We will consider the weighted response time under any policy, P, which obeys the scale-free property and follows the SJF completion order. In Lemma 5.7, we derive an expression for the weighted response time under the policy P as a function of the scale-free constants $\boldsymbol{\omega}^P$. In Theorem 5.8 we then minimize this expression to find the optimal scale-free constants and the optimal allocation function.

Lemma 5.7. *Consider a policy P which obeys the scale-free property and which completes jobs in shortest-job-first order. We define*

$$\omega_i^P = \frac{\sum_{j=1}^{i-1} \theta_j^P(t)}{\theta_i^P(t)} \qquad \forall 1 < i \leq M,\ 0 \leq t < T_i^P$$

and $\omega_1^P = 0$. We can then write the weighted response time under policy P as a function of $\boldsymbol{\omega}^P = (\omega_1^P, \omega_2^P, \ldots, \omega_M^P)$ as follows

$$F^P(\boldsymbol{\omega}^P) \quad = \quad \frac{1}{s(n)} \sum_{k=1}^{M} x_k \cdot \left[z(k) s(1 + \omega_k^P) - z(k-1) s(\omega_k^P) \right]$$

Proof. To analyze the total weighted response time under P we will relate P to a simpler policy, P', which is much easier to analyze. We define P' to be

$$\theta_i^{P'} = \theta_i^{P'}(t) = \theta_i^P(0) \qquad \forall 1 \leq i \leq M.$$

Importantly, each job receives some initial optimal allocation at time 0 which does not change over time under P'. Since allocations under P' are constant we have that

$$T_k^{P'} \quad = \quad \frac{x_k}{s(\theta_k^{P'})}.$$

We can now derive equations that relate T_k^P to $T_k^{P'}$.

By Theorem 5.5, during the interval $[T_{k+1}^{P'}, T_k^{P'}]$,

$$\omega_i^P = \omega_i^{P'} \qquad \forall 1 \leq i \leq k.$$

Note that a fraction of the system $\sum_{i=k+1}^{M} \theta_i^{P'}$ is unused during this interval, and hence by Lemma 5.4, we have that

$$T_k^{P'} - T_{k+1}^{P'} \quad = \quad \frac{T_k^P - T_{k+1}^P}{s(\theta_1^{P'} + \cdots + \theta_k^{P'})}.$$

Let $\alpha_k = \frac{1}{s(\theta_1^{P'} + \cdots + \theta_k^{P'})}$ be the scaling factor during this interval.

If we define $x_{M+1} = 0$ and $T_{M+1}^P = 0$, we can express the weighted response time under policy P, F^P, as

$$\begin{aligned} F^P &= z(M)T_M^P + z(M-1)(T_{M-1}^P - T_M^P) + \cdots + z(2)(T_2^P - T_3^P) + z(1)(T_1^P - T_2^P) \\ &= \sum_{k=1}^{M} z(k)(T_k^P - T_{k+1}^P) \\ &= \sum_{k=1}^{M} z(k)\frac{T_k^{P'} - T_{k+1}^{P'}}{\alpha_k}. \end{aligned}$$

We can now further expand this expression in terms of the job sizes, using the fact that $s(ab) = s(a) \cdot s(b)$, as follows:

$$\begin{aligned} F^P =& \sum_{k=1}^{M} z(k)\frac{\frac{x_k}{s(\theta_k^{P'} n)} - \frac{x_{k+1}}{s(\theta_{k+1}^{P'} n)}}{\alpha_k} \\ =& \frac{1}{s(n)} \sum_{k=1}^{M} z(k) \cdot \left[x_k s(1 + \omega_k^{P'}) - x_{k+1} s(\omega_{k+1}^{P'})\right] \\ =& \frac{1}{s(n)} \sum_{k=1}^{M} x_k \cdot \left[z(k)s(1 + \omega_k^P) - z(k-1)s(\omega_k^P)\right] \end{aligned} \tag{5.5}$$

as desired. □

We now have an expression for the weighted response time of any policy P which obeys the scale-free property and completes jobs in SJF order. Since the optimal policy obeys these properties, the choice of P which minimizes the above expression for weighted response time must be the optimal policy. In Theorem 5.8 we find a closed form expression for the optimal scale-free constants.

Theorem 5.8 (Optimal Scale-Free Constants)**.** *The scale-free constants of the optimal policy are given by the expression*

$$\omega_k^* = \frac{1}{\left(\frac{z(k)}{z(k-1)}\right)^{\frac{1}{1-p}} - 1} \qquad \forall 1 < k \leq M.$$

Proof. Consider a policy P which obeys the scale-free property and completes jobs in SJF order. Let $F^P(\boldsymbol{\omega}^P)$ be the expression for weighted response time for P from Lemma 5.7. Our goal is to find a closed form expression for $\boldsymbol{\omega}^*$, the scale-free constants which minimize the above expression for weighted response time.

A sufficient condition for finding $\boldsymbol{\omega}^*$ is that a policy completes jobs in SJF order *and* satisfies the following first-order conditions:

$$\frac{\partial F^P}{\partial \omega_k^P} = 0 \qquad \forall 1 \leq k \leq M$$

The second-order conditions are satisfied trivially. Note that solely satisfying the first order conditions is insufficient, because the resulting solution is not guaranteed to respect the SJF completion order. Luckily, we can show that any solution which satisfies the first-order conditions also completes jobs in SJF order. To see this, we will begin by finding the allocation function, $\mathbf{\Theta}^P(t)$, which satisfies the first-order conditions.

The first order conditions, obtained by differentiating (5.5), give

$$z(k)s'(1+\omega_k^P) - z(k-1)s'(\omega_k^P) = 0 \qquad \forall 1 \le k \le M$$

and hence

$$\omega_k^P = \frac{1}{\left(\frac{z(k)}{z(k-1)}\right)^{\frac{1}{1-p}} - 1} \qquad \forall 1 < k \le M$$

We can show that the values of $\Theta_k^P(t)$ are increasing in k (see B.1). That is, smaller jobs always have larger allocations than larger jobs under $\mathbf{\Theta}^P(t)$. This implies that $\mathbf{\Theta}^P(t)$ follows the SJF completion order, since a larger job cannot complete before a smaller job unless it receives a larger allocation for some period of time. $\mathbf{\Theta}^P(t)$ therefore satisfies the sufficient condition for optimality. We thus have found a closed form expression for the optimal scale free constants. That is,

$$\omega_k^* = \frac{1}{\left(\frac{z(k)}{z(k-1)}\right)^{\frac{1}{1-p}} - 1} \qquad \forall 1 < k \le M$$

as desired. □

We now derive the optimal allocation function in Theorem 5.9.

Theorem 5.9 (Optimal Allocation Function)**.** *At time t, when $m(t)$ jobs remain in the system,*

$$\theta_i^*(t) = \begin{cases} \left(\frac{z(i)}{z(m(t))}\right)^{\frac{1}{1-p}} - \left(\frac{z(i-1)}{z(m(t))}\right)^{\frac{1}{1-p}} & 1 \le i \le m(t) \\ 0 & i > m(t) \end{cases}$$

where

$$z(k) = \sum_{i=1}^{k} w_i.$$

We refer to the optimal allocation policy which uses $\boldsymbol{\theta}^*(t)$ as heSRPT.

Proof. We can now solve a system of equations to derive the optimal allocation function. Consider a time, t, when there are $m(t)$ jobs in the system. Since the optimal completion order is SJF, we know that the jobs in the system are specifically jobs $1, 2, \ldots, m(t)$. We know that the allocation to jobs $m(t)+1, \ldots, M$ is 0, since these jobs have been completed. Hence, we have that

$$\theta_1^* + \theta_2^* + \ldots + \theta_{m(t)}^* = 1.$$

Furthermore we have $m(t) - 1$ constraints provided by the expressions for $\omega_2^*, \omega_3^*, \ldots, \omega_{m(t)}^*$.

$$\omega_2^* = \frac{\theta_1^*(t)}{\theta_2^*(t)}, \omega_3^* = \frac{\theta_2^*(t) + \theta_1^*(t)}{\theta_3^*(t)}, \cdots, \omega_{m(t)}^* = \frac{\sum_{i=1}^{m(t)-1} \theta_i^*(t)}{\theta_{m(t)}^*(t)}.$$

These can be written as

$$\begin{aligned}
\theta_1^*(t) &= \omega_2^* \theta_2^*(t) \\
\theta_1^*(t) + \theta_2^*(t) &= \omega_3^* \theta_3^*(t) \\
&\cdots \\
\theta_1^*(t) + \theta_2^*(t) + \cdots + \theta_{m(t)-1}^*(t) &= \omega_{m(t)}^* \theta_{m(t)} \\
\theta_1^* + \theta_2^* + \ldots + \theta_{m(t)}^* &= 1
\end{aligned}$$

and then rearranged as

$$\begin{aligned}
\theta_{m(t)}^*(t) &= \frac{1}{1 + \omega_{m(t)}^*} \\
\theta_{m(t)-1}^*(t) &= \frac{\theta_{m(t)}^*(t)\omega_{m(t)}^*}{1 + \omega_{m(t)-1}^*} = \frac{\omega_{m(t)}^*}{(1 + \omega_{m(t)-1}^*)(1 + \omega_{m(t)}^*)} \\
&\cdots \\
\theta_2^*(t) &= \frac{\theta_3^*(t)\omega_3^*}{1 + \omega_2^*} = \frac{\omega_3^* \cdots \omega_{m(t)}^*}{(1 + \omega_2^*) \cdots (1 + \omega_{m(t)}^*)} \\
\theta_1^*(t) &= \frac{\theta_2^*(t)\omega_2^*}{1 + \omega_1^*} = \frac{\omega_2^* \cdots \omega_{m(t)}^*}{(1 + \omega_1^*) \cdots (1 + \omega_{m(t)}^*)}.
\end{aligned}$$

We can now plug in the known values of ω_i^* and find that

$$\boldsymbol{\theta}_i^*(t) = \left(\frac{z(i)}{z(m(t))}\right)^{\frac{1}{1-p}} - \left(\frac{z(i-1)}{z(m(t))}\right)^{\frac{1}{1-p}} \quad \forall 1 \leq i \leq m(t)$$

This argument holds for any $m(t) \geq 1$, and hence we have fully specified the optimal allocation function. □

Taken together, the results of this section yield an expression for weighted response time under the optimal allocation policy. This expression is stated in Theorem 5.10.

Theorem 5.10 (Optimal Weighted Response Time)**.** *Given a set of M jobs of size $x_1 \geq x_2 \geq \ldots \geq x_M$, the weighted response time, F^*, under the optimal allocation policy $\boldsymbol{\theta}^*(t)$ is given by*

$$F^* = \frac{1}{s(n)} \sum_{k=1}^{M} x_k \cdot \left[z(k)^{\frac{1}{1-p}} - z(k-1)^{\frac{1}{1-p}}\right]^{1-p}$$

where

$$z(k) = \sum_{i=1}^{k} w_i.$$

5.5 Discussion and Evaluation

We now examine the impact of the results shown in Section 5.4.

Section 5.5.1 shows how the results of Section 5.4 can be applied to provide the optimal policy with respect to mean slowdown and mean response time.

We perform a numerical evaluation of heSRPT in Section 5.5.2. We compare heSRPT to several competitor policies from the literature and show that heSRPT not only outperforms these policies as expected, but often reduces slowdown by over 30%.

The above results apply in the common case where all jobs are present at time 0, but it is also interesting to consider the online case where jobs arrive over time. To address the online case, we use heSRPT as the basis of a heuristic policy. Section 5.5.3 compares our heuristic policy, Adaptive-heSRPT, to other heuristic allocation policies from the literature. Adaptive-heSRPT vastly outperforms other heuristic policies in this online case.

5.5.1 Applying heSRPT

The optimality results of Section 5.4.3 were stated in terms of any weighted response time metric where weights favor small jobs. Specifically, this class of metrics can be used to find the optimal policy with respect to several popular metrics including mean slowdown and mean response time. This is summarized in the following corollary.

Corollary 5.11 (Optimal Mean Slowdown and Mean Response Time)**.** *If we define*

$$w_i = \frac{1}{x_i/s(n)} \quad \textit{and thus} \quad z(k) = \sum_{i=1}^{k} \frac{1}{x_i/s(n)},$$

Theorem 5.9 yields the optimal policy with respect to mean slowdown and Theorem 5.10 yields the optimal total slowdown.

If we define

$$w_i = 1 \quad \textit{and thus} \quad z(k) = k,$$

Theorem 5.9 yields the optimal policy with respect to mean response time and Theorem 5.10 yields the optimal total response time.

This corollary follows directly from the definitions of mean slowdown and mean response time, respectively. Specifically, our results hold in these cases because the weights used in both cases favor small jobs.

One might ask which, if any, of our results hold in the case where weights do not favor small jobs. If weights do not favor small jobs it is easy to construct an example where one should not complete jobs in SJF order. Giving a job of any size a sufficiently high weight can cause it to complete first under the optimal policy. In addition to obviously contradicting Theorem 5.2, the SJF completion order was exploited in the proof of Theorem 5.8. It is worth noting, however, that regardless of the weights used, the optimal policy will obey the scale free property of Theorem 5.5.

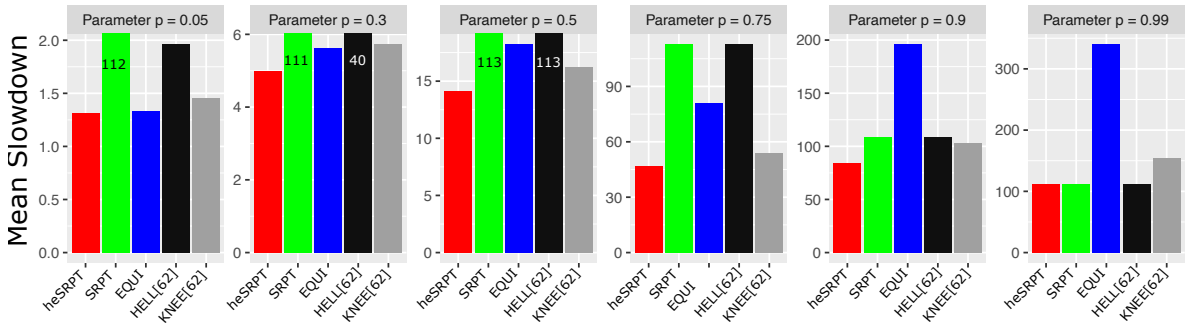


Figure 5.5: Mean slowdown of heSRPT and allocation policies from the literature in the offline setting with all jobs present at time 0. Evaluation assumes $n = 1,000,000$ cores and $M = 500$ jobs whose sizes are Pareto($\alpha = .8$) distributed. Each graph shows mean slowdown for one value of the speedup parameter, p, where $s(k) = k^p$. heSRPT often dominates by over 30%.

5.5.2 Numerical Evaluation: Offline Setting

We now compare the optimal mean slowdown under heSRPT with the mean slowdown under policies from the literature.

Our comparison continues to assume that all jobs are present at time 0. While we have a closed form expression for the optimal mean slowdown under heSRPT, we wish to compare heSRPT to policies for which there is no closed form analysis. Hence, we perform a numerical analysis of heSRPT and several policies from the literature.

We compare heSRPT to the following list of competitor policies:

SRPT allocates the entire system to the single job with shortest remaining processing time. While SRPT is optimal when $p = 1$, we expect SRPT to perform poorly when jobs make inefficient use of cores. When all jobs are present at time 0, SRPT is equivalent to the RS policy [104] which minimizes mean slowdown in an M/G/1.

EQUI allocates an equal fraction of the cores to each job at every moment in time. `EQUI` has been analyzed using competitive analysis [24, 25] in similar models of parallelizable jobs, and was shown to be optimal in expectation when job sizes are unknown and exponentially distributed in Chapter 6. Other policies such as *Intermediate-SRPT* [49] reduce to `EQUI` in our model where the number of jobs, M, is assumed to be less than the number of cores, n.

HELL is a heuristic policy proposed in [62] which, similarly to heSRPT, tries to balance system efficiency with biasing towards short jobs. HELL defines a job's efficiency to be the function $\frac{s(k)}{k}$. HELL then iteratively allocates cores. In each iteration, HELL identifies the job which can achieve highest ratio of efficiency to remaining processing time, and allocates to this job the cores required to achieve this maximal ratio. This process is repeated until all cores are allocated. While HELL is consistent with the goal of heSRPT, the specific ratio that HELL uses is just a heuristic.

KNEE is the other heuristic policy proposed in [62]. KNEE defines a job's *knee allocation* to be the number of cores for which the job's marginal reduction in run-time falls below some threshold, α. KNEE then iteratively allocates cores. In each iteration, KNEE identifies the job with the lowest knee allocation and gives this job its knee allocation. This process repeats until all

cores are allocated. Because there is no principled way to choose this α, we perform a brute-force search of the parameter space and present the results given the best α parameter we found. Hence, results for KNEE are an optimistic prediction of KNEE's performance.

We evaluate heSRPT and the competitor policies in a system of $n = 1,000,000$ cores and $M = 500$ jobs whose sizes are Pareto($\alpha = .8$) distributed. The speedup parameter p is set to values between 0.05 and 0.99. Figure 5.5 shows the results of this analysis.

heSRPT outperforms every competitor policy in every case as expected. When p is low, `EQUI` is within 1% of heSRPT, but `EQUI` is over $3\times$ worse than heSRPT when $p = 0.99$. Conversely, when $p = 0.99$, SRPT is nearly optimal. However, SRPT is an order of magnitude worse than heSRPT when $p = 0.05$. While HELL performs similarly to SRPT in most cases, it is 50% worse than optimal when $p = 0.05$. The KNEE policy is the best of the competitor policies that we consider. In the worst case, when $p = 0.99$, KNEE is roughly 50% worse than heSRPT. However, these results for KNEE required brute-force tuning of the allocation policy. We examined several other job size distributions and in all cases we saw similar trends.

Because the competitor policies described in this section were designed to minimize mean response time, we now compare the optimal mean response time under heSRPT to the mean response time of these competitor policies. The competitor policies remain unchanged from their descriptions in this section, and the conditions of our analysis (numbers of cores, job size distribution, speedup function) remain the same as well.

Figure 5.6 shows the results of our analysis. We see that the results with respect to mean response time are generally similar to the results of Figure 5.5. heSRPT once again outperforms every competitor policy by at least 30% in at least one case. Hence, the results shown in Figure 5.5 are not only caused by the fact that the competitor policies optimize for a different metric. Even when comparing policies with respect to mean response time, each of the competitor policies can be far from optimal.

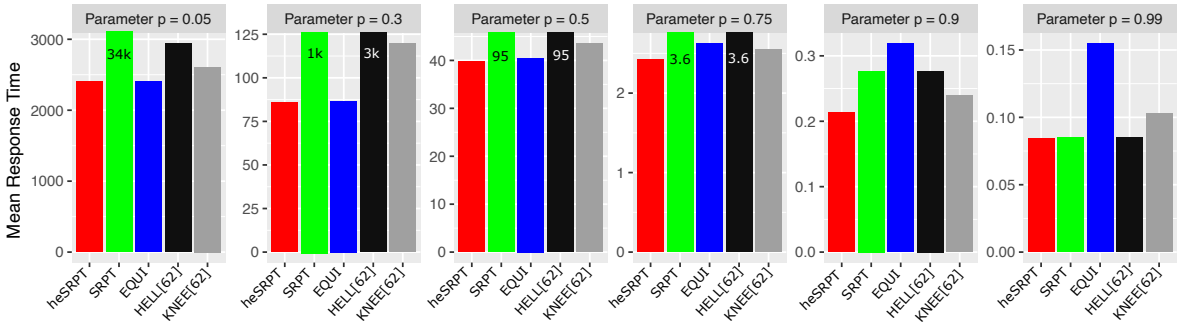


Figure 5.6: A comparison of the optimal mean response time under heSRPT to other allocation policies found in the literature. Each policy is evaluated on a system of $n = 1,000,000$ cores and a set of $M = 500$ jobs whose sizes are drawn from a Pareto distribution with shape parameter .8. Each graph shows the mean response time under each policy with various values of the speedup parameter, p, where the speedup function is given by $s(k) = k^p$. heSRPT outperforms every competitor by at least 30% in at least one case.

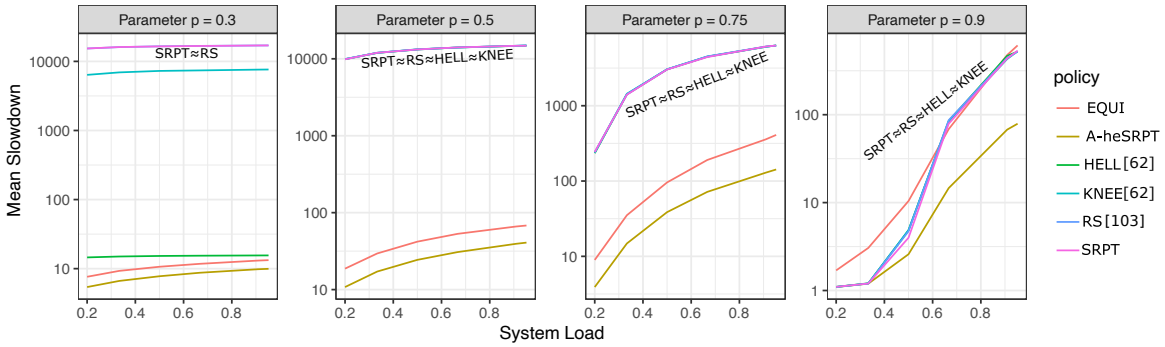


Figure 5.7: Mean slowdown of A-heSRPT and policies from the literature in the online setting. Simulations assume $n = 10,000$ cores and a Poisson arrival process. Job sizes are Pareto($\alpha = 1.5$) distributed. Each graph shows mean slowdown for one value of the speedup parameter, p, where $s(k) = k^p$. A-heSRPT often dominates by an order of magnitude.

5.5.3 Numerical Evaluation: Online Setting

While we have shown that heSRPT provides the optimal mean slowdown in the case where all jobs are present at time 0, minimizing the slowdown of a stream of arriving parallelizable jobs remains an open theoretical problem.

To understand the added complexity of the online setting, consider the problem of allocating cores to two jobs at time 0, while knowing that an arrival will occur at time 1. The optimal policy might try to complete one job quickly and allow the arrival to compete for cores with the second job, or it might try and complete both of the original jobs before time 1 in order to reduce the response time of the arriving job. In general, it is no longer sufficient to compare two jobs and decide which to complete first. One must also decide how many jobs to complete before the next arrival. This prevents the results in this chapter from generalizing cleanly to the online case.

We therefore use simulation to compare the performance of heSRPT to the competitor policies described above in an online setting where jobs arrive over time. For this comparison, we define a heuristic online policy, inspired by heSRPT, which we refer to as Adaptive-heSRPT (A-heSRPT). A-heSRPT recalculates the allocation to each job every time a job departs from *or arrives to* the system, using the heSRPT allocations as defined in Theorem 5.9. All of the other competitor policies we examine generalize to online setting in a similar way, reevaluating their allocation decisions on each arrival or departure.

For the sake of completeness, we also examine an additional competitor policy in the online case, the **RS** policy [104] which is known to minimize mean slowdown online for non-parallelizable jobs in an M/G/1 system [48]. The RS policy allocates all cores to the job with the lowest product of remaining size (R) times initial size (S). RS is identical to SRPT if all jobs are present at time 0, but must be considered separately in the online case.

Figure 5.7 shows that A-heSRPT outperforms all competitor policies in every case, *often by several orders of magnitude*. When jobs are highly parallelizable, policies such as SRPT and RS which aggressively favor small jobs can occasionally compete with A-heSRPT. However, as load increases and the allocation decision becomes more complex, A-heSRPT vastly outperforms

the competition. Conversely, when jobs have a low level of parallelizability (low p), SRPT, RS, HELL and KNEE fail to maintain the efficiency of the system and perform poorly. EQUI, which maximizes efficiency but does not favor small jobs, follows the opposite trend – EQUI performs decently when p is low, but it performs poorly when p is high. A-heSRPT is the only policy which is able to handle high and low values of p over a range of system loads.

5.6 Conclusion

This chapter addresses the problem of scheduling parallelizable jobs of known size in order to minimize an important class of weighted response time metrics. When designing a scheduling policy, it is tempting to use an efficiency-maximizing policy such as EQUI. However, when job sizes are known, an optimal policy might be willing to sacrifice system efficiency in order to favor smaller jobs. Strictly favoring short jobs by using an SRPT policy sacrifices too much system efficiency, and is far from optimal in general. This chapter shows that the optimal policy, heSRPT, must carefully balance the tradeoff between instantaneous system efficiency and favoring short jobs. While our optimality results are derived in the case where all jobs are present at time 0, heSRPT serves as the basis for an online policy that performs well in practice.

From Section 5.5, it is clear that systems with known job sizes stand to benefit greatly from implementing heSRPT. However, this chapter also has important implications for systems where job sizes are currently not available to the system. Many of these systems do not make job size information available to the scheduler largely because the importance of such information is not well-understood by the systems community. For example, work has been done to better predict the speedup function a data center job will receive [23]. Similar attempts can be made to predict the inherent sizes of data center jobs. The results of this chapter show that, if we could develop mechanisms for accurately predicting job sizes in data centers, response times in these systems could be significantly reduced.

Having considered both the case where job sizes are completely unknown and the case where job sizes are known exactly, it is natural to ask how a scheduling policy should behave when given *partial* information about the sizes of the jobs. For example, if the job sizes are unknown, but are known to follow a Pareto distribution, the expected remaining size of job will change depending on how long the job has been running. Scheduling jobs with generally-distributed sizes in multi-server systems remains as one of the most important open problems in the queueing community [33]. While recent work has looked at scheduling non-parallelizable jobs with unknown sizes in a multi-server system [90], these results require heavy-traffic analysis. Hence, the question of how to schedule *parallelizable* jobs of unknown, generally distributed sizes remains an open problem at the forefront of performance modeling research [37].

We have now seen one way in which a scheduling policy may be able to reduce job response times by sacrificing the instantaneous system efficiency. However, this chapter still assumes that all jobs follow a single speedup function. In the next two chapters we will see that additional, analogous tradeoffs will arise when jobs are permitted to follow different speedup functions. Here, rather than sacrificing instantaneous system efficiency to favor short jobs, an optimal policy may defer parallelizable work to preserve the future efficiency of the system. When jobs follow multiple

speedup functions *and* have known sizes, an allocation policy must weigh multiple competing tradeoffs simultaneously.

Chapter 6

Scheduling With Multiple Speedup Functions

6.1 Introduction

In both Chapter 4 and Chapter 5 we have restricted our discussion to the case where all jobs follow the same speedup function. This could be the case if, for example, a system was tasked with processing jobs that represented instances of a single application being invoked with different inputs. It is natural to ask, however, how a scheduling policy should behave when some jobs are known to be more parallelizable than others.

Scheduling jobs which follow different speedup functions is particularly important because modern parallel workloads can be astonishingly diverse [100]. Jobs from these heterogeneous workloads can differ dramatically in both their inherent sizes and in how effectively they can be parallelized [23]. This heterogeneity often occurs when different jobs in a given workload represent fundamentally different types of computation. For instance, a simple database lookup which retrieves a single record may only be capable of running on a single core and may complete in just milliseconds. On the other hand, data analytics queries generally require scanning entire tables and aggregating statistics over large sets of records [76]. As such, databases are generally designed to allow these analytics queries to be highly parallelizable [58]. As we will see in this chapter, employing scheduling policies which explicitly account for the heterogeneity of the workloads they serve can dramatically improve system performance.

6.1.1 Scheduling Tradeoffs

This chapter considers the case where jobs may belong to one of two *classes*, each of which has its own speedup function, s_i. We will assume that job sizes are once again unknown and exponentially distributed, but that the scheduler can identify which class a job belongs to. That is, the scheduler knows that some jobs are more parallelizable than others.

This chapter will show how we can optimally schedule parallelizable jobs which are heterogeneous with respect to their speedup functions. Figure 6.1 shows an illustration of the tradeoffs a

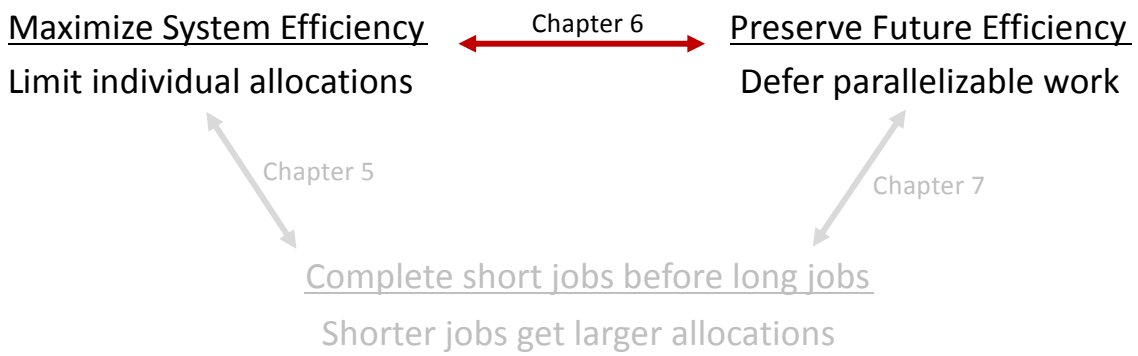


Figure 6.1: When job sizes are known to the system, a scheduling policy must balance a tradeoff between maximizing instantaneous system efficiency maximizing future system efficiency.

policy must balance, updated to reflect the case where jobs follow different speedup functions.

Key Insight

The scheduling policies in this chapter will have to change in two principle ways in order to effectively leverage the system's knowledge of each job's speedup function.

First, it should be clear that when jobs follow different speedup functions, `EQUI` is no longer guaranteed to maximize instantaneous system efficiency. We argued in Chapter 4 that `EQUI` maximized instantaneous system efficiency by equalizing each job's marginal benefit from receiving additional cores. Extending this reasoning to the case where jobs follow different speedup functions will require us to derive a new efficiency-maximizing policy that gives larger allocations to the more parallelizable jobs in the system.

Second, this chapter will show that when jobs follow different speedup functions, maximizing instantaneous system efficiency is no longer sufficient for minimizing mean response time. In Chapter 4, when jobs could not be differentiated on the basis of their sizes or speedup functions, the order in which we completed the jobs had no effect on the overall mean response time. However, when jobs follow different speedup functions, there can be a massive difference between completing a more parallelizable job and completing a less parallelizable job. For example, consider the case where the system is processing one perfectly parallelizable job, and one job that cannot be parallelized at all. If the perfectly parallelizable job is completed first, the system will only be able to effectively utilize one core to complete the remaining job. On the other hand, if the non-parallelizable job is completed first, the remaining perfectly parallelizable job can make use of all n cores. As this example illustrates, an optimal policy must balance a tradeoff between the instantaneous system efficiency in the present state and the *future* efficiency of the system as new jobs arrive and are completed. Specifically, an optimal may choose to *defer parallelizable work*, sacrificing some instantaneous system efficiency by reducing the allocations to the more parallelizable jobs in order to keep these more flexible jobs in the system in the future.

6.1.2 Contributions

In Section 6.3 we begin by considering the case where the speedup functions for each class can be of a very general form – any two concave, increasing speedup functions such that $s_1(k) < s_2(k) \quad \forall k > 1$. While we can numerically compute an optimal policy in this case, the generality of the model makes the problem difficult to solve analytically.

Then, in Section 6.4 we will consider an important special case where jobs are in one of two particular classes. The first class of jobs, which we call *elastic*, consists of jobs which are perfectly parallelizable. That is, elastic jobs follow a speedup function $s_E(k) = k$ for all $k \geq 0$. The second class of jobs, which we refer to as *inelastic*, consists of jobs which are not parallelizable. That is, inelastic jobs follow a speedup function $s_I(k) = 1$ for all $k \geq 1$. We see that, in this case, we can derive the form of the optimal policy, even if the elastic and inelastic jobs follow different job size distributions.

The contributions of this chapter are as follows:

- In Section 6.3, we consider the case where the two classes of jobs may follow different speedup functions of general forms. In this case, we can numerically compute an optimal policy which balances the tradeoff between present and future efficiency by deferring parallelizable work.
- In Section 6.3 we also analyze a policy called GREEDY*, which defers parallelizable work in a limited manner, and performs near-optimally in practice.
- Next, in Section 6.4, we consider the case where jobs are either elastic or inelastic. We propose two natural scheduling policies which aim to minimize the mean response time across jobs. First, the *Elastic-First* policy gives strict preemptive priority to elastic jobs and aims to minimize mean response time by maximizing the rate at which jobs depart the system. Second, the *Inelastic-First* policy gives strict preemptive priority to inelastic jobs. By deferring elastic work for as long as possible, Inelastic-First maximizes average system efficiency. It is not immediately obvious if either of these policies is optimal, or which policy is better.
- We show in Section 6.4.3 that if elastic and inelastic jobs follow the same exponential size distribution, Inelastic-First is optimal with respect to mean response time. This argument uses precedence relations to show that deferring elastic work increases the long run efficiency of the system.
- Then, in Section 6.4.4, we show that in the case where elastic jobs are larger on average than inelastic jobs, Inelastic-First is optimal with respect to mean response time. This requires the introduction of a novel sample path argument. Our key insight is that Inelastic-First minimizes the expected amount of inelastic work in the system as well as the expected *total* work in the system. As long as elastic jobs are larger than inelastic jobs on average, this suffices for minimizing mean response time.
- In the case where elastic jobs are smaller on average than inelastic jobs, Inelastic-First is no longer optimal. We illustrate this via a counterexample in Section 6.4.5 which shows that Elastic-First can outperform Inelastic-First. In order to determine when Elastic-First outperforms Inelastic-First, we perform the first analysis of both the Elastic-First and Inelastic-First scheduling policies in Section 6.5. This analysis leverages recent techniques for solving high-dimensional Markov chains. Our analytical results match simulation.

6.2 Our Model

Our goal is to derive and analyze policies which minimize the mean response time across jobs. In this chapter, WLOG, we assume that each of the n cores processes jobs with a rate of 1 unit of work per second. Hence, a job's size is equal to its running time on a single core. We assume that job sizes are unknown to the system, and are drawn from exponential distributions. We consider the case where jobs belong to one of two *classes* of jobs, where each class of jobs follows its own corresponding speedup function.

6.2.1 General Speedup Functions

In section 6.3, we begin by considering the case where the speedup functions for each class follow a very general form. We assume that class 1 jobs arrive according to a Poisson process with rate λ_1 and follow a speedup function $s_1(k)$ while class 2 jobs arrive according to a Poisson process with rate λ_2 and follow a speedup function $s_2(k)$. We assume only that s_1 and s_2 are any two increasing, concave functions such that

$$s_1(k) < s_2(k) \quad \forall k > 1.$$

We assume that the job sizes of both classes are unknown to the system and are exponentially distributed with rate μ.

We model the system under any policy π as a continuous time Markov chain where each state denotes the number of jobs belonging to each class that are currently in the system. That is, we define a continuous time Markov process $\{(N_1^\pi(t), N_2^\pi(t)) \colon t \geq 0\}$ where

$$(N_1^\pi(t), N_2^\pi(t)) \in \mathbb{Z}^2_{\geq 0}, \qquad \forall t \geq 0.$$

We define $N_1^\pi(t)$ to be the number of class 1 jobs in system at time t and we define $N_2^\pi(t)$ to be the number of class 2 jobs in system at time t. We let the state $(N_1^\pi(t), N_1^\pi(t)) = (i, j)$ denote that there are i class 1 jobs and j class 2 jobs currently in the system.

Because job sizes are exponential and arrivals occur according to a Poisson process, at any moment in time t, the distributions of remaining job sizes and the distributions of times until the next arrival for each job class can be fully specified by the numbers of jobs belonging to each class which are currently in the system. Hence, we will only consider policies which are *stationary* and *deterministic*, meaning the policy π makes the same allocation decision at every time t, given that the system is in state (i, j). Specifically, we define $\pi_1(i, j)$ to be the number of cores allocated to class 1 jobs in state (i, j) under policy π, and we define $\pi_2(i, j)$ to be the number of cores allocated to class 2 jobs in state (i, j) under policy π.

6.2.2 Elastic and Inelastic Jobs

Because this general formulation of the problem is hard to solve analytically, Section 6.4 considers a special case where each job is either *elastic* or *inelastic*. Elastic jobs follow a speedup function

$s_E(k)$ and inelastic jobs follow a speedup function $s_I(k)$ where

$$s_E(k) = k \quad \forall k \geq 0 \qquad s_I(k) = \begin{cases} k & 0 \leq k \leq 1 \\ 1 & k > 1 \end{cases}$$

That is, elastic jobs are perfectly parallelizable while inelastic jobs receive no benefit from running on more than 1 core.

Note that all of the results presented in this chapter hold equally if inelastic jobs can run on up to some fixed number of cores, $C < n$. We can simply renormalize our allocation policies to consider allocating in units of $\frac{n}{C}$ cores. After renormalizing, inelastic jobs can once again receive up to one unit of allocation while elastic jobs can receive any number of units of allocation. While our results do not depend on the value of C, we consider the case where $C = 1$ for the sake of simplifying our notation.

We define λ_E, λ_I, $\pi_E(i,j)$, $\pi_I(i,j)$, N_E^π, and N_I^π to be the same quantities as in the general case, but corresponding to elastic and inelastic jobs respectively. Furthermore, we allow the elastic and inelastic job size distributions to differ. We assume that elastic job sizes are exponentially distributed with rate μ_E while inelastic job sizes are exponentially distributed with rate μ_I. We let X_E and X_I be random variables representing the sizes of an elastic job or an inelastic job respectively.

We refer to a policy π as *work conserving* if and only if, in any state (i,j),

$$\pi_I(i,j) + \pi_E(i,j) \geq i,$$

and

$$\pi_I(i,j) + \pi_E(i,j) \geq n \cdot \mathbb{1}_{\{j>0\}}.$$

That is, π never leaves cores idle if there is an eligible job in the system. In Appendix C.1 we show that there exists an optimal policy for processing elastic and inelastic jobs which is also work conserving. It therefore suffices to only consider work conserving policies throughout our analysis.

Given that we only consider work conserving policies, note that

$$\pi_I(i,j) \leq i \qquad \forall (i,j) \in \mathbb{Z}^2_{\geq 0},$$

$$\pi_E(i,j) \leq n \cdot \mathbb{1}_{\{j>0\}} \qquad \forall (i,j) \in \mathbb{Z}^2_{\geq 0},$$

and

$$\pi_I(i,j) + \pi_E(i,j) \leq n \qquad \forall (i,j) \in \mathbb{Z}^2_{\geq 0}.$$

In general, $\pi_I(i,j) + \pi_E(i,j)$ could be less than n if there are not a sufficient number of jobs to use all n cores, or if π chooses to idle cores instead of allocating them to an eligible job.

We define the *system load*, ρ to be

$$\rho \equiv \frac{\lambda_I}{n\mu_I} + \frac{\lambda_E}{n\mu_E}. \tag{6.1}$$

In Appendix C.2 we show that for any work conserving policy, π, $(N_I^\pi(t), N_E^\pi(t))$ is an ergodic Markov chain if $\rho < 1$. Because there exists an optimal work conserving policy, (6.1) is necessary for stability under *any* policy π'. We therefore only consider the regime where $\rho < 1$.

In the case of elastic and inelastic jobs, we will track several stochastic quantities in our system. We define the total number of jobs in the system, $N^\pi(t)$, as

$$N^\pi(t) = N_I^\pi(t) + N_E^\pi(t).$$

We also define $W^\pi(t)$ to be the *total work* in the system under policy π at time t, where total work is the sum of the remaining sizes of all jobs in the system. Similarly, we let $W_E^\pi(t)$ and $W_I^\pi(t)$ be the *total elastic work* and the *total inelastic work* in the system under policy π at time t. These quantities are the sums of the remaining sizes of all elastic or inelastic jobs respectively. When referring to the corresponding steady-state quantities, we omit the argument t.

We will investigate the performance of two allocation policies for processing elastic and inelastic jobs, *Elastic-First* (EF) and *Inelastic-First* (IF). EF gives strict preemptive priority to elastic jobs, and processes jobs in first-come-first-serve (FCFS) order within each job class. That is, in any state (i, j) where $j > 0$, EF allocates all n cores to the elastic job with the earliest arrival time. In any state (i, j) where $j = 0$, EF allocates one core to each inelastic job, in FCFS order, until either all jobs have received a core or all n cores have been allocated. By contrast, IF gives strict preemptive priority to inelastic jobs while processing jobs in FCFS order within each job class. Under IF, in any state (i, j) where $i < n$, one core is allocated to each inelastic job and the remaining $n - i$ cores are allocated to the elastic job with the earliest arrival time if there is one. In any state (i, j) where $i \geq n$, all n cores are allocated to the inelastic jobs with the n earliest arrival times.

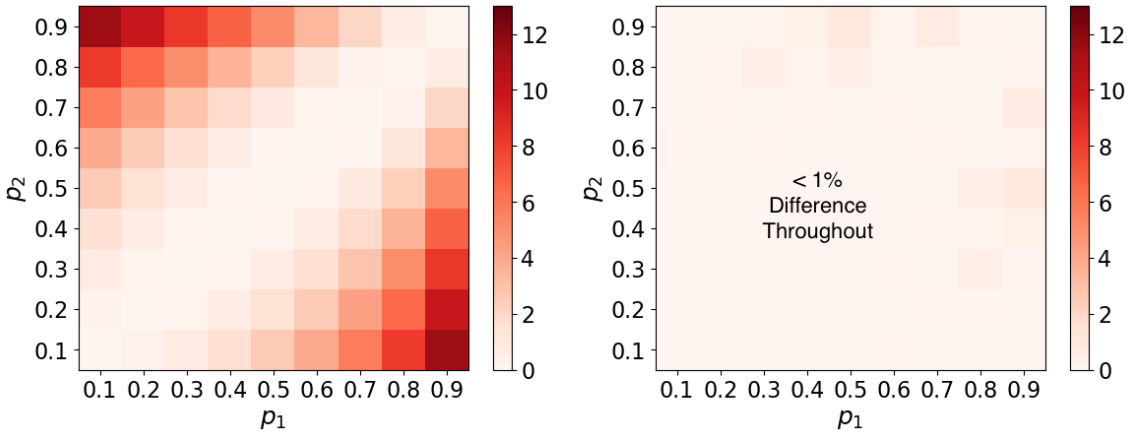
6.3 General Speedup Functions

In this section, we consider the case where jobs may have different speedup functions of a very general form. To facilitate this generality in the forms of the speedup functions, we will assume throughout this section that the sizes of jobs from both classes are exponentially distributed with rate μ.

From Chapter 4 recall that, when all jobs were exponentially distributed and followed a *single speedup function*, the optimal scheduling policy with respect to mean response time was EQUI. We will see in Section 6.3.1 that, in the case of multiple speedup functions, EQUI is no longer the optimal policy. In Section 6.3.2, we propose a class of policies called GREEDY which maximize the departure rate in every state. We then describe the optimal GREEDY policy, GREEDY* in Section 6.3.3. While GREEDY* is not optimal in general (see Section 6.3.5), we show that GREEDY* performs near-optimally in a wide range of settings (Sections 6.3.4 and 6.3.5). Finally in Section 6.3.6, we return to fixed-width policies and explain why they are insufficient when there are multiple speedup functions.

6.3.1 EQUI is No Longer Optimal

We have already seen that EQUI is optimal when jobs are homogeneous with respect to speedup. One might assume that, since EQUI bases its decisions on the number of jobs in the system rather than the jobs' speedup functions, EQUI could continue to perform well when there are multiple



(a) EQUI vs. OPT

(b) GREEDY* vs. OPT

Figure 6.2: Heat maps showing the percentage difference in the mean response time in the system, $\mathbb{E}[T]$, between (a) EQUI and OPT and between (b) GREEDY* and OPT, in the case of two speedup functions where s_1 and s_2 are Amdahl's law with parameters p_1 and p_2 respectively. Here $\mathbb{E}[X] = \frac{1}{2}$ and $\lambda_1 = \lambda_2 = 5$. The axes represent different values of p for each class. GREEDY*, EQUI, and OPT were evaluated numerically using the MDP formulation given in Section 6.3.4. These heat maps look similar under various values of λ_1 and λ_2.

speedup functions. However, it turns out that EQUI's performance is suboptimal even when there are just two speedup functions (see Figure 6.2a). While EQUI's performance is actually close to optimal in the cases where s_1 and s_2 are similar, we see that EQUI's performance relative to the optimal policy becomes worse as the difference between the speedup functions increases.

To see why EQUI is suboptimal in this case, recall that EQUI's optimality stems from the fact that it maximizes the instantaneous system efficiency in every state when jobs follow a single speedup function (see proof of Theorem 4.3). When jobs are permitted to have different speedup functions, maximizing the instantaneous system efficiency requires allocating more cores to class 2 jobs and fewer cores to class 1 jobs.

6.3.2 A GREEDY Class of Policies

We have seen that EQUI fails to maximize the instantaneous system efficiency when there are multiple speedup functions. Would a policy that maximizes the instantaneous system efficiency be optimal in this case? We define the GREEDY class of policies to be the policies which achieve the maximum possible instantaneous system efficiency in every state.

To describe the policies in GREEDY, we again consider the case where jobs belong to one of two job classes. For any state (i, j) where there are i class 1 jobs and j class 2 jobs, instantaneous system efficiency is proportional to the total rate of departures in the state. Hence, we will consider maximizing the total rate of departures via a two step process. *First*, a policy π must decide how many cores, $\pi_1(i, j)$, to allocate to the i class 1 jobs. The remaining $\pi_2(i, j) = n - \pi_1(i, j)$ cores

will be allocated to class 2 jobs. *Second*, the policy must decide how to divide the $\pi_1(i,j)$ cores among the class 1 jobs and the $\pi_2(i,j)$ cores among the class 2 jobs. We have seen that, when dividing cores among a set of jobs with a single speedup function, EQUI maximizes the total rate of departures of this set of jobs. For a given choice of $\pi_1(i,j)$, the $\pi_1(i,j)$ cores should thus be evenly divided among the class 1 jobs and the $\pi_2(i,j)$ cores should be evenly divided among the class 2 jobs in order to maximize the total rate of departures. Thus, only the first decision remains.

To find an allocation of cores which maximizes the rate of departures, we first define $\beta(i,j)$, the maximum rate of departures from the state (i,j):

$$\beta(i,j) = \max_{\alpha \in [0,n]} is_1\left(\frac{\alpha}{i}\right)\mu + js_2\left(\frac{n-\alpha}{j}\right)\mu. \tag{6.2}$$

Then, we can say that a policy, G, is in GREEDY if and only if

$$G_1(i,j) \in \left\{\alpha : is_1\left(\frac{\alpha}{i}\right)\mu + js_2\left(\frac{n-\alpha}{j}\right)\mu = \beta(i,j)\right\}. \tag{6.3}$$

This same two-step process will generalize to the case when jobs follow more than 2 speedup functions.

Crucially, note that GREEDY is truly a *class* of policies, since, in a given state, there may be multiple choices of $G_1(i,j)$ which satisfy (6.3). That is, there could be multiple allocations which achieve the maximal total rate of departures. For example, consider a system with 4 cores where both jobs classes have the same service rate $\mu = \frac{1}{\mathbb{E}[X]}$. If there are 4 class 1 jobs and 4 class 2 jobs, any choice of $G_1(i,j) \in [0,4]$ results in the maximal rate of departures, $n\mu$. This begs the question of *which* policy from the GREEDY class achieves the best performance.

6.3.3 The Best GREEDY Policy: GREEDY*

We now define GREEDY*, a policy which dominates all other GREEDY policies with respect to mean response time. Consider two GREEDY policies, G and G'. In any state (i,j), both policies achieve the same maximal rate of departures. However, G might achieve this rate by having a higher departure rate for class 1 jobs and a lower departure rate for class 2 jobs as compared to G'. If class 1 jobs are less parallelizable than class 2 jobs, we say that G "defers parallelizable work" in this state, which is a strategy that could benefit G in the future.

The GREEDY* policy is the GREEDY policy which in all states opts to defer parallelizable work when possible. Specifically, in the case of two job classes: GREEDY* allocates $G_1^*(i,j)$ cores to class 1 jobs (the less parallelizable class), where $G_1^*(i,j)$ is the maximum value of $\pi_1(i,j)$ satisfying (6.3). That is,

$$G_1^*(i,j) = \max\left\{\alpha : is_1\left(\frac{\alpha}{i}\right)\mu + js_2\left(\frac{n-\alpha}{j}\right)\mu = \beta(i,j)\right\}.$$

In other words, $G_1^*(i,j)$ allows GREEDY* to attain the maximal rate of departures while *also* maximizing the rate at which class 1 jobs are completed. Theorem 6.1 shows that GREEDY* dominates all other GREEDY policies.

Theorem 6.1. *For any GREEDY policy, G,*

$$\mathbb{E}[T]^{\textit{GREEDY}^*} \leq \mathbb{E}[T]^G.$$

Proof. Consider the performance of GREEDY* and G on the state space $\mathcal{S} = \{(i,j) : i,j \in \mathbb{N}\}$. We will use the technique of precedence relations (see, e.g., [3, 17]) to compare the mean number of customers, $\mathbb{E}[N]$, under GREEDY* to that under G. This requires that we define a value function for G, $V^G(i,j)$, and a cost function, $c(i,j)$. We define the cost of being in state (i,j) to be

$$c(i,j) = i + j$$

so that the average cost of performing policy G is equal to the mean number of customers, $\mathbb{E}[N]$. We also define $\gamma_i^G(i,j)$ to be the rate of departures of class i jobs from the state (i,j) under policy G.

We then define $V^G(i,j)$ to be the asymptotic total difference in the accrued cost under G when starting in state (i,j) as opposed to some designated reference state. We require the following lemma which establishes a useful property of V^G.

Lemma 6.2. *For any GREEDY policy, G, and any $(i,j) \in \mathcal{S}$,*

$$V^G(i+1,j) > V^G(i,j+1).$$

Proof. We begin by defining V_n^G as follows:

$$V_{n+1}^G(i,j) = A_n^G(i,j) + I_n^G(i,j)$$

where

$$A_n^G(i,j) = c(i,j) + \lambda_1 \left(V_n^G(i+1,j) - V_n^G(i,j)\right) + \lambda_2 \left(V_n^G(i,j+1) - V_n^G(i,j)\right)$$

and

$$I_n^G = \gamma_1^G(i,j) \left(V_n^G((i-1)^+,j) - V_n^G(i,j)\right) + \gamma_2^G(i,j) \left(V_n^G(i,(j-1)^+) - V_n^G(i,j)\right)$$

and $V_0^G(i,j) = i + j + \frac{i}{i+j+1}$ for all $(i,j) \in \mathcal{S}$.

Recall that $\gamma_i^G(i,j)$ denotes the departure rate of class i jobs from state (i,j) under policy G, λ_i denotes the arrival rate of class i jobs, and $c(i,j) = i + j$. From [81] we know that

$$\lim_{n\to\infty} V_n^G(i,j) - V_n^G(y_1,y_2) = V^G(i,j) - V^G(y_1,y_2).$$

Thus, if we can prove that our claim holds for $V_n^G(i,j)$ for all $n \geq 0$, then it must hold for $V^G(i,j)$ as well (see, for example, [56]). We will now prove that all three of the following properties of V_n^G hold for all $n \geq 0$ by induction:

1. $V_n^G(i+1,j) > V_n^G(i,j)$
2. $V_n^G(i,j+1) > V_n^G(i,j)$

3. $V_n^G(i+1,j) > V_n^G(i,j+1)$

Note that the first two properties will be necessary for our proof of the third property, and the lemma follows directly from the third property.

We can easily verify that all three properties hold when $n = 0$ due to our choice of V_0^G. We now wish to show that if these properties hold for V_n^G, they must hold for V_{n+1}^G.

To prove property 1, we wish to show that

$$V_{n+1}^G(i+1,j) - V_{n+1}^G(i,j) = A_n^G(i+1,j) - A_n^G(i,j) + I_n^G(i+1,j) - I_n^G(i,j) > 0.$$

We can easily see that $A_n^G(i+1,j) - A_n^G(i,j) > 0$, since the cost function $c(i,j)$ is increasing in i and $V_n^G(i,j)$ is increasing in i by the property 1 of the inductive hypothesis. To see that $I_n^G(i+1,j) - I_n^G(i,j) > 0$, we can expand the terms as follows:

$$\begin{aligned}
&I_n^G(i+1,j) - I_n^G(i,j) \\
&= \gamma_1^G(i,j)\left(V_n^G(i,j) - V_n^G((i-1)^+,j)\right) \\
&+ \gamma_2^G(i,j)\left(V_n^G(i,j) - V_n^G(i,(j-1)^+)\right) \\
&+ \gamma_2^G(i+1,j)\left(V_n^G(i+1,(j-1)^+) - V_n^G(i,j)\right) \\
&+ \left(1 - \gamma_1^G(i+1,j) - \gamma_2^G(i+1,j)\right)\left(V_n^G(i+1,j) - V_n^G(i,j)\right).
\end{aligned}$$

Each line here is positive by the inductive hypothesis and the fact that

$$\left(1 - \gamma_1^G(i+1,j) - \gamma_2^G(i+1,j)\right) \geq 0$$

since the system was uniformized to one. Thus property 1 holds. The proof of property 2 follows a very similar argument.

To show property 3 we wish to show that

$$V_{n+1}^G(i+1,j) - V_{n+1}^G(i,j+1) = A_n^G(i+1,j) - A_n^G(i,j+1) + I_n^G(i+1,j) - I_n^G(i,j+1) > 0.$$

We can easily see that $A_n^G(i+1,j) - A_n^G(i,j+1) > 0$ by the inductive hypothesis. To see that $I_n^G(i+1,j) - I_n^G(i,j+1) > 0$, we can expand the terms as follows:

$$\begin{aligned}
&I_n^G(i+1,j) - I_n^G(i,j+1) \\
&= \gamma_1^G(i,j+1)\left(V_n^G(i,j) - V_n^G((i-1)^+,j+1)\right) \\
&+ \gamma_2^G(i+1,j)\left(V_n^G(i+1,(j-1)^+) - V_n^G(i,j)\right) \\
&+ \left(\gamma_1^G(i,j+1) + \gamma_2^G(i,j+1) - \gamma_1^G(i+1,j) - \gamma_2^G(i+1,j)\right)\left(V_n^G(i+1,j) - V_n^G(i,j)\right) \\
&+ \left(1 - \gamma_1^G(i,j+1) - \gamma_2^G(i,j+1)\right)\left(V_n^G(i+1,j) - V_n^G(i,j+1)\right).
\end{aligned}$$

We know, by assumption, that $s_1(k) < s_2(k)$ for $k > 1$, and thus the coefficient $\gamma_1^G(i,j+1) + \gamma_2^G(i,j+1) - \gamma_1^G(i+1,j) - \gamma_2^G(i+1,j)$ is non-negative. Therefore, all terms in this sum are positive by the inductive hypothesis, and property 3 holds.

All three properties therefore hold by induction, and $V^G(i+1,j) > V^G(i,j+1)$ as desired. □

We now prove Theorem 6.1 by contradiction. We begin by assuming that there exists a GREEDY policy $G \neq \text{GREEDY}^*$ which is optimal in terms of $\mathbb{E}[N]$ and thus $\mathbb{E}[N]^G \leq \mathbb{E}[N]^{G'}$ for any GREEDY policy G'. Since $G \neq \text{GREEDY}^*$, there exists some state, $(i,j) \in \mathcal{S}$ where G and GREEDY* take different actions. Note that both i and j must be non-zero in this state, because otherwise there is only one action which will achieve the maximal rate of departures, and G and GREEDY* must therefore take the same action.

We now consider a policy G' which takes the same action as GREEDY* in state (i,j), and the same action as G in every other state. We can apply the technique of precedence relations described in [3, 17] to show that $\mathbb{E}[N]^{G'} < \mathbb{E}[N]^G$. G' can now be obtained from G by taking $\gamma_2^G(i,j) - \gamma_2^{G^*}(i,j)$ away from the total completion rate of class 2 jobs and adding $\gamma_1^{G^*}(i,j) - \gamma_1^G(i,j)$ to the total completion rate of class 1 jobs, where G^* denotes GREEDY*. Theorem 3.1 in [17] tells us that $\mathbb{E}[N]^{G'} < \mathbb{E}[N]^G$ if:

$$\left(\gamma_1^{G^*}(i,j) - \gamma_1^G(i,j)\right) V^G(i-1,j) < \left(\gamma_2^G(i,j) - \gamma_2^{G^*}(i,j)\right) V^G(i,j-1).$$

To see that this property holds, first note that

$$\begin{aligned}&\left(\gamma_1^{G^*}(i,j) - \gamma_1^G(i',j')\right) - \left(\gamma_2^G(i,j) - \gamma_2^{G^*}(i,j)\right)\\ =&\left(\gamma_1^{G^*}(i,j) + \gamma_2^{G^*}(i,j)\right) - \left(\gamma_1^G(i,j) + \gamma_2^G(i,j)\right) = 0\end{aligned}$$

since GREEDY* and G have the same (maximal) total rate of departures in every state. Thus,

$$\gamma_1^{G^*}(i,j) - \gamma_1^G(i,j) = \gamma_2^G(i,j) - \gamma_2^{G^*}(i,j).$$

By Lemma 6.2, we know that

$$V^G(i-1,j) < V^G(i,j-1).$$

This implies that $\mathbb{E}[N]^{G'} < \mathbb{E}[N]^G$ which contradicts our assumption that G is optimal in terms of $\mathbb{E}[N]$. By Little's Law, we can reformulate this in terms of $\mathbb{E}[T]$. □

While GREEDY* is the best GREEDY policy, it will turn out that it is not optimal (see Section 6.3.5). Hence, we now turn our attention to computing the optimal policy.

6.3.4 Computing the Optimal Policy

The optimal policy, OPT, must not only consider the current state of the system when choosing how to determine the best partition $(\pi_1(i,j), \pi_2(i,j))$, but must also consider the probabilities of transitioning to future states as well. To find a policy which balances this tradeoff between performance in the current state and future states, we formulate the problem as a Markov Decision Process (MDP).

We will consider an MDP with state space $\mathcal{S} = \{(i,j) : i,j \in \mathbb{N}\}$, where x_i represents the number of class i jobs in the system. The action space in any state is given by

$$\mathcal{A} = \{(\pi_1(i,j), \pi_2(i,j)) : \pi_1(i,j) + \pi_2(i,j) = n\}.$$

Let the arrival rate of class i jobs be given by λ_i. Recall that, given an allocation of a_i cores to x_i type i jobs, it is optimal to run the x_i jobs on these cores using EQUI. Thus, given a state (i, j) and an action $(\pi_1(i, j), \pi_2(i, j))$, the total departure rate of jobs from each class of jobs is given by

$$\mu_1(\pi_1(i,j), i) := \min\{\pi_1(i,j), i\}\mu s\left(\max\left\{1, \frac{\pi_1(i,j)}{i}\right\}\right)$$

and

$$\mu_2(\pi_2(i,j), i) := \min\{\pi_2(i,j), i\}\mu s\left(\max\left\{2, \frac{\pi_2(i,j)}{i}\right\}\right).$$

We choose the cost function

$$c(i, j) = i + j$$

such that the *average cost per period* equals the average number of jobs in the system, $\mathbb{E}[N]$. We uniformize the system at rate 1 (always achievable by scaling time) and find that Bellman's optimality equations [81] for this MDP are given by

$$\mathbb{E}\left[N^{OPT}\right] + V^{OPT}(i,j) = A^{OPT}(i,j) + H^{OPT}(i,j),$$

where

$$\begin{aligned} A^{OPT}(i,j) = \; & c(i,j) + \lambda_1\left(V^{OPT}(i+1,j) - V^{OPT}(i,j)\right) \\ & + \lambda_2\left(V^{OPT}(i,j+1) - V^{OPT}(i,j)\right), \end{aligned} \tag{6.4}$$

$$H^{OPT}(i,j) = V^{OPT}(i,j) + \min_{(\pi_1(i,j),\pi_2(i,j))\in\mathcal{A}}\Bigg\{$$

$$\begin{aligned} & \mu_1(\pi_1(i,j), i)\left(V^{OPT}\left((i-1)^+, j\right) - V^{OPT}(i,j)\right) \\ & +\mu_2(\pi_2(i,j), j)\left(V^{OPT}\left(i, (j-1)^+\right) - V^{OPT}(i,j)\right)\Bigg\}. \end{aligned} \tag{6.5}$$

Here, the value function $V^{OPT}(i, j)$ denotes the asymptotic total difference in accrued costs when using the optimal policy and starting the system in state (i, j) instead of some reference state. While these equations are hard to solve analytically, the optimal actions can be obtained numerically by defining

$$V^{OPT}_{n+1}(i,j) = A^{OPT}_n(i,j) + H^{OPT}_n(i,j)$$

with $V^{OPT}_0(\cdot,\cdot) = 0$. Here, A^{OPT}_n and H^{OPT}_n are defined as in (6.4) and (6.5), but in terms of V^{OPT}_n. We can then perform *value iteration* to extract the optimal policy [64]. We use the results of this value iteration to compare the performance of OPT to several other policies in Figure 6.2 and Figure 6.3.

Note that using the same MDP formulation when there exists only one speedup function results in a much simpler expression for V^{OPT} which clearly yields EQUI.

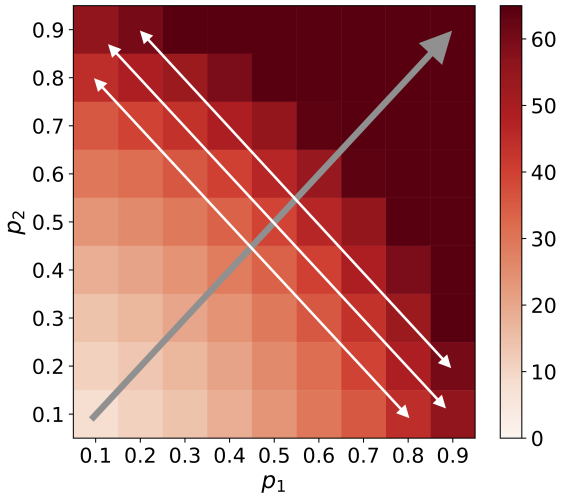


Figure 6.3: Heat map showing the percentage difference in the mean response time, $\mathbb{E}[T]$, between JSQ-Chunk with the optimal k^* and OPT , in the case of two speedup functions where s_1 and s_2 are Amdahl's law with parameters p_1 and p_2 respectively. Here $\mathbb{E}[X_1] = \mathbb{E}[X_2] = 1$ and $\lambda_1 = \lambda_2 = 2$. The axes represent the value of p_i for each class. OPT was evaluated numerically using the MDP formulation given in Section 6.3.4 and the results for JSQ-Chunk come from analysis.

6.3.5 GREEDY* is Near-Optimal

Surprisingly, even GREEDY* is not optimal for minimizing mean response time as shown in Figure 6.2b (although it is always within 1% of OPT in the figure). The same intuition that led us to believe that GREEDY* is the best GREEDY policy can be used to explain why GREEDY* is not optimal. We have already seen that deferring parallelizable work is advantageous to GREEDY*. However, we were only comparing GREEDY* to policies with maximal overall departure rate. It turns out that the advantage of deferring parallelizable work can be so great that a policy stands to benefit from achieving a submaximal overall rate of departures in order to defer parallelizable work.

6.3.6 Does Fixed-Width Scheduling Work?

We saw in Chapter 4 that, when jobs of unknown size follow a single speedup function, the JSQ-Chunk policy with the optimal chunk size k^* will often achieve near-optimal performance (see Section 4.5.2). Is this still true when jobs are permitted to have different speedup curves? Figure 6.3 compares JSQ-Chunk with OPT for the case of two speedup functions. We see two trends:

Trend 1: Along the thick arrow in Figure 6.3, JSQ-Chunk becomes further from OPT as $p_1 + p_2$ increases.

Trend 1 makes intuitive sense since, when jobs are more parallelizable, OPT will be able to more effectively exploit this parallelism while JSQ-Chunk will be limited by being restricted to use a single k^* for each job. Nonetheless, trend 1 becomes irrelevant as the number of cores, n, increases, since JSQ-Chunk converges to OPT when p_1 equals p_2 (single speedup function). Thus, we are more interested in trend 2.

Trend 2: Along any thin diagonal in Figure 6.3 ($p_1 + p_2 =$ constant), JSQ-Chunk becomes further from OPT as we move outward on the diagonal.

Trend 2 follows from two observations. First, Theorem 6.3 proves that JSQ-Chunk's performance is fixed along these diagonals. Second, we have reason to believe that the mean response time under OPT should decrease as we move further outward along these diagonals (see Observation 6.4). Together, these observations explain trend 2. The rest of this section discusses Theorem 6.3 and Observation 6.4.

Theorem 6.3. *Given two speedup functions, s_1 and s_2, where s_1 follows Amdahl's law with parameter p_1 and s_2 follows Amdahl's law with parameter p_2, and $\lambda_1 = \lambda_2$, the mean response time under JSQ-Chunk with the optimal k^* is constant for any (p_1, p_2) such that $p_1 + p_2 = c$, where c is a constant.*

Proof. In the case of two speedup functions, under JSQ-Chunk with level of parallelization, k, X_k from (2.1) becomes:

$$X_k = \begin{cases} \frac{X}{s_1(k)} & \text{w.p.} \quad \frac{\lambda_1}{\lambda_1+\lambda_2} \\ \frac{X}{s_2(k)} & \text{w.p.} \quad \frac{\lambda_2}{\lambda_1+\lambda_2} \end{cases}.$$

We now prove that, for any given k, $\mathbb{E}[X_k]$ is constant whenever $p_1 + p_2 = c$. By definition, when $\lambda_1 = \lambda_2$,

$$\mathbb{E}[X_k] = \frac{1}{2} \cdot \frac{\mathbb{E}[X]}{s_1(k)} + \frac{1}{2} \cdot \frac{\mathbb{E}[X]}{s_2(k)}. \tag{6.6}$$

Since s_1 and s_2 are both instances of Amdahl's law, we can use a property of Amdahl's law that states

$$\frac{1}{2} \cdot \frac{1}{s_1(k)} + \frac{1}{2} \cdot \frac{1}{s_2(k)} = \frac{1}{s_3(k)}, \tag{6.7}$$

where $s_3(k)$ is the speedup function for Amdahl's law with parameter $p_3 = \frac{p_1+p_2}{2}$. Combining (6.6) and (6.7) we have

$$\mathbb{E}[X_k] = \frac{\mathbb{E}[X]}{s_3(k)},$$

which is a constant, provided that $p_1 + p_2 = c$.

As stated in Section 4.4.2, the performance of JSQ-Chunk depends *only* on $\mathbb{E}[X_k]$. Hence, along any diagonal where $p_1 + p_2 = c$, the mean response time under JSQ-Chunk with chunk size k remains constant. Since any choice of k leads to constant performance along the diagonal, setting k to k^* will also keep JSQ-Chunk constant along this diagonal. Hence, the mean response time under JSQ-Chunk with the optimal k^* is constant when $p_1 + p_2 = c$. □

Observation 6.4. *Along any diagonal where $p_1 + p_2 = c$, where c is a constant and $\lambda_1 = \lambda_2$, we expect the mean response time under OPT to decrease as $\mid p_1 - p_2 \mid$ increases.*

Although the mean job size is constant along this diagonal, when $\mid p_1 - p_2 \mid$ is higher, the optimal policy has additional information about which jobs will take longer to run. In effect, this allows OPT to favor jobs which benefit more from parallelization and leads to a lower overall mean response time. A similar effect was observed in [106], where assigning jobs different service rates lowered optimal mean response time.

Remark 6.5. *Although Theorem 6.3 and Observation 6.4 assume that $\lambda_1 = \lambda_2$, they are easily generalized to cases where the arrival rates are not equal. In these cases, the diagonals along which $\mathbb{E}[X_k]$ will be constant will have a different slope. Nonetheless, trend 2 will still occur along these diagonals.*

Unlike trend 1, which disappears as the number of cores, n, increases, we do not expect trend 2 to vanish. This is supported by our evaluation of OPT under higher values of n (not shown). We thus conclude that JSQ-Chunk does not perform near-optimally when jobs follow multiple speedup functions.

6.4 The Case of Elastic and Inelastic Jobs

The results of the prior section showed that, when the form of our speedup functions is unconstrained, computing an optimal policy can be computationally intensive. In this section, we consider an important special case where jobs are assumed to either be perfectly parallelizable, or not at all parallelizable.

6.4.1 Elastic and Inelastic Jobs in the Real World

It is common to find systems which use a shared set of resources to process both elastic and inelastic jobs. In such settings, elastic jobs typically have more inherent work than the inelastic jobs. For example, consider a cluster which must process a stream of many MapReduce jobs [22]. From the cluster's point of view, this workload produces a stream of *map stages* and *reduce stages*. Map stages (elastic) are designed to be highly parallelized and do the bulk of the computational work. Reduce stages (inelastic) are inherently sequential and do much less total work than a map stage. As another example, modern machine learning frameworks [67] advocate the use of a single platform for both the training and serving of models. Training jobs (elastic) are large, requiring large data sets and many training epochs. Distributed training methods such as distributed stochastic gradient descent are also designed to scale out across an arbitrary number of nodes [61]. Once a model has been trained, serving the model (inelastic), which consists of feeding a computed model a single data point in order to retrieve a single prediction, is done sequentially and requires comparatively little processing power.

It is less common for elastic jobs to be smaller than inelastic jobs in practice, given the overhead involved in writing parallel code. If the amount of inherent work required for a job is small to begin with, system developers may not choose to add the additional data structures and synchronization mechanisms required to make the job elastic. One exception is HPC workloads. In this setting, there are often both malleable jobs (elastic) [30] and jobs with hard requirements (inelastic). While malleable jobs are designed to run on any number of cores, jobs with hard requirements demand a fixed number of cores. It is unclear which class of jobs we would expect to involve more inherent work.

6.4.2 Optimal Scheduling of Elastic and Inelastic Jobs

This section will consider both of the above cases. Recall that we allow elastic and inelastic jobs to follow different exponential size distributions with rates μ_E and μ_I respectively. First, we show that if $\mu_I \geq \mu_E$, then IF is optimal for minimizing mean response time. Second, we show that if $\mu_I < \mu_E$, then IF is not necessarily optimal.

In Section 6.4.3, we begin by considering the special case where $\mu_I = \mu_E$. In this case where we have homogeneous sizes, our analysis follows directly from Theorem 6.1 in the previous section. Unfortunately, we will show that the argument used to extend Theorem 6.1 does not extend to the case where $\mu_I \neq \mu_E$.

In Section 6.4.4, we therefore develop a new argument to handle the case where $\mu_I \geq \mu_E$. Using a novel sample path argument, we prove that IF is the optimal policy in this case.

Lastly, in section 6.4.5, we consider the case where $\mu_I < \mu_E$. Here, we construct a very simple example demonstrating that IF is not optimal in this environment. Furthermore, in this example, we show the policy EF actually outperforms IF. We do not know what policy is optimal in this regime.

6.4.3 Optimality when $\mu_I = \mu_E$

We first consider the case where $\mu_I = \mu_E$. In this case, IF is optimal with respect to minimizing mean response time. As stated in Section 6.1.1, the optimal policy should balance the tradeoff between completing jobs quickly and preserving future system efficiency. When $\mu_I = \mu_E$, IF maximizes future system efficiency without reducing the instantaneous system efficiency. We argue this formally in Theorem 6.6 by leveraging Theorem 6.1.

Theorem 6.6. *If jobs are either elastic or inelastic,* IF *is optimal with respect to minimizing mean response time when $\mu_I = \mu_E$.*

Proof. Following the terminology of Section 6.3, we can consider GREEDY and GREEDY* in the context of the case where jobs are either elastic or inelastic. In this case, we can clearly apply Theorem 6.1 to see that

$$\mathbb{E}[T]^{\text{GREEDY}^*} = \min_{\pi \in GREEDY} \mathbb{E}[T^\pi]. \tag{6.8}$$

To leverage this result, we note that when $\mu_I = \mu_E$ in our model, a policy is in GREEDY if and only if it does not idle servers unnecessarily.

We now argue that IF, which is non-idling, must be equivalent to GREEDY*. Consider the server allocations made by GREEDY* in any state (i, j). In states where IF allocates zero servers to elastic jobs, $\texttt{IF}_E(i, j)$ is clearly minimal. In any state (i, j) where $\texttt{IF}_E(i, j) > 0$, servers cannot be reallocated from elastic jobs to inelastic jobs, since all i inelastic jobs must already be in service. Hence, reducing $\texttt{IF}_E(i, j)$ in this case results in a policy which is not in GREEDY. $\texttt{IF}_E(i, j)$ is therefore minimal amongst GREEDY policies in any state (i, j), and IF is equivalent to GREEDY*.

We show in Appendix C.1 that there exists an optimal policy which is non-idling. Hence, when $\mu_I = \mu_E$, there is an optimal policy in GREEDY. This implies that GREEDY* must be optimal

with respect to mean response time. Thus, IF, which is equivalent to GREEDY*, is optimal with respect to mean response time. □

Why the prior argument does not generalize

Unfortunately, the results of Section 6.3 do not extend to the case where $\mu_I \neq \mu_E$. In particular, the proof of Theorem 6.1 uses a precedence relation between any two states $(i, j-1)$ and $(i-1, j)$. This claim essentially states that a policy π in state (i, j) would perform better by transitioning to state $(i-1, j)$ than it would by transitioning to state $(i, j-1)$. In the case where $\mu_I = \mu_E$, this makes perfect intuitive sense. In this case, both states $(i-1, j)$ and $(i, j-1)$ contain the same amount of expected total work. Hence, it is better to be in state $(i-1, j)$, which benefits from having an additional elastic job. Consider how this intuition changes when $\mu_I > \mu_E$. In this case, state $(i, j-1)$ has less expected total work, but state $(i-1, j)$ has more expected elastic work. It turns out that the precedence relation shown in Section 6.3 no longer holds when $\mu_I \neq \mu_E$. Moreover, even if the precedence relations were to hold when $\mu_I > \mu_I$, Theorem 6.1 would yield that GREEDY* is optimal amongst GREEDY policies, not optimal amongst all policies. We must therefore devise a new argument to reason about the optimal allocation policy for elastic and inelastic jobs when these jobs follow different size distributions.

6.4.4 Optimality when $\mu_I \geq \mu_E$

We will show IF is optimal in the more general case of $\mu_I \geq \mu_E$. While our goal is to minimize mean response time, we note that via Little's Law [35], it suffices to minimize the mean total number of jobs in the system. [1]

First, we start by defining a class of policies $\mathcal{P}$ which serve inelastic jobs on a first-come-first-serve (FCFS) basis; elastic jobs can be served in any order. In more detail, a policy π is said to be in class $\mathcal{P}$ if the following hold true:

1. π is work-conserving.
2. π serves inelastic jobs in FCFS order. In particular, if π allocates N servers to inelastic jobs at time t (N may be fractional, and there may be more than N inelastic jobs in the systems), the allocation must give $\lfloor N \rfloor$ servers to the $\lfloor N \rfloor$ inelastic jobs with the earliest arrival times. If there is a remaining fraction of a server, it may then be allocated to the inelastic job with the next earliest arrival time.

Clearly, IF $\in \mathcal{P}$.

Road map: Theorem 6.7 argues that we only need to compare IF to policies in $\mathcal{P}$. Specifically, $\mathcal{P}$ contains some optimal policy that minimizes the mean number of jobs in system and mean response time.

Next, in Theorem 6.8 we present a novel sample path argument which shows that IF has stochastically less work in the system than any policy in $\mathcal{P}$. We will directly leverage this fact to show that, out of all policies $\pi \in \mathcal{P}$, IF has the least expected inelastic work in system and also the least expected total work in system.

[1] Little's Law states that for any ergodic system with average total arrival rate λ, the mean response time, $\mathbb{E}[T]$ is related to the mean total number of jobs in system, $\mathbb{E}[N]$ via the formula $\mathbb{E}[T] = \frac{\mathbb{E}[N]}{\lambda}$.

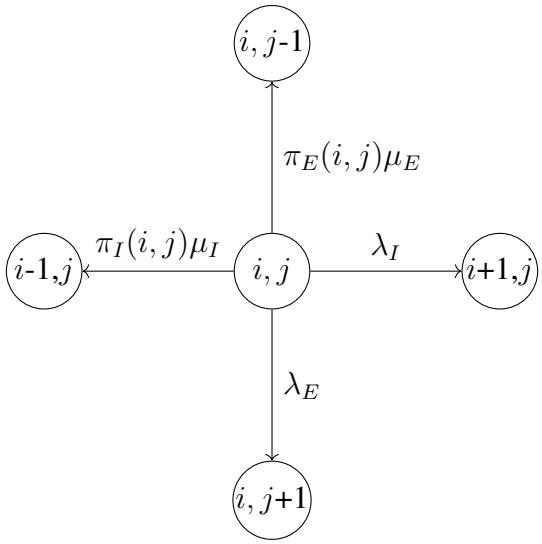


Figure 6.4: The Markov chain $(N_I^\pi(t), N_E^\pi(t))$ for a stationary, deterministic, work-conserving allocation policy, π.

Finally, In Theorem 6.10 we show that, of all policies in $\mathcal{P}$, IF minimizes the expected number of jobs in system. Thus, by Little's Law, IF is optimal with respect to mean response time.

Analysis. We now present Theorem 6.7.

Theorem 6.7. *The class $\mathcal{P}$ contains a policy π which minimizes both mean response time and mean number of jobs in system. Specifically*

$$\mathbb{E}\left[N^\pi\right] = \min_{\pi'} \left\{\mathbb{E}\left[N^{\pi'}\right]\right\},$$

and

$$\mathbb{E}\left[T^\pi\right] = \min_{\pi'} \left\{\mathbb{E}\left[T^{\pi'}\right]\right\},$$

where N^π is the total number of jobs in the system in steady-state under policy π, and T^π is the response time of a job in the system under π in steady-state.

Proof. Recall that we will consider only stationary, deterministic, work-conserving policies which make allocation decisions based on state (i, j). Let π be a stationary, deterministic, work-conserving policy with the minimal mean number of jobs in system. Figure 6.4 shows the transition rates out of state (i, j) under π.

We see that the transition rates out of the current state (i, j) under policy π depend solely on the number of servers allocated to each type of job. Thus, neither the order in which we serve the jobs nor how many jobs of each type are running matter. In particular, we can construct a policy π' such that, for any state (i, j),

$$\pi_I(i,j) = \pi'_I(i,j) \qquad \text{and} \qquad \pi_E(i,j) = \pi'_E(i,j)$$

and π' serves inelastic jobs in FCFS order. The policy π' has the same Markov chain as π, so the expected numbers of jobs in system under π and π' are identical. Because π is work-conserving, π' is also work-conserving. Hence, π' is in $\mathcal{P}$ and achieves the minimal mean number of jobs in system. □

The power of Theorem 6.7 is that, to show IF is optimal with respect to mean response time, it now suffices to show:

$$\mathbb{E}\left[N^{\text{IF}}\right] \leq \mathbb{E}\left[N^{\pi}\right] \qquad \forall \pi \in \mathcal{P}. \tag{6.9}$$

However, it is hard to directly compare the *numbers of jobs* under different policies. We get around this roadblock by instead analyzing how the *remaining work* in the system under IF relates to other policies $\pi \in \mathcal{P}$. In particular, we obtain the following strong result.

Theorem 6.8. *For all policies $\pi \in \mathcal{P}$, if we assume that*

$$(N_I^{\pi}(0), N_E^{\pi}(0)) = (N_I^{\text{IF}}(0), N_E^{\text{IF}}(0)),$$

then:

$$W^{\text{IF}}(t) \leq_{ST} W^{\pi}(t) \text{ and } W_I^{\text{IF}}(t) \leq_{ST} W_I^{\pi}(t) \qquad \forall t \geq 0,$$

where $W^{\pi}(t)$ is the total remaining work under policy π at time t, $W_I^{\pi}(t)$ is the remaining inelastic work under policy π at time t, and $\leq_{ST}$ denotes stochastic dominance.

Proof. Fix an arbitrary policy $\pi \in \mathcal{P}$, and let us consider a fixed *arrival sequence*, that is, a fixed sequence of arrival times and job sizes. We couple π and IF under this sequence. Here, it suffices to consider arrival sequences where the total number of job arrivals up to any time t is finite, as this occurs with probability 1.

Recall that $W_I^{\pi}(t)$ and $W_E^{\pi}(t)$ are respectively the remaining inelastic and elastic work in the system at time t under scheduling policy π. Furthermore, also recall that $W^{\pi}(t)$, the total work at time t, is given by:

$$W^{\pi}(t) = W_I^{\pi}(t) + W_E^{\pi}(t).$$

In order to show the desired stochastic dominance relations, it will suffice to show that on any such arrival sequence

$$W_I^{\text{IF}}(t) \leq W_I^{\pi}(t) \quad \text{and} \quad W^{\text{IF}}(t) \leq W^{\pi}(t) \quad \forall t \geq 0.$$

First, we see it is immediate that, under our arrival sequence, $W_I^{\text{IF}}(t) \leq W_I^{\pi}(t)$ for all $t \geq 0$. Since IF and π process inelastic jobs in FCFS order, each inelastic job enters service at least as early under IF as it does under π. Furthermore, IF never preempts inelastic jobs. Hence, at each time t, the remaining size of each inelastic job that has arrived by time t is no larger under IF than it is under π. Since the inelastic work in system is just the sum of the remaining sizes of inelastic jobs, the total inelastic work at time t under IF is less than the total inelastic work at time t under π.

It remains to show that

$$W^{\text{IF}}(t) \leq W^{\pi}(t) \qquad \forall t \geq 0. \tag{6.10}$$

We prove our claim by induction. For a base case, it is clear that $W^{\text{IF}}(0) \leq W^{\pi}(0)$, as the policies have the same set of jobs at time zero, and no work has been completed. For any time t, we partition the interval $[0, t)$ into subintervals $[t_i, t_{i+1})$ such that either

1. IF allocates all n servers on $[t_i, t_{i+1})$, or
2. IF allocates strictly less than n servers on $[t_i, t_{i+1})$.

We now induct on i, and show that $W^{\text{IF}}(t_i) \leq W^{\pi}(t_i)$ implies $W^{\text{IF}}(t_{i+1}) \leq W^{\pi}(t_{i+1})$.

If the interval $[t_i, t_{i+1})$ falls into case (1), IF is completing work at the maximal rate of any policy. In particular, IF completes exactly $(t_{i+1} - t_i) \cdot n$ work on $[t_i, t_{i+1}]$. Let ω denote the work completed by π on $[t_i, t_{i+1})$. Then, we must have $\omega \leq (t_{i+1} - t_i) \cdot n$. Since IF and π experience the same set of arrivals on this interval, we have:

$$\begin{aligned} W^{\pi}(t_{i+1}) - W^{\text{IF}}(t_{i+1}) &= (W^{\pi}(t_i) - \omega) - (W^{\text{IF}}(t_i) - (t_{i+1} - t_i) \cdot n) \\ &= (W^{\pi}(t_i) - W^{\text{IF}}(t_i)) + ((t_{i+1} - t_i) \cdot n - \omega) \\ &\geq 0. \end{aligned}$$

Thus, we have $W^{\text{IF}}(t_{i+1}) \leq W^{\pi}(t_{i+1})$, as desired.

If the interval $[t_i, t_{i+1})$ falls into case (2), IF allocates strictly less than n servers on $[t_i, t_{i+1})$. We aim to show that $W^{\text{IF}}(t_{i+1}) \leq W^{\pi}(t_{i+1})$. Observe that IF can have no elastic jobs in its system on $[t_i, t_{i+1})$. This is because we have defined IF to be work-conserving. Hence, if there was an elastic job, IF would run it on all available servers.

Observe that, assuming no elastic job arrives at time t_{i+1},

$$W^{\text{IF}}(t_{i+1}) = W_I^{\text{IF}}(t_{i+1}).$$

Likewise, we know

$$W^{\pi}(t_{i+1}) = W_I^{\pi}(t_{i+1}) + W_E^{\pi}(t_{i+1}) \geq W_I^{\pi}(t_{i+1}).$$

We get the inequality above because π cannot have negative elastic work at time t_{i+1}. Finally, we have

$$\begin{aligned} W^{\pi}(t_{i+1}) - W^{\text{IF}}(t_{i+1}) &= (W_I^{\pi}(t_{i+1}) + W_E^{\pi}(t_{i+1})) - W_I^{\text{IF}}(t_{i+1}) \\ &= (W_I^{\pi}(t_{i+1}) - W_I^{\text{IF}}(t_{i+1})) + W_E^{\pi}(t_{i+1}) \\ &\geq W_I^{\pi}(t_{i+1}) - W_I^{\text{IF}}(t_{i+1}) \\ &\geq 0, \end{aligned}$$

where the last inequality follows from the fact that $W_I^{\text{IF}}(t') \leq W_I^{\pi}(t')$ for all $t' \geq 0$. Thus, we have $W^{\text{IF}}(t_{i+1}) \leq W^{\pi}(t_{i+1})$.

Note that some elastic work could arrive at exactly time t_{i+1}. However, this increases the total work in both systems by the same amount and thus has no effect on the ordering of these quantities.

Thus, for any interval $[t_i, t_{i+1})$, if $W^{\text{IF}}(t_i) \leq W^{\pi}(t_i)$, then we have $W^{\text{IF}}(t_{i+1}) \leq W^{\pi}(t_{i+1})$. Since $W^{\text{IF}}(0) \leq W^{\pi}(0)$, it follows that this inequality holds at the end of the last subinterval. The end of this final subinterval is exactly time t. Thus, for any $t \geq 0$, we have $W^{\text{IF}}(t) \leq W^{\pi}(t)$, as desired.

We have thus found a coupling of π and IF such that the amount of total work and the amount of inelastic work in each system is ordered at every moment in time. This implies that

$$W_I^{\text{IF}}(t) \leq_{ST} W_I^{\pi}(t) \quad \forall t \geq 0$$

and

$$W^{\text{IF}}(t) \leq_{ST} W^{\pi}(t) \quad \forall t \geq 0$$

as desired. □

In other words, IF is the best policy in $\mathcal{P}$ for minimizing remaining inelastic and total work in the system. One possible explanation for this is that, by deferring parallelizable work, IF ensures that all n servers are saturated with work for as long as possible.

We now understand that, out of all policies in $\mathcal{P}$, IF is optimal with respect to minimizing both expected remaining inelastic work *and* expected remaining total work at any time t. We now establish a relationship between expected remaining work and expected number of jobs in system.

Lemma 6.9. *For any policy π, we have:*

$$\mathbb{E}[W_I^{\pi}] = \frac{1}{\mu_I}\mathbb{E}[N_I^{\pi}] \qquad \textit{and} \qquad \mathbb{E}[W_E^{\pi}] = \frac{1}{\mu_E}\mathbb{E}[N_E^{\pi}],$$

where W_I^{π} and N_I^{π} are respectively the inelastic work and number of inelastic jobs in the system in steady-state under policy π, and where the size of an inelastic job is $X_I \sim Exp(\mu_I)$. W_E^{π}, N_E^{π}, and μ_E are the analogous quantities for elastic jobs.

Proof. We do the proof for the inelastic relationship, but the proof for the elastic relationship is identical. Let the random variable $N_I^{\pi}(t)$ denote the number of inelastic jobs in the system under policy π at time t. For every $\ell \in \{1, \ldots, N_I^{\pi}(t)\}$ we define $R_{\ell,I}^{\pi}(t)$ as the *remaining size* of inelastic job ℓ under policy π at time t.

Recall $W_I^{\pi}(t)$ is the remaining inelastic work in system at time t under policy π. We have the following equivalence:

$$W_I^{\pi}(t) = \sum_{\ell=1}^{N_I^{\pi}(t)} R_{\ell,I}^{\pi}(t).$$

By the memoryless property of the exponential distribution, the remaining size of jobs $\ell \in \{1, \ldots, N_I^{\pi}(t)\}$ are exponentially distributed. Specifically, $R_{\ell,I}^{\pi}(t) \sim Exp(\mu_I)$, for any policy π or time t. Thus, $N_I^{\pi}(t)$ and $R_{\ell,I}^{\pi}(t)$ are independent and we have that

$$\begin{aligned}\mathbb{E}[W_I^{\pi}(t)] &= \mathbb{E}[R_{\ell,I}^{\pi}(t)] \cdot \mathbb{E}[N_I^{\pi}(t)] \\ &= \mathbb{E}[X_I] \cdot \mathbb{E}[N_I^{\pi}(t)].\end{aligned}$$

As shown in Appendix C.2, $\mathbb{E}[N_I^{\pi}(t)]$ converges to $\mathbb{E}[N_I^{\pi}]$ as $t \to \infty$. This implies the convergence of $\mathbb{E}[W_I^{\pi}(t)]$. Thus, taking the limit as $t \to \infty$ yields

$$\mathbb{E}[W_I^{\pi}] = \mathbb{E}[X_I] \cdot \mathbb{E}[N_I^{\pi}] = \frac{1}{\mu_I} \cdot \mathbb{E}[N_I^{\pi}],$$

as desired [2]. □

We can now show that IF has the lowest expected number of jobs in system when $\mu_I \geq \mu_E$.

Theorem 6.10. *For any policy π, if $\mu_I \geq \mu_E$, we have:*

$$\mathbb{E}\left[N^{\text{IF}}\right] \leq \mathbb{E}\left[N^{\pi}\right].$$

And via Little's Law, we have:

$$\mathbb{E}\left[T^{\text{IF}}\right] \leq \mathbb{E}\left[T^{\pi}\right].$$

Proof. Because there exists an optimal work-conserving policy in $\mathcal{P}$, it suffices to consider any policy $\pi \in \mathcal{P}$. We write total work under π as $W^{\pi} = W_I^{\pi} + W_E^{\pi}$. Likewise, we have the equality $N^{\pi} = N_I^{\pi} + N_E^{\pi}$. First, from Lemma 6.9, we have the following equalities:

$$\mathbb{E}[W_I^{\pi}] = \frac{1}{\mu_I}\mathbb{E}[N_I^{\pi}] \qquad \text{and} \qquad \mathbb{E}[W_E^{\pi}] = \frac{1}{\mu_E}\mathbb{E}[N_E^{\pi}].$$

Furthermore, by the stochastic dominance results of Theorem 6.8,

$$\mathbb{E}[W_I^{\text{IF}}] \leq \mathbb{E}[W_I^{\pi}] \quad \text{and} \quad \mathbb{E}[W^{\text{IF}}] \leq \mathbb{E}[W^{\pi}]$$

Thus, we have:

$$\begin{aligned}
\mathbb{E}\left[N^{\text{IF}}\right] &= \mathbb{E}\left[N_I^{\text{IF}} + N_E^{\text{IF}}\right] \\
&= \mu_I \mathbb{E}\left[W_I^{\text{IF}}\right] + \mu_E \mathbb{E}\left[W_E^{\text{IF}}\right] \\
&= (\mu_I - \mu_E)\mathbb{E}\left[W_I^{\text{IF}}\right] + \mu_E \mathbb{E}\left[W_I^{\text{IF}} + W_E^{\text{IF}}\right] \\
&\leq (\mu_I - \mu_E)\mathbb{E}\left[W_I^{\pi}\right] + \mu_E \mathbb{E}\left[W_I^{\pi} + W_E^{\pi}\right] \qquad (6.11) \\
&= \mu_I \mathbb{E}\left[W_I^{\pi}\right] + \mu_E \mathbb{E}\left[W_E^{\pi}\right] \\
&= \mathbb{E}[N_I^{\pi}] + \mathbb{E}[N_E^{\pi}] = \mathbb{E}[N^{\pi}]
\end{aligned}$$

Note, we leverage the fact $\mu_I \geq \mu_E$ in (6.11). If $\mu_E > \mu_I$, then $\mu_I - \mu_E$ would be negative, so we would not be able establish a relationship like (6.11). This completes the proof. □

We have therefore established that IF is optimal with respect to mean response time when $\mu_I \geq \mu_E$.

6.4.5 Failure when $\mu_I < \mu_E$

Now, we consider the case when $\mu_I < \mu_E$. Here, we demonstrate that IF is not optimal in minimizing mean response time. In fact, IF is not even optimal in the simplified environment where there are only two servers and no arrivals. We construct our counterexample in Theorem 6.11 below.

[2]Technically, we have only proven that $\mathbb{E}[W_I^{\pi}(t)]$ converges to *some value*, but not that it converges to $\mathbb{E}[W_I^{\pi}]$. This would be sufficient for our subsequent results. In fact, $\mathbb{E}[W_I^{\pi}(t)]$ converges to $\mathbb{E}[W_I^{\pi}]$ as $t \to \infty$, but we omit this proof for brevity.

Theorem 6.11. *In general,* IF *is not optimal for minimizing mean response time when* $\mu_I < \mu_E$.

Proof. Assume we have $n = 2$ servers, $\mu_E = 2\mu_I$, and there are no arrivals. We show that, if the system starts with two inelastic jobs and one elastic job, the policy EF outperforms IF.

We directly compute the mean response time for both policies, starting with IF. We let T^{IF} denote response time under IF, and T^{EF} denote response time under elastic first. We have:

$$\mathbb{E}\left[T^{\text{IF}}\right] = \frac{3}{2\mu_I} + \frac{2}{\mu_I + \mu_E} + \frac{\mu_I}{\mu_I + \mu_E}\left(\frac{1}{2\mu_E}\right) + \frac{\mu_E}{\mu_I + \mu_E}\left(\frac{1}{\mu_I}\right) = \frac{35}{12\mu_I}.$$

On the other hand, we see:

$$\mathbb{E}[T^{\text{EF}}] = \frac{3}{2\mu_E} + \frac{2}{2\mu_I} + \frac{1}{\mu_I} = \frac{33}{12\mu_I}.$$

In particular, we have $\mathbb{E}[T^{\text{EF}}] < \mathbb{E}\left[T^{\text{IF}}\right]$. Thus, in general, IF is not optimal when $\mu_I < \mu_E$. In fact, in this environment, we see EF outperforms IF. □

6.5 Response Time Analysis with Elastic and Inelastic Jobs

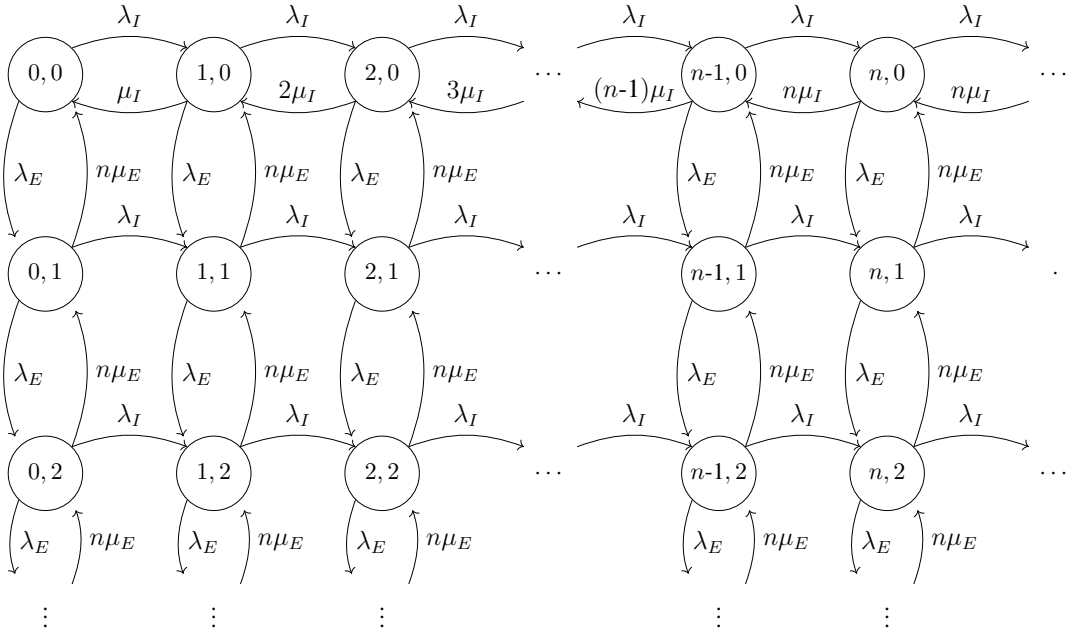
From the results of Section 6.4, we know that IF is optimal with respect to mean response time when $\mu_I \geq \mu_E$. However, Section 6.4 also shows that EF can outperform IF when $\mu_I < \mu_E$. This begs the question of which allocation policy, IF or EF, performs better for given values of μ_I and μ_E.

In this section we derive the mean response time for EF under a range of values of μ_I, μ_E, λ_I, λ_E, and n. First, in Section 6.5.1, we present the Markov chains for EF. This Markov chain is 2D-infinite. Then, in Section 6.5.2, we present a technique from the stochastic literature called Busy Period Transitions [77, 78] which reduces the 2D-infinite chain to a 1D-infinite chain. Although Busy Period Transitions produce an approximation, it is known to be highly accurate, with errors of less than 1% [39, 40, 41, 77, 78]. Finally, in Section 6.5.3, we apply standard Matrix-Analytic methods to solve the 1D-infinite Markov chain, obtaining the stationary distribution and finally the mean response time under EF. The analysis for the IF policy is similar, and thus we defer it to Appendix C.3.

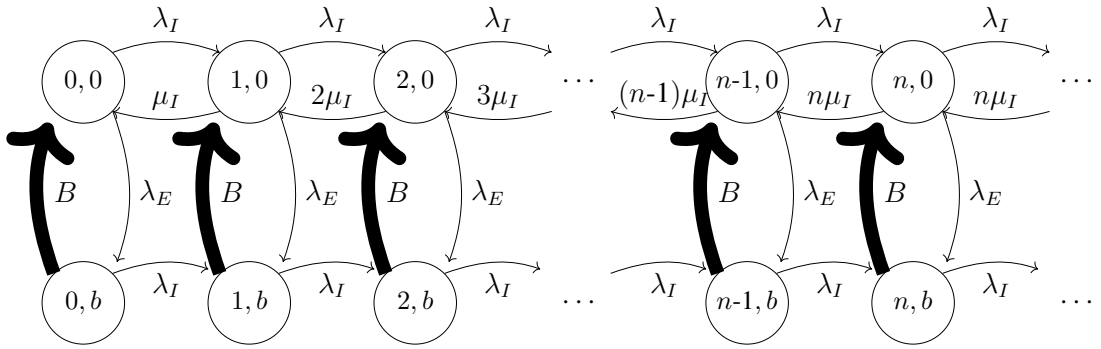
The results of our analysis for IF and EF are shown in Figures 6.6, 6.7, and 6.8. We compared our analysis with simulation, and all numbers agree within 1%. Note that in Section 6.3, we used MDP-based techniques that required truncating the state space and were computationally intensive. The techniques presented in this section do not require truncating the state space, can be tuned to arbitrary precision, and are comparatively efficient.

Figure 6.6 presents an overview of our results, showing only the relative performance of IF and EF as the system load, ρ, is moved from (a) low load to (b) medium load to (c) high load. In every case, IF outperforms EF when $\mu_I \geq \mu_E$, as expected from Theorem 6.10. When $\mu_I < \mu_E$, Figure 6.6 shows us that EFcan outperform IF, and that the region where EF is better grows as ρ increases.

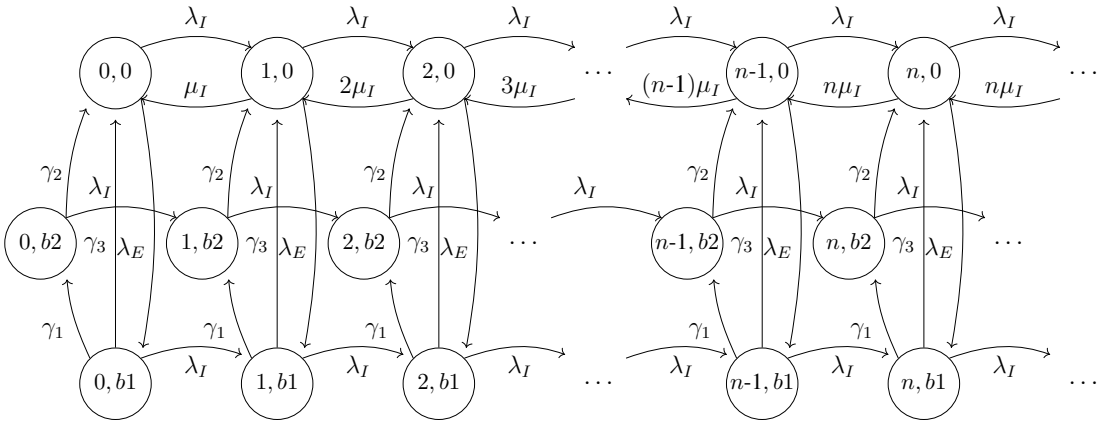
Figure 6.7 shows the absolute mean response times under IF and EF as a function of μ_I. We again examine the system under various fixed values of ρ. The dotted lines at $\mu_I = 1$ denote the



(a) Full EF chain

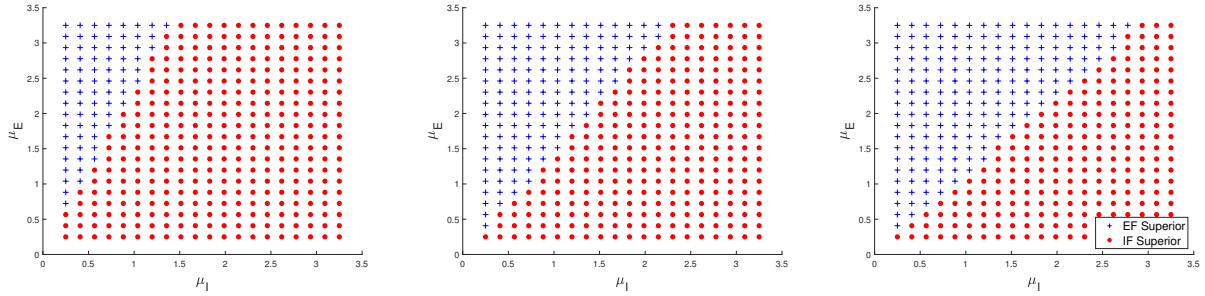


(b) EF chain with special states



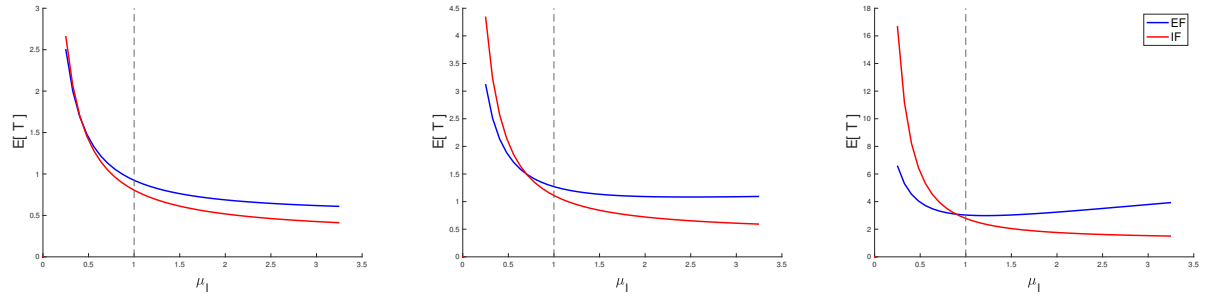
(c) Final 1D EF chain

Figure 6.5: The transformation of the 2D-infinite EF chain to a 1D-infinite chain via the busy period transformation. Special states representing an $M/M/1$ busy period are shown in (b). These busy periods are approximated by a Coxian distribution in (c).



(a) Low load, $\rho = .5$ (b) Med. Load, $\rho = .7$ (c) High load, $\rho = .9$

Figure 6.6: Heat maps showing the relative performance of IF and EF as a function of μ_I and μ_E when $n = 4$. We fix load ρ and vary μ_I and μ_E. To offset the changes to μ_I and μ_E, we change λ_I and λ_E to keep ρ constant. In every graph, $\lambda_I = \lambda_E$. The red circles represent settings where IF dominates EF. The blue $+$'s represent cases where EF dominates IF. As ρ increases, the region where EF dominates IF grows. However, as expected, when $\mu_I \geq \mu_E$ IF dominates EF for all loads.

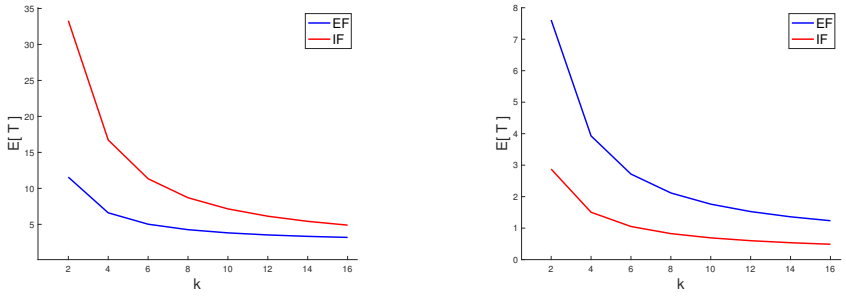


(a) Low load, $\rho = .5$ (b) Med. load, $\rho = .7$ (c) High load, $\rho = .9$

Figure 6.7: Graphs showing the absolute mean response times under IF and EF as a function of μ_I when $n = 4$. In each graph, we fix system load, ρ, and set $\mu_E = 1$. We then vary μ_I. To offset the changes in μ_I, we change λ_I and λ_E to keep ρ constant. In every graph, $\lambda_I = \lambda_E$. The dotted lines at $\mu_I = 1$ denote the case where $\mu_I = \mu_E$. Thus IF is optimal to the right of this line, while EF may dominate IF to the left of this line. We see that the allocation policy has a major impact on mean response time.

case where $\mu_I = \mu_E$. We therefore know that IF is optimal to the right of this line in every graph, while EF may dominate IF to the left of this line. We see that our choice of allocation policy has a major impact on mean response time.

While Figures 6.6 and 6.7 assume that $n = 4$, our analysis works equally well with any number of servers, n. Figure 6.8 shows how the mean response time under IF and EF changes as n increases while system load, ρ, remains constant.



(a) $\mu_I = .25$, $\mu_E = 1$ (b) $\mu_I = 3.25$, $\mu_E = 1$

Figure 6.8: Graphs showing the mean response time under IF and EF as a function of the number of servers, n under high load ($\rho = 0.9$). The values of μ_I and μ_E are chosen to represent the extreme ends of Figure 6.7c (where the performance gap between the policies is the largest). Even when $n = 16$, the difference between IF and EF remains large.

6.5.1 Markov Chains for IF and EF

Figure 6.5a shows the Markov chain which exactly describes EF. The corresponding IF chain is given in Appendix C.3. Recall that the state (i, j) denotes having i inelastic jobs and j elastic jobs in the system. This chain is infinite in 2 dimensions – the number of inelastic jobs and the number of elastic jobs. Because there is no general method for solving 2D-infinite Markov chains, we provide a technique for converting this chain to a 1D-infinite Markov chain in Section 6.5.2.

6.5.2 Converting From 2D-Infinite to 1D-Infinite

To reduce the dimensionality of the Markov chain for EF we follow a three step process:
Step 1: Response time of elastic jobs is trivial. Under EF, elastic jobs have preemptive priority over inelastic jobs. Thus, their behavior is independent of the state of inelastic jobs in the system. We can therefore model the response time of elastic jobs as an $M/M/1$ queueing system with arrival rate λ_E and service rate $n\mu_E$, which is well understood in the queueing literature [55]. What remains is to understand the response time of the inelastic jobs.
Step 2: The busy period transformation. Looking at Figure 6.5a, we notice that the chain has a repeating structure when there is at least 1 elastic job in the system ($j \geq 1$). We leverage this repeating structure to reduce the Markov chain for EF to a 1D-infinite chain. Specifically, while there are elastic jobs in the system, EF does not process any inelastic jobs. The length of time where EF is not processing any inelastic jobs can be viewed as an $M/M/1$ *busy period*. In an $M/M/1$ system, a busy period is defined to be the time between when a job arrives into an empty system until the system empties. In our case, this busy period is the time from when an elastic job arrives into a system with no elastic jobs until the system next has 0 elastic jobs. In Figure 6.5b, we show how to the entire portion of the Markov chain where $j \geq 0$ with a set of special states which represent the duration of an $M/M/1$ busy period for the elastic jobs.
Step 3: Creating 1D chain for inelastic jobs. Looking at Figure 6.5b, we note the bolded tran-

sition arrows (labeled "B") emanating from the busy period states. Because the duration of an $M/M/1$ busy period is not exponentially distributed, we must replace these special transitions with a mixture of exponential states (a Coxian distribution) which accurately approximates the duration of a busy period. A technique for matching the first three moments of the busy period with a Coxian is given in [77]. The 1D-infinite chain resulting from this technique is described in Figure 6.5c.

We use the same three-step technique to make an analogous simplification of the Markov chain for IF (see Appendix C.3).

Given these 1D-infinite chains, we now apply standard matrix analytic techniques to solve for mean response time.

6.5.3 Matrix Analytic Method

We now explain how to analyze IF and EF using the 1D-infinite Markov chains developed in the previous section. We do this by applying matrix analytic methods [57, 72, 73]. Matrix analytic methods are iterative procedures which compute the stationary distribution of a repeating, 1D-infinite Markov chain.

Consider Figure 6.5c, which shows the 1D-infinite chain for EF. Observe that each column of this chain, after the first column, has identical transitions. The idea of matrix analytic methods is to represent the stationary distribution of column $j+1$ as a product of the stationary distribution of column j and some unknown matrix R. The matrix R is determined iteratively through a numeric procedure [57, 72, 73]. This procedure yields the stationary distribution of the chain. Using the stationary distribution we can easily determine the mean number of inelastic jobs, and hence the mean response time for inelastic jobs (recall that the response time for elastic jobs under EF is trivial). An analogous argument can be applied to solve the 1D-infinite chain for IF.

6.6 Conclusion

This chapter establishes optimality results and provides the first stochastic analysis of policies for scheduling jobs which are heterogeneous with respect to parallelizability. We consider the case where jobs belong to one of two classes, with one class of jobs being more parallelizable than the other. While it is intuitive that a scheduling policy should try and maximize instantaneous system efficiency just as EQUI did in the single-speedup function case, an optimal policy must also *defer parallelizable work* in order to preserve the future efficiency of the system. We show that a policy called GREEDY*, which both maximizes instantaneous system efficiency and defers parallelizable work when possible, achieves a near-optimal mean response time in many cases.

Additionally, we show that when jobs are either elastic or inelastic, GREEDY* can perform optimally. In this case, the *Inelastic-First* (IF) policy, which gives strict preemptive priority to inelastic jobs, is equivalent to GREEDY*. We prove that IF is optimal for minimizing the mean response time across jobs in the common case where elastic jobs are larger on average than inelastic jobs. We also provide an analysis of mean response times under the Elastic-First (EF) and Inelastic-First (IF) policies.

It is interesting to note that some systems [23, 82] employ heuristic policies that could be considered part of the GREEDY class of policies. However, these policies are based solely on a intuitive understanding of instantaneous system efficiency. As a result, we are unaware of any system which currently employs a GREEDY* policy, despite the fact that GREEDY* is provably the best GREEDY policy. By failing to defer parallelizable work, these systems are performing suboptimally. It would therefore be interesting to the compare the performance of a numerically computed optimal policy to one of these heuristic policies given a real-world workload of jobs with different speedup functions.

The most pressing theoretical question that this chapter poses is how one should schedule elastic and inelastic jobs when elastic jobs are smaller on average than inelastic jobs. This chapter shows that, in this case, `EF` can outperform `IF`. However, we do not show that `EF` is the optimal allocation policy. We can in fact prove that, in each state, the optimal policy either gives strict priority to an elastic job or strict priority to the inelastic jobs in the system. However, numerically computed optimal policies appear to use a mixture of actions, prioritizing inelastic jobs in some states and prioritizing elastic jobs in other states. We conjecture that an optimal policy in this case has the form of a *threshold policy*. That is, there is some finite region of the state space outside of which the optimal action is to prioritize the smaller elastic jobs. Unfortunately, a proof of this property has remained elusive.

As it turns out, our results on scheduling elastic and inelastic jobs provide important intuition about another important model of parallelizable jobs. In many cases, a parallelizable job is neither fully elastic nor fully inelastic, but rather is composed of different *phases*. That is, a job may alternate between being fully parallelizable at some times and non-parallelizable at other times. In Chapter 7, we will use the intuition developed in this chapter to develop optimal scheduling policies for parallelizable jobs composed of elastic and inelastic phases.

Chapter 7

Scheduling Jobs with Phases

7.1 Introduction

To this point, we have considered several different variants of the same basic model described in Section 2. Aside from some technical assumptions, we have focused on how different assumptions about the job sizes (known vs. unknown) and different assumptions about the speedup functions (single speedup function vs. multiple speedup functions) change the kinds of scheduling policies we should use to minimize job response times. In this final chapter, however, we consider use cases where the scheduler knows more about each job than just its overall size and speedup function.

In many systems, each parallelizable job is actually composed of a series of distinct *phases*, where each phase has its own corresponding size and speedup function. The difficulty in scheduling such jobs arises largely from the fact that each job's parallelizability is not constant over time. That is, a job may be alternate between being highly parallelizable or not at all parallelizable depending on the type of computation it is doing at each moment in time.

A wide variety of computer systems process jobs composed of phases. Databases explicitly invoke the elastic and inelastic phases of a database query during query execution [110]. Cluster schedulers [23], distributed computing platforms (e.g. Hadoop [92] and Apache Spark [108]), distributed machine learning frameworks [62], and supercomputers all process jobs composed of a mixture of highly parallelizable and highly sequential phases. Furthermore, it is common for the scheduling policies in these systems to be aware of each job's current phase [74, 96].

Although the above systems have the ability to track the current phase of each running job, the systems do not effectively leverage this information to make optimal scheduling decisions. In this chapter, we therefore address the problem of *phase-aware scheduling* by designing scheduling policies that use the available phase information about each job to reduce job response times.

We will examine this problem by looking focusing on the example of scheduling in databases, where a single query will alternate between highly parallelizable phases and non-parallelizable phases. Specifically, modern databases translate queries into a pipeline composed of multiple phases corresponding to different database operations [75]. For example, a phase that corresponds to a sequential table scan is generally *elastic*, while a phase corresponding to a table join might be *inelastic*. Figure 7.1 shows that this phenomenon holds for a variety of queries from the Star

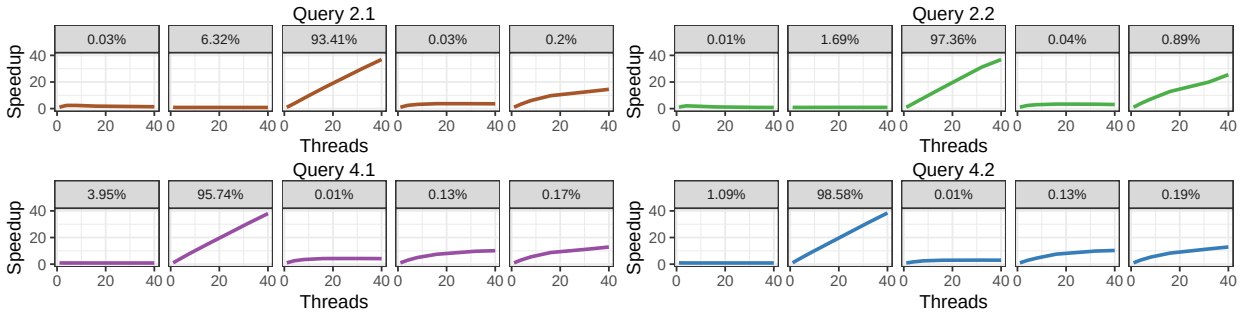


Figure 7.1: Speedup functions for each phase of four queries from the Star Schema Benchmark. Queries were executed using the NoisePage database [75]. Phases are either elastic (highly parallelizable) or inelastic (highly sequential). The percentages denote the fraction of time spent in each phase when the query was run on a single core. Despite the queries spending most of their time in elastic phases, the overall speedup function of each query is highly sublinear due to Amdahl's law.

Schema Benchmark [76].

7.1.1 Scheduling Tradeoffs

In this chapter, we consider the case where jobs are composed of elastic and inelastic *phases*. We will consider cases where the scheduler has limited information about each job's phases and phase sizes, and cases where the exact sequence and sizes of a job's phases are known to the scheduler.

This chapter will show how we can construct optimal *phase-aware* scheduling policies, which account for the different phases that each job moves through over time. Figure 7.2 shows an illustration of the tradeoffs that such a policy must balance, updated to reflect the case where jobs are composed of elastic and inelastic phases.

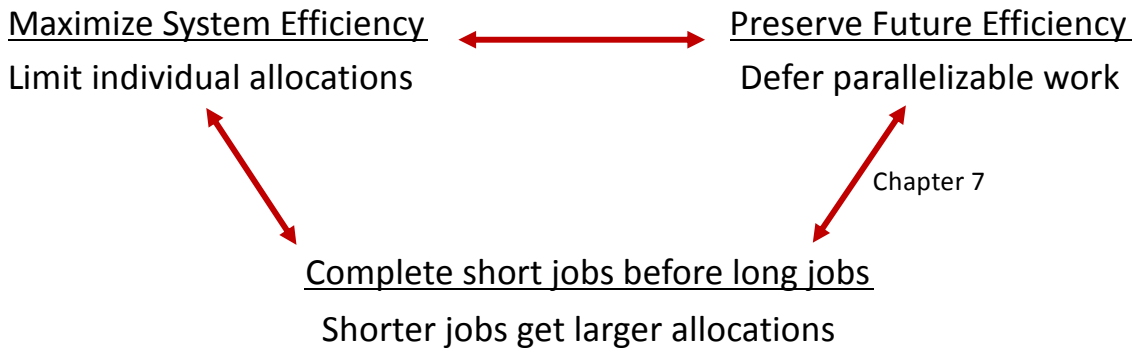


Figure 7.2: Designing good phase-aware scheduling policies involves balancing multiple tradeoffs simultaneously.

Key Insight

It turns out that, in order to construct good phase-aware scheduling policies, we often have to balance all of the tradeoffs shown in Figure 7.2 simultaneously. If a job is known to have a shorter remaining size, or a shorter *expected* remaining size, a scheduling policy should favor this job. However, we should avoid allocating multiple cores to a job if it is in an inelastic phase. Given a set of jobs that are in a mixture of elastic and inelastic phases, a scheduling policy should still defer parallelizable work to preserve the future efficiency of the system. That is, it is generally beneficial to give priority to jobs in inelastic phases. To make matters even more complex, it could be the case that some jobs are in short elastic phases while other jobs are in long inelastic phases. Here, we will see that our choice of scheduling policy will depend on what is known about the exact sequence of phases comprising each job in the system. In short, to derive new phase-aware scheduling policies, we must combine the intuition and mathematical results developed in various prior chapters in order to address all of the tradeoffs shown in Figure 7.2.

7.1.2 Contributions

Although real-world systems process jobs composed of phases, and these systems are often aware of the current phase of each job, phase-aware scheduling remains an open problem. Hence, our first contribution is a stochastic model of jobs composed of elastic and inelastic phases. Under this model, we derive a provably optimal scheduling policy.

As we saw in Chapter 6, when scheduling jobs that are either entirely elastic or entirely inelastic, the IF policy was optimal with respect to mean response time. This raises the question of whether IF can be generalized to a policy that performs well when scheduling jobs composed of elastic and inelastic phases. In this chapter we show that IF, which gives priority to jobs in inelastic phases, is the optimal policy for scheduling jobs composed of elastic and inelastic phases in many cases. This optimal policy greatly outperforms both the PA-FCFS policy used in real systems and the EQUI policy proposed for scheduling jobs composed of phases in the worst-case literature. Because our optimality results require some simplifying assumptions, we validate IF's performance through a range of simulations, including a simulation of a database running queries from the Star Schema Benchmark [76].

The specific contributions of this chapter are summarized as follows:

- We first present a novel model of parallelizable jobs composed of elastic and inelastic phases where the scheduler knows, at all times, what phase a job is in. This model generalizes the models used in previous chapters, where we assumed that all jobs followed a single speedup function throughout their lifetimes.
- We prove that the *Inelastic First* (IF) policy, which *defers parallelizable work* by giving strict priority to jobs which are in an inelastic phase, is optimal under our model. Because the proof of optimality requires a complex coupling argument, we break this claim down by considering special cases which are easier to understand. We begin by proving the optimality of IF in simpler models in Section 7.4 before proving our more general claim in Section 7.5.
- In Section 7.6, we perform an extensive simulation-based performance evaluation to illus-

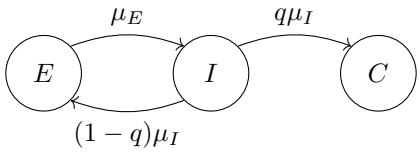


Figure 7.3: The Markov chain governing the evolution of a multi-phase job when running on a single core. E refers to the elastic phase, I refers to the inelastic phase, and C is the completion state.

trate that IF outperforms a range of scheduling policies. Even in settings that violate the assumptions of our model, IF can perform nearly 30% better than the PA-FCFS policy used in modern databases and a factor of 3 better than the EQUI policy advocated for in the worst-case theory literature.

- Lastly, in Section 7.7, we perform a case study on scheduling in databases where queries consist of elastic and inelastic phases. In this setting, the scheduler sometimes has additional information about each query beyond just the query's current phase. We show how to generalize IF to leverage this additional information and improve upon state-of-the-art database scheduling by roughly 50% in simulation.

7.2 Our Model

In this section, we present a model of jobs composed of distinct phases running in a system consisting of n homogeneous cores.

Multi-phase Jobs

We begin by noting that in a wide range of systems applications, job phases are either highly parallelizable or highly sequential. Figure 7.1 shows that database queries often follow this pattern. A similar phenomenon applies in systems using a map-reduce paradigm [22] where parallelizable map stages are interlaced with sequential reduce stages. Machine learning training jobs also consist of highly parallelizable iterations of distributed gradient descent followed by a sequential step that coalesces the results on a central parameter server [62]. Hence, while job phases could potentially experience intermediate parallelizability, we will consider the highly practical case where job phases are either *elastic*, perfectly parallelizable, or *inelastic*, totally sequential.

To model the duration of each job phase, we define a phase's *inherent size* to be the amount of time it takes the phase to complete when run on a single core. This is analogous to and compatible with our notion of a job's size, in that a job's inherent size is the sum of the sizes of the job's phases. For analytical tractability, we will assume that inelastic phase sizes are distributed as $\texttt{Exp}(\mu_I)$ and elastic phase sizes are distributed as $\texttt{Exp}(\mu_E)$, and that all phase sizes are independently distributed. Although the scheduler often knows the current phase of each job in the system, it is less common in real systems for the scheduler to know the full sequence of phases comprising

each job, or the size of each phase. Hence, we will generally assume that the scheduler knows the current phase of each job, but that the scheduler does not know the future phases or any of the phase sizes of a job. In Section 7.7, we consider scheduling in databases, where it is common for the database to have additional information about each job's phases and phase sizes.

Our definition of elastic and inelastic phases is analogous to our definition of elastic and inelastic jobs in Chapter 6. Elastic and inelastic phases follow the speedup functions $s_E(k)$ and $s_I(k)$ as defined in Section 6.2.2. That is, elastic phases are perfectly parallelizable, while inelastic phases receive no benefit from running on more than one core. The result is that, when run on k cores, and elastic phase of size X_E will complete in time

$$\frac{X_E}{s_E(k)}$$

and an inelastic phase of size X_I will complete in time

$$\frac{X_I}{s_I(k)}.$$

Note that this definition is compatible with our notion of elastic and inelastic jobs. Specifically, a job is elastic if all of its phases are elastic, and a job is inelastic if all of its phases are inelastic.

Because the sizes of a job's phases are assumed to be exponentially distributed and unknown to the system, we model a multi-phase job via a continuous-time Markov chain, as shown in Figure 7.3. We model each job via a Markov chain consisting of three states: an E state that denotes that the job is in an elastic phase, an I state that denotes that the job is in an inelastic phase, and an absorbing state, C, that denotes that the job has been completed. Each arriving job can either start in the E state or in the I state. We assume that a job can only transition to the completion state from the inelastic state. This is realistic for a wide range of systems where the results of a parallel computation must be sequentially coalesced and returned to the user [22, 44, 108]. It also simplifies our analysis without weakening our results (see Remark 7.3). We define q to be the probability that a job completes after an inelastic phase; with probability $1 - q$ the job will transition to an elastic phase.

We assume that all jobs evolve according to the same underlying Markov chain. However, the exact number of phases and the sizes of the phases belonging to each job can be different. Under this model, the expected total inherent size of a job depends on whether the job begins with an E phase or an I phase, and is given by the following expressions:

$$\begin{aligned}
\mathbb{E}[\text{Job size if start in } E] &= \left(\frac{1}{\mu_E} + \frac{1}{\mu_I}\right)\frac{1}{q} \\
\mathbb{E}[\text{Job size if start in } I] &= \left(\frac{1}{\mu_E} + \frac{1}{\mu_I}\right)\frac{1}{q} - \frac{1}{\mu_E}
\end{aligned}$$

We refer to the completion of a job's final inelastic phase as a *job completion*. We refer to the completion of any of the job's phases as a *transition*. An *inelastic transition* occurs when an inelastic phase is completed and an *elastic transition* occurs when an elastic phase is completed.

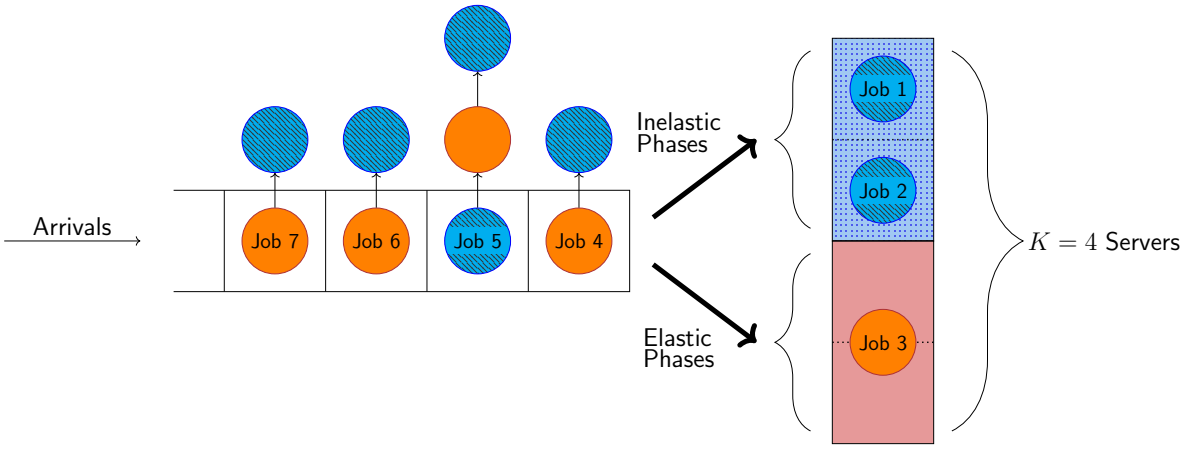


Figure 7.4: The central queue and cores for our system. Jobs 1-7 are all modeled by the Markov chain presented in Figure 7.3. We use solid orange to illustrate the elastic phases of jobs, and crosshatched blue to illustrate the inelastic phases. While we assume the number of remaining phases is unknown to the scheduler, we have drawn out the remaining phases to illustrate job structure. Here, there are $n = 4$ cores. Two cores are allocated to jobs in an inelastic phase (Jobs 1 and 2), and two cores are allocated to a single job in an elastic phase (Job 3).

Scheduling Policies

A *scheduling policy*, π, determines how to allocate the n cores to the jobs in the system at every moment in time. Although our policies are fully preemptive, we assume that policies only change their allocation at times of job arrivals, transitions, or job completions. When a job is in an inelastic phase, it can be allocated up to one core (i.e., fractional allocations are admitted). When a job is in an elastic phase, it can be allocated any number of cores up to n.

To find an optimal scheduling policy, it suffices to only consider policies that do not idle cores unnecessarily. This follows from Theorem C.1, which showed that there exists a non-idling optimal scheduling policy for scheduling jobs consisting of either a single elastic phase or a single inelastic phase. The proof of C.1 can be trivially extended to the case where jobs consist of multiple phases. Hence, we only consider non-idling scheduling policies.

Because the sizes of a job's phases are exponentially distributed and unknown to the system, we first consider policies that make allocation decisions based only on the type of each job's current phase. In Section 7.7.1, however, we discuss the case where scheduling policies may have additional information about the phase sizes for each job.

This chapter focuses on the analysis of the *Inelastic First* (`IF`) policy, which we generalize from Chapter 6 to handle the case where jobs are composed of multiple phases. Specifically, at every moment in time, `IF` gives strict priority to jobs that are in inelastic phases. If there are i jobs in the system that are in an inelastic phase, then `IF` allocates $\min\{i, n\}$ cores to these jobs. `IF` then allocates any remaining cores to a job in an elastic phase if such a job exists, otherwise these extra cores remain idle. Our intuition is that this generalization of `IF` will remain a good policy because it continues to *defer parallelizable work*. We will show that `IF` again increases system

efficiency by keeping flexible elastic phases in the system, ensuring that all n cores can remain utilized.

We show that IF is optimal with respect to minimizing mean response time. In addition to deferring parallelizable work, IF is a good policy because it is *also* able to favor jobs with smaller expected remaining sizes. Observe that IF does not require any knowledge of the job parameters (μ_I, μ_E, and q). Thus, optimally scheduling multi-phase jobs can be done regardless of whether these parameters are known to the system.

Arrival Processes and Metrics

In this chapter, we allow for an arbitrary arrival process. To be precise, we first define an *arrival time sequence* as two fixed, infinite sequences, $(t_n)_{n\geq 1}$ and $(\ell_n)_{n\geq 1}$, where t_n is the time at which the nth job arrives and $\ell_n \in \{E, I\}$ denotes whether the arriving job begins with either an E phase or an I phase. We define an *arrival time process* as a distribution over arrival sequences.

We define the *response time* of the nth job under policy π to be the time from when the job arrives until it completes. We denote this quantity by the random variable $T_\pi^{(n)}$. We let T_π denote the steady-state response time whenever this quantity exists.

As an example, consider the case where the arrival time process is a Poisson process with rate λ and each job starts with an E phase with probability r_E and with an I phase with probability r_I. Then we can define the system load as:

$$\rho = \text{System load } = \frac{\lambda \cdot \mathbb{E}[\text{Job size}]}{n},$$

where

$$\mathbb{E}[\text{Job size}] = r_E \cdot \mathbb{E}[\text{Job size if start in } E] + r_I \cdot \mathbb{E}[\text{Job size if start in } I] \ .$$

In this setting, if $\rho < 1$, the steady-state mean response time under policy π exists and is denoted by $\mathbb{E}[T_\pi]$.

Stochastically Minimizing the Number of Jobs in System

We show that IF minimizes the steady-state mean response time across jobs. To show this, we prove a series of claims about the number of jobs in the system at any point in time. Namely, we argue that IF stochastically maximizes the number of jobs completed by any point in time. This is equivalent to saying IF stochastically minimizes the number of jobs in system at any point in time.

To reason about the number of completions by time t, we will count the number of elastic and inelastic transitions as well as the number of job completions. We define $C_\pi(t)$ to be the number of job completions by time t under policy π. We define $I_\pi(t)$ (and $E_\pi(t)$) to be the number of inelastic (resp. elastic) transitions under policy π by time t. Finally, we define $I_\pi(s,t)$ to be the number of inelastic transitions under π on the interval $(s,t]$ and we define $E_\pi(s,t)$ and $C_\pi(s,t)$ analogously.

With respect to the number of jobs in system, let $N_\pi(t)$ denote the number of jobs present at time t, under policy π. We define $N_\pi^E(t)$ to be the number of jobs in an elastic phase at time t under π and we define $N_\pi^I(t)$ to be the number of jobs in an inelastic phase at time t under π.

7.3 Overview of Theorems

In this section, we provide an overview of the theoretical results in Sections 7.4 and 7.5.

7.3.1 Main Result

We first state the main theorem in full generality. At a high level, the theorem states that IF is the most effective policy in terms of completing jobs. More specifically, we show that the number of jobs completed by any point in time under IF stochastically dominates the number of jobs completed by the same time under any other algorithm.

Theorem 7.1. *Consider a n core system serving multi-phase jobs. The policy IF stochastically maximizes the number of jobs completed by any point in time. Specifically, for a policy A, let $C_A(t)$ denote the number of jobs completed by time t and let $N_A(t)$ denote the number of jobs in the system at t. Then under any arbitrary arrival time process, $C_{\text{IF}}(t) \geq_{st} C_A(t)$ for all times $t \geq 0$. Consequently, $N_{\text{IF}}(t) \leq_{st} N_A(t)$ for all times $t \geq 0$.*

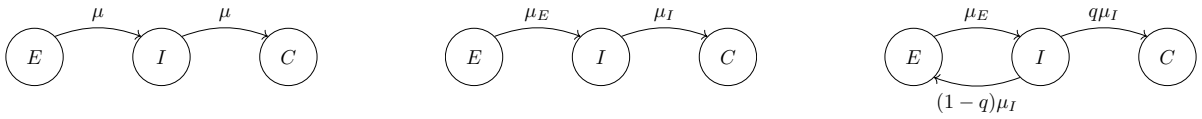
We can leverage Theorem 7.1 to derive far-reaching results about job response time. In particular, if the arrival time process is a renewal process[1], we can show that IF minimizes the steady-state mean response time. We formalize this idea in the following immediate corollary of Theorem 7.1.

Corollary 7.2. *Suppose the same system setup as in Theorem 7.1. For any arbitrary policy A, let T_A be the steady-state job response time when it exists. If the arrival time process is a renewal process, then $\mathbb{E}[T_{\text{IF}}] \leq \mathbb{E}[T_A]$.*

Proof. By Theorem 7.1, we know IF stochastically minimizes the number of jobs in the system at any point in time. Since the arrival time process is a renewal process, this implies that IF minimizes the steady-state mean number of jobs in the system. By Little's law, minimizing the mean number of jobs in the system suffices for minimizing the steady-state mean response time. □

Theorem 7.1 and its corollary show that IF succeeds by both *deferring parallelizable work* and working on jobs with smaller expected remaining sizes. Specifically, while elastic phases can be completed more quickly by parallelizing across all cores, there are benefits to keeping elastic phases in the system. These elastic phases are flexible and can ensure that all n cores remain utilized. It is also possible to allocate some cores to inelastic phases without significantly increasing the run time of an elastic phase. Furthermore, jobs in inelastic phases have smaller expected remaining sizes. Hence, deferring parallelizable work also results in favoring shorter

[1] By a renewal process, we mean the inter-arrival times $t_n - t_{n-1}$ are i.i.d., and that the initial phases of jobs p_n are i.i.d. as well.



(a) Two-phase job with equal rate

(b) Two-phase job with unequal rates

(c) Multi-phase job

Figure 7.5: The three job structures we consider. E refers to the elastic state, I refers to the inelastic state, and C refers to the completion state. In Figure 7.5(a), jobs just have two phases, both with inherent size distributed as $\texttt{Exp}(\mu)$. In Figure 7.5(b), jobs still have two phases. Phase I has size distributed as $\texttt{Exp}(\mu_I)$, and phase E has size distributed as $\texttt{Exp}(\mu_E)$. In Figure 7.5(c), we add in potential transitions from phase I to phase E.

jobs. For these reasons, the optimal policy, IF, defers as much parallelizable work as possible without over-allocating to inelastic phases.

Remark 7.3. *One might assume that* IF *benefits not from deferring parallelizable work, but rather from how we have defined our model, where jobs in inelastic phases have smaller expected remaining sizes. However, as we show in Section 7.7, simply favoring jobs with smaller remaining sizes (as done by the* PA−SRPT *policy) is not nearly as important as deferring parallelizable work in real-world settings.*

7.3.2 How We Prove Theorem 7.1

We now provide a road map for how we prove Theorem 7.1. At a high level, we note that it suffices to find a coupling between two systems, one running IF and one running an arbitrary policy A, under which $C_{\text{IF}}(t) \geq C_A(t), \forall t \geq 0$. However, finding such a coupling is difficult due to the complicated job structure in Figure 7.3. In particular, the inherent size distributions are different between the two phases and jobs are composed of an unknown number of elastic and inelastic phases.

We therefore begin by considering several simpler job structures, as seen in Figure 7.5. The simplest job structure has elastic and inelastic phases with the same size distribution, and does not allow transitions from an inelastic phase to an elastic phase. We then add the complexities gradually back to the model. In each case, we argue that studying the number of inelastic transitions suffices to understand the total number of job completions. Recalling that $I_A(t)$ is the number of inelastic transitions under A by time t, we prove that, for any policy A, there exists a coupling under which $I_{\text{IF}}(t) \geq I_A(t)$ for all $t \geq 0$. We then argue that, under this coupling, $C_{\text{IF}}(t) \geq C_A(t)$ for all $t \geq 0$.

In Section 7.4.1, we start with the simplest job structure as shown in Figure 7.5(a). In this structure, jobs consist of a single elastic phase followed by a single inelastic phase. Moreover, we assume the inherent sizes of both the elastic and inelastic phases are identically distributed as $\texttt{Exp}(\mu)$. We refer to such jobs as *two-phase jobs with equal rates*. We are able to couple two systems experiencing jobs of this structure by (1) having them experience the same sequence of arrivals and (2) splitting time into roughly uniform chunks of length $\texttt{Exp}(n\mu)$. At the end of each

chunk of time, the systems will both potentially experience a job transition. By splitting time into "busy" and "idle" periods under this coupling (as defined in the proof of Lemma 7.4), we prove the desired result.

We then consider the slightly more complicated job structure shown in Figure 7.5(b) in Section 7.4.2. In this job structure, jobs again consist of a single elastic phase followed by a single inelastic phase. However, we drop the assumption that the inherent sizes of elastic and inelastic phases are identically distributed. We refer to such jobs as *two-phase jobs with unequal rates*. Although having unequal rates between phases complicates the splitting of time into roughly equal blocks, we work around this by leveraging a trick called *uniformization*. More specifically, we reformulate this more general job structure as a Markov chain in which the elastic and inelastic phases have the same inherent size distribution, but with some additional self-loop transitions added to the chain. We then expand our existing coupling argument by coupling the transition outcomes of the two systems.

Finally, we consider the general job structure as shown in Figure 7.5(c) in Section 7.5. In this job structure, jobs have alternating elastic and inelastic phases, each with a different service rate. We refer to such jobs as *multi-phase jobs*. This case seems quite different from the previous settings, since now an inelastic transition can produce an elastic phase. However, we show that such a transition is equivalent to a job completion followed immediately by an arrival of a job beginning with an elastic phase. Using this argument, we show how a coupling in the general case follows from our coupling in the cases with two-phase jobs.

The above arguments all leverage the fact that, for a given sample path under our model, the policy that has completed more inelastic phases by time t will also have completed more jobs. This is a direct result of our assumption that the final phase of each job will be an inelastic phase. In the case where jobs may finish with an elastic phase, an optimal policy may give priority to a job in an elastic phase if the job has a sufficiently small expected remaining size. We examine the case where jobs finish with elastic phases in Section 7.6 and find that IF still frequently performs well.

7.4 Two-Phase Jobs

7.4.1 Two-Phase Jobs with Equal Rates

We first consider two-phase jobs with equal rates. These are jobs that consist of a single elastic phase followed by a single inelastic phase where both phases have inherent size distributed as $\texttt{Exp}(\mu)$, as illustrated in Figure 7.5(a).

Lemma 7.4. *Consider a n core system serving two-phase jobs with equal rates. Consider any policy A and let the policies* IF *and A start from the same initial conditions and have the same arrival time process. Then there exists a coupling between* IF *and A such that $I_{\texttt{IF}}(t) \geq I_A(t)$ and $C_{\texttt{IF}}(t) \geq C_A(t)$ for all $t \geq 0$, where $I_{\texttt{IF}}(t)$ (resp. $I_A(t)$) is the number of inelastic transitions by time t under* IF *(resp. A), and $C_{\texttt{IF}}(t)$ (resp. $C_A(t)$) is the number of jobs completed by time t under* IF *(resp. A).*

We begin by first describing the coupling we will use to prove Lemma 7.4. This coupling will serve as a building block for subsequent arguments in this chapter. We then present the proof of

Lemma 7.4.

Note that for two-phase jobs with equal rates, every inelastic job transition is also a completion, so we have $I_A(t) = C_A(t)$ for all times $t \geq 0$ under any policy A. Therefore, to prove Lemma 7.4, it suffices to construct a coupling under which $I_{\text{IF}}(t) \geq I_A(t)$ for all $t \geq 0$. Then the claim $C_{\text{IF}}(t) \geq C_A(t)$ follows directly.

Coupling IF and A

Let S_{IF} be the system running IF and S_A be the system running any arbitrary policy A. The high-level intuition of the coupling is as follows. Since both phases, inelastic and elastic, have inherent size $\text{Exp}(\mu)$, we can parse time into blocks of length $\text{Exp}(n\mu)$. At the end of each of these blocks, both systems will potentially experience a job transition. Outside of these points of time, no job transitions can occur. This simplifies the counting of job completions and inelastic transitions. Arrivals do not change the number of transitions or job completions, and hence we do not need assumptions on the arrival time process.

Job arrivals: We assume that the two systems, S_{IF} and S_A, have the same number of jobs in each phase at time 0 (for instance, seven jobs in an inelastic phase, and three jobs in an elastic phase). Formally, we assume that $N^E_{\text{IF}}(0) = N^E_A(0)$ and $N^I_{\text{IF}}(0) = N^I_A(0)$.

We fix an arrival time sequence that is shared between S_{IF} and S_A. Recall that an arrival sequence is a fixed sequence of arrival times $(t_n)_{n\geq 1}$ and a corresponding binary sequence $(\ell_n)_{n\geq 1}$, where t_n is the time the nth overall job arrival occurs in both systems and $\ell_n \in \{E, I\}$ determines which phase a job starts in.

Job transitions and departures: Suppose the current time is t. We generate a random variable $X \sim \text{Exp}(n\mu)$, that is shared by both systems. Suppose s is the next unrealized arrival time in the arrival sequence. If $s < t + X$, we allow the arrival to occur simultaneously into both systems. We then set $t \leftarrow s$, and return to the beginning of this paragraph. If $s > t + X$, then we set the current time to be $t \leftarrow t + X$, and then select one of the n cores uniformly at random[2] (we select the same core in both systems). If a system is running a job in its inelastic phase on this randomly selected core, the job completes. Likewise, if the core is running a job in its elastic phase, the system experiences an elastic transition, producing an inelastic phase. Lastly, if the core selected is idling, nothing happens. This event (which may or may not result in a transition/departure) will be referred to as a *potential transition*. In general, a time where either an arrival or potential transition occurs will be referred to as an *event time*.

Additionally, if at time t the system is serving i inelastic jobs, we assume they are running on cores 1 through i. If an elastic job is being served, it is run on cores $i + 1$ through e, where e is some number less than or equal to n. The remaining cores are left idle.

[2]For the sake of simplicity, we assume that jobs can only be allocated an integral number of cores. However, our result generalizes to the case where allocations are fractional. When allocations are fractional, we treat the cores as a continuous interval, $[0, n]$ and generate $U \sim \text{Unif}[0, n]$. The type of phase running at the corresponding point in the interval $[0, n]$ determines what type of transition occurs.

Proof of Lemma 7.4

Proof. We proceed by induction on event times, as defined in Section 7.4.1. We parse time into two types of periods: *busy periods* where S_{IF} utilizes all n of its cores, and *idle periods* where S_{IF} idles at least one of its cores at any point in time. Let t_0 be the start of a period (either busy or idle). We show below that, if $I_{\text{IF}}(t_0) \geq I_A(t_0)$, then, at any point of time t during the current period, $I_{\text{IF}}(t) \geq I_A(t)$. This claim is sufficient for proving Lemma 7.4 since time can be partitioned into disjoint alternating busy and idle periods. To get a sense of how time is partitioned, refer to Figure 7.6.

As a base case for our induction, observe that $I_{\text{IF}}(0) = I_A(0)$. Depending on the initial conditions, time $t = 0$ will serve as either the start of the first busy period or the first idle period.

Busy Periods: We first consider the case where time t_0 marks the start of a busy period, and assume inductively that $I_{\text{IF}}(t_0) \geq I_A(t_0)$. We show that $I_{\text{IF}}(t) \geq I_A(t)$ for all times t in the busy period by contradiction. Assume for contradiction that there is some earliest time s in the busy period such that $I_{\text{IF}}(s) < I_A(s)$.

First, we argue that

$$N^E_{\text{IF}}(t_0) \leq N^E_A(t_0). \tag{7.1}$$

If $t_0 = 0$, this follows directly from the shared initial conditions of S_{IF} and S_A. Now suppose $t_0 > 0$. Observe that, since t_0 marks the beginning of a busy period, immediately before time t_0, all of the jobs in S_{IF} must be in the inelastic phase. That is, $N^E_{\text{IF}}(t_0-) = 0$, and thus $N^E_{\text{IF}}(t_0-) \leq N^E_A(t_0-)$. Lastly, the event that happens at t_0 can only be job arriving since t_0 is the start of a busy period. Since the arrival occurs simultaneously in both systems, it follows that $N^E_{\text{IF}}(t_0) \leq N^E_A(t_0)$, as desired.

Next, let τ be the time for the event preceding the event at s. We claim that

$$I_{\text{IF}}(\tau) = I_A(\tau), \text{ and } N_{\text{IF}}(\tau) = N_A(\tau), \tag{7.2}$$

where $N_{\text{IF}}(\tau) = N_A(\tau)$ follows from $I_{\text{IF}}(\tau) = I_A(\tau)$. This is because the number of inelastic transitions determines the number of job completions and both systems experience the same arrival sequence. The claim $I_{\text{IF}}(\tau) = I_A(\tau)$ is true since $I_{\text{IF}}(t)$ is non-decreasing, $I_A(\tau)$ can increase by at most 1 at time s, and s is the earliest time during the busy period for which $I_{\text{IF}}(s) < I_A(s)$. More specifically, we can conclude that at time s, S_{IF} experiences an elastic transition, whereas S_A experiences an inelastic transition. This holds because $I_{\text{IF}}(s) < I_A(s)$ and $I_{\text{IF}}(\tau) = I_A(\tau)$ if and only if $I_{\text{IF}}(s) = I_{\text{IF}}(\tau)$ and $I_A(s) = I_A(\tau) + 1$. This implies that S_A experiences an inelastic transition at time s. Furthermore, S_{IF} experiences an elastic transition at time s since `IF` does not idle cores during a busy period.

Now, we can claim that

$$E_{\text{IF}}(t_0, s) \leq E_A(t_0, s). \tag{7.3}$$

Since we have shown that $N^E_{\text{IF}}(t_0) \leq N^E_A(t_0)$ in (7.1), it suffices to show that $N^E_{\text{IF}}(s) \geq N^E_A(s)$. Per our coupling, the previous paragraph implies that S_{IF} is running fewer inelastic jobs on the interval $[\tau, s)$ than S_A. Since S_{IF} always runs the maximal number of inelastic jobs, we have that

$N^I_{\text{IF}}(\tau) < N^I_A(\tau)$. Moreover, since $N_{\text{IF}}(\tau) = N_A(\tau)$ by (7.2), we know that $N^E_{\text{IF}}(\tau) > N^E_A(\tau)$, and thus $N^E_{\text{IF}}(s) \geq N^E_A(s)$.

Finally, let M denote the number of potential transitions during $(t_0, s]$. Since S_{IF} is never idling cores between times t_0 and s, we have the identities:

$$M = E_{\text{IF}}(t_0, s) + I_{\text{IF}}(t_0, s), \text{ and } M \geq E_A(t_0, s) + I_A(t_0, s).$$

Consequently, utilizing (7.3) and rearranging, we have that:

$$I_{\text{IF}}(t_0, s) \geq I_A(t_0, s). \tag{7.4}$$

Moreover, recall that by definition, $I_{\text{IF}}(s) = I_{\text{IF}}(t_0) + I_{\text{IF}}(t_0, s)$ and $I_A(s) = I_A(t_0) + I_A(t_0, s)$. Since we assumed $I_{\text{IF}}(t_0) \geq I_{\text{IF}}(t_0)$, and we know that $I_{\text{IF}}(t_0, s) \geq I_A(t_0, s)$ by (7.4), we have $I_{\text{IF}}(s) \geq I_A(s)$, a contradiction. This completes the induction step for busy periods.

Idle periods: Next, we consider the case that time t_0 marks the beginning of an idle period, and again inductively assume that $I_{\text{IF}}(t_0) \geq I_A(t_0)$. To show $I_{\text{IF}}(t) \geq I_A(t)$ for all times t in the idle period, we once again proceed by contradiction. That is, suppose there is some earliest time s in the period such that $I_{\text{IF}}(s) < I_A(s)$.

First, observe that, since S_{IF} always chooses to idle at least one core during the idle period, there cannot be any elastic phase jobs in the system. That is, $N^E_{\text{IF}}(t) = 0$ for all times t in the idle period.

Letting τ be defined again as the time for the event preceding the event at s, by a similar reasoning to before, we must have that

$$I_{\text{IF}}(\tau) = I_A(\tau), \tag{7.5}$$

and that, at time s, S_{IF} does not have a transition, whereas S_A experiences an inelastic transition.

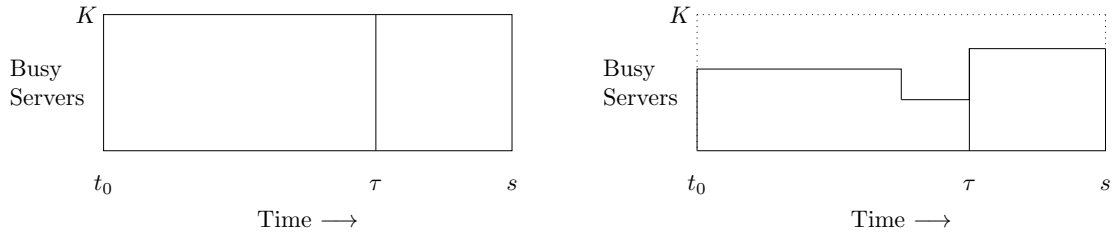
Now we show that we have a contradiction. First note that the equality $I_{\text{IF}}(\tau) = I_A(\tau)$ in (7.5) implies that $N_{\text{IF}}(\tau) = N_A(\tau)$. Next, since at time s, S_{IF} does not have a transition but S_A experiences an inelastic transition, per our coupling, S_{IF} is running strictly fewer jobs in the inelastic phase than S_A. Since S_{IF} has no elastic jobs, this implies that $N_{\text{IF}}(\tau) < N_A(\tau)$, leading to a contradiction. This completes the induction step for idle periods.

□

7.4.2 Two-Phase Jobs with Unequal Rates

We now consider two-phase jobs with unequal rates, as illustrated in Figure 7.5(b). In this case, the inherent sizes of elastic phases are distributed as $\texttt{Exp}(\mu_E)$, and the inherent sizes of inelastic phases are distributed as $\texttt{Exp}(\mu_I)$. In this section, we will show how to generalize the coupling in Section 7.4.1 to establish Lemma 7.5, below.

Lemma 7.5. *Consider a n core system serving two-phase jobs with unequal rates. Consider any policy A and let the policies* IF *and A start from the same initial conditions and have the same*



(a) Busy period

(b) Idle period

Figure 7.6: Examples of busy and idle periods used in the proof of Lemma 7.4. The figures both show the relative positions of the times t_0, τ, and s. The value n refers to the number of cores, and the heights of the boxes indicate how many cores $S_{\texttt{IF}}$ allocates to jobs.

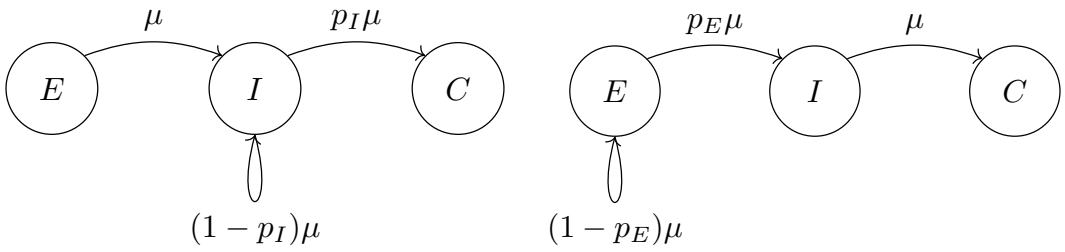
arrival time process. Then there exists a coupling between IF *and* A *such that* $I_{\texttt{IF}}(t) \geq I_A(t)$ *and* $C_{\texttt{IF}}(t) \geq C_A(t)$ *for all* $t \geq 0$*, where* $I_{\texttt{IF}}(t)$ *(resp.* $I_A(t)$*) is the number of inelastic transitions by time* t *under* IF *(resp.* A*), and* $C_{\texttt{IF}}(t)$ *(resp.* $C_A(t)$*) is the number of jobs completed by time* t *under* IF *(resp.* A*).*

It is not trivial to apply the coupling in Section 7.4.1 to the situation where different phases (elastic and inelastic) have different exponential rates (μ_E and μ_I). The key component of our coupling in Section 7.4.1 was that we could parse time into blocks of length $\texttt{Exp}(n\mu)$ to keep both systems in sync. Now, the size of the blocks could depend on which types of phases (elastic or inelastic) are being served, and thus may be unequal between the two systems.

To tackle this problem, we leverage the technique of Markov chain *uniformization*. In uniformization, we find a rate μ that is at least as large as any transition rate in the Markov chain. Our goal is to set the total transition rate out of both the elastic and inelastic states to be μ without changing the behavior of the Markov chain. For instance, if $\mu_E > \mu_I$, we take $\mu := \mu_E$. Clearly, the total transition rate out of the elastic state is already μ in this case. For the inelastic state, because $\mu_I < \mu$, we add a self-loop transition that brings the total transition rate out of the state up to μ. We can then rewrite both μ_I and the new self-loop transition in terms of μ. Specifically, we can think of first transitioning out of the inelastic state with some total rate μ, and then either moving to the completion state or experiencing a self-loop with probability p_I or $1 - p_I$ respectively. By choosing $p_I = \frac{\mu_I}{\mu}$, we can re-write μ_I as $p_I\mu$. We then write the self-loop transition rate as $(1 - p_I)\mu$. Figure 7.7(a) shows the resulting uniformized Markov chain.

It is easy to confirm that, since we have only added some self-loop transitions, the uniformized Markov chain is equivalent to the original Markov chain (Figure 7.5(b)). The case where $\mu_E < \mu_I$ is symmetric and the uniformized Markov chain for this case is shown in Figure 7.7(b).

To present our coupling, we will use the uniformized job model. Under the uniformized job model, $I_A(t)$ can differ from the total number of job completions, $C_A(t)$. However, for the coupling we present, we can use the number of inelastic transitions to directly recover the total number of job completions.



(a) Two-phase job, unequal rates, $\mu_E > \mu_I$ (b) Two-phase job, unequal rates, $\mu_E < \mu_I$

Figure 7.7: Two cases of uniformizing two-phase jobs with unequal rates. In Figure 7.7(a), $\mu_E > \mu_I$, so we take our dominating rate as $\mu := \mu_E$. We then take $p_I := \frac{\mu_I}{\mu}$ and set the total transition rate out of the inelastic state to be μ. With probability $1 - p_I$, after completing an inelastic phase, we immediately start another one. With probability p_I, the job completes and exits the system. Figure 7.7(b) shows the analogous case where $\mu_E < \mu_I$.

System Coupling

Our goal in the coupling is twofold. Once again, we want to chop up time into blocks of length $\texttt{Exp}(n\mu)$ to keep S_{IF} and S_A roughly in sync. Additionally, we want to construct a coupling where reasoning about the number of inelastic transitions, $I_A(t)$, suffices for reasoning about total job completions, $C_A(t)$. That is, we want to find a coupling under which $I_{\text{IF}}(t) \geq I_A(t), \forall t \geq 0$ implies $C_{\text{IF}}(t) \geq C_A(t), \forall t \geq 0$.

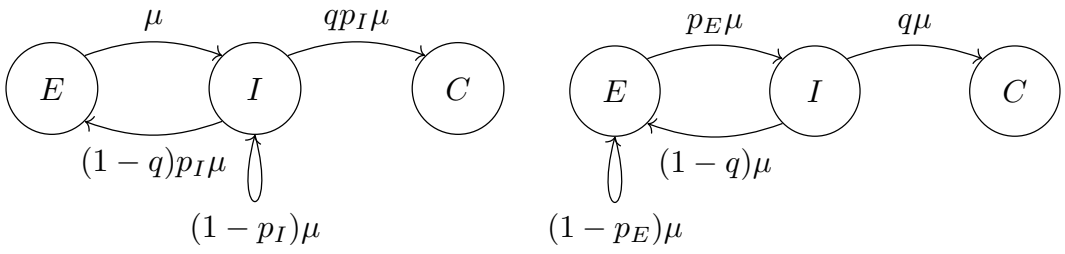
Job arrivals: S_{IF} and S_A share the same arrival time sequence, and start with the same initial conditions.

Job transitions and departures: Our coupling in this case closely follows the coupling in Section 7.4.1. Specifically, the current time t is updated in the same manner as in Section 7.4.1. However, we handle potential transitions slightly differently, due to uniformization. We only discuss the case $\mu_I < \mu_E$, as the reverse case can be handled symmetrically.

When $\mu_I < \mu_E$, we take our dominating rate to be $\mu := \mu_E$. We generate an infinite sequence, $(X_n)_{n\geq 1}$, of i.i.d. $Bern(p_I)$ random variables, where $p_I = \frac{\mu_I}{\mu}$. The realizations of $(X_n)_{n\geq 1}$ are shared between the two systems. These *coin flips* will determine whether an inelastic transition results in a self-loop or in a job completion.

Throughout time, both systems keep track of the total number of inelastic transitions which have occurred. More concretely, each system starts with its own counter, n, which is initialized to 0. For each system, if we randomly select a core holding an inelastic job while experiencing a potential transition, we increment this system's counter ($n \leftarrow n + 1$). We then check position n of the shared infinite sequence of coin flips. If $X_n = 1$, the inelastic job completes and exits the system. Otherwise, we have a self-loop transition and no job exits the system. Hence, for any policy, π, at time t, we have

$$C_\pi(t) = \sum_{i=1}^{I_\pi(t)} X_i.$$



(a) Multi-phase job, $\mu_E > \mu_I$ (b) Multi-phase job, $\mu_E < \mu_I$

Figure 7.8: Two cases of uniformizing multi-phase jobs. In Figure 7.8(a), $\mu_E > \mu_I$, so we take our dominating rate as $\mu := \mu_E$. We then take $p_I := \frac{\mu_I}{\mu}$, and set the the total transition rate out of the inelastic state to be μ. With probability $1 - p_I$, after completing an inelastic phase, we immediately start another one. With complementary probability p_I, the job does one of two things. With probability q, it completes. Otherwise, with probability $1 - q$, it begins an elastic phase. Figure 7.8(b) shows the analogous case where $\mu_E < \mu_I$.

Since S_{IF} and S_A share a common sequence of coin flips, $I_{\text{IF}}(t) \geq I_A(t)$ implies $C_{\text{IF}}(t) \geq C_A(t)$.

Proof of Lemma 7.5

Since we only need to show $I_{\text{IF}}(t) \geq I_A(t), \forall t \geq 0$, we can use the proof of Lemma 7.4 verbatim to prove Lemma 7.5.

7.5 Optimality in the General Case

We now consider the fully general multi-phase job structure, as seen in Figure 7.5(c). In order to prove Theorem 7.1, it suffices to prove Lemma 7.6 below.

Lemma 7.6. *Consider a n core system serving multi-phase jobs. Consider any policy A and let the policies* IF *and A start from the same initial conditions and have the same arrival time process. Then there exists a coupling between* IF *and A such that $I_{\text{IF}}(t) \geq I_A(t)$ and $C_{\text{IF}}(t) \geq C_A(t)$ for all $t \geq 0$, where $I_{\text{IF}}(t)$ (resp. $I_A(t)$) is the number of inelastic transitions by time t under* IF *(resp. A), and $C_{\text{IF}}(t)$ (resp. $C_A(t)$) is the number of jobs completed by time t under* IF *(resp. A).*

As in Section 7.4.2, we use uniformization to rewrite the job structure of Figure 7.5(c) so that the elastic and inelastic phase transitions have equal rates. There are two possible uniformizations here, once again depending on how μ_E and μ_I relate. Determining the dominating rate μ and transition probabilities p_I or p_E is the same as in Section 7.4.2, and the two possible uniformized Markov chains are shown in Figure 7.8. With these job structures in mind, we present the system coupling which allows us to prove the optimality of IF.

7.5.1 System coupling

As in Section 7.4.1, we construct a coupling that keeps systems S_A and S_{IF} in sync with respect to potential transition times and that allows us to use $I_A(t)$ to reason about $C_A(t)$.

Job arrivals: As in Sections 7.4.1 and 7.4.2, we let S_{IF} and S_A share the same arrival time sequence and start with the same initial conditions.

Job transitions and departures: For the most part, the transition process is similar to the uniformized case presented in Section 7.4.2. However, while we previously only needed a single infinite sequence of i.i.d. Bernoulli random variables, here we will need two. We state the two cases ($\mu_E < \mu_I$ and $\mu_E > \mu_I$) separately, as they differ slightly in their construction.

First, we consider $\mu_E < \mu_I$ (Figure 7.8(b)). Here, instead of a single sequence of coin flips, we have two shared sequences of coin flips. The first sequence, $(X_n)_{n\geq 1}$, is an i.i.d. sequence of $Bern(p_E)$ random variables. If $X_n = 1$, the nth elastic transition results in an elastic phase completion, producing an inelastic phase. Otherwise, if $X_n = 0$, the elastic transition does not result in a phase completion. The second sequence, $(Y_n)_{n\geq 1}$, is a sequence of i.i.d. $Bern(q)$ random variables. Recall that q is the probability that the completion of an inelastic phase will result in a job completion. If $Y_n = 0$, the nth inelastic transition results in the creation of an elastic phase. If $Y_n = 1$, the nth inelastic transition results in a job completion.

The case when $\mu_E > \mu_I$ (Figure 7.8(a)) is slightly more complex. Here, we do not have any self-loops for elastic phases. However, there are three possible outcomes for inelastic phases. We therefore keep track of two sequences of i.i.d. Bernoulli random variables, $(X_n)_{n\geq 1}$ and $(Y_n)_{n\geq 1}$. In the first sequence, X_n is distributed as $Bern(p_I)$. In the second sequence, Y_n is distributed as $Bern(q)$.

If $X_n = 0$, the nth inelastic transition does not result in the completion of an inelastic phase. If $X_n = 1$, the nth inelastic transition results in a phase completion, and we then examine the sequence (Y_n). If the nth inelastic transition results in the mth overall inelastic phase completion, we check Y_m. If $Y_m = 1$, the job completes, and if $Y_m = 0$, the job transitions to an elastic phase.

Because S_{IF} and S_A share the same sequence of coin flips, comparing the number of inelastic transitions between systems is equivalent to comparing the number of job completions. That is, if $I_{\text{IF}}(t) \geq I_A(t), \forall t \geq 0$, then $C_{\text{IF}}(t) \geq C_A(t)$.

7.5.2 Proof of Lemma 7.6

Multi-phase jobs add an extra layer of complexity which prevents us from directly leveraging the arguments used in Lemmas 7.4 and 7.5. When an inelastic phase completes, there are two possible outcomes: either an elastic phase will be produced or a job will complete. Our insight is that we can view the creation of an elastic phase as a job completion immediately followed by an arrival of a job in an elastic phase. This reduction puts us back in the case of two-phase jobs with unequal rates, allowing us to invoke Lemma 7.5. We formalize this argument in the proof of Lemma 7.6 below.

Proof of Lemma 7.6. Consider the above coupling under any given sample path. First, we replace each inelastic transition that produces an elastic phase with a different type of transition. Namely, we replace these transitions with a job completion followed immediately by an arrival of a job in an elastic phase. We will refer to this replacement as our *re-framing* of the sample path. Observe that the schedules produced in S_{IF} and S_A remain the same under the re-framing. While the number of job completions by any point t, $C_{\text{IF}}(t)$ and $C_A(t)$, may change under this re-framing, the key insight is that the number of inelastic transitions, $I_{\text{IF}}(t)$ and $I_A(t)$ respectively, remains identical. Thus, if we can argue that IF maximizes the number of inelastic transitions by any point in time under the re-framing, it does so in the original environment as well. This is sufficient for proving that $C_{\text{IF}}(t) \geq C_A(t), \forall t \geq 0$ under the original sample path.

Second, observe that our proof of Lemma 7.5 still holds if we allow additional arrivals at potential transition times, so long as these arrivals occur simultaneously in both systems. However, under our re-framing, the arrivals we add may not occur simultaneously in S_{IF} and S_A since they are generated by inelastic transitions to elastic phases. We address this issue by establishing the following claim.

Claim. *Let the sequence of additional arrival times under the re-framing be (t_n) in S_{IF} and (s_n) in S_A. For any $n \geq 1$, we have $t_n \leq s_n$, i.e. the nth additional arrival occurs in S_{IF} before it occurs in S_A.*

We will prove this claim below, allowing us to complete the proof of Lemma 7.6. Specifically, for any time t, let n be the index such that $t_n \leq t < t_{n+1}$. The claim tells us that S_A experiences additional arrivals at $s_1 \geq t_1, s_2 \geq t_2, \ldots, s_{n+1} \geq t_{n+1}$. However, we can view S_A as a system that has additional arrivals at $t_1, t_2, \ldots, t_{n+1}$, but chooses to not schedule these additional arrivals until after $s_1, s_2, \ldots, s_{n+1}$. Then by Lemma 7.5, we have $I_{\text{IF}}(t) \geq I_A(t)$, which completes the proof of Lemma 7.6. □

Proof of the claim. We will show inductively that the nth additional arrival occurs in S_{IF} before it does is S_A. We first argue that $t_1 \leq s_1$. Observe that, on the time interval $[0, t_1 \wedge s_1]$, S_{IF} and S_A experience precisely the same sequence of arrivals. Hence, by Lemma 7.5, IF maximizes the number of inelastic transitions by any time $t \in [0, t_1 \wedge s_1]$. In particular, it maximizes the number inelastic transitions by time $t_1 \wedge s_1$. Since S_{IF} and S_A share the same sequences of Bernoulli random variables, it must be that the system with more inelastic transitions experiences the first inelastic to elastic transition, and hence $t_1 \leq s_1$. Now note that the schedule produced by S_A is identical to that produced by a policy which receives the additional arrival at time t_1 instead of time s_1, but just chooses to ignore its existence until later on. This allows us to assume that the extra arrival into S_A occurs at t_1 instead of s_1. We then observe that, on the interval $[0, s_2 \wedge t_2]$, if systems S_{IF} and S_A experience the same sequence of arrivals then IF maximizes the number of inelastic transitions by time $s_2 \wedge t_2$. Hence, we have that $t_2 \leq s_2$.

Using the same iterative argument, it follows that S_A and S_{IF} can be assumed to experience the same sequence of arrivals up to time $s_n \wedge t_n$, and thus by Lemma 7.5 we have that $t_n \leq s_n$. □

Having proven Lemma 7.6, we can now prove Theorem 7.1.

Theorem 7.1. *Consider a n core system serving multi-phase jobs. The policy* IF *stochastically maximizes the number of jobs completed by any point in time. Specifically, for a policy A, let $C_A(t)$ denote the number of jobs completed by time t and let $N_A(t)$ denote the number of jobs in the system at t. Then under any arbitrary arrival time process, $C_{\text{IF}}(t) \geq_{st} C_A(t)$ for all times $t \geq 0$. Consequently, $N_{\text{IF}}(t) \leq_{st} N_A(t)$ for all times $t \geq 0$.*

Proof. Lemma 7.6 implies the existence of a coupling such that $C_{\text{IF}}(t) \geq C_A(t), \forall t \geq 0$. Consequently, $C_{\text{IF}}(t) \geq_{st} C_A(t), \forall t \geq 0$. Since the number of jobs in the system at time t is just the total number of arrivals by time t minus the total number of completions by time t, the claim $N_{\text{IF}}(t) \leq_{st} N_A(t), \forall t \geq 0$ also readily follows from Lemma 7.6. □

7.6 Evaluation

The analysis of Section 7.5 has shown that, when jobs have the structure presented in Figure 7.3, IF is optimal. In particular, IF minimizes the steady-state mean response time for any settings of μ_E, μ_I, q, and any arrival time process such that the system is stable.

The purpose of this section is three-fold. First, we examine the benefit of doing IF as opposed to other scheduling policies used in real-world systems or proposed in the literature. Second, we will relax the assumption that the phases are exponentially distributed and consider a range of phase size distributions from low-variability to high-variability. Third, we will consider jobs that consist of a deterministic number of phases and may end with an elastic phase, relaxing our assumption that jobs follow the underlying structure illustrated in Figure 7.3. We find that, even with these relaxed assumptions, IF is almost always a great choice compared to all other competitor policies.

We begin by describing the competitor policies in Section 7.6.1. Then we show the comparisons to IF via simulation in Sections 7.6.2 and 7.6.3.

7.6.1 Competitor Scheduling Policies

We compare IF to three competitor policies.

EQUI is a phase-unaware policy for scheduling parallelizable jobs that has been widely advocated for in the worst-case [24, 25, 26] theoretical literature. EQUI divides severs equally among all jobs in the system. That is, if there are N jobs in the system, each job receives an allocation of $\frac{n}{N}$ cores regardless of its current phase.

Phase-Aware First-Come-First-Served (PA−FCFS**)** is a popular policy in systems applications because it is easy to implement with little space or time overhead. PA−FCFS proceeds by iteratively looking at the next earliest arriving job in the system and allocating as many cores as possible to this job until all cores have been allocated. (A job in an inelastic phase is obviously allocated only 1 core.)

Elastic-First (EF**)** gives strict priority to the earliest arriving job in an elastic phase. If no jobs are in an elastic phase, cores are allocated to any jobs in an inelastic phase in FCFS order. Intuitively, EF seems like it might perform well in cases where elastic phases are smaller than inelastic phases on average. In this case, EF can be thought of as a greedy policy which continuously minimizes

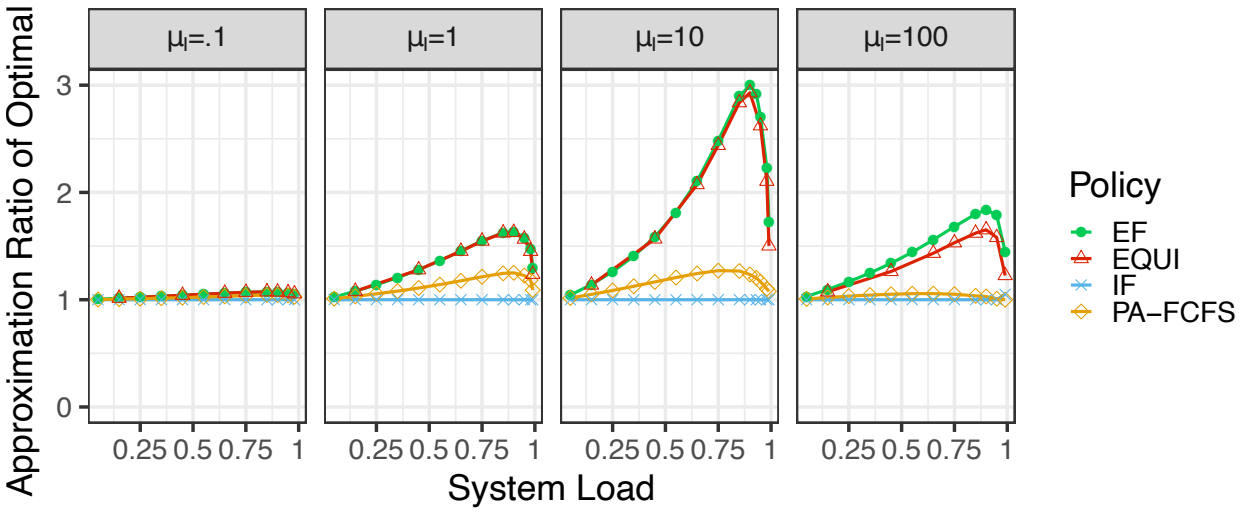


Figure 7.9: The approximation ratio of mean response time under EQUI, EF, PA−FCFS, and IF when compared with the optimal mean response time. Phases have exponentially distributed inherent sizes. IF is optimal (see Section 7.5) and thus has an approximation ratio of 1. In each case, $n = 100$, $\mu_E = 1$, $q = 0.2$, and jobs arrive according to a Poisson process. All jobs begin with an elastic phase. Results are shown as μ_I varies from $\mu_I = 0.1$ (the rare case where inelastic phases are long compared to elastic phases) to $\mu_I = 100$ (the more common case where inelastic phases are short compared to elastic phases).

the expected time until the next phase completion. This is analogous to the GREEDY* policy proposed in Chapter 6. However, we will see that this intuition is wrong.

7.6.2 Evaluation of Policies Under Our Job Model

Figure 7.9 shows the results of simulations comparing the performance of IF, EQUI, PA−FCFS, and EF under the model defined in Section 7.2 as we vary μ_I. Each simulation consists of 100 million job completions. Although we have already proven the optimality of IF in these cases, Figure 7.9 shows that the improvement of IF over the competitor policies is significant. In this small sample of the parameter space, IF outperforms PA−FCFS by up to 25%, and outperforms EF and EQUI by as much as a factor of 3. It is interesting to note that IF outperforms EF even when $\mu_E > \mu_I$. Even in this case, EF suffers from its failure to defer parallelizable work.

7.6.3 Jobs with Alternate Phase Size Distributions

Although we have shown that IF is optimal when phase sizes are exponentially distributed, we wish to further show that IF outperforms other policies under a range of phase size distributions. To examine the sensitivity of IF's performance to the underlying phase size distributions, we examine different distributions with a range of variances. Specifically, we consider the case where phases are Weibull distributed and the *squared coefficient of variation*, C^2, of the phase size distri-

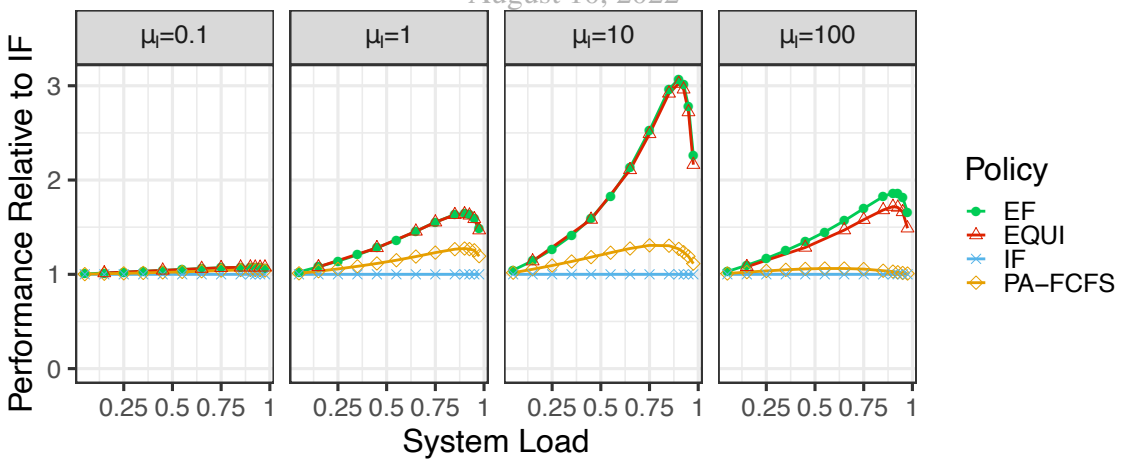
bution is both higher and lower than that of an exponential distribution.[3]

Figure 7.10 shows the performance of each competitor policy in simulation relative to the performance of IF (hence, the performance of IF is always normalized to 1). In most cases, IF is still the best of the four policies by a wide margin.

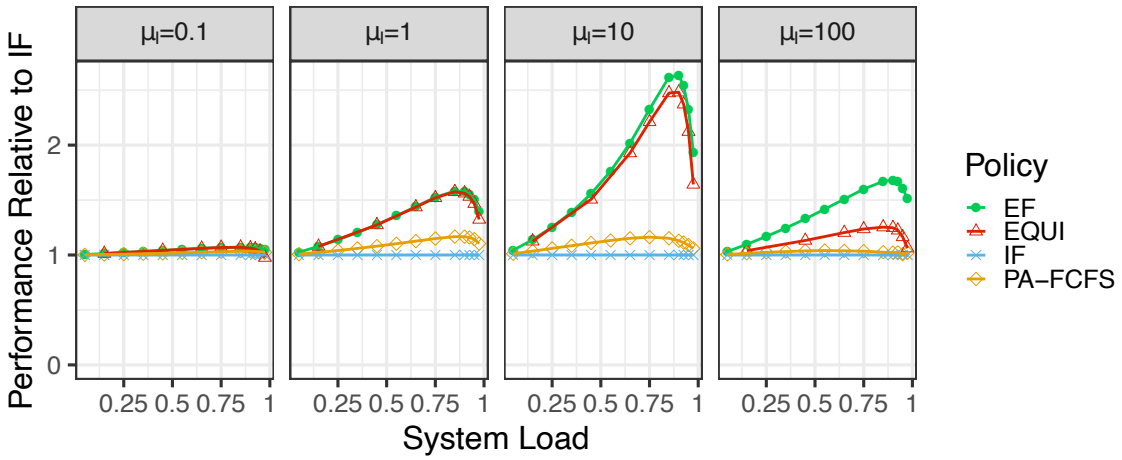
When $C^2 = 50$ we do find examples where EQUI outperforms IF. Here, when job sizes are highly variable, EQUI benefits from its insensitivity to the variance of the job size distribution (see Lemma 4.1). Specifically, because the phase size distributions have decreasing failure rates, working on phases with the *least attained service* will generally result in completing smaller phases before larger phases [2]. EQUI biases in this direction by dividing cores equally amongst all jobs in the system.

The relative performance of the competitor policies compared to IF also depends on the distribution of the number of phases comprising each job. For instance, when $q = 1$ and all jobs begin with an elastic phase, IF and PA−FCFS are equivalent policies. However, as q decreases, for a given system load, the gap between IF and PA−FCFS widens as it becomes increasingly likely that PA−FCFS will make a mistake and give priority to an elastic phase. Hence, although Figure 7.10 shows that EQUI can outperform IF when $q = .2$, in the case where $q = .025$ IF again outperforms EQUI at all loads.

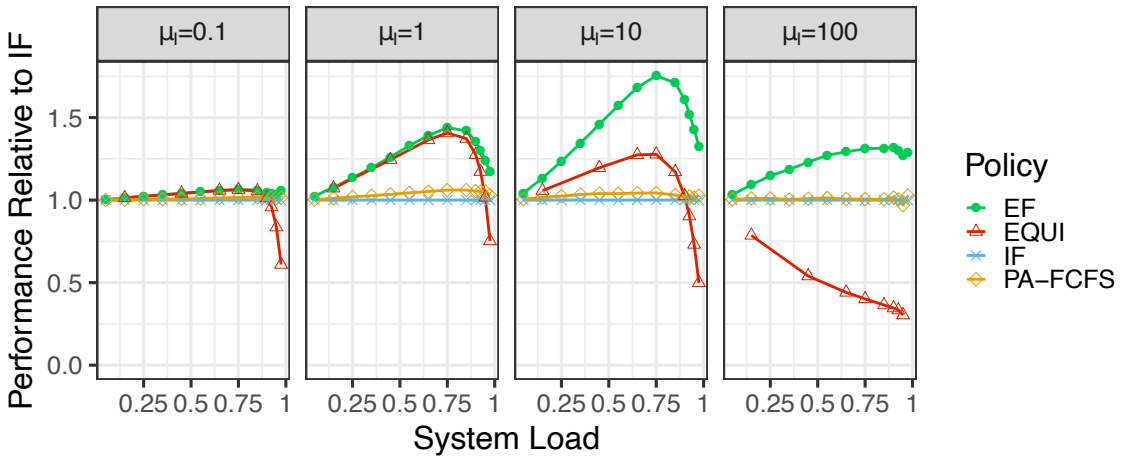
[3]A Weibull distribution with shape parameter $k = 1$ collapses to an exponential distribution ($C^2 = 1$). Adjusting k changes the distribution to have either higher C^2 ($k < 1$) or lower C^2 ($k > 1$) than an exponential distribution.



(a) $C^2 = 0.5$



(b) $C^2 = 5$



(c) $C^2 = 50$

Figure 7.10: The approximation ratio of mean response time under EQUI, EF, PA−FCFS, and IF, all compared with IF, when phases follow a Weibull distribution. In each case, $n = 100$, $\mu_E = 1$, $q = 0.2$, and jobs arrive according to a Poisson process. IF typically still outperforms the competitor policies. When jobs are highly variable ($C^2 = 50$), EQUI outperforms IF due to its insensitivity to job size variance.

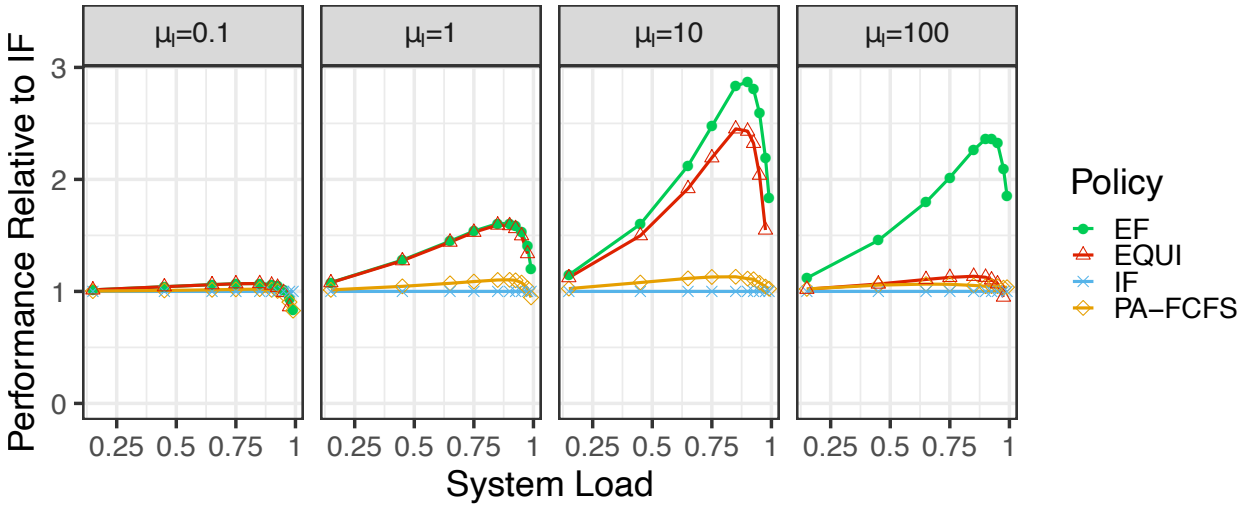


Figure 7.11: The approximation ratio of mean response time under EQUI, EF, PA−FCFS, and IF, all compared with IF, when jobs consist of a deterministic number of phases. In each case, $n = 100$, $\mu_E = 1$, and jobs arrive according to a Poisson process. Each job consists of 5 phases whose sizes are Weibull distributed with $C^2 = 5$. Each phase, including the last phase, is chosen to be either elastic or inelastic with probability .5. Although these changes to the job structure violate our theoretical assumptions, IF is still generally the best of the policies we consider.

7.6.4 Jobs with Alternate Structures

We can also examine the effect of the job structure on the performance of IF by considering cases where the sequence of a job's phases is not determined by a Markov chain of the form shown in Figure 7.3. Specifically we consider the case where the total number of job phases is deterministic. Furthermore, we allow each of the job phases to be either elastic or inelastic, including the final phase of each job. The results of these experiments look largely the same as the results in Figure 7.10, and an example of these results is shown in Figure 7.11. Although IF is not optimal in this case, it is still frequently the best of the policies we consider.

One notable exception where IF becomes worse than the competitor policies occurs when $\mu_I = 0.1$ and system load is high. In this case, the benefit of doing IF has been diminished in several ways. First, the benefit of deferring parallelizable work is realized when there are very few jobs in the system, but the scheduling policy is still able to utilize all n cores because at least one job is in an elastic phase. Under very high load, the probability of having few jobs in the system vanishes, decreasing the benefit of deferring parallelizable work. Second, the change in job structure has made it possible for IF to work on jobs with longer expected remaining total sizes. This is an effect of both allowing jobs to end in elastic phases and making the total number of phases for each job deterministic. Finally, we are considering the case where $\mu_I = 0.1$, meaning inelastic phases are ten times larger than elastic phases on average. Because IF prioritizes inelastic phases, this particular choice of the phase size distributions further increases the chance that IF suffers by completing jobs with larger total sizes before jobs with smaller total sizes. Despite all of these challenges, IF still performs within 18% of the best competitor policy.

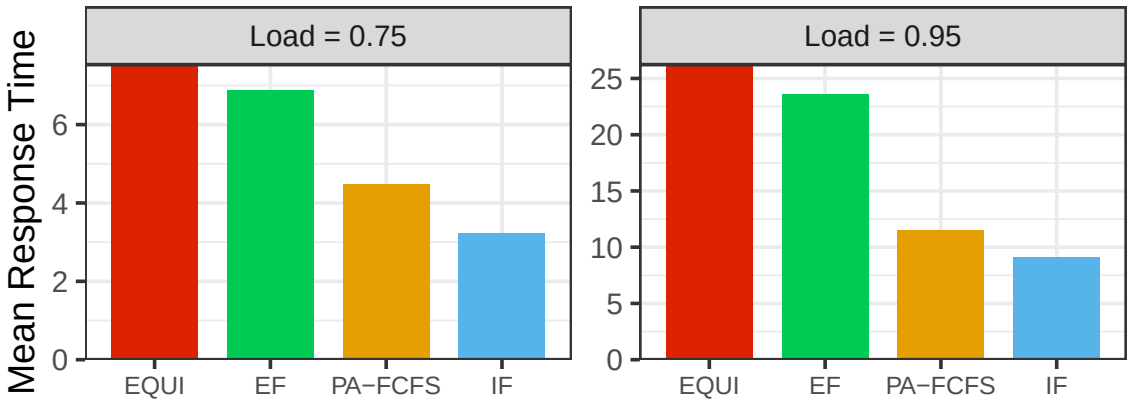


Figure 7.12: The mean response time of EQUI, EF, PA−FCFS, and IF processing a workload consisting of a mixture of 5 queries from the Star Schema Benchmark. We assume Poisson arrivals. Although this workload violates our modeling assumptions, IF is still the best policy by a wide margin. IF improves upon the next best policy, the PA−FCFS policy used in the NoisePage database, by up to 30%.

7.7 Case Study: Scheduling in Databases

Throughout this chapter, we have drawn inspiration for our model from a range of systems including modern databases. In this section, we consider whether the scheduling policies that we have proposed work well for real database workloads. Because this section specifically considers scheduling in databases, we will refer to a scheduling policy as allocating cores to *queries* rather than allocating cores to jobs. Real database workloads differ from our modeling assumptions in two ways. First, phase sizes are not exponentially distributed. Second, the sequence of phases for each query is not determined by an underlying Markov chain. In this case study, we ask whether IF will still perform well under these real-world conditions.

To answer this question, we perform simulations using a workload consisting of a mixture of five queries from the Star Schema Benchmark. The ordering of phases and the phase sizes in our simulations are based on the timings of actual queries running in the NoisePage database, as shown in Figure 7.1. Each query consists of a deterministic number of elastic and inelastic phases (either six or seven total phases per query). Of the five queries we consider, four queries end with an inelastic phase and one query ends with an elastic phase. For each arrival in our simulation, a query type is drawn uniformly at random. The query sizes have a squared coefficient of variation of $C^2 = 0.41$, meaning that query sizes are not highly variable in this case. In particular, note that these queries clearly deviate from the job structure illustrated in Figure 7.3.

Figure 7.12 shows the results of these simulations. The ordering of the policies with respect to mean response time is the same as what we observed in Section 7.6. In particular, IF is again consistently the best of the policies we consider, and IF outperforms the PA−FCFS policy used in the current version of NoisePage by up to 30%.

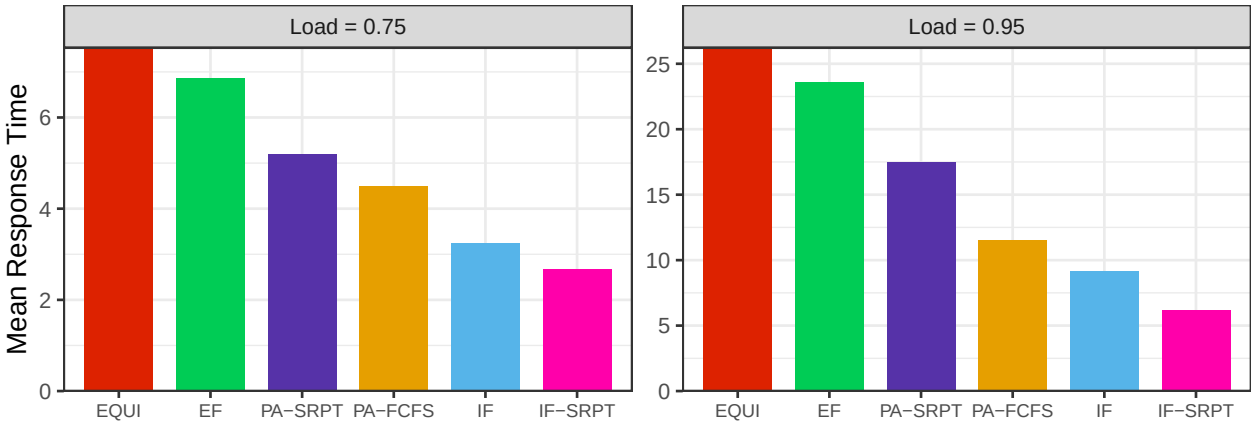


Figure 7.13: **Comparison with size-aware policies.** The mean response time of EQUI, EF, PA−SRPT, PA−FCFS, IF and IF−SRPT processing a workload consisting of a mixture of 5 queries from the Star Schema Benchmark. We assume Poisson arrivals. IF−SRPT can improve upon IF by 33% by leveraging query size information. Notably, PA−SRPT performs worse than PA−FCFS despite attempting to leverage size information.

7.7.1 Size-Aware Scheduling

Although the focus of this chapter has been the setting where job sizes are unknown to the scheduler, we recognize that schedulers in real-world databases often have knowledge of the size of each query phase and the number of phases comprising each query. Specifically, the query planner in the NoisePage database on which we have based our simulations can provide the scheduler with information about the sequence of phases for each query and an estimate of phase sizes in addition to information about the current phase.

Historically, when job sizes are known, the performance modeling community has advocated for reducing mean response time by trying to complete smaller jobs before larger jobs [93]. This begs the question of whether the phase-aware policies developed in this chapter can be improved by adapting them to favor short jobs. Notably, NoisePage and many other databases use a PA−FCFS policy, and do not leverage the available information about query sizes to make better scheduling decisions. Would favoring short queries improve response times in these systems?

Our theorems in Sections 7.4 and 7.5 have shown the importance of *deferring parallelizable work* by giving priority to inelastic phases in order to maintain the overall efficiency of the system. In the case where phase sizes are known, it is not immediately clear how to balance the objectives of favoring shorter queries and deferring parallelizable work.

Size-aware Scheduling Policies. We now consider two size-aware scheduling policies that favor queries with smaller *remaining total size*, the sum of the remaining sizes of all of a query's remaining phases. As we will see, one of the scheduling policies performs well because it manages to both favor short queries and grant strict priority to inelastic phases. However, the other policy, which favors the shortest queries in the system but does not otherwise defer parallelizable work, does even worse than PA−FCFS.

The first policy we consider is an adaptation of IF to the case where query sizes are known

to the scheduler. We call this new policy *Inelastic-First-Shortest-Remaining-Processing-Time* (IF−SRPT) because it combines IF with the ubiquitous SRPT scheduling policy. IF−SRPT gives strict priority to inelastic phases over elastic phases in the same manner as IF. However, among inelastic phases, IF−SRPT gives priority to the phases belonging to the queries with the smallest remaining total sizes. Likewise, when choosing to run an elastic phase, IF−SRPT will choose the elastic phase belonging to the query with the smallest remaining total size.

Our second policy is a *Phase-Aware SRPT* policy, which we refer to as PA−SRPT. PA−SRPT gives strict priority to the phases belonging to the queries with the smallest remaining total sizes, regardless of whether a phase is elastic or inelastic. However, PA−SRPT is phase-aware in that it avoids allocating too many cores to inelastic phases. Hence, if the query with the smallest remaining total size is in an inelastic phase, PA−SRPT will allocate one core to this query. If the next smallest query is in an elastic phase, PA−SRPT will allocate the remaining $n-1$ cores to this second smallest query. Although PA−SRPT does not explicitly defer parallelizable work, it biases more strongly towards the shortest queries in the system than IF−SRPT does.

We again evaluate these policies using a workload based on the Star Schema Benchmark, and the results are shown in Figure 7.13. Unsurprisingly, IF−SRPT is the best performer. It benefits from biasing its allocations towards shorter queries while still deferring parallelizable work. This leads IF−SRPT to achieve a mean response time which can be 33% lower than that of IF, and 47% lower than that of the PA−FCFS policy used in NoisePage. What is more counterintuitive is that PA−SRPT performs quite poorly. In fact, PA−SRPT is worse than PA−FCFS in both cases shown in Figure 7.13, and IF−SRPT outperforms PA−SRPT by up to a factor of 3.

7.7.2 Why PA-SRPT is worse than PA-FCFS

As seen in this chapter, deferring parallelizable work is vital to reducing mean response time. PA−SRPT suffers from its failure to defer parallelizable work. It is not immediately clear, however, why PA−SRPT is even worse than PA−FCFS, given that neither policy explicitly defers parallelizable work.

Although neither PA−SRPT nor PA−FCFS explicitly defers parallelizable work, we can see that PA−SRPT suffers because it inadvertently defers far less parallelizable work than PA−FCFS. We define the *percentage of deferred parallelizable work* under a given policy at time t to be the number of cores allocated to inelastic phases divided by the number of cores that IF would allocate to inelastic phases. We can then consider the long-run time-average percentage of deferred parallelizable work under various policies. We normalize this quantity using the allocations under IF because IF allocates as many cores to inelastic phases as possible without being wasteful. As a result, IF defers 100% of parallelizable work by definition. Phase-unaware policies, such as EQUI, can defer more than 100% of parallelizable work by wastefully allocating too many cores to inelastic phases.

Figure 7.14 shows that PA−SRPT defers far less parallelizable work than PA−FCFS, leading PA−SRPT to perform poorly. The poor performance of PA−SRPT is due to the structure of one particular query, which is both small in total size and ends with an elastic phase. PA−SRPT tends to give strict priority to this type of query, while PA−FCFS treats all queries equally. This leads PA−SRPT to defer less parallelizable work than PA−FCFS. Figure 7.14 also shows that, while

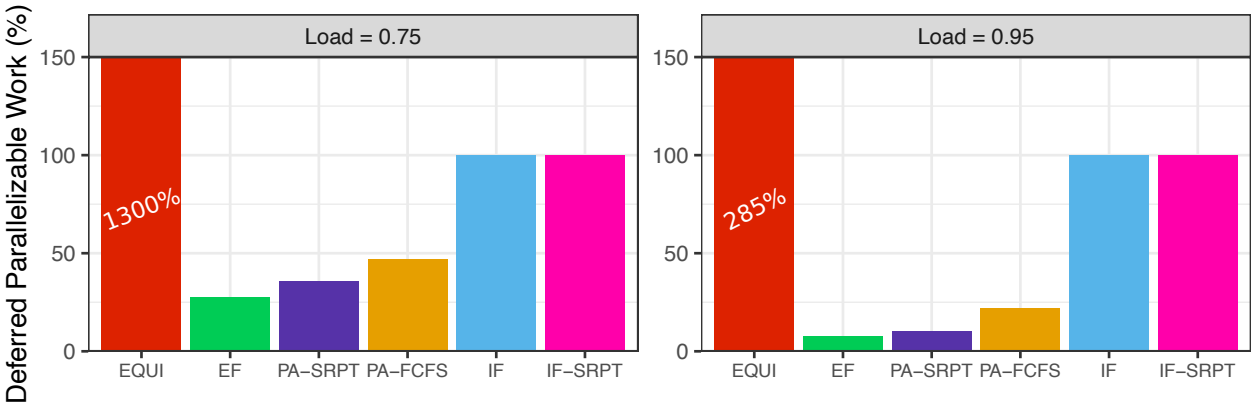


Figure 7.14: The percentage of deferred parallelizable work under `EQUI`, `EF`, `PA−SRPT`, `PA−FCFS`, `IF` and `IF−SRPT` given a workload consisting of a mixture of 5 queries from the Star Schema Benchmark. `IF−SRPT` defers 100% of parallelizable work, but `PA−SRPT` defers even less parallelizable work than `PA−FCFS`.

`IF−SRPT` manages to favor short jobs, it still defers parallelizable work. `IF−SRPT` is able to both defer 100% of parallelizable work *and* prioritize shorter queries, leading to lower mean response time.

7.7.3 Implementing an Improved Database Scheduler

Given the significant performance improvement promised by the above simulations, we implemented new scheduling policies for the NoisePage database [75]. NoisePage is an open-source database which schedules queries using the concept *morsel-driven parallelism* [58]. In NoisePage, each elastic phase is divided into a set of tasks which can be run concurrently. Inelastic phases are represented as a single task, and therefore cannot be parallelized. A phase is considered complete when all of its tasks are complete. A scheduling policy is responsible for mapping the tasks of each query onto a pool of worker threads. For example, the default scheduling policy in NoisePage prioritizes tasks belonging to earlier-arriving queries, and is therefore analogous to the `PA−FCFS` policy described in this chapter.

We implemented scheduling policies inspired by `IF` and `IF−SRPT` in NoisePage. This required us to create mechanisms for passing task metadata to the scheduling policy. Namely, we passed information to the scheduler about whether each task belongs to an elastic or inelastic phase, and we passed the scheduler the overall remaining size of each query. Query sizes are determined by an offline profiling step. We also built a new scheduling framework that is capable of enforcing arbitrary priority schemes between queries while avoiding the synchronization and lock contention that can arise from a centralized scheduler design. In our design, each worker thread is associated with its own set of independent run queues, with each queue representing a different task priority level. For example, in our implementation of `IF`, each worker thread has two queues. One queue holds tasks belonging to elastic phases and one queue holds tasks that represent inelastic phases. When looking for its next task, a worker thread gives priority to tasks in its inelastic task queue.

Inelastic tasks are then served FCFS order. To accommodate more complex priority schemes, each queue is also maintained as a priority queue. For example, in our implementation of `IF-SRPT`, the remaining size of each query is used to determine the relative priorities of the tasks in each inelastic task queue. `IF-SRPT` then gives priority to inelastic tasks belonging to queries with shorter remaining sizes. Running tasks are never preempted, so priorities are enforced at a task-level granularity. New tasks are immediately enqueued onto various run queues using a Join-Shortest-Queue dispatching policy [31].

To evaluate our new scheduling policies, we constructed a benchmark based on the queries from the Star Schema Benchmark (SSB). Our benchmark uses a Poisson arrival process where the query type of each new arrival is chosen uniformly at random from the full set of SSB queries. We ran our benchmark on a 40-core, 2-socket machine [4]. We found that using `IF` improved mean response time by 25% and using `IF-SRPT` improved mean response time by 30% over the default `PA-FCFS` scheduling policy used in NoisePage. These experiments show that, despite the differences between our theoretical model and our real-world implementation, we can still realize the significant performance gains promised by our theoretical results and simulations.

One modeling decision that likely impacts the performance of our new scheduling policies is that our model does not consider memory or caching effects. As a result, our new scheduling policies do not consider the state of the various on-chip caches or the virtual memory system. Intuitively, it seems that a policy like `PA-FCFS` should perform fewer preemptions, and could thus benefit from having a lower cache miss ratio. To measure this type of effect, we ran experiments using `perf` [80] to monitor the performance of the memory hierarchy. We found that the default scheduling policy under NoisePage results in a last-level cache miss ratio of 36.2% while our `IF` and `IF-SRPT` policies resulted in last-level cache miss ratios of 36.4% and 40.4% respectively. That is, the number of cache misses is roughly 10% higher under `IF-SRPT` than under `PA-FCFS` or `IF`. We therefore conclude that caching effects do have the potential to limit the performance of `IF-SRPT`, but that these effects also appear to be minor in comparison to the benefits of prioritizing short queries and deferring parallelizable work. Similar measurements of virtual memory performance, such as TLB hit ratios, were roughly identical between the different scheduling policies.

Another difference between our model and our implementation is that we do not model each task of an elastic phase individually. Our assumption is that, as long as the number of tasks in an elastic phase is fairly large in comparison to the number of cores in the system, the kinds of straggler effects [85] captured by a more detailed model should be small relative to the overall response time of a query. The benefit of this assumption is tractability — modeling queries at the individual task level results in a DAG model of parallelism, which has proven difficult to analyze (see Chapter 3). By default, NoisePage splits elastic operations into a large number of relatively small tasks, so our modeling assumption is reasonable for a wide range of systems. Nonetheless, a more detailed model could more faithfully capture the dynamics of a real-world database, and may be necessary in order to accurately capture the performance of larger-scale systems where the number of cores can outstrip the number of tasks comprising an elastic operation. Furthermore, although the process for splitting an elastic phase into individual tasks is currently done statically,

[4]Model: Intel Xeon Gold 5218R

an improved scheduling policy may want to dynamically control how elastic phases are divided based on the state of the system. Our implementation results show that the IF and IF−SRPT policies provide a useful starting point for policies which want to further optimize query execution at the individual task level.

7.8 Conclusion

This chapter addresses the optimal scheduling of parallelizable jobs that consist of different numbers of elastic and inelastic phases. In Chapter 6 we saw that if jobs were entirely elastic or entirely inelastic, IF was optimal in many cases. Viewed through the lens of the model used in this chapter, the results of Chapter 6 essentially consider a special case where each job only consists of one phase. Given that the model presented in this chapter is much more general, allowing for an arbitrary number of phases per job, it is not obvious that our results from Section 6.4 should generalize cleanly. However, we prove in this chapter that the IF policy, which defers parallelizable work, is optimal in a strong sense: for any number of cores, n, for any system load, ρ, for any arrival process (including adversarial arrivals), and when jobs can each consist of an arbitrary number of phases. While our proofs require that the phases have exponentially distributed sizes and that jobs all end with inelastic phases, experimental evaluation shows that the dominance of IF typically extends to cases when job sizes and structures deviate from these assumptions. Furthermore, IF only needs to know the current phase of each job in order to schedule jobs optimally. That is, IF does not require knowledge of the job parameters, μ_I, μ_E, and q. We also show that IF performs well in simulation under database workloads where job structures, phase sizes, and phase types can vary widely.

In the setting of scheduling in databases, it is common that the scheduler not only knows the phase of each job but also has knowledge of the job's size. When job sizes are known, it is natural to consider scheduling policies which favor short jobs. We consider two such policies: the PA−SRPT policy which strictly favors jobs with the shortest remaining total sizes and the IF−SRPT policy which both defers parallelizable work and favors short jobs. We find that IF−SRPT is far superior to PA−SRPT and performs even better than IF. This somewhat counterintuitive result underscores the importance of deferring parallelizable work when scheduling parallelizable jobs composed of phases.

One may ask why we bothered to include EQUI in our performance evaluation for this chapter, since it is easy to predict that nearly any phase-aware policy should be better than EQUI. However, the fact that policies like IF and IF−SRPT outperform EQUI (and other *phase-unaware* policies) by such a large margin could have important implications for the design of future systems. As the gap between phase-aware and phase-unaware scheduling policies continues to widen, system designers should be increasingly tempted to pursue designs which permit the efficient implementation of phase-aware scheduling policies. In fact, one could argue that the benefits of phase-aware scheduling have already had one major influence on computer systems. The benefits of separating elastic and inelastic work into separate phases are at least partially responsible for the popularity of the MapReduce paradigm [22] in distributed computing. The results of this chapter argue that we should be continuing to look for opportunities to enable phase-aware scheduling in a wide variety

of systems.

It is worth noting that the same kind of open theoretical problem that we faced at the end of Chapter 6 remains open at the end of this chapter as well. In chapter 6, we were unable to provide optimality results when elastic jobs were smaller than inelastic jobs on average. If we consider the case where phase sizes are known to the system, we face the same kind of challenges in deriving an optimal policy. In particular, our intuition tells us that a policy should both favor short jobs and defer parallelizable work, leading us to recommend the `IF-SRPT` policy. However, it is possible that an optimal policy could do even better by occasionally using `PA-SRPT` in cases where the remaining size of a job in an elastic phase is *much smaller* than the remaining size of a job in an inelastic phase. We conjecture that one could outperform `IF-SRPT` by dynamically switching between `IF-SRPT` and `PA-SRPT` based on the amount of elastic work and the number of jobs in the system. Making this decision in a principled way, however, remains an open problem.

Chapter 8

Conclusion and Future Work

8.1 Conclusion

This thesis has considered several of the most important settings in which a system is tasked with scheduling parallelizable jobs. Throughout this thesis, we have demonstrated the power of our three-pronged approach to developing policies for scheduling parallelizable jobs:

First, we develop stochastic models that are suited to the particular setting we are considering. In Chapter 4, we modeled systems where the scheduler does not have job size information by assuming that job sizes were unknown and exponentially distributed. In Chapter 5, we modeled systems where the scheduler can estimate jobs sizes by assuming that job sizes are known exactly to the system. Similarly, Chapter 4 and Chapter 5 model cases where all jobs are instances of the same program and thus follow the same speedup function, while Chapter 6 and Chapter 7 model cases where the workload is composed of different types of jobs that can follow different speedup functions.

Second, each chapter considers the how the particular model being considered impacts the kind of scheduling policy we should use. When the scheduler has no information about job sizes and all jobs follow the same speedup function, we saw that the `EQUI` policy performs optimally by maximizing instantaneous system efficiency (Chapter 4). However, when job sizes are known (Chapter 5) or speedup functions vary from job to job (Chapter 6 and Chapter 7), an optimal policy may be able to reduce mean response time by sacrificing instantaneous system efficiency.

Third, we examine the performance of our new scheduling policies either analytically or through simulation to show the benefit of our principled approach to scheduling parallelizable jobs. We conclude that policies such as heSRPT (Chapter 5) or `IF-SRPT` (Chapter 7) stand to dramatically improve performance in real-world systems that process parallelizable jobs.

8.2 Scheduling Tradeoffs

This thesis demonstrates how, by formally understanding the problem of scheduling parallelizable jobs, we can design scheduling policies that balance the complex set of *tradeoffs* shown in Figure 8.1.

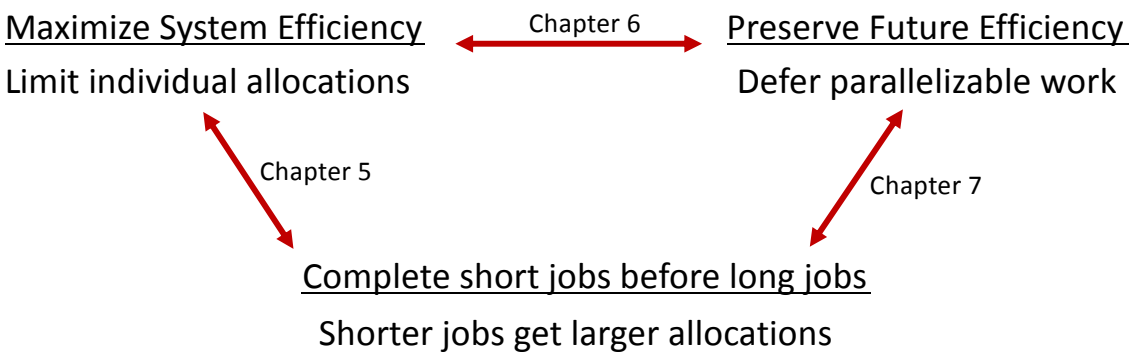


Figure 8.1: The tradeoffs balanced by the scheduling policies described in this thesis.

Chapter 2 formally develops the notion of *instantaneous system efficiency*, a metric which describes how well a set of jobs is able to make use of a corresponding set of core allocations. In Chapter 4, we see that instantaneous system efficiency is maximized by ensuring that no single job receives too many cores. Here, we can show that when job sizes are unknown and exponentially distributed, and all jobs follow the same speedup function, the `EQUI` policy is optimal because it maximizes instantaneous system efficiency. However, if the system has more information about job sizes or speedup functions, finding an optimal policy becomes much more complex.

In particular, when job sizes are known to the system (Chapter 5), it is possible to reduce the mean response time across jobs by giving larger allocations to smaller jobs. However, one cannot simply use an SRPT policy which allocates all of the cores to the smallest job in the system. Heavily favoring any one individual job contradicts the lesson learned in Chapter 4, that system efficiency is maximized by sharing cores equally. Hence, an optimal policy must balance a tradeoff between the high efficiency of a policy like `EQUI`, and the benefits of completing short jobs before long jobs. We derive and analyze the optimal policy, *high-efficiency SRPT*, which lies between `EQUI` and SRPT and gets the best features of both policies.

Similarly, when jobs are known to follow different speedup functions (Chapter 6 and Chapter 7), we must judiciously decide how to favor individual jobs in the system. In this setting, more parallelizable jobs are able to make more efficient use of additional cores than less parallelizable jobs. Thus, we conclude that in order to maximize instantaneous system efficiency, we should favor the more parallelizable jobs. However, favoring the more parallelizable jobs can have unintended consequences. Favoring the more parallelizable jobs will cause the more parallelizable jobs to complete quickly. The system will then be left with only less parallelizable jobs to work on. Hence, although favoring the more parallelizable jobs may maximize system efficiency in the present, doing so may lower system efficiency *in the future*. We therefore develop policies such as GREEDY* and `IF` that *defer parallelizable work*. These policies balance a tradeoff between maximizing the instantaneous system efficiency, and preserving the *future* system efficiency.

Finally, in Chapter 7, we consider complex cases where all of the above tradeoffs must be balanced simultaneously. That is, we must consider three competing goals of maximizing instantaneous system efficiency, preserving future system efficiency, and favoring short jobs. We find that the `IF-SRPT` policy is able to balance these tradeoffs and significantly outperform existing polices for scheduling queries from the Star Schema Database Benchmark. This result demonstrates

how a principled approach to scheduling parallelizable jobs can lead to policies which perform well both in theory *and* in practice.

8.3 Impact

While many systems employ heuristic policies when scheduling parallelizable jobs, these policies generally rely solely on intuition rather than a formal understanding of the scheduling problem. This leads to suboptimal performance in systems such as [23, 82] which employ GREEDY policies, but do not use GREEDY*. In some cases, a heuristic approach can actually make system performance worse. For example, [102] advocates using a heuristic-based version of the `PA-SRPT` policy instead of `PA-FCFS` to schedule database queries composed of multiple phases. However, we show in Chapter 7 that `PA-SRPT` can be worse than `PA-FCFS` because it fails to defer parallelizable work. Hence, real-world systems stand to benefit greatly from the results of this thesis.

Note, however, that the results of this thesis are not solely intended to prescribe the use of a particular scheduling policy in an existing system, but to also describe the potential benefits of building systems which provide additional information to their schedulers. For example, the developers of the NoisePage database asked us to model, based on their benchmarks, how much benefit they could expect from building a phase-aware scheduler, since this would represent a significant development effort. Using simulations derived from database benchmarks, we were able to show that phase-aware scheduling would significantly improve mean query latency, and was thus worth implementing. Each of the scenarios considered in this thesis represents a similar opportunity for improvement for system developers. By building schedulers with a greater awareness of job size and scalability information, our theoretical results show that we can greatly reduce response times in real-world systems.

8.4 Future Work

Although we address a wide range of settings in this thesis, the problem of how to schedule parallelizable jobs is far from solved.

The most prominent of these questions is how one should schedule jobs with more complex sequences of phases. For example, job phases could change according to a Markov process consisting of a greater number of phases with generally distributed phase sizes and speedup functions which lie in between the elastic and inelastic speedups considered in this thesis. Furthermore, the Markov process corresponding to each job could be unique. This model begins to resemble the well-studied DAG model, and thus promises to be harder to analyze. However, the generality of this model makes it well-suited to accurately modeling a wide variety of parallel computations.

Another direction to consider is to extend the analyses of this thesis to consider the tail of response time rather than the mean. System developers and administrators are often interested in the 99th percentile (P99) of response time, for example. It is interesting to ask whether policies like heSRPT which favor shorter jobs in order to reduce mean response time could actually be bad for the P99 of response time since they bias away from longer jobs which are more likely

to populate the tails of the response time distribution. The question here is whether heSRPT is effective enough in reducing queueing times to offset its bias against large jobs.

Finally, the question of how to schedule parallelizable jobs in systems composed of heterogeneous servers is becoming increasingly important. Data centers are increasingly composed of servers with different processors, different ratios of memory to CPU, and different hardware acceleration capabilities [100]. This can cause the run time of a job to depend heavily on not just how many servers it is allocated, but the type of servers it is allocated. These trends are being mirrored at the individual server level, and even on consumer-grade machines such as laptops and mobile devices[54]. This heterogeneity poses important new tradeoffs not considered in this thesis. For instance if several jobs are capable of running faster on a GPU than a CPU, which job should be run on which type of hardware? Should we favor shorter jobs, maximize system efficiency, or balance load between separate CPU and GPU queues? Fully accounting for the growing heterogeneity of modern systems will require both significant theoretical work and significant systems work to make modern schedulers capable of measuring and balancing these new tradeoffs.

Appendix A

Scheduling Without Job Sizes

A.1 Mixed-Random-Chunk

In this section, we explore Mixed-Random-Chunk (MRC), which uses two chunk sizes: k_1 and k_2. We will also restrict ourselves to consider policies which balance load between chunks according to the size of a chunk. That is, given that jobs are dispatched to an average of k cores, the average per-core arrival rate should be $\Lambda k/n$.

Under Mixed-Random-Chunk, we group a_1 of the n cores into chunks of size k_1. We call these chunks size-k_1 chunks. The remaining $a_2 := n - a_1$ cores are grouped into size-k_2 chunks. Like before, we assume that k_1 divides a_1, and k_2 divides a_2.

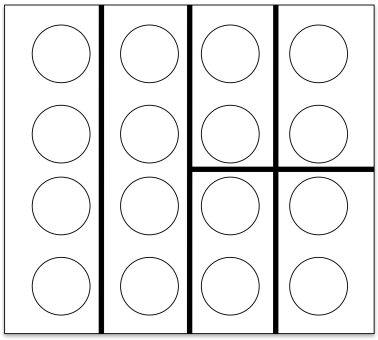


Figure A.1: A system with $n = 16$ under a Mixed-Random-Chunk policy. Here, $a_1 = 8$ of the cores are grouped into chunks of size $k_1 = 4$, and $a_2 = 8$ of the cores are grouped into chunks of size $k_2 = 2$.

When a job arrives, we choose a core uniformly at random (with probability $\frac{1}{n}$). The job is then parallelized across all cores in the chosen core's chunk. We can see via Poisson splitting that the arrivals to a size-k_i chunk form a Poisson process with rate $\frac{\Lambda k_i}{n}$. Under this policy, the level of parallelization is defined to be $k = (a_1k_1 + a_2k_2)/n$. Hence, the average per-core arrival rate is

$$\Lambda_k = \frac{a_1}{n} \cdot \frac{\Lambda k_1}{n} + \frac{a_2}{n} \cdot \frac{\Lambda k_2}{n} = \frac{\Lambda k}{n},$$

and Mixed-Random-Chunk obeys the load balancing property outlined above.

Mean Response Time

To derive the mean response time $\mathbb{E}[T]$ under Mixed-Random-Chunk, we must condition on the type of chunk to which an arrival is sent. We denote by E_i the event that the arrival is sent to a size k_i chunk. Recall that the arrival rate to a core in a size-k_i chunk is $\Lambda_{k_i} = \Lambda k_i/n$ and the size of a job piece at this core is distributed as $X_{k_i} = X/\mathrm{s}(k_i)$. This information suffices to calculate the mean response time of a job sent to a size-k_i chunk, $\mathbb{E}[T \mid E_i]$ (see Theorem 4.4 and its proof). Next, note that a job will be sent to a size-k_i chunk with probability a_i/n. Hence, we can find the mean response time for the entire system by conditioning:

$$\mathbb{E}[T]^{\text{MRC}} = \frac{a_1}{n}\mathbb{E}[T \mid E_1] + \frac{a_2}{n}\mathbb{E}[T \mid E_2]. \tag{A.1}$$

Optimizing a_1 and a_2

In (A.1) observe that a_1 and a_2 are decision variables that can be chosen to minimize mean response time. Surprisingly, we find in Theorem A.1 that the optimal choice is to have only *one* chunk size.

Theorem A.1. *The mean response time under Mixed-Random-Chunk is minimized when the number of size-k_1 chunks, a_1, is either 0 or n.*

Proof. Since $a_2 = n - a_1$, (A.1) can be rewritten as

$$\mathbb{E}[T] = \frac{a_1}{n}\left(\mathbb{E}[T \mid E_1] - \mathbb{E}[T \mid E_2]\right) + \mathbb{E}[T \mid E_2].$$

This expression is of the form $\mathbb{E}[T] = a_1 x + y$, and thus the mean response time is linear in a_1. We can also see that x and y are independent of a_1 and a_2. This follows since, for any core belonging to a chunk of size k_i, neither the service requirement, $X_{k_i} = X_{k_i}/s(k_i)$, nor the arrival rate $\Lambda_{k_i} = \Lambda k_i/n$ depend on a_i. Thus, our linear expression is either increasing or decreasing in a_1, and will thus be minimized at a boundary where $a_1 = 0$ or $a_1 = n$. □

Variance of Response Time

Not only is mean response time not improved by having multiple chunk sizes, but neither is the variance of response time, as shown in Theorem A.2.

Theorem A.2. *The variance of the response time, $\mathrm{Var}(T)$, under Mixed-Random-Chunk is minimized when the number of size-k_1 chunks, a_1, is either 0 or n.*

Proof. By conditioning we have that the variance of the response time is given by

$$\begin{aligned}\mathrm{Var}(T) =& \mathbb{E}[T - \mathbb{E}[T]]^2 = \mathbb{E}\left[T^2\right] - \mathbb{E}[T]^2 \\ =& \frac{a_1}{n}\mathbb{E}\left[T^2 \mid E_1\right] + \frac{a_2}{n}\mathbb{E}\left[T^2 \mid E_2\right] \\ & - \left(\frac{a_1}{n}\mathbb{E}[T \mid E_1] + \frac{a_2}{n}\mathbb{E}[T \mid E_2]\right)^2.\end{aligned}$$

Again, the substitution of $a_2 = n - a_1$ and subsequent algebraic manipulation leads to

$$\begin{aligned}\mathrm{Var}(T) = &- a_1^2 \left(\frac{\mathbb{E}\left[T \mid E_1\right] + \mathbb{E}\left[T \mid E_2\right]}{n} \right)^2 \\ &+ \frac{a_1}{n} \Big(\mathbb{E}\left[T^2 \mid E_1\right] - \mathbb{E}\left[T^2 \mid E_2\right] \\ &\quad - 2\mathbb{E}\left[T \mid E_1\right] \mathbb{E}\left[T \mid E_2\right] + 2\mathbb{E}\left[T \mid E_2\right]^2 \Big) \\ &\quad + \mathbb{E}\left[T^2 \mid E_2\right] - \mathbb{E}\left[T \mid E_2\right]^2 .\end{aligned}$$

Note that this expression is of the form $\mathrm{Var}(T) = -a_1^2 x + a_1 y + z$, and thus the variance of the response time is quadratic in a_1. Furthermore, it is clear from the expression that x is positive, since mean response times must be positive. The variance of the response time, then, is a concave quadratic in a_1. Since the number of cores, a_1, assigned to size-k_1 chunks takes an integer value between 0 and n, this means that the variance of the response time is minimized on a boundary, when $a_1 = 0$ or $a_1 = n$. □

Why Multiple Chunk Types Do Not Help

The issue is that the benefit of having larger chunks available is usually outweighed by a reduction in the stability region of the system. We have therefore chosen to omit the bulk of our analysis of these policies.

A.2 The Nelson-Philips Approximation

Here we state the approximation of mean response time under JSQ as given in [70].

Let $M/M/c/JSQ$ denote a c core system with total arrival rate Λ and service rate $\mu = 1/\mathbb{E}[X]$ at each core. Let

$$\rho := \frac{\Lambda}{c\mu}.$$

We then define the following terms:

$$
\begin{aligned}
a(\rho) &= 1 - \frac{c\rho}{c+4} \\
b(\rho) &= \frac{c\rho}{(c+4)(c-1)} \\
\xi(\rho) &= \frac{\rho(1 - c\rho^{c-1} + (c-1)\rho^c)}{(1-\rho)(1-\rho^c)} \\
q(\rho) &= a(\rho) + b(\rho)\xi(\rho) \\
S(\rho) &= \frac{c(1-\rho)}{1-\rho^c}\{\rho^c + q(1-\rho^c)\} \\
\alpha_1 &= 0.0455, \alpha_2 = 0.7678, \gamma_1 = 0.0216, \gamma_2 = 0.0045 \\
r_c &= \gamma_1 \log_2(c) + \gamma_2 \\
i_c &= -1/\log_2\{\alpha_1 \log_2(c) + \alpha_2\} \\
R(\rho) &= \frac{1}{1 - 4r_c\rho^{i_c}(1-\rho^{i_c})} \\
A(\rho) &= \left[\sum_{n=0}^{c-1} (c\rho)^n/n!\right] + (c\rho)^c/(c!(1-\rho)) \\
P_c(\rho) &= (c\rho)^c/(c!(1-\rho)A(\rho)) \\
W_{M/M/c}(\rho) &= \frac{1}{\mu}\frac{P_c(\rho)}{c(1-\rho)}
\end{aligned}
$$

The term $W_{M/M/c}(\rho)$ is the mean waiting time in an $M/M/c$ system. The term $S(\rho)$ is an approximation of the mean length of the shortest queue in the $M/M/c/JSQ$ system. $R(\rho)$ is an experimentally derived error correction term. The mean response time in the $M/M/c/JSQ$ system given in [70] is:

$$\mathbb{E}[T]^{JSQ} \approx W_{M/M/c}(\rho)S(\rho)R(\rho) + \frac{1}{\mu}.$$

Appendix B

Scheduling With Job Sizes

B.1 Proof of Lemma B.1

Lemma B.1. *Let $\mathbf{\Theta}^P(t)$ be the allocation function which satisfies the following first-order conditions:*

$$\frac{\partial F^P}{\partial \omega_k^P} = 0 \qquad \forall 1 \le k \le M.$$

Then for any t, $\Theta_k^P(t)$ is increasing in k for $1 \le k \le m(t)$.

Proof. Following the same argument as Theorem 5.9, we can see that the expression for $\mathbf{\Theta}^P(t)$ is

$$\Theta_i^P(t) = \left(\frac{z(i)}{z(m(t))}\right)^{\frac{1}{1-p}} - \left(\frac{z(i-1)}{z(m(t))}\right)^{\frac{1}{1-p}} \qquad \forall 1 \le i \le m(t).$$

Note that $\frac{1}{1-p} > 1$, so $i^{\frac{1}{1-p}}$ is convex in i. We know that

$$z(i) - z(i-1) = w_i$$

and

$$z(i+1) - z(i) = w_{i+1}.$$

Furthermore, $w_{i+1} \ge w_i$. Hence, by convexity,

$$\begin{aligned} z(i)^{\frac{1}{1-p}} - z(i-1)^{\frac{1}{1-p}} &< (z(i) + w_i)^{\frac{1}{1-p}} - (z(i-1) + w_i)^{\frac{1}{1-p}} \\ &< (z(i) + w_{i+1})^{\frac{1}{1-p}} - z(i)^{\frac{1}{1-p}} \\ &< z(i+1)^{\frac{1}{1-p}} - z(i)^{\frac{1}{1-p}} \end{aligned}$$

This implies that $\Theta_i^P(t)$ is increasing in i for $1 \le i \le m(t)$. □

Appendix C

Scheduling With Multiple Speedup Functions

C.1 Idling Policies

We define a policy to be *idling* if it leaves one or more servers idle rather than allocating them to some eligible jobs.

Theorem C.1. *For any policy π which unnecessarily idles servers there exists a non-idling policy π' such that*

$$\mathbb{E}[T^{\pi'}] \leq \mathbb{E}[T^{\pi}].$$

Hence, there exists an optimal policy which is non-idling.

Proof. Consider any policy π which idles servers unnecessarily in one or more states. We will construct a new policy, π', which is identical to π in every state where π does not idle servers unnecessarily. In each state (i, j) where π *does* idle server unnecessarily, if $j > 0$, π' will allocate all of π's idle servers to the elastic job with the earliest arrival time. If $j = 0$, π' will instead allocate π's idle servers to each unserved (or underserved) inelastic job in FCFS order.

We now compare the performance of π to π' on any fixed arrival sequence of elastic and inelastic jobs. Suppose π first unnecessarily idles servers at time t, and suppose π gives jobs constant allocations on the time interval $(t, t + \delta]$. We reallocate the idle servers during this time interval in order to match the allocations π' would make. No job received fewer servers as a result of this transformation, and at least one job received additional servers during $(t, t + \delta]$. Each job which received additional servers during $(t, t + \delta]$ had its response time decreased, and no jobs had their response time increased. Furthermore, after this interchange, the schedule now reflects the allocation decisions that π' would make.

We now proceed to the next time, t', in the schedule where there are unnecessarily idle servers. Note, this idle space may exist because it is part of the policy π, or because an earlier interchange caused a job to complete earlier, creating some idle servers at time t'. In either case, we simply perform the same interchange as before, decreasing the response time of some jobs without increasing the response time of any jobs.

Note that each interchange causes the earliest occurrence of unnecessary idle servers in the schedule to occur at a later time. We therefore iterate this argument until all idle time either vanishes or occurs after the completion of the last job in the arrival sequence. At this point, the schedule reflects the actions taken by π', and hence the mean response time under π' is no larger than the mean response time under π. Hence, given any optimal idling policy π, we can construct a non-idling policy π' which also optimal. □

C.2 Lyapunov Stability of Work Conserving Policies

Theorem C.2. *For any work-conserving policy π, the associated Markov chain $\{(N_I^\pi(t), N_E^\pi(t)) \colon t \geq 0\}$ has a stationary distribution. If we define (N_I^π, N_E^π) to be a random element that follows this stationary distribution, then*

$$\lim_{t\to\infty}(N_I^\pi(t), N_E^\pi(t)) \stackrel{d}{=} (N_I^\pi, N_E^\pi). \tag{C.1}$$

Furthermore,

$$\lim_{t\to\infty}\mathbb{E}[N_I^\pi(t)] = \mathbb{E}[N_I^\pi], \tag{C.2}$$

and

$$\lim_{t\to\infty}\mathbb{E}[N_E^\pi(t)] = \mathbb{E}[N_E^\pi]. \tag{C.3}$$

Proof. To prove this claim, it suffices to show the drift results below, which allows us to apply the Foster-Lyapunov theorem [94] to show the convergence in distribution in (C.1) and apply the bounds in [34] to show the convergence of expectations in (C.2) and (C.3).

Consider the following Lyapunov function $V : \mathbb{Z}_{\geq 0}^2 \to \mathbb{R}_{\geq 0}$ for the Markov chain $\{(N_I^\pi(t), N_E^\pi(t)) \colon t \geq 0\}$:

$$V(i,j) = \frac{i}{n\mu_I} + \frac{j}{n\mu_E}.$$

Then its drift $\Delta V(i,j)$ can be written as

$$\Delta V(i,j) = \sum_{(i',j')} r_{(i,j)\to(i',j')}(V(i',j') - V(i,j)),$$

where $r_{(i,j)\to(i',j')}$ is the rate of transition from state (i,j) to state (i',j'). Note that for any (i,j) and (i',j'),

$$|V(i',j') - V(i,j)| < \frac{1}{n\min\{\mu_I,\mu_E\}}.$$

We now show that for the finite set, $F = \{(i,j) \colon i+j \leq n\}$, we have

$$\Delta V(i,j) \leq -\epsilon \qquad \forall (i,j) \notin F$$

for some $\epsilon > 0$. Let (i,j) be any state not in F, i.e., $i+j > n$. By definition,

$$\Delta V(i,j) = \frac{\lambda_I}{n\mu_I} + \frac{\lambda_E}{n\mu_E} - \left(\frac{\pi_I(i,j)\mu_I}{n\mu_I} + \frac{\pi_E(i,j)\mu_E}{n\mu_E}\right).$$

Because π is assumed to be a work conserving policy, and there are at least n jobs in system, we know that

$$\pi_I(i,j) + \pi_E(i,j) = n.$$

Furthermore, we have assumed that

$$\rho = \frac{\lambda_I}{n\mu_I} + \frac{\lambda_E}{n\mu_E} = 1 - \epsilon < 1$$

for some $\epsilon > 0$. Hence, we have that

$$\Delta V(i,j) = \rho - 1 = -\epsilon$$

as desired. We can therefore conclude that the Markov chain induced by π is positive recurrent and the convergence in distribution in (C.1) follows.

Note that for any $(i,j) \in F$, $V(i,j) \geq \frac{1}{\max\{\mu_I,\mu_E\}}$. Then extending Theorem 2.3 of [34] to continuous-time Markov chains using uniformization implies that

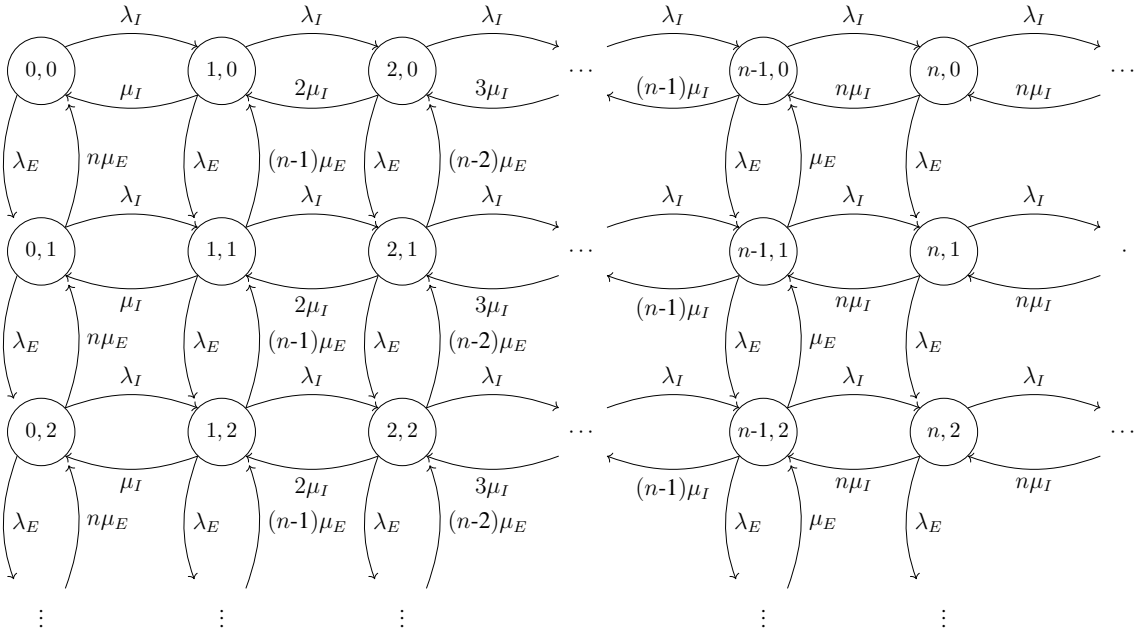
$$\sup_{t\geq 0} \mathbb{E}[(V(N^\pi(t)))^2] < \infty.$$

Therefore, $\{N_I^\pi(t), t \geq 0\}$ and $\{N_E^\pi(t), t \geq 0\}$ are uniformly integrable, which implies the convergence of expectations in (C.2) and (C.3). □

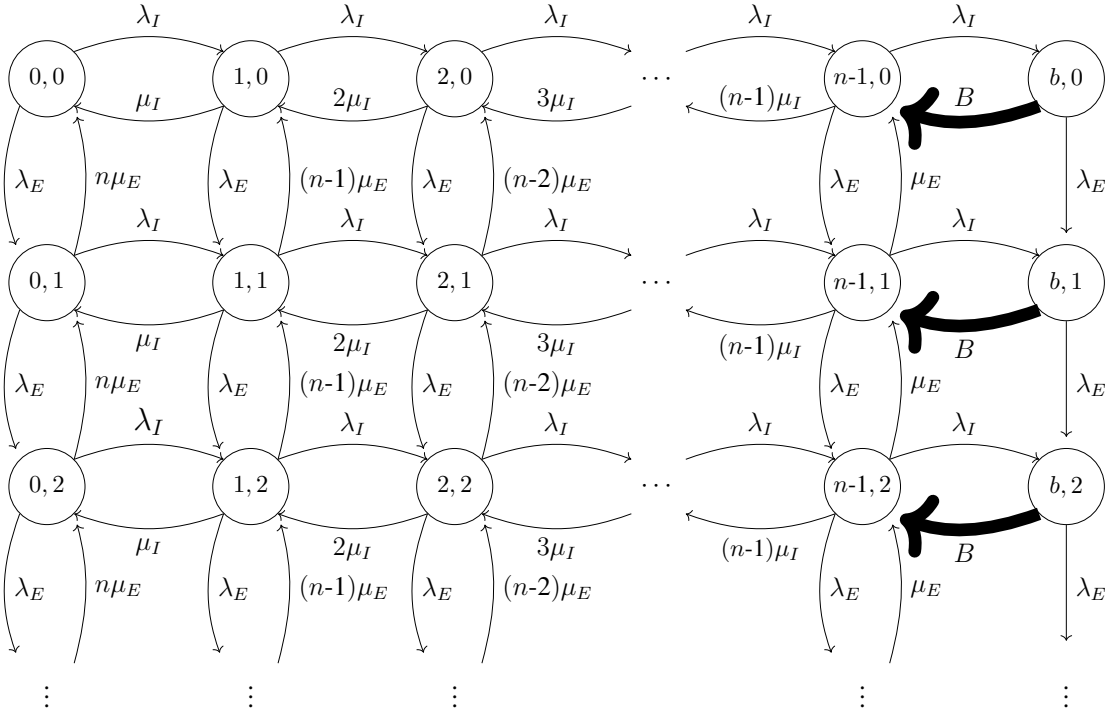
C.3 Markov Chains for IF

Figure C.1a shows the Markov chain for IF. To analyze this chain we will use a busy period transformation as described in Section 6.5.2.

We note that the inelastic jobs under IF see an $M/M/n$ queueing system, and hence their mean response time is known. We therefore only need to consider the mean response time of elastic jobs under IF. When there are more than n inelastic jobs in the system under IF, elastic jobs receive no service. The time from when there are first n inelastic jobs in the system until there are $n-1$ inelastic jobs is exactly an $M/M/1$ busy period. Hence, we perform the same busy period transformation described in Section 6.5.2 to the Markov chain for IF. This results in a 1D-infinite Markov chain which we can analyze using matrix analytic methods. We depict the busy period transformation for IF in Figure C.1

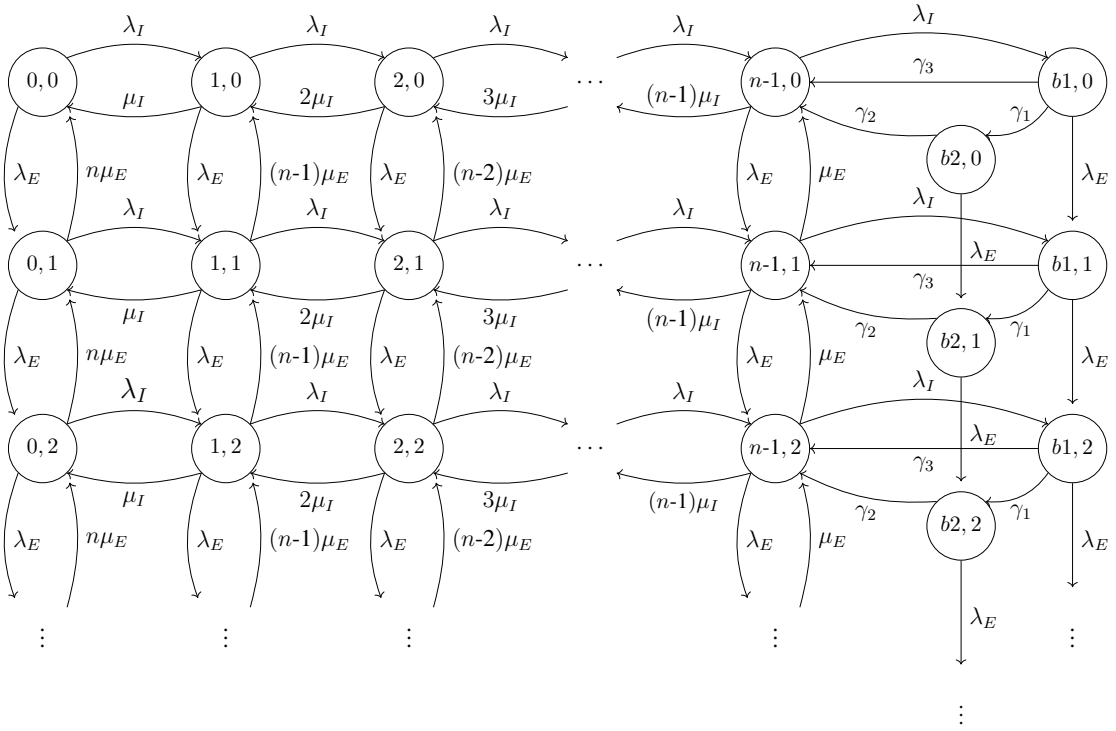


(a) Full IF chain



(b) IF chain with special states

Figure C.1: The transformation of the 2D-infinite IF chain to a 1D-infinite chain via the busy period transformation. Special states representing an $M/M/1$ busy period are shown in (b). (continued on next page)



(c) Final 1D `IF` chain

Figure C.1: (continued) The busy periods from (b) are approximated by a Coxian distribution in (c)

Bibliography

[1] S. Aalto, A. Penttinen, P. Lassila, and P. Osti. On the optimal trade-off between SRPT and opportunistic scheduling. In *SIGMETRICS*. ACM, 2011. 3.3

[2] Samuli Aalto, Urtzi Ayesta, and Rhonda Righter. On the Gittins index in the M/G/1 queue. *Queueing Systems*, 63(1):437–458, 2009. 7.6.3

[3] I. Adan, G. J. J. A. N. van Houtum, and J. van der Wal. Upper and lower bounds for the waiting time in the symmetric shortest queue system. *Annals of Operations Research*, 48: 197–217, 1994. 3.3, 6.3.3, 6.3.3

[4] Kunal Agrawal, Jing Li, Kefu Lu, and Benjamin Moseley. Scheduling parallel DAG jobs online to minimize average flow time. In *SIAM Symposium on Discrete Algorithms*, pages 176–189. SIAM, 2016. 3.2

[5] Eitan Bachmat and Hagit Sarfati. Analysis of size interval task assigment policies. *Performance Evaluation Review*, 36(2):107–109, 2008. 3.3

[6] Nikhil Bansal and Kirk Pruhs. Server scheduling in the Lp norm: a rising tide lifts all boat. In *STOC*, pages 242–250, 2003. 5.1.4

[7] F. Baskett, K. M. Chandy, R. Muntz, and F. G. Palacios. Open, closed, and mixed networks of queues with different classes of customers. *Journal of the ACM*, 22:248–260, 1975. 4.3.1

[8] B. Berg, J.P. Dorsman, and M. Harchol-Balter. Towards optimality in parallel scheduling. *ACM POMACS*, 1(2), 2018. 1.6

[9] B. Berg, M. Harchol-Balter, B. Moseley, W. Wang, and J. Whitehouse. Optimal resource allocation for elastic and inelastic jobs. In *Proceedings of the 32nd ACM Symposium on Parallelism in Algorithms and Architectures*, pages 75–87, 2020. 1.6

[10] B. Berg, R. Vesilo, and M. Harchol-Balter. heSRPT: Parallel scheduling to minimize mean slowdown. *Performance Evaluation*, 144:102–147, 2020. 1.6

[11] B. Berg, J. Whitehouse, B. Moseley, W. Wang, and M. Harchol-Balter. The case for phase-aware scheduling of parallelizable jobs. *Performance Evaluation*, 153:102246, 2022. 1.6

[12] Dimitris Bertsimas and José Niño-Mora. Conservation laws, extended polymatroids and multiarmed bandit problems; a polyhedral approach to indexable systems. *Mathematics of Operations Research*, 21(2):257–306, 1996. 5.1.3

[13] Guy E Blelloch, Phillip B Gibbons, and Yossi Matias. Provably efficient scheduling for languages with fine-grained parallelism. *Journal of the ACM (JACM)*, 46(2):281–321, 1999.

3.2

[14] Robert D Blumofe and Charles E Leiserson. Space-efficient scheduling of multithreaded computations. *SIAM Journal on Computing*, 27(1):202–229, 1998. 3.2

[15] Robert D Blumofe and Charles E Leiserson. Scheduling multithreaded computations by work stealing. *Journal of the ACM (JACM)*, 46(5):720–748, 1999. 3.2

[16] T. Bonald and A. Proutière. Insensitivity in processor-sharing networks. *Performance Evaluation*, 49:193–209, 2002. 4.3.1

[17] A. Bušić, I. Vliegen, and A. Scheller-Wolf. Comparing Markov chains: aggregation and precedence relations applied to sets of states, with applications to assemble-to-order systems. *Mathematics of Operations Research*, 37:259–287, 2012. 6.3.3, 6.3.3

[18] M. Chowdhury, Y. Zhong, and I. Stoica. Efficient coflow scheduling with varys. In *SIGCOMM*, volume 44, pages 443–454. ACM, 2014. 3.3

[19] Richard Cole. Parallel merge sort. *SIAM Journal on Computing*, 17(4):770–785, 1988. 1.1

[20] Richard W Conway, Louis W Miller, and William L Maxwell. *Theory of scheduling*. Dover, 2003. 3.3

[21] Dinesh Das, Jiaqi Yan, Mohamed Zait, Satyanarayana R Valluri, Nirav Vyas, Ramarajan Krishnamachari, Prashant Gaharwar, Jesse Kamp, and Niloy Mukherjee. Query optimization in oracle 12c database in-memory. *Proceedings of the VLDB Endowment*, 8(12):1770–1781, 2015. 5.1

[22] Jeffrey Dean and Sanjay Ghemawat. MapReduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008. 3.1, 6.4.1, 7.2, 7.8

[23] Christina Delimitrou and Christos Kozyrakis. Quasar: Resource-efficient and qos-aware cluster management. *ACM SIGPLAN Notices*, 49(4):127–144, 2014. 1.2, 3.1, 5.6, 6.1, 6.6, 7.1, 8.3

[24] J. Edmonds. Scheduling in the dark. *Theoretical Computer Science*, 1999. 3.2, 4.1.2, 4.3, 5.5.2, 7.6.1

[25] J. Edmonds and K. Pruhs. Scalably scheduling processes with arbitrary speedup curves. SODA '09, pages 685–692. ACM, 2009. 3.2, 3.2, 5.5.2, 7.6.1

[26] Jeff Edmonds, Donald D Chinn, Tim Brecht, and Xiaotie Deng. Non-clairvoyant multiprocessor scheduling of jobs with changing execution characteristics. *Journal of Scheduling*, 6 (3):231–250, 2003. 3.2, 7.6.1

[27] Jeff Edmonds, Sungjin Im, and Benjamin Moseley. Online scalable scheduling for the l_k-norms of flow time without conservation of work. In *Proceedings of the Twenty-Second Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 109–119. SIAM, 2011. 3.2, 3.2

[28] Isaac Grosof, Ziv Scully, and Mor Harchol-Balter. Srpt for multiserver systems. *Performance Evaluation*, 127:154–175, 2018. 3.3

[29] Isaac Grosof, Mor Harchol-Balter, and Alan Scheller-Wolf. Stability for two-class

multiserver-job systems. *arXiv preprint arXiv:2010.00631*, 2020. 3.3

[30] A. Gupta, B. Acun, O. Sarood, and L. Kalé. Towards realizing the potential of malleable jobs. In *HiPC*. IEEE, 2014. 6.4.1

[31] V. Gupta, M. Harchol-Balter, K. Sigman, and W. Whitt. Analysis of join-the-shortest-queue routing for web server farms. *Performance Evaluation*, 2007. 4.4.2, 7.7.3

[32] Varun Gupta, Karl Sigman, Mor Harchol-Balter, and Ward Whitt. Insensitivity for ps server farms with jsq routing. *ACM SIGMETRICS Performance Evaluation Review*, 35(2):24–26, 2007. 3.3

[33] Varun Gupta, Mor Harchol-Balter, JG Dai, and Bert Zwart. On the inapproximability of m/g/k: why two moments of job size distribution are not enough. *Queueing Systems*, 64(1): 5–48, 2010. 5.6

[34] Bruce Hajek. Hitting-time and occupation-time bounds implied by drift analysis with applications. *aap*, 14(3):502–525, 1982. C.2

[35] M. Harchol-Balter. *Performance Modeling and Design of Computer Systems: Queueing Theory in Action*. Cambridge University Press, 2013. 4.5.2, 6.4.4

[36] Mor Harchol-Balter. Task assignment with unknown duration. *Journal of the ACM*, 49(2): 260–288, March 2002. 3.3

[37] Mor Harchol-Balter. Open problems in queueing theory inspired by datacenter computing. *Queueing Systems*, 97(1):3–37, 2021. 5.6

[38] Mor Harchol-Balter. The multiserver job queueing model. *Queueing Systems*, 100(3):201–203, 2022. 3.3

[39] Mor Harchol-Balter, Cuihong Li, Takayuki Osogami, Alan Scheller-Wolf, and Mark Squillante. Task assignment with cycle stealing under central queue. In *Proceedings of the 23rd International Conference on Distributed Computing Systems*, pages 628–637, Providence, RI, May 2003. 6.5

[40] Mor Harchol-Balter, Cuihong Li, Takayuki Osogami, Alan Scheller-Wolf, and Mark Squillante. Cycle stealing under immediate dispatch task assignment. In *Proceedings of the 15th ACM Symposium on Parallel Algorithms and Architectures*, pages 274–285, San Diego, CA, June 2003. 6.5

[41] Mor Harchol-Balter, Takayuki Osogami, Alan Scheller-Wolf, and Adam Wierman. Multi-server queueing systems with multiple priority classes. *Queueing Systems: Theory and Applications*, 51(3–4):331–360, 2005. 6.5

[42] Mor Harchol-Balter, Alan Scheller-Wolf, and Andrew Young. Surprising results on task assignment in server farms with high-variability workloads. In *ACM Sigmetrics 2009 Conference on Measurement and Modeling of Computer Systems*, pages 287–298, 2009. 3.3

[43] Nikos Hardavellas, Ippokratis Pandis, Ryan Johnson, Naju Mancheril, Anastassia Ailamaki, and Babak Falsafi. Database servers on chip multiprocessors: Limitations and opportunities. In *Proceedings of the Biennial Conference on Innovative Data Systems Research*, 2007. 3.1

[44] Stavros Harizopoulos and Anastassia Ailamaki. A case for staged database systems. In *CIDR*, 2003. 7.2

[45] John L Hennessy and David A Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2011. 2

[46] M. D. Hill and M. R. Marty. Amdahl's law in the multicore era. *Computer*, 41:33–38, 2008. 4.2, 5.1.2, 5.2

[47] Yige Hong and Weina Wang. Sharp waiting-time bounds for multiserver jobs. *arXiv preprint arXiv:2109.05343*, 2021. 3.3

[48] Esa Hyytiä, Samuli Aalto, and Aleksi Penttinen. Minimizing slowdown in heterogeneous size-aware dispatching systems. *SIGMETRICS*, 40(1):29–40, 2012. 5.5.3

[49] Sungjin Im, Benjamin Moseley, Kirk Pruhs, and Eric Torng. Competitively scheduling tasks with intermediate parallelizability. *TOPC*, 3(1):4, 2016. 2, 3.2, 5.5.2

[50] David B Jackson, Heather L Jackson, and Quinn O Snell. Simulation based HPC workload analysis. In *IPDPS 2001*, pages 8–pp. IEEE, 2000. 5.2

[51] H. Jahanjou, E. Kantor, and R. Rajaraman. Asymptotically optimal approximation algorithms for coflow scheduling. In *SPAA*, pages 45–54. ACM, 2017. 3.3

[52] Kostis Kaffes, Timothy Chong, Jack Tigar Humphries, Adam Belay, David Mazières, and Christos Kozyrakis. Shinjuku: Preemptive scheduling for μsecond-scale tail latency. In *16th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 19)*, pages 345–360, 2019. 3.1

[53] Samir Khuller and Manish Purohit. Brief announcement: Improved approximation algorithms for scheduling co-flows. In *SPAA*, pages 239–240. ACM, 2016. 3.3

[54] Myungsun Kim, Kibeom Kim, James R Geraci, and Seongsoo Hong. Utilization-aware load balancing for the energy efficient operation of the big. little processor. In *2014 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1–4. IEEE, 2014. 8.4

[55] L. Kleinrock. *Queueing Systems, Volume II: Computer Applications*. 1976. 4.4.1, 6.5.2

[56] G. M. Koole. Monotonicity in Markov reward and decision chains: Theory and applications. *Foundations and Trends in Stochastic Systems*, 1:1–76, 2006. 6.3.3

[57] Guy Latouche and V. Ramaswami. *Introduction to Matrix Analytic Methods in Stochastic Modeling*. ASA-SIAM, Philadelphia, 1999. 6.5.3

[58] Viktor Leis, Peter Boncz, Alfons Kemper, and Thomas Neumann. Morsel-driven parallelism: A numa-aware query evaluation framework for the many-core age. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 743–754, 2014. 3.1, 5.1, 6.1, 7.7.3

[59] Charles E Leiserson, Neil C Thompson, Joel S Emer, Bradley C Kuszmaul, Butler W Lampson, Daniel Sanchez, and Tao B Schardl. There's plenty of room at the top: What will drive computer performance after moore's law? *Science*, 368(6495), 2020. 1

[60] Stefano Leonardi and Danny Raz. Approximating total flow time on parallel machines.

Journal of Computer and System Sciences, 73(6):875–891, 2007. 1.2, 3.2

[61] Xiangru Lian, Ce Zhang, Huan Zhang, Cho-Jui Hsieh, Wei Zhang, and Ji Liu. Can decentralized algorithms outperform centralized algorithms? a case study for decentralized parallel stochastic gradient descent. In *Advances in Neural Information Processing Systems*, pages 5330–5340, 2017. 6.4.1

[62] S. Lin, M. Paolieri, C. Chou, and L. Golubchik. A model-based approach to streamlining distributed training for asynchronous SGD. In *MASCOTS*. IEEE, 2018. 2, 3.1, 5.5.2, 7.1, 7.2

[63] Don Lipari. The slurm scheduler design. *SLURM User Group. http://slurm. schedmd. com/slurm_ug_2012/SUG-2012-Scheduling. pdf*, 2012. 3.1, 4.1

[64] S. A. Lippman. Semi-Markov decision processes with unbounded rewards. *Management Science*, 19:717–731, 1973. 6.3.4

[65] Milo MK Martin, Mark D Hill, and Daniel J Sorin. Why on-chip cache coherence is here to stay. *Communications of the ACM*, 55(7):78–89, 2012. 1.1

[66] J. McCool, M. Robison, and A. Reinders. *Structured Parallel Programming: Patterns for Efficient Computation*. Elsevier, 2012. 4.2

[67] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I Jordan, et al. Ray: A distributed framework for emerging AI applications. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 561–577, 2018. 1.2, 3.1, 5.1, 6.4.1

[68] S. Muthukrishnan, R. Rajaraman, A. Shaheen, and J. Gehrke. Online scheduling to minimize average stretch. In *FOCS*. IEEE, 1999. 5.2

[69] Deepak Narayanan, Fiodar Kazhamiaka, Firas Abuzaid, Peter Kraft, Akshay Agrawal, Srikanth Kandula, Stephen Boyd, and Matei Zaharia. Solving large-scale granular resource allocation problems efficiently with pop. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, pages 521–537, 2021. 3.1

[70] R. Nelson and T. Philips. An approximation for the mean response time for shortest queue routing with general interarrival and service times. *Performance Evaluation*, 1993. 4.4.2, 4.4.2, A.2

[71] Randolph Nelson and Asser N Tantawi. Approximate analysis of fork/join synchronization in parallel queues. *IEEE transactions on computers*, 37(6):739–743, 1988. 3.3

[72] Marcel F. Neuts. *Matrix-Geometric Solutions in Stochastic Models*. Johns Hopkins University Press, 1981. 6.5.3

[73] Marcel F. Neuts. *Structured Stochastic Matrices of M/G/1 Type and Their Applications*. Marcel Dekker, 1989. 6.5.3

[74] Thu D Nguyen, Raj Vaswani, and John Zahorjan. Using runtime measured workload characteristics in parallel processor scheduling. In *Workshop on Job Scheduling Strategies for Parallel Processing*, pages 155–174. Springer, 1996. 7.1

[75] NoisePage. Noisepage. 2021. `https://noise.page`. (document), 5.1, 7.1, 7.1, 7.7.3

[76] Patrick O'Neil, Elizabeth O'Neil, Xuedong Chen, and Stephen Revilak. *The Star Schema Benchmark and Augmented Fact Table Indexing*, pages 237—-252. 2009. 1.5, 6.1, 7.1, 7.1.2

[77] Takayuki Osogami and Mor Harchol-Balter. Closed form solutions for mapping general distributions to quasi-minimal PH distributions. *Performance Evaluation*, 63(6):524–552, 2006. 6.5, 6.5.2

[78] Takayuki Osogami, Mor Harchol-Balter, and Alan Scheller-Wolf. Analysis of cycle stealing with switching times and thresholds. In *Proceedings of ACM Sigmetrics*, pages 184–195, San Diego, CA, June 2003. 6.5

[79] Patrick O'Neil, Elizabeth O'Neil, Xuedong Chen, and Stephen Revilak. The star schema benchmark and augmented fact table indexing. In *Technology Conference on Performance Evaluation and Benchmarking*, pages 237–252. Springer, 2009. 2

[80] Perf. perf: Linux profiling with performance counters. 2021. `https://perf.wiki.kernel.org/`. 7.7.3

[81] M. L. Puterman. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. John Wiley & Sons, Chichester, 1994. 6.3.3, 6.3.4

[82] Aurick Qiao, Sang Keun Choe, Suhas Jayaram Subramanya, Willie Neiswanger, Qirong Ho, Hao Zhang, Gregory R Ganger, and Eric P Xing. Pollux: Co-adaptive cluster scheduling for goodput-optimized deep learning. In *15th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 21)*, 2021. 1.2, 3.1, 5.1, 6.6, 8.3

[83] Zhen Qiu, Cliff Stein, and Yuan Zhong. Minimizing the total weighted completion time of coflows in datacenter networks. In *SPAA*, pages 294–303. ACM, 2015. 3.3

[84] Runtian Ren and Xueyan Tang. Clairvoyant dynamic bin packing for job scheduling with minimum server usage time. In *SPAA*, pages 227–237. ACM, 2016. 3.1

[85] X. Ren, G. Ananthanarayanan, A. Wierman, and M. Yu. Hopper: Decentralized speculation-aware cluster scheduling at scale. *ACM SIGCOMM Computer Communication Review*, 45 (4):379–392, 2015. 3.1, 7.7.3

[86] Dennis M Ritchie and Ken Thompson. The unix time-sharing system. *Bell System Technical Journal*, 57(6):1905–1929, 1978. 1

[87] Gerald Sabin, Matthew Lang, and P Sadayappan. Moldable parallel job scheduling using job efficiency: An iterative approach. In *Workshop on Job Scheduling Strategies for Parallel Processing*, pages 94–114. Springer, 2006. 3.1

[88] Bilal Sadiq and Gustavo De Veciana. Balancing SRPT prioritization vs opportunistic gain in wireless systems with flow dynamics. In *International Teletraffic Congress*. IEEE, 2010. 5.1.3

[89] Z. Scully, G. Blelloch, M. Harchol-Balter, and A. Scheller-Wolf. Optimally scheduling jobs with multiple tasks. In *Proceedings of the ACM Workshop on Mathematical Performance Modeling and Analysis*, 2017. 4.3.3

[90] Ziv Scully, Isaac Grosof, and Mor Harchol-Balter. Optimal multiserver scheduling with unknown job sizes in heavy traffic. *Performance Evaluation*, 145:102150, 2021. 5.6

[91] Mehrnoosh Shafiee and Javad Ghaderi. Brief announcement: a new improved bound for coflow scheduling. In *SPAA*, pages 91–93. ACM, 2017. 3.3

[92] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. In *2010 IEEE 26th symposium on mass storage systems and technologies (MSST)*, pages 1–10, 2010. 7.1

[93] Donald R Smith. A new proof of the optimality of the shortest remaining processing time discipline. *Operations Research*, 26(1):197–199, 1978. 5.1.1, 7.7.1

[94] R. Srikant and Lei Ying. *Communication Networks: An Optimization, Control and Stochastic Networks Perspective*. Cambridge Univ. Press, New York, 2014. C.2

[95] S. Srinivasan, S. Krishnamoorthy, and P. Sadayappan. A robust scheduling strategy for moldable scheduling of parallel jobs. In *Proceedings of the IEEE International Conference on Cluster Computing*, CLUSTER '03, pages 92–99, 2003. 3.1

[96] Nathan R Tallent and John M Mellor-Crummey. Effective performance measurement and analysis of multithreaded applications. In *Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 229–240, 2009. 7.1

[97] Thomas N Theis and H-S Philip Wong. The end of moore's law: A new beginning for information technology. *Computing in Science & Engineering*, 19(2):41–50, 2017. 1

[98] Muhammad Tirmazi, Adam Barker, Nan Deng, Md E Haque, Zhijing Gene Qin, Steven Hand, Mor Harchol-Balter, and John Wilkes. Borg: the next generation. In *Proceedings of the Fifteenth European Conference on Computer Systems*, pages 1–14, 2020. 3.1, 4.1

[99] Alexey Tumanov, Timothy Zhu, Jun Woo Park, Michael A Kozuch, Mor Harchol-Balter, and Gregory R Ganger. Tetrisched: global rescheduling with adaptive plan-ahead in dynamic heterogeneous clusters. In *Proceedings of the Eleventh European Conference on Computer Systems*, pages 1–16, 2016. 3.1

[100] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes. Large-scale cluster management at Google with Borg. In *EUROSYS*. ACM, 2015. 3.1, 4.1, 6.1, 8.4

[101] Rares Vernica, Michael J Carey, and Chen Li. Efficient parallel set-similarity joins using mapreduce. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 495–506, 2010. 3.1

[102] Benjamin Wagner, André Kohn, and Thomas Neumann. Self-tuning query scheduling for analytical workloads. In *Proceedings of the 2021 International Conference on Management of Data*, pages 1879–1891, 2021. 1.2, 3.1, 8.3

[103] Weina Wang, Mor Harchol-Balter, Haotian Jiang, Alan Scheller-Wolf, and Rayadurgam Srikant. Delay asymptotics and bounds for multitask parallel jobs. *Queueing Systems*, 91 (3):207–239, 2019. 3.3

[104] Adam Wierman, Mor Harchol-Balter, and Takayuki Osogami. Nearly insensitive bounds on SMART scheduling. *SIGMETRICS*, 33(1):205–216, 2005. 5.5.2, 5.5.3

[105] Wencong Xiao, Romil Bhardwaj, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, Zhenhua Han, Pratyush Patel, Xuan Peng, Hanyu Zhao, Quanlu Zhang, et al. Gandiva: Introspective cluster scheduling for deep learning. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, pages 595–610, 2018. 3.1

[106] Y. Xu, A. Scheller-Wolf, and K. P. Sycara. The benefit of introducing variability in single-server queues with application to quality-based service domains. *Operations Research*, 63: 233–246, 2015. 6.3.6

[107] David D Yao. Dynamic scheduling via polymatroid optimization. In *IFIP International Symposium on Computer Performance Modeling, Measurement and Evaluation*, pages 89–113. Springer, 2002. 5.1.3

[108] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, pages 15–28, 2012. 7.1, 7.2

[109] X. Zhan, Y. Bao, C. Bienia, and K. Li. PARSEC3.0: A multicore benchmark suite with network stacks and SPLASH-2X. *SIGARCH*, 44:1–16, 2017. (document), 2, 4.2, 5.2, 5.2

[110] Jingren Zhou, John Cieslewicz, Kenneth A Ross, and Mihir Shah. Improving database performance on simultaneous multithreading processors. In *Proceedings of the 31st Very Large Data Bases Conference (VLDB)*, 2005. 7.1

[111] Hang Zhu, Kostis Kaffes, Zixu Chen, Zhenming Liu, Christos Kozyrakis, Ion Stoica, and Xin Jin. Racksched: A microsecond-scale scheduler for rack-scale computers. In *14th USENIX Symposium on Operating Systems Design and Implementation ({OSDI} 20)*, pages 1225–1240, 2020. 3.1