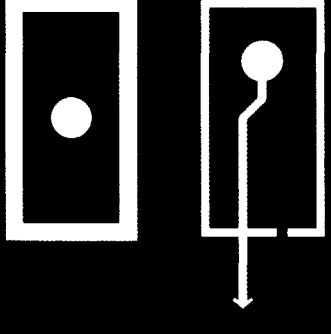
Share Scheduling in Distributed Systems



Jan de Jongh

Share Scheduling in Distributed Systems

Propositions Supplement to the Ph.D. Thesis of Jan de Jongh

1. The objectives of share scheduling must take into account the actual number of jobs present in the system (*this thesis*).
2. In general, both the global and the local scheduling policies affect the performance of share scheduling in distributed systems (*this thesis*).
3. The availability of a job-migration mechanism substantially improves the achievable performance of share scheduling in distributed systems (*this thesis*).
4. Static scheduling policies are not sufficient for achieving share-scheduling objectives in distributed systems (*this thesis*).
5. In the war on terror, detecting communications is at least as much of a challenge as deciphering them.
6. The use of virtual functions in object-oriented software designs does not agree very well with the principle of 'information hiding'.
7. The advent of Power-Line Communications (the use of power lines for data communications) poses a serious threat to many applications that use the electromagnetic spectrum.
8. A government advocating free markets should allow more competition for itself as well.
9. Massive parallelism within a software application is no guarantee for speed, massive use of a software application is no guarantee for quality.
10. The typical maximum zapping speed of contemporary television sets is not sufficient for the current large number of TV channels, and certainly not for their average quality.

Share Scheduling in Distributed Systems

Proefschrift

ter verkrijging van de graad van doctor
aan de Technische Universiteit Delft,
op gezag van de Rector Magnificus prof.dr.ir. J.T. Fokkema,
voorzitter van het College voor Promoties,
in het openbaar te verdedigen op maandag 18 februari 2002 om 13:30 uur

door

Johannes Frederik Cornelis Maria DE JONGH,

elektrotechnisch ingenieur,

geboren te Raamsdonk.

Dit proefschrift is goedgekeurd door de promotor:
Prof.dr.ir. H.J. Sips

Toegevoegd promotor:
Dr.ir. D.H.J. Epema

Samenstelling promotiecommissie:

Rector Magnificus,	voorzitter
Prof.dr.ir. H.J. Sips,	Technische Universiteit Delft, promotor
Dr.ir. D.H.J. Epema,	Technische Universiteit Delft, toegevoegd promotor
Prof.dr.ir. O.J. Boxma,	Technische Universiteit Eindhoven
Prof. M. Livny Ph.D.,	University of Winsconsin, Madison WI, USA
Prof.dr.ir. P. Van Mieghem,	Technische Universiteit Delft
Prof.dr. P.M.A. Sloot,	Universiteit van Amsterdam
Dr.ir. A.J.C. van Gemund,	Technische Universiteit Delft

Front cover: The front cover shows the Static Vertical Partitioning policy (defined in Sections 5.5.3 and 6.5.4) at work in a homogeneous ten-processor system with four groups, viz., red, green, yellow, and blue. The rectangles represent the processors. Each processor is allocated to a group, as indicated by its color. The circles represent jobs. Upon departure of the yellow job on the seventh processor, the policy probes three processors (indicated by means of the thicker rectangles), and migrates the green job from the tenth to the seventh processor. This decision is taken by virtue of rule M1 of SVP's migration policy.

Share Scheduling in Distributed Systems / J.F.C.M. de Jongh
Proefschrift Technische Universiteit Delft - Met lit. opg.
Met samenvatting in het Nederlands
ISBN 90-9015128-1
NUGI 811
Keywords: distributed systems, scheduling, queueing theory, simulation

Correspondence: JanDeJongh@ACM.Org

Printed by Ponsen en Looijen B.V., Wageningen, The Netherlands

Voor John

Rostvast verankerd
blijf je in onze gedachten

Contents

Chapter 1

Introduction

A distributed system is a collection of cooperating computers. In the past two decades, the use of distributed systems has increased dramatically. Such systems have several advantages over uniprocessors, such as improved performance and increased fault tolerance. Nowadays, it is feasible to build computer systems with enormous processing capacities by interconnecting many small computers. An example of such a high-performance system is the Distributed ASCI (Advanced School for Computing and Imaging) Supercomputer [57], a set of four clusters of workstations at Dutch universities interconnected by Asynchronous Transfer Mode (ATM) links. Many compute-intensive applications, such as those found in the areas of weather forecasting, VLSI design, and other numerical computations, can benefit from the capacity of such superclusters. Therefore, distributed systems have become an attractive alternative for expensive supercomputers.

The aim of *share scheduling*, which is the subject of this thesis, is to provide prespecified shares of the total system capacity to groups of jobs. Such scheduling is desired, for instance, in systems where users pay for their shares of the system capacity. Numerous other objectives and policies for scheduling in distributed systems have been studied. For instance, much effort has been put into the design of policies that distribute the load among the processors in order to minimize the job response time. Share scheduling in uniprocessors and in networks has recently gained attention. However, relatively little is known on share scheduling in distributed systems. In this thesis, we formally state the objectives of share scheduling, and we propose and evaluate policies for share scheduling in distributed systems.

We start this introductory chapter with the system and workload models used throughout this thesis. Subsequently, we present a survey on scheduling in distributed systems. Then we introduce and motivate share scheduling, and present an overview of related work. Finally, we give an outline of the remainder of the thesis.

1.1 System Models

Our system model consists of P processors, numbered $1, \ldots, P$. Processor p has capacity c_p, with $c_p > 0$, $p = 1, \ldots, P$. The capacity of a processor is defined as its speed relative to a reference processor with unit capacity. The capacity lost due to context switching and to other operating-system overhead (for instance, due to accounting) can be compensated

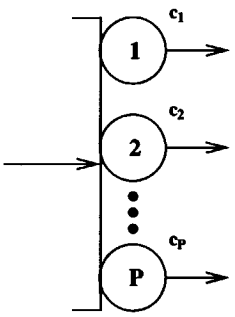


Figure 1.1: The Multiprocessor Model.

for in c_p. We assume that

$$c_1 \geq \cdots \geq c_P. \tag{1.1}$$

The *total capacity* c of the system is defined as

$$c \triangleq \sum_{p=1}^{P} c_p. \tag{1.2}$$

A system is called *homogeneous* when $c_1 = \cdots = c_P$.

We present three versions of the system model, representing a broad range of computer architectures: uniprocessors, multiprocessors, and distributed systems.

1.1.1 The Uniprocessor Model

A uniprocessor consists of a single processor ($P = 1$). In a uniprocessor, the decision when to serve which job is made by a *local* scheduling policy. Examples of such policies are First-Come First-Served (FCFS) and Processor Sharing (PS). Local scheduling policies have been the subject of intensive research in the past decades [1, 4, 14, 16, 35, 51, 55, 86, 87] and will be studied in Chapter 3 of this thesis. In practical implementations, the processor cannot serve multiple jobs simultaneously, unlike with policies such as Processor Sharing. Because the latter policies can be approximated with round-robin policies [14], we allow simultaneous service to multiple jobs. This means that the local scheduling policy must also decide at what rate the processor serves these jobs, if serving more than one.

1.1.2 The Multiprocessor Model

In the multiprocessor system model as shown in Figure 1.1, P processors share a single job queue, $P > 1$. All processors have fast access to this queue. We assume that in a multiprocessor, jobs can be transferred from one processor to another without cost. A single scheduling policy decides when each of the processors serves which job. We will introduce such a policy in Chapter 2, when stating the objectives of share scheduling.

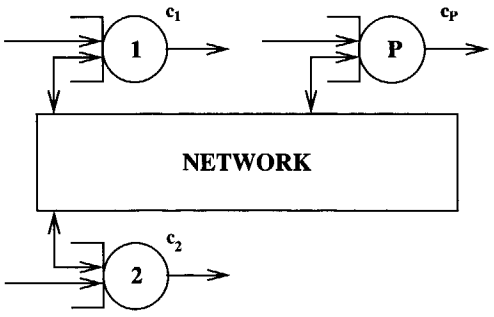


Figure 1.2: The Distributed-System Model.

1.1.3 The Distributed-System Model

The distributed-system model, as shown in Figure 1.2, is modeled after Wang and Morris [84]. The system consists of P machines connected by a network, $P > 1$. Each machine is equipped with a single processor. In other words, we do not consider interconnections of multiprocessors. We assume that the graph describing the network is connected, i.e., every machine can communicate with every other machine in the system, and that the processors must contend for the network on an equal basis. Apart from that, we will not be concerned with the internal structure of the network.

The main difference with multiprocessor systems is that in a distributed system, information about the system state is spread across the different processors. Since the costs of communication between processors are higher than those in a multiprocessor system, it is difficult to have a consistent view of the system state. Also, in many cases, migrating a job from one processor to another is very costly in terms of network bandwidth and service delay. In other words, the queue access time is much larger for a remote processor than for a local processor (i.e., the processor holding the queue in its memory). In order to obtain an indication of the potential benefits of job migration, we will study systems with and without job migration.

Unlike in uniprocessors and in multiprocessors, scheduling in distributed systems has two components. The *global* scheduling policy decides to which processor an arriving job must be sent, and when to migrate which jobs. At each processor, the *local* scheduling policy decides when the processor serves which of the jobs present in its queue. Clearly, scheduling in distributed systems is much more complicated than in uniprocessors and in multiprocessors.

1.2 Workload Models

An important aspect of performance evaluation of computer systems is a realistic model of the workload. In our model, the workload consists of *jobs*, each of which belongs to one of G groups, numbered $1, \ldots, G$. A job corresponds to an amount of work to be performed, for instance, the execution of a program. An essential restriction on a job is that it can only be served by a single processor at a time.

In more elaborate workload models, jobs consist of *tasks* with precedence constraints. These constraints dictate that the execution of a task cannot start before one or more other tasks have finished. Two tasks for which all precedence constraints are met can run simultaneously on different processors. These models are important with the advent of parallel programming. However, due to their complexity, such models will not be considered in this thesis, despite their usefulness.

We consider two types of workload, viz. *non-permanent* and *permanent* jobs, explained below. The workload type of a job is unknown to the scheduling policies. For both types of workload, the service times of jobs of group g are independent, identically distributed random variables with probability distribution function $F_g(x)$ and expectation $1/\mu_g$. We assume jobs are compute-bound, i.e., they are always ready for execution.

1.2.1 Non-Permanent Jobs

Non-permanent jobs are jobs that arrive, are executed, and leave the system. In our model, non-permanent jobs of group g arrive according to a Poisson process with arrival rate λ_g. According to the terminology of queueing networks [3], the non-permanent jobs of a group form an *open job class*. Models with non-permanent jobs are used extensively for performance evaluation of computer systems. When a group of jobs consists of the jobs submitted by many users, the use of a Poisson arrival process is justified by the Palm-Khintchine theorem [39]. This theorem states that the superposition of many independent renewal processes forms, in the limit, a Poisson process. However, for systems with a small number of users, the use of Poisson arrival processes can no longer be justified. For such systems, models with permanent jobs are probably more realistic.

1.2.2 Permanent Jobs

Permanent jobs are jobs that are replaced immediately upon departure by a new arrival to the system. In other words, their number remains constant. The permanent jobs of a group form a *closed job class*, according to the terminology in [3]. Permanent jobs account for the fact that users wait for the completion of their jobs before submitting a new one.

1.3 Overview of Scheduling in Distributed Systems

An important task in a distributed system is the management of a variety of resources, such as memory, I/O devices, files, and processors. This thesis is concerned with *processor scheduling*, or simply *scheduling*: the allocation of processor time to jobs. We consider multi-user distributed systems that are primarily used for compute-intensive applications with large service-time requirements. It goes without saying that proper scheduling of their most-wanted resources—the processors—is essential for such applications.

Before introducing and motivating share scheduling in Section 1.4, we present an overview of previous work on scheduling in distributed systems. First, we introduce some frequently used terminology in Section 1.3.1, and we propose a simple taxonomy in Section 1.3.2. The taxonomy holds equally well for share-scheduling policies. Using this vocabulary for scheduling, we proceed with an overview of the more classical objectives and performance measures in Section 1.3.3. We introduce some well-known scheduling

mechanisms in Section 1.3.4. Finally, we provide a general overview of the literature on scheduling policies (not restricted to share scheduling) in Sections 1.3.5 and 1.3.6.

1.3.1 Terminology

There exists a large body of terminology concerning scheduling in distributed systems. This terminology is not always used in a consistent way in the literature. Instead of pointing out all possible inconsistencies, we will define a terminology that is sufficient for the purposes of this thesis.

Kendall's Notation

We follow Kendall's description of queueing systems [50] as X|Y|Z, where X refers to the interarrival time distribution, Y to the service-time distribution, and Z denotes the number of processors. For instance, the D|D|1 queue has deterministic interarrival and service times, while in the M|M|1 queue, these times have exponential distributions. Both systems are uniprocessors.

Global and Local Scheduling

In distributed systems, we distinguish between *global* and *local* scheduling. Global scheduling decides which processor serves which jobs. In contrast, local scheduling decides when and at what rate to serve jobs on a single processor. The global and local scheduling decisions are taken according to the global and local scheduling *policies*. Because local scheduling policies were already in use in early uniprocessors, they have been studied extensively, for instance with queueing models (e.g., [14, 51]). A classification of local scheduling policies is postponed until Chapter 3, in which we also study their performance with respect to share scheduling.

Static and Dynamic Global Policies

We distinguish between *static* and *dynamic* global policies [22, 24]. Static global policies only use *a priori* information for scheduling decisions, whereas dynamic policies use information on the state of the system that can only be obtained at runtime. The main advantage of static policies is their simplicity, since they do not need to gather and process system-state information, as dynamic policies do. On the other hand, dynamic policies have the ability to adapt to changes in the workload.

The distinction between static and dynamic global policies was further refined by Wang and Morris [84], by introducing multiple levels of information usage. Although we do not adopt their classification in this thesis, we will sometimes distinguish dynamic policies based on the type and amount of information they require.

Static policies sometimes require detailed *a priori* information on the workload. For instance, some models with stochastic workloads and static global policies, such as those studied by de Souza e Silva and Gerla [21, 22], assume that the traffic intensities are known beforehand. Jobs are routed probabilistically to processors, and the objective is usually to determine the routing probabilities such that, for instance, the average job response time is minimized. The models are often used for capacity management. For

instance, with these models one can identify bottlenecks in a distributed system in steady state and study the effects of removing them. However, a change in the (assumed) traffic intensities cannot be reacted upon by the policy and the resulting performance will no longer be optimal.

In other models with static policies, the complete set of jobs is known beforehand, as well as the arrival time and service time of every job. These models are often referred to as *job-shop scheduling*. In these models, the objective is to schedule the jobs in space and time such that for instance the *makespan* is minimized. The makespan is the elapsed time until all jobs in the job set have finished execution. Some problems in deterministic scheduling can be solved in polynomial time, but many others are NP-hard [58]. The models play an important role in, for instance, manufacturing and parallel computing [44, 58].

For most general-purpose distributed systems, the assumption that the workload can be described in full detail in advance, is quite unrealistic. Therefore, in this thesis we will concentrate on dynamic global policies, although some simple static policies are studied for comparison.

Deterministic and Probabilistic Global Policies

Another distinction is that between *deterministic* and *probabilistic* global policies [24, 25, 84]. A deterministic policy does not use any random element when taking scheduling decisions. In other words, a deterministic policy will always take the same decision in cases where the workload and the system state (the latter only when the policy is dynamic) are identical. On the other hand, a probabilistic global policy uses one or more random elements, and so, scheduling decisions cannot always be predicted. The static policies in which jobs are routed probabilistically to processors, briefly mentioned in the previous section, are good examples of probabilistic global policies. Dynamic policies are often probabilistic in order to break ties. For example, the Join Shortest Queue (JSQ) policy assigns an arriving job to the processor with the smallest number of jobs in its queue. If there is more than one such processor, one of them can be chosen at random, and the resulting policy is probabilistic. If one would choose to pick the lowest-numbered processor, the resulting policy would be deterministic.

Central and Distributed Global Policies

We distinguish between *central* and *distributed* global policies [11]. Central policies run on a single processor, whereas in distributed policies, the scheduling authority is distributed among processors (usually in a symmetric way). A possible drawback of central policies is the fact that the central processor can become a performance bottleneck, especially in large systems or when complicated policies are involved. On the other hand, with a central policy it is usually easier to maintain complete up-to-date information on the system state because all scheduling decisions are taken at a single point. This can significantly improve the quality of the decisions as compared to distributed policies. For the latter, information gathering requires finding a balance between the amount of network traffic and the accuracy of the information on the system state. If all processors inform each other about scheduling events, the network may become congested. On the other hand, if

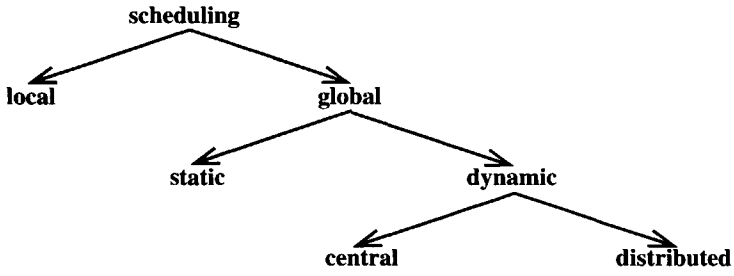


Figure 1.3: Our taxonomy of scheduling.

they don't, scheduling decisions will be based on incomplete information about the system state.

Source-Initiative and Server-Initiative Distributed Global Policies

Distributed policies can be further classified as *source initiative* or *server initiative* [84]. In the former, the processor on which a job arrives takes the initiative to find a suitable processor to serve the job. In the latter, processors search for suitable jobs to serve. Source-initiative and server-initiative policies are also named sender-initiated and receiver-initiated policies, respectively [25].

Non-Preemptive and Preemptive Global Policies

Finally, we distinguish between *non-preemptive* and *preemptive* global policies. In non-preemptive global policies, once a job is taken into service on a processor, it remains there until completion. In preemptive policies, a job in service can be preempted and moved to another processor, where service continues at the point where the job was interrupted.

1.3.2 A Scheduling Taxonomy

Various taxonomies of scheduling policies in distributed systems have been devised, for instance by Wang and Morris [84], Casavant and Kuhl [10], and Jacqmot et al. [42]. The taxonomy used in this thesis is outlined in Figure 1.3. It is modeled after Casey [11], who proposed a simple yet elegant hierarchical taxonomy. As in Section 1.3.1, we distinguish between local and global scheduling and between global static and global dynamic scheduling. However, our definition of *static* scheduling is different from Casey's, who defines it as 'the *a priori* assignment of tasks to processing elements'. In other words, the difference between static and dynamic scheduling in [11] is determined by the *time* of the assignment. Our definition focuses on the *nature* of the information used for scheduling decisions, and conforms to more recent work, for instance [22]. Our distinction between central and distributed scheduling is equal to Casey's *centralized* and *decentralized* scheduling.

1.3.3 Objectives and Performance Measures

The design of global and local scheduling policies may aim at various objectives. In order to measure, predict, and compare the effectiveness of scheduling policies with respect to these objectives, suitable performance measures must be chosen. A common objective is the minimization of the average job response time. A heuristic approach to this is to distribute the load among the processors. The need for load distribution is justified by, amongst others, the research of Livny and Melman [61], and Rommel [73], who showed that in a distributed system without load distribution, there is a rather high probability that processors are idle while there are jobs waiting for service at other processors. Two classes of load-distribution policies exist [34]: *load-sharing* policies aim at keeping as many processors busy as possible, whereas *load-balancing* policies aim, in addition, at equalizing the queue lengths at all processors.

In order to evaluate the usefulness of load balancing in homogeneous distributed systems, Rommel [73] introduced the *Probability of Load-Balancing Success (PLBS).* The PLBS $\mathbf{P}_{\text{lbs}}(n, m)$ is defined as the probability that at least one processor is underloaded (having n or fewer jobs), while at least one processor is overloaded (having m or more jobs, $m > n$). A low value of $\mathbf{P}_{\text{lbs}}(n, m)$ indicates that the workload is well-spread across the processors. The PLBS can be used to compare the performances of different load-balancing and load-sharing strategies. For load-sharing policies, $\mathbf{P}_{\text{lbs}}(0, 2)$ should be used. Unfortunately, this definition of the PLBS is less meaningful in systems with processors of different speeds.

One of the main problems with the use of the overall mean reponse time as a performance measure is that it depends on the distribution of the job arrivals among the processors. Clearly, a load-balancing policy will have fewer problems when the workload arrives in a perfectly balanced fashion among the processors than when it does not so. Wang and Morris [84] introduced a single performance measure addressing this problem. They considered a system consisting of K equal *server* processors and N *source* processors, connected by means of a communication network, as shown in Figure 1.4. The distinction between source and server processors is one from a logical viewpoint: physical processors can be both a source and a server processor at the same time. Apart from this distinction, our model of distributed systems outlined in Section 1.1.3 is identical. Jobs arrive at the source processors according to Poisson arrival processes with rates $\lambda_1, \ldots, \lambda_N$. The job service times are taken from a single random variable with mean $1/\mu$. The *utilization* ρ is defined as

$$\rho \triangleq \frac{\lambda_1 + \cdots + \lambda_N}{K\mu}.$$

For the performance evaluation of different global scheduling policies, they define the *Q-factor* $Q_A(\rho)$ of scheduling policy A at utilization ρ as

$$Q_A(\rho) \triangleq \frac{R_{FCFS}}{\sup_{\frac{1}{K\mu}\sum_{i=1}^{N}\lambda_i=\rho}\{\max_i\{R_{i,A}\}\}} \tag{1.3}$$

where R_{FCFS} is the mean response time over all jobs under the global *First-Come First-Served* scheduling policy (if it were possible to implement it), and $R_{i,A}$ is the mean response time of jobs originating from source i under policy A. The denominator in

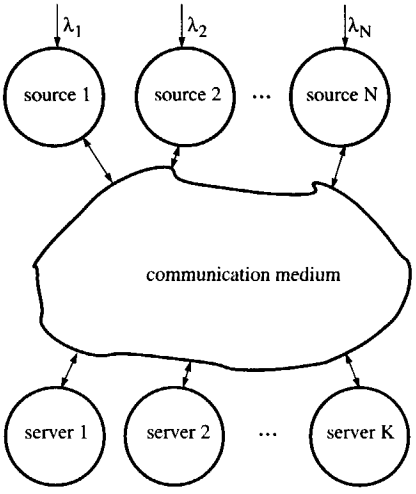


Figure 1.4: System model after Wang and Morris [84].

(1.3) is the supremum of the maximum response times among the sources, taken over all possibilities of the N arrival processes resulting in a fixed utilization ρ. Clearly, the definition of $Q_A(\rho)$ captures the ability of policy A to perform well in all cases where the workload is highly unbalanced among the sources.

Krueger and Livny [56] studied several combinations of global and local policies. They addressed a number of performance measures, viz., the average and the standard deviation of the *wait time*, which is the total time a job spends waiting for service, and the average and the standard deviation of the *wait time ratio*, which is the wait time per unit of service. With this set of performance measures, they evaluated the suitability of the scheduling policies for different objectives, including fairness. As with the Q-factor defined by Wang and Morris, these performance measures are all in some way related to the expected job response time.

For compute-intensive workloads, a natural and meaningful objective is to provide prespecified fractions of the total processing capacity to jobs or groups of jobs [26, 64]. Scheduling policies that try to meet this objective are named *share-scheduling policies*; they are the main subject of this thesis. A more elaborate motivation and the precise objectives will be given in Section 1.4 and in Chapter 2. Unfortunately, the response-time related performance measures described thus far are not very useful for share scheduling, since there is no direct relationship beteen the average job response time and the fraction of the system capacity used to serve jobs of a particular group. Hence, a new set of performance measures will be defined in Chapter 2. An overview of the literature on share scheduling will be given in Section 1.5.

1.3.4 Scheduling Mechanisms

Scheduling policies cannot be implemented without supporting mechanisms. A mechanism is a basic hardware or software facility (or a combination of both). The distinction between policies and mechanisms is a useful way of studying scheduling at different levels of abstraction. Although the study of scheduling mechanisms is not part of this thesis, some are important for the sequel, so we will outline them briefly.

We distinguish between *local* and *global* mechanisms. A local mechanism only involves a single processor, whereas a global mechanism involves two or more processors.

Some important local mechanisms are *preemption*, *priorities*, and *feedback*. Preemption is the ability to stop a job, save its state, and resume it at a later time. Preemption was already required in time-sharing systems to provide reasonable response times to interactive jobs, while at the same time serving compute-intensive background jobs. In some systems, jobs can be given priorities. These priorities control the execution order of jobs, and they may be statically assigned or dynamically adjusted. A combination of preemption and priorities can be found in *feedback* systems. In systems with feedback, there are multiple job queues, each of which holds jobs with identical priorities. A job is served until it terminates or until a timer expires. In the latter case, the job is fed back to the tail of a lower-priority queue. The special case where there are no priorities (i.e., there is just a single priority queue) is referred to as *round robin* scheduling.

Two well-known global mechanisms are *initial placement* and *job migration*. Initial placement allows a job to be transferred from one processor to another as long as its execution has not yet started. Job migration allows the transfer of a job at any time. Obviously, preemption is a requisite for the implementation of job migration. As pointed out by Zayas [89], migration of jobs can be expensive in terms of network delay and network load, especially when the jobs have large address spaces.

1.3.5 Static Scheduling Policies

Many static probabilistic global policies for scheduling in distributed systems have been analyzed with queueing theory; we will discuss only a selection of papers. De Souza e Silva and Gerla [21] studied a heterogeneous system with sites connected by a communication network. Each site consists of a central service center modeling the CPU and one or more service centers modeling, for instance, disk stations. The workload consists of closed job chains local to the sites, which model background load, and open job chains arriving at the sites according to Poisson processes. Each service center has a First-Come First-Served, Processor Sharing, or Infinite Server queueing discipline, with exponentially distributed service times. Upon arrival at a site, a job of an open chain can be executed either locally or be sent to another site that is able to serve the job (i.e., *site constraints* must be obeyed). This decision is taken independently of the state of the system and is controlled by routing probabilities. The authors' objective is to find optimal routing probabilities such that a weighted sum of the response times over all job classes is minimized. Since a product-form solution [3] exists, an efficient algorithm for the calculation of these probabilities was found. A more general system was studied in [22], in which jobs consist of multiple tasks that must be executed in sequence. Once a task has completed at some site, the job can be transferred to another site (or perhaps *must* be transferred, due to the per-task site constraints).

Tantawi and Towsley [81] studied an arbitrarily connected distributed system with jobs arriving at the sites according to independent Poisson processes. Each site can be an arbitrary queueing network, as long as it has a product-form solution; there are no site constraints in the model. Jobs can be executed either locally or be sent to another site. In the latter case, the job incurs a communication delay. When the job is terminated at a site, a reply is sent back to the originating site. A job can be transferred only once. As in the previous model, the routing decisions are taken probabilistically and independently of the state of the system, and the resulting queueing network has a product-form solution. An algorithm is proposed that computes the minimum average response time taken over all jobs for a given set of system parameters. As it turns out, under optimal load balancing the set of sites can be partitioned into three groups. The first group consists of sites that send some or all arriving jobs to other sites and do not receive jobs from other sites. These sites are called *source* sites. The second group consists of sites that do not send jobs but receive jobs from at least one other site. These are the *sink* sites. The last group are the so-called *neutral* sites: sites that process all their jobs locally and neither send nor receive jobs. By a parametric analysis, the authors also studied the effect of varying the communication delay on the partitioning of sites. As the communication delay increases, more and more source and sink nodes turn into neutrals, as the cost of transferring a job to another site outweighs the benefits of using a faster site.

Ross and Yao [74] analyzed a heterogeneous model consisting of sites connected by a broadcast communication network. Jobs arrive according to Poisson processes and are either *generic* and can be served by any site, or are *dedicated* and can only be served locally. The authors' objective is twofold. First, the generic jobs are redistributed according to the routing probabilities in order to minimize the average response time. Secondly, the jobs are scheduled locally such that response-time constraints are met for the dedicated jobs. The authors found that taking into account the local scheduling discipline can lead to a significant performance improvement.

1.3.6 Dynamic Scheduling Policies

Chow and Kohler [13] compare three dynamic central policies with a static probabilistic policy in heterogeneous distributed systems. In their model, jobs arrive according to a Poisson process at a central global scheduler, which sends the jobs to one of m First-Come First-Served processors. The service times are exponentially distributed. The mean service time on processor i is $1/\mu_i$. In the static policy, a job is sent to processor i with probability π_i, $i = 1, \ldots, m$. The probabilities are chosen to be proportional to the corresponding processor speeds. This policy is compared to three dynamic policies: (1) the *Minimum Response Time Policy* (MRTP), (2) the *Minimum System Time Policy* (MSTP), and (3) the *Maximum Throughput Policy* (MTP).

In MRTP, an arriving job is sent to the processor with the least expected response time, i.e., the processor i for which $(n_i + 1)/\mu_i$ is minimal, where n_i is the number of jobs at processor i including the one in service.

In MSTP, an arriving job is sent to a processor i such that the expected time to serve *all* jobs in the system to completion (including the arriving job) is minimal. The expected time $T(n_1, \ldots, n_m)$ to complete all jobs in the system when it is in state $(n_1, \ldots, n_m)$ is the sum of the expected time until the first departure and the expected time to finish the

remaining jobs. In other words,

$$T(n_1,\ldots,n_m) = \frac{1}{\sum_{j=1,n_j\neq 0}^{m}\mu_j} + \sum_{k=1,n_k\neq 0}^{m}\left(\frac{\mu_k}{\sum_{j=1,n_j\neq 0}^{m}\mu_j}T(n_1,\ldots,n_k-1,\ldots,n_m)\right), \tag{1.4}$$

with

$$T(0,\ldots,0,n_k,0,\ldots,0) = \frac{n_k}{\mu_k}. \tag{1.5}$$

The difference between MRTP and MSTP may not be obvious at first sight, but minimizing (1.4) does not in general minimize the expected response time for an arriving job. For example, this is the case when $m = 2$, $\mu_1 = 2$, $\mu_2 = 1$, $n_1 = 4$, and $n_2 = 1$. Now, MRTP sends an arriving job to processor 2, because $(n_1+1)/\mu_1 = 2.5$ and $(n_2+1)/\mu_2 = 2$. However, MSTP sends it to processor 1, since $T(n_1+1,n_2) = 1279/486$ and $T(n_1,n_2+1) = 1292/486$.

In MTP, an arriving job is sent to the processor i such that the expected throughput of the system during the next interarrival period is maximal. In other words, MTP maximizes the expected number of departures before the next arrival. Let $TP_i(n_i)$ denote the expected throughput at processor i during an average interarrival period starting with n_i jobs at processor i, and let $TP(n_1,\ldots,n_m)$ denote the total throughput, in other words, $TP(n_1,\ldots,n_m) = \sum_{i=1}^{m} TP_i(n_i)$. Since arrivals are Poisson, we have the following expression for the probability density function $f(\tau)$ of the interarrival period:

$$f(\tau) = \lambda e^{-\lambda\tau}.$$

Similarly, departures at each processor are Poisson, and the probability mass function Q_{ik} of the number of departures at processor i during an interarrival period with n_i jobs at the start of the period is given by

$$\begin{aligned} Q_{ik} &= \frac{(\mu_i\tau)^k e^{-\mu_i\tau}}{k!},\ 0 \le k < n_i, \\ Q_{in_i} &= \sum_{k=n_i}^{\infty}\frac{(\mu_i\tau)^k e^{-\mu_i\tau}}{k!}. \end{aligned}$$

The expression for $TP_i(n_i)$ becomes:

$$\begin{aligned} TP_i(n_i) &= \int_0^{\infty}\frac{\sum_{k=1}^{n_i} kQ_{ik}}{\tau}f(\tau)\,d\tau \\ &= \lambda\left[\sum_{k=1}^{n_i-1}\left(1-\frac{n_i}{k}\right)\left(\frac{\mu_i}{\lambda+\mu_i}\right)^k - n_i\ln\left(\frac{\lambda}{\lambda+\mu_i}\right)\right]. \end{aligned}$$

Upon arrival of a job, MTP sends the job to the processor k for which

$$TP(n_1,\ldots,n_k+1,\ldots,m)$$

is maximal. In case of a tie, the fastest processor is chosen. As opposed to both MRTP and MSTP, MTP needs information on the arrival rate of jobs. In [13], only two-processor systems were considered and in such systems it was found that each of the three dynamic

policies significantly reduces the average response time as compared to the static policy. (For the static policy, the expected job response time can be easily expressed analytically; for the dynamic policies, numerical approximations were used.) Of the three dynamic policies, MTP performs best, but the difference with MRTP is rather small.

Another example of a dynamic central policy is implemented in the CONDOR system developed at the University of Wisconsin, Madison, by Litzkow, Livny, and Mutka [60, 67, 66, 65]. CONDOR is a batch queueing system implementing a central load-sharing policy, and it runs on UNIX™ workstations. The policy is to match idle workstations and jobs waiting for service. In CONDOR, fair access to idle workstations is controlled by the *up-down* algorithm. The jobs submitted for execution on a CONDOR cluster are associated with the workstation on which they have been submitted. For each such workstation i, a *schedule index* $SI[i]$ is maintained. A job submitted from workstation i has preemptive priority over one from workstation j when $SI[i] < SI[j]$. The entries in the SI-table are updated on the following occasions: periodically, when a job leaves the system, and when a workstation becomes available for CONDOR. The periodic updates of $SI[i]$ for workstation i depend on the number of jobs submitted from this workstation that are awaiting execution and on the number of jobs submitted from this workstation currently executing in the CONDOR cluster.

In [67], Mutka and Livny show by means of simulation that the up-down algorithm outperforms both a random allocation of processors to waiting jobs and a round-robin allocation. With the up-down algorithm, the quality of service to users imposing only a light load on the system was not harmed by increasing workloads due to other users.

Many dynamic distributed policies have been proposed and evaluated in the literature. Wang and Morris [84] evaluated source-initiative and server-initiative global policies with varying degrees of required information, including static policies. For instance, some policies use the busy status of servers, while others use their queue lengths or even the remaining service times of jobs. We already partly described their system model in Section 1.3.3. At both the sources and servers, jobs are handled First-Come First-Served. A key result obtained by the authors is that there exist large performance differences among the global scheduling policies, making the choice of a scheduling policy a critical design decision. Also, with the same level of information available, server-initiative policies outperform source-initiative policies.

Eager, Lazowska, and Zahorjan [24] address the question what level of complexity a load-distribution policy should have. In their model, a number of identical processors are connected by a broadcast communication network. Jobs arrive at the nodes according to Poisson processes with the same arrival rates. Jobs' service times are identically distributed random variables. The performance measure used is the average response time. Two components of the global scheduling policy are identified. First, the *transfer* policy determines whether to serve an arriving job locally or remotely. The transfer policy used is to process a job locally if the current queue length at the processor is smaller than a certain threshold T. Second, the *location* policy decides to which processor a job selected for transfer should be sent. Three location policies of increasing complexity are considered. First, the *Random* policy selects a processor at random. This location policy is static, since it requires no system information. In order to prevent instability, a job can be transferred at most L_t times, where L_t is called the *transfer limit*. Secondly, the *Threshold* policy randomly probes up to L_p (the *probe limit*) processors and sends the

job to the first processor with a queue length smaller than T. If no suitable processor is found after L_p probes, the job is served locally. Thirdly, the *Shortest* policy randomly probes L_p processors and sends the job to the processor with the smallest number of jobs in its queue, unless that would place the processor's queue length above the threshold T, in which case the job must be served locally after all. The main result is that Threshold performs much better than Random for a wide range of system parameters, even for small values of the threshold T. The performance of Shortest is not significantly better than that of Threshold, even though Shortest uses more information about the state of the system. An important conclusion is therefore that the potential benefits of load distribution can be achieved to a large extent with relatively simple global policies.

1.4 Share Scheduling

As explained in Section 1.3.3, load-distributing policies can be used to improve the job reponse time in distributed systems. However, sometimes this objective is not meaningful [26]. This is especially true for long-running, non-interactive jobs. For such jobs, users are not really concerned about their response time (they usually know it is very large), but instead, want them to make progress at certain rates. As a refinement to load-distributing policies, *share-scheduling* policies attempt *to provide groups of jobs (for instance, all jobs belonging to a single user or to a department) with prespecified fractions (shares) of the total processing capacity of a multiprocessor or a distributed system.*

An example of an environment in which share scheduling was desired was at IBM [64], on a distributed system consisting of RS/6000 machines running UNIX, called the 'Compute Power Server Cluster', used for numerically intensive scientific computations, with only little interactive use. Several departments funded the system. The workload consisted mainly of compute-intensive jobs with service times in the order of at least minutes, but typically hours, or even days. How to distribute the processing power among the departments? This is where share scheduling comes into play. The cluster replaced VM mainframes, for which much effort had been put into the design of share scheduling. Unfortunately, share scheduling in distributed systems is much more complicated than share scheduling on uniprocessors or multiprocessors.

The foremost motivation for share scheduling is *fairness*: a group of jobs should be given the processing capacity it is entitled to, because this has been agreed upon, or because the user funds the system in exchange for a share. This objective of share scheduling cannot be accomplished easily by load-distributing policies. In a multi-user environment, in which distributed systems usually operate, load-distributing policies often do not take into account the user to whom a job belongs. This implies that a user can increase his share of the total system capacity by submitting more jobs, at the cost of the shares of other users. This undesired and frustrating behavior is discouraged by share-scheduling policies, since users will become aware that submitting more jobs only affects the obtained shares of their other jobs, not those of other users.

Ideally, with share scheduling, the total service rate of a group of jobs is independent of the other groups[1]. Therefore, users can make an estimate of the response time of each

[1]In Chapter 2, we will explain that this is not a good objective: One also wants to allocate unused shares instead of wasting system capacity. As a result, the share allocated to a user *should* depend on

job, provided they know the required service times of their jobs. So the system becomes more *predictable* to its users.

An additional feature of share scheduling is that it can be used to encourage users to certain behaviors, such as the use of batch queues or job submission during off-peak hours. For instance, while a user normally obtains 10% of the system capacity, 30% can be promised if he submits jobs to a batch system that schedules jobs say from 5PM to 9AM.

An important restriction in this thesis is that we consider only compute-intensive, long-running jobs on a time scale of minutes or hours, which justifies the negligence of several overhead effects, such as those of context switching. The applications we have in mind include weather forecasting, simulations, and VLSI design. We also restrict ourselves to distributed systems; shared-memory multiprocessor systems are only considered as a reference for the performance of share-scheduling policies in distributed-memory systems. By comparing the performance of share-scheduling policies on multiprocessors, for which we assume that job migration is free, and distributed systems, one can gain insight into the controversial benefits of job migration in distributed systems [36, 89]. A final restriction in this thesis is that we do not consider jobs consisting of multiple tasks, such as for instance studied in [2]. Such jobs are often used to model parallel programs.

Although in this thesis we restrict ourselves to the application of share scheduling in distributed systems, many of the concepts, mechanisms, and policies can be applied to other systems as well, such as network routers and manufacturing systems. For example, if a router in a network routes several traffic streams, it may be desired to ensure that the router capacity is distributed fairly among the traffic streams, as pointed out by Greenberg and Madras [35]. A major difference between this example and our situation is the time scale of scheduling events. For compute-intensive workloads in distributed systems, this is in the order of minutes or hours, whereas for routing packets this is in the order of microseconds.

1.5 Related Literature on Share Scheduling

Share scheduling has been a topic of research in the area of uniprocessors for about 15 years. In the early literature on the subject [38, 49, 29, 64], one usually speaks of *(fair) share scheduling*, whereas in more recent literature [82, 83, 79, 80], the terminology has changed to *proportional-share scheduling*. We discuss the literature on share scheduling in three sections, considering uniprocessors and multiprocessors, distributed systems, and networks in turn.

1.5.1 Share Scheduling in Uniprocessors and Multiprocessors

One of the earliest implementations of share scheduling in uniprocessors was the Fair Share Scheduler for UNIX, designed by Henry [38]. Apart from the two usual parameters used to calculate the priority of a UNIX process[2] (the recently obtained CPU time and

the presence of the other groups. For the sake of this discussion, we assume that every group has enough jobs present in the system in order to obtain its share.

[2]For this discussion, a UNIX process is identical to a 'job'.

the so-called NICE value), Henry introduces a third parameter that takes into account the CPU usage of *all* processes in what the author calls a 'Fair Share Group:' a group of processes for which share objectives are to be met. When a fair-share group is not using all of its allocated share of the processor, the scheduler distributes the excess share to other groups in proportion to their allocated shares. Another share scheduler for uniprocessor UNIX systems is described by Kay and Lauder [49]. Essick [29] describes a share scheduler for the PRISMOS operating system. As opposed to the work of Henry and Kay and Lauder, Essick emphasizes *short-term* fairness. A recent commercial example of a share-scheduling system is the Share Scheduler for CONVEXOS[3] [20]. This scheduler allows the system administrator to create a hierarchy of share groups, and many tools are available to monitor CPU usage, update databases used by the scheduler, and report long-term CPU usage. Of more recent date is the MAUI scheduler [40, 41]. This scheduler features fair-share scheduling and is in use on systems for high-performance computing, such as IBM SP2 systems and LINUX clusters.

Waldspurger and Weihl [82] proposed an elegant mechanism for share scheduling called *lottery* scheduling, and implemented it on top of the MACH microkernel to allow precise control of the execution rates of threads. In lottery scheduling, resource allocation is controlled by so-called lottery *tickets*. Access to a resource is granted to the client with the winning ticket in a lottery. Clearly, fairness is guaranteed probabilistically since each client holding tickets will eventually win the lottery. Analogously to monetary systems, a ticket is represented as a number of *units* in a *ticket currency*. The currency abstraction is used to share tickets among multiple clients. For each currency, its name, a list of tickets backing the currency, the total number of units active in the currency, and a list of tickets issued in the currency are maintained. All currencies originate from the *base currency* which has no backing tickets. Currencies can be split up into subcurrencies, enabling a share to be split up in a hierarchical way. A client's obtained share of a resource is proportional to the total value in base units of the tickets possessed by the client. A currency is devaluated when more units of it are issued, since the total value (in base units) of the tickets in the currency equals the total value of the backing tickets. The notion of a currency is especially useful in cases where a ticket holder (for instance, a user) needs to be able to create more tickets without increasing their total value. Several extensions and variations to lottery scheduling are presented in [82], such as *ticket transfers* (e.g., passing tickets from a blocked client to the server in a remote procedure call) and *compensation tickets* (to compensate clients for not consuming the share of the resource they are entitled to). The authors present the results of a number of experiments, showing that the execution rates of clients closely match their allocated shares. As expected, the deviations are somewhat larger for larger ratios of the allocated shares and over smaller intervals of time. Nevertheless, lottery scheduling allows very fine-grained control over resource usage. The authors conclude that lottery scheduling can be used to control effectively the relative execution rates of jobs, with little overhead.

A deterministic approach to control the relative execution rates of competing jobs was also presented by Waldspurger and Weihl [83]. In *stride scheduling*, each job is assigned a *ticket* value which specifies the share of a resource this job should obtain relative to other jobs. The time interval between successive selections for a unit of service is inversely proportional to a job's ticket value. A representation of this interval is the *stride* of the

[3]CONVEX and CONVEXOS are registered trademarks of Hewlett-Packard.

job. The stride of a job is not measured in real time units, but in virtual time units called *passes*. The *pass* field of a job represents the virtual time index for its next selection. In the basic algorithm, after the expiration of a quantum, the scheduler selects the job with the minimum pass for service, and advances the pass of the job by its stride. The results obtained with a prototype implementation of stride scheduling in LINUX, indicate that the job execution rates closely match the desired rates, closer than achieved with lottery scheduling.

Another deterministic approach for the control of obtained shares has been presented by Fong and Squillante [32], who propose a scheduling policy called *Time-Function Scheduling* (TFS). A group of jobs with similar performance requirements is called a job *class* and is assigned a *time function* and a job queue. There are K job classes. The time function value $F_k(t)$ of a job in class k defines its priority, where t is the elapsed time since the job was last appended to the tail of the run queue of class-k jobs. Jobs are preempted after the expiration of a time quantum, when the job is blocked, and when the state of the scheduler changes (for instance, when a new job arrives). When a job of class k is preempted, it is appended to the tail of the class-k run queue. The scheduler only considers the jobs at the head of the run queues to determine which of them should be allocated the next time quantum. Two possible choices for the F_k are constant and linear functions. If $F_k(t) = k$, $k = 1, \ldots, K$, class i has absolute priority over class j when $i > j$. If $F_k(t) = a_k \cdot t$, $k = 1, \ldots, K$, *each* job of group i and *each* job of group j are served at ratio a_i/a_j. The capabilities of TFS to provide different execution rates among jobs, as well as the overhead imposed by the scheduler, are comparable to those of lottery scheduling, but the variance in response-time ratios is significantly smaller. Also, the convergence of TFS to the allocation goals is faster and more stable than with lottery scheduling.

Epema [27] derives expressions for the steady-state shares of groups of compute-intensive jobs in multiprocessor UNIX systems. The model presented in the paper is an extension of that of Hellerstein [37] for uniprocessor UNIX systems. The obtained shares of jobs can be controlled with their NICE values, in effect achieving share scheduling. The model in [27] matches reality well, as was shown by comparing it with simulation results of the scheduling policies in several UNIX multiprocessors, and by actual measurements in such systems. Unfortunately, the relations between the NICE values of a set of jobs and their obtained shares are rather complicated, and are different for SYSTEM V, BSD[4]4.3, and MACH. Also, not all ratios of obtained shares can be achieved.

Stoica, Abdel-Wahab, and Jeffay study a scheduling policy called *Earliest Eligible Virtual Deadline First* (EEVDF) for proportional-share scheduling [79, 80]. Every job has a positive weight which determines the relative share of processor time it is entitled to. The weight of job j is denoted by w_j. The policy is based upon a notion of virtual time. Virtual time progresses at a rate equal to the inverse of the sum of the weights of the jobs present in the system, so that in each unit of virtual time, every job can receive an amount of processor time equal to its weight. The virtual time $V(t)$ at time t is defined as

$$V(t) \triangleq \int_0^t \frac{1}{\sum_{j \in \mathcal{A}(\tau)} w_j} d\tau, \tag{1.6}$$

where $\mathcal{A}(t)$ is the set of active jobs at time t. For each request by a job, an eligible time

[4]Berkeley Software Distribution.

and a deadline are computed. Both are points in virtual time (possibly in the past), and they correspond to the amount of service the job has received when it issues the request, and to this amount plus the amount of service requested, respectively. Service is granted to the job with the earliest deadline whose eligible time is not in the future. It is shown that EEVDF enjoys the property that as long as a job keeps requesting service, the amount of service it has obtained is never more off from what it is entitled to than the maximum quantum size. In fact, the maximum quantun size is shown to be the best lower bound that can be achieved by any proportional-share policy. In addition, it is shown that the policy can deal with adding and removing jobs, and with changing weights very well. The usefulness of EEVDF was demonstrated with a prototype CPU scheduler under FREEBSD.

1.5.2 Share Scheduling in Distributed Systems

Share scheduling in distributed systems has received little attention. Moruzzi and Rose [64] introduce a central share scheduler for clusters of UNIX workstations. The scheduler periodically gathers data on CPU usage on each machine, recalculates the priority of each process based on the obtained and the required shares, and modifies the job priorities on the machines. The authors report that the share scheduler is a valuable tool for the management of the cluster. De Jongh studies central scheduling policies for share scheduling in [45], and distributed policies in [47]. We extend his results in Chapters 5 and 6, respectively. Epema and De Jongh describe the objectives, performance measures, and basic policies for share scheduling in [28]. In the paper, the authors summarize results obtained in Chapters 2, 3, 4, and 5.

1.5.3 Share Scheduling in Networks

In this thesis, we restrict ourselves to share scheduling in distributed systems. However, share scheduling is also becoming more and more important in communication networks, where it is usually called *(weighted) fair queueing.* We will describe a selection of the literature on fair queueing, for the purpose of illustrating the differences with scheduling compute-intensive jobs in distributed systems.

Especially in (packet-switching) networks designed for high-performance computing or for high-bandwidth applications, there is an increasing need for fairness, for guaranteed bandwidth, and for guaranteed maximum transfer delay. An example application that requires guaranteed bandwidth and guaranteed maximum delay is real-time video, which must display video frames at fixed intervals. It is vital that the queueing policies at the intermediate gateways protect the traffic streams of such applications from those of ill-behaved sources [23].

The most important differences between scheduling compute-intensive jobs and packet scheduling in networks are:

1. In networks, the time scale on which to achieve fair scheduling is much smaller (in the order of microseconds or less). Therefore, the computational complexity of a gateway scheduling policy is much more important than that of a local scheduling policy in a distributed system.

2. In networks, scheduling is mostly nonpreemptive because the transmission of a packet usually cannot be interrupted. As a result, many of the problems with share scheduling in networks are due to the nonpreemptive nature of packet scheduling. In general-purpose distributed systems, the availability of a job-preemption mechanism is practically a prerequisite.

3. In networks, the reception of a packet at a gateway takes a non-trivial amount of time compared to the time to process the packet (i.e., take the routing decision and transmit the packet). Therefore, an important design decision for gateway scheduling is whether or not one allows the transmission of a packet to start before the entire packet is received. This plays no role in scheduling compute-intensive jobs.

For the special case in which the traffic streams at a gateway should be treated equally, what one really needs is a queueing discipline known as Head-of-the-Line Processor Sharing (HOL-PS, e.g., [35]). In HOL-PS, a single server serves at equal rates all jobs at the heads of a number of queues (of network packets) simultaneously. That way, the server capacity is fairly distributed among the traffic flows present. The generalization of HOL-PS to unequal shares is known as Generalized Processor Sharing (GPS, e.g. [71]). In GPS, each traffic source i has an associated positive weight ϕ_i. At all times, GPS serves a traffic source i in first-come first-served order at rate proportional to ϕ_i, provided source i is backlogged (i.e., work for source i is present). Clearly, if r is the total service rate of the server, GPS guarantees a service rate of $(\phi_i / \sum_j \phi_j) \times r$ to source i at all times that source i is backlogged.

Unfortunately, HOL-PS and GPS are hard to implement in data networks since one usually cannot preempt the handling of a packet. On the other hand, a cyclic service discipline fails to be fair when the packet sizes of different flows do not correspond with the required service rates. One of the key problems in gateway scheduling is therefore to find a nonpreemptive, fair, and computationally-efficient policy that selects the next packet to be sent.

Demers, Keshav, and Shenker [23] study a scheduling policy named Fair Queueing (FQ). In FQ, from among the packets that have completely arrived at a gateway, the one that would finish transmission first in the *bit-by-bit round-robin policy* is the next to be transmitted. The authors show that FQ has several important advantages over First-Come First-Served treatment of packets, notably increased fairness, lower delay for sources using less than their full share of bandwidth, and protection from ill-behaved sources.

Greenberg and Madras [35] analyze queueing models of fair scheduling policies in networks. The authors propose two policies called *Fair Queueing based on Starting times* (FQS) and *Fair Queueing based on Finishing times* (FQF) that are to approximate HOL-PS. In FQS the job that would *start* earliest under the HOL-PS discipline is selected for service, while FQF selects the job that would *finish* earliest under HOL-PS. Only one job (packet) is served at a time, and once a job is taken into service, it is served to completion. The FQF policy is identical to the original FQ policy in [23]. By means of operational analysis, the authors show that both FQS and FQF can be used as substitutes for HOL-PS. Their results include the following upper bound on the difference between the service allocations to any queue q under HOL-PS, and under FQS and FQF, provided

there exists a maximum job service time s_{MAX}:

$$|a^q_{\text{HOL-PS}}(0,t) - a^q_{\text{FQ}}(0,t)| \leq s_{\text{MAX}}, \tag{1.7}$$

where $a^q_{\text{HOL-PS}}(0,t)$ and $a^q_{\text{FQ}}(0,t)$ denote the amounts of service time given to queue q during interval $[0,t)$ under HOL-PS, and under FQS and FQF, respectively. In fact, this bound is the best possible under any non-preemptive policy. Since FQF requires *a priori* knowledge about the jobs' service times, whereas FQS does not, the latter policy is preferred.

Unfortunately, FQS is expensive to implement in high-speed networks because of its computational complexity, as pointed out by Shreedhar and Varghese [76]. They propose an alternative algorithm derived from round-robin, called Deficit Round-Robin (DRR). The algorithm allows for the queues to have different shares (in terms of desired throughputs). The basic principle of DRR is to associate with each queue a variable containing the service-time deficit from the past. When a job (packet) with a large service time arrives at a queue, servicing this job is delayed in favor of other queues until the corresponding *deficit value* is large enough. Obviously, this requires the algorithm to have knowledge of the service time of a job when it enters the system. In data networks this is reasonable, since it is simply the size of a packet. The key result of [76] is that DRR provides fairness at a much lower computational cost than FQF.

Parekh and Gallager [71] extend the fair-queueing policy of [23] to unequal shares, and introduce a nonpreemptive policy called Packet-by-packet Generalized Processor Sharing (PGPS), also known as Weighted Fair Queueing (WFQ). In PGPS, when the server becomes available at time t, the policy selects the packet that would complete service first in GPS if no packets would arrive after time t. The authors show that PGPS closely approximates GPS, and they derive tight bounds on the worst-case packet delay when the sources are constrained by a leaky-bucket policy.

1.6 Outline of the Thesis

We end this introductory chapter with an outline of the remainder of this thesis. We describe the research questions, the organization of the thesis, and its main research contributions.

1.6.1 Research Questions

In the remainder of this thesis, we focus on two questions:

1. What are the objectives and performance measures for share scheduling in mathematical terms?

2. What are suitable policies for share scheduling in uniprocessors, multiprocessors, and especially distributed systems?

1.6.2 Organization of the Thesis

This thesis is organized as follows. We define the objectives and performance measures in Chapter 2. We introduce three objectives:

1. Making sure that each of the groups obtains its share.

2. Preventing processors from being unnecessarily idle.

3. Making sure that the jobs within a single group obtain equal shares.

As it turns out, the differences between uniprocessor, multiprocessor, and distributed systems play an important role in setting share-scheduling objectives. Also, due to the possible absence of groups at some times, constant target shares for all groups may not be adequate.

The design and analysis of local and global share-scheduling policies is covered in Chapters 3 through 6. A key result of this thesis is that share scheduling is rather trivial in uniprocessors and in multiprocessor systems, given the proper policies we will introduce, but is problematic in distributed systems with no or costly job migration.

In Chapter 3, we examine the performance of some local scheduling policies. Classical policies, such as First-Come First-Served, Head-of-the-Line, and Processor Sharing, do not perform well over a broad range of the workload parameters. However, we introduce two new policies, viz. Up-Down and Group-Priority Processor Sharing, of which especially the latter performs extremely well. Both policies are reasonably easy to implement.

Global *static* policies are considered in Chapter 4, which consists of two parts. A deterministic model shows how share scheduling can be done effectively on multiprocessor systems. In the second part of Chapter 4, we study a probabilistic model of a distributed system without job migration. Essentially, this model is much different from the one studied in the first part of this chapter, and the results emphasize the sharp contrast in complexity and share-scheduling performance between multiprocessors and distributed systems. Even though the static probabilistic policy is not considered viable for implementation on a general-purpose distributed system (due to unrealistic assumptions on the workload), it gives us a feeling for the problems we are faced with in such systems. Moreover, the results obtained are used in later chapters for comparison with dynamic policies.

In Chapter 5, we examine global *dynamic central* policies, in which a single processor takes all scheduling decisions. Analytical solutions for global dynamic policies are hard, if not impossible, to obtain, and therefore we compare various dynamic policies by means of simulations. Two strategies are compared. In the first, we try to have each processor run at least one job of each group. The local policy then provides the proper shares to all groups. The idea is that once each group obtains its share locally on each processor, it will obtain its share globally. In the second, the set of processors is partitioned among the groups, and jobs are only served by their group's processor subset. In short, the second approach gives better performance, but is more complex than the first approach. Also, the second approach leaves processors unnecessarily idle more often than the first approach.

In Chapter 6, we propose and compare global *dynamic distributed* policies, in which the scheduling authority is distributed among the processors in the system. Distributed scheduling policies are complicated because the current knowledge of a processor may be incomplete or out of date. The policies proposed are distributed variants of the central policies introduced in Chapter 5. The policies use a mechanism called *probing* in order to obtain information on the system state. Since only a fraction of the processors is probed

to arrive at a scheduling decision, only partial information on the system state is available to the policy. We will show that the performance degradation due to the lack of complete state information is negligible if at least 30% of the processors is probed. However, the scalability of share-scheduling policies remains an open problem.

Finally, in Chapter 7, we summarize our conclusions and suggest future research. In Appendix A, we explain some of the simulation techniques used. We also present the results of some well-known tests we performed on the quality of the random-number generator used in our simulations.

1.6.3 Main Contributions of the Thesis

The main contributions of this thesis are:

- We state the objectives of share scheduling in precise mathematical terms in Chapter 2. In doing so, we explicitly take into account the shares obtained by groups of jobs, whereas related work mostly studies job response and waiting times.
- We introduce new performance measures for the evaluation of share-scheduling policies in Chapter 2.
- We evaluate the share-scheduling performance of common uniprocessor scheduling policies in Chapter 3.
- We derive necessary and sufficient conditions for providing shares to jobs in a multiprocessor system in Section 4.2.
- We introduce and evaluate share-scheduling policies for distributed systems in Section 4.3 and Chapters 5 and 6. We consider the effects of both the global and the local policies on the performance of a policy. Also, using distributed policies for share scheduling (as we do in Chapter 6) has not been studied in detail before.
- We show that our share-scheduling objectives, as outlined in Chapter 2, can be met in realistic distributed systems.
- We show that in general, dynamic share-scheduling policies outperform random splitting.
- We show that in distributed systems, both the local and the global policies have a significant effect on the share-scheduling performance.
- We show that, under realistic assumptions, job migration substantially improves share-scheduling performance. This contradicts some previous work on the benefits of job migration.

Chapter 2

Objectives and Performance Measures of Share Scheduling

2.1 Introduction

In this chapter, we consider the objectives of share scheduling in uniprocessors, multiprocessors, and distributed systems. We state these objectives in terms of so-called *shares*. A share is a fraction of the total system capacity. We define the *obtained job share* in Section 2.2 as the fraction of the total system capacity currently used to serve a particular job. Subsequently, we define the *obtained group share* as the share obtained by all jobs together in a group, and the *total obtained share* as the share obtained by all jobs together, regardless of the group to which they belong. Since the objective of share scheduling is to keep the obtained shares close to prespecified shares, it is important to know some characteristics of obtained shares. In particular, in Section 2.2 we derive upper bounds on the obtained shares.

In Section 2.3, we introduce two idealized key scheduling polices. These policies allow total control of the obtained job shares on uniprocessors and in multiprocessor systems, respectively. By studying these policies, we show what shares can theoretically be obtained on these systems.

Subsequently, we introduce the three objectives of share scheduling. In Section 2.4, we state our first and most important objective, the *delivery of feasible group shares*. The *feasible group share* of group g is the target for the obtained group share of group g. Our objective is to minimize the differences between the feasible and the obtained group shares for all groups. However, as it turns out, the definition of the feasible group share is not trivial. Our starting point is the definition of the so-called *required group share*. The required group shares are constant, and they sum to one. Unfortunately, providing the required group share to a group is not always possible. In short, obstructions occur when there is a lack of jobs of a group; when, due to inhomogeneity, there is a mismatch between the shares and the processor capacities; and when the proper allocation of processor time to jobs requires the availability of job migration. For instance, a group with no jobs present in the system cannot be given any share of the system. We state a reasonable, uniform definition of the *feasible* group share. According to this definition, the feasible group share only depends on the number of jobs of that group present, on the number of processors and their capacities, and on the required group share.

In Section 2.5, we state our second objective, *the minimization of capacity loss*, or, in other words, using as much of the system's capacity as possible. We state this objective in terms of the total feasible share, which is the target for the total obtained share. Capacity loss occurs when some processor is idle while two or more jobs reside at another processor.

In Section 2.6, we state our third objective, *internal fairness*, which aims at equal shares for the jobs within a single group. This objective is stated in terms of the feasible job share, which is the target for the obtained job share.

We conclude this chapter with a summary in Section 2.7.

2.2 Obtained Shares

In this section, we define the obtained share of a group of jobs, and derive some properties of obtained shares.

2.2.1 Preliminary Definitions

Jobs arrive at the system according to one or more interarrival-time processes. These processes determine the time between the arrivals of two consecutive jobs. The *arrival time* of job j is denoted by A_j. Once a job j is completed, it leaves the system at its *departure time* D_j. The *response time* R_j of job j is defined as

$$R_j \triangleq D_j - A_j \ . \tag{2.1}$$

The *service time* S_j of job j is its response time on a unit-capacity processor serving no other jobs; by definition, the response time of a job with service time s on a processor with capacity c' is s/c'. We assume that service times are finite and not known in advance.

We define the *job set* $J(t)$ *at time* t as the set of jobs present in the system at time t:

$$J(t) \triangleq \{j | A_j \leq t < D_j\} \ . \tag{2.2}$$

We define the set $J_{*p}(t)$ of jobs on processor p ($p = 1, \ldots, P$), the set $J_{g*}(t)$ of jobs of group g ($g = 1, \ldots, G$), and the set $J_{gp}(t)$ of jobs of group g on processor p, all at time t, in an analogous way. The number of elements in some set Ω is denoted by $|\Omega|$.

2.2.2 The Definition of Obtained Shares

For each job $j \in J(t)$, we define the *remaining work* $W_j^{\mathrm{J}}(t)$ *at time* t as the time it would take to serve the job to completion on a unit-capacity processor. The *service rate* $\sigma_j^{\mathrm{J}}(t)$ *of job* j *at time* t ($A_j \leq t < D_j$) is defined as

$$\sigma_j^{\mathrm{J}}(t) \triangleq -\lim_{\tau \downarrow t} \frac{d W_j^{\mathrm{J}}(\tau)}{d\tau} \ . \tag{2.3}$$

The *obtained share* $o_j^{\mathrm{J}}(t)$ *of job* j *at time* t ($A_j \leq t < D_j$) is defined as

$$o_j^{\mathrm{J}}(t) \triangleq \frac{\sigma_j^{\mathrm{J}}(t)}{c} \ . \tag{2.4}$$

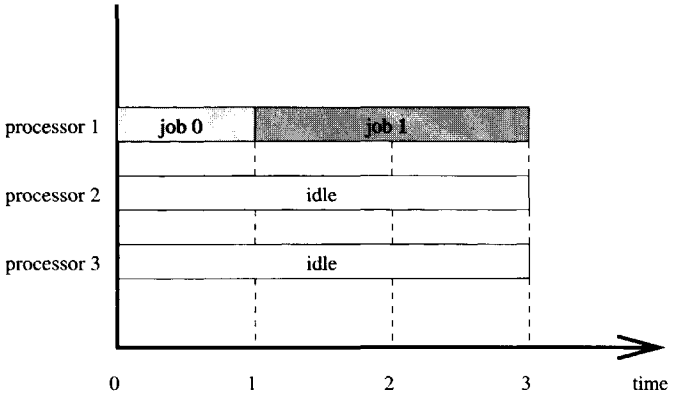


Figure 2.1: The Gantt chart for the First-Come First-Served policy.

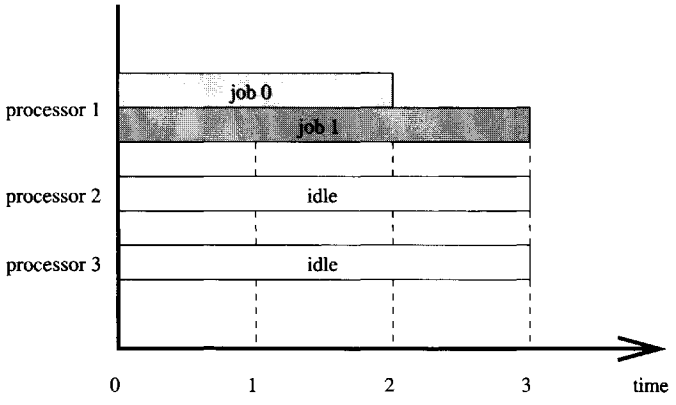


Figure 2.2: The Gantt chart for the Processor-Sharing policy.

In words, $o_j^{\mathrm{J}}(t)$ is the fraction of the total system capacity used to serve job j. In order for (2.3) and (2.4) to make sense, we need to assume that $W_j^{\mathrm{J}}(t)$ is right-differentiable. This is indeed the case for all scheduling policies considered, for which $W_j^{\mathrm{J}}(t)$ is always a piecewise-linear, continuous function of t.

Let us illustrate $W_j^{\mathrm{J}}(t)$ and $o_j^{\mathrm{J}}(t)$ by considering an example system with $P = 3$, $c_1 = 2$, and $c_2 = c_3 = 1$. For simplicity, we assume that there is no job migration and that jobs are only served by processor 1, or wait in its queue. At time $t = 0$, job 1 with service time $S_1 = 4$ enters the system, job 0 is already present, and $W_0^{\mathrm{J}}(0) = 2$. There are no other jobs present in the system, nor do any other jobs arrive. We consider both the First-Come First-Served (FCFS) and Processor-Sharing (PS) policies. We show Gantt charts for the execution of the two jobs (0 and 1) under FCFS and PS in Figures 2.1 and 2.2, respectively. In Figures 2.3 and 2.4, we show $W_1^{\mathrm{J}}(t)$, $\sigma_1^{\mathrm{J}}(t)$, and $o_1^{\mathrm{J}}(t)$ for the First-Come First-Served (FCFS) and Processor-Sharing (PS) policies, respectively. When FCFS is

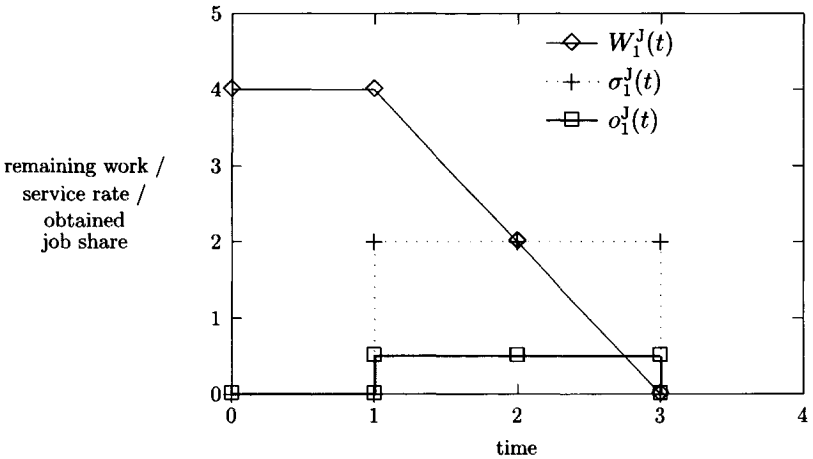


Figure 2.3: The remaining work $W_1^J(t)$, the service rate $\sigma_1^J(t)$, and the obtained job share $o_1^J(t)$ for the First-Come First-Served policy.

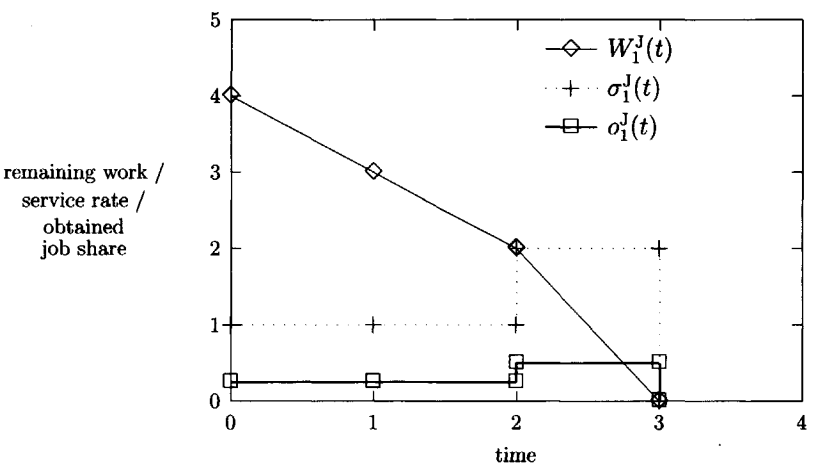


Figure 2.4: The remaining work $W_1^J(t)$, the service rate $\sigma_1^J(t)$, and the obtained job share $o_1^J(t)$ for the Processor-Sharing policy.

used, job 1 is taken into service at $t = 1$ (after job 0 has terminated), and departs at $t = 3$. During the service time of job 1, $W_1^{\mathrm{J}}(t)$ is a linearly decreasing function of t with slope -2. Therefore, $\sigma_1^{\mathrm{J}}(t) = 2$, and $o_1^{\mathrm{J}}(t) = \sigma_1^{\mathrm{J}}(t)/4 = 1/2$. When PS is used, job 1 is taken into service immediately at $t = 0$, sharing processor 1 with job 0 until $t = 2$, so $\sigma_1^{\mathrm{J}}(t) = 1$, and $o_1^{\mathrm{J}}(t) = 1/4$. At $t = 2$, job 0 leaves the system and job 1 is processed at full rate on processor 1 ($\sigma_1^{\mathrm{J}}(t) = 2$, $o_1^{\mathrm{J}}(t) = 1/2$). Job 1 leaves the system at $t = 3$.

We define the *obtained group share* $o_{gp}^{\mathrm{G}}(t)$ *of group* g *on processor* p, the *obtained group share* $o_g^{\mathrm{G}}(t)$ *of group* g, and the *total obtained share* $o^{\mathrm{T}}(t)$ (all at time t) as

$$o_{gp}^{\mathrm{G}}(t) \triangleq \sum_{j \in J_{gp}(t)} o_j^{\mathrm{J}}(t), \tag{2.5}$$

$$o_g^{\mathrm{G}}(t) \triangleq \sum_{j \in J_{g*}(t)} o_j^{\mathrm{J}}(t), \tag{2.6}$$

$$o^{\mathrm{T}}(t) \triangleq \sum_{j \in J(t)} o_j^{\mathrm{J}}(t). \tag{2.7}$$

In the example system described earlier, it is obvious that $o^{\mathrm{T}}(t) = 1/2$ from $t = 0$ to $t = 3$, and $o^{\mathrm{T}}(t) = 0$ from $t = 3$ onwards.

2.2.3 Some Properties of Obtained Shares

Before turning our attention to the objectives of share scheduling, we will discuss some properties of obtained shares. We have,

$$\int_{A_j}^{D_j} o_j^{\mathrm{J}}(t)\, dt = \frac{1}{c} \int_{A_j}^{D_j} \sigma_j^{\mathrm{J}}(t)\, dt = \frac{S_j}{c}, \tag{2.8}$$

which follows from the definition of the obtained job share in (2.4), and from

$$W_j^{\mathrm{J}}(A_j) = S_j,$$

together with

$$W_j^{\mathrm{J}}(D_j) = 0.$$

Furthermore, we clearly have

$$0 \leq o_j^{\mathrm{J}}(t) \leq \frac{c_1}{c} \leq 1. \tag{2.9}$$

The next property puts an upper bound on the sum of the obtained job shares of any set of jobs. Clearly, for any set $\{1, \ldots, J\}$ of jobs we have:

$$\sum_{j=1}^{J} o_j^{\mathrm{J}}(t) \leq \frac{1}{c} \times \sum_{p=1}^{\min(J,P)} c_p. \tag{2.10}$$

Since (2.10) imposes upper bounds on the total obtained share and the obtained group shares, we define the *maximum obtainable total share* $m^{\mathrm{T}}(t)$ *at time* t as

$$m^{\mathrm{T}}(t) \triangleq \frac{1}{c} \times \sum_{p=1}^{\min(|J(t)|,P)} c_p, \tag{2.11}$$

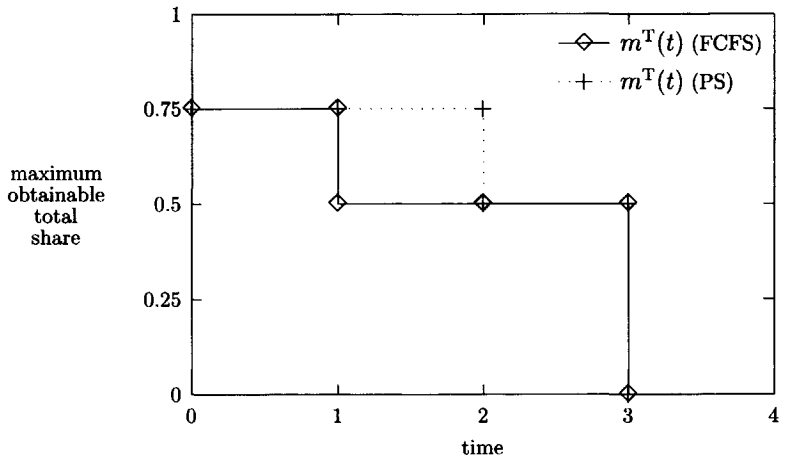


Figure 2.5: The maximum obtainable total share $m^{\mathrm{T}}(t)$ for the First-Come First-Served and Processor-Sharing policies in the example system of Section 2.2.2.

and the *maximum obtainable group share* $m_g^{\mathrm{G}}(t)$ *of group* g *at time* t as

$$m_g^{\mathrm{G}}(t) \triangleq \frac{1}{c} \times \sum_{p=1}^{\min(|J_{g*}(t)|,P)} c_p. \tag{2.12}$$

In homogeneous systems, (2.11) and (2.12) reduce to

$$m^{\mathrm{T}}(t) = \min\left(\frac{|J(t)|}{P}, 1\right), \tag{2.13}$$

$$m_g^{\mathrm{G}}(t) = \min\left(\frac{|J_{g*}(t)|}{P}, 1\right). \tag{2.14}$$

By (2.6), (2.7), (2.10), (2.11), and (2.12), we have, whether the system is homogeneous or not,

$$o^{\mathrm{T}}(t) \leq m^{\mathrm{T}}(t), \tag{2.15}$$

$$o_g^{\mathrm{G}}(t) \leq m_g^{\mathrm{G}}(t). \tag{2.16}$$

For the example system described in Section 2.2.2, $m^{\mathrm{T}}(t)$ is shown as a function of time in Figure 2.5.

2.3 Two Basic Scheduling Policies

Before discussing the objectives of share scheduling in detail, we introduce two important scheduling policies, one for uniprocessors and one for multiprocessors and distributed systems. Each of these allows full control over the obtained shares of the jobs present and should be seen as a reference policy for the delivery of shares.

2.3.1 Job-Priority Processor-Sharing in Uniprocessors

Our first policy is for uniprocessors. In Job-Priority Processor Sharing (JPPS), each job present on a processor has a *weight*, and the processor serves all jobs simultaneously, each at a rate proportional to its weight. It is obvious that with JPPS, any job on the processor can be given any obtained job share. So JPPS provides us with full control over the ratios of obtained shares on a per-job basis, and is therefore an ideal share-scheduling policy.

The idea of discriminating among jobs is not new. In Chapter 3, we describe variants of JPPS in which the weight of a job depends on which group a job belongs to.

2.3.2 Job-Priority Processor-Sharing in Multiprocessors and Distributed Systems

Our second policy is for multiprocessors and distributed systems with free job migration. In *multiprocessor job-priority processor sharing* (MJPPS), which is a generalization of JPPS described above, each job j has a *weight* w_{jp} on processor p, which is the fraction of time job j spends on processor p. We assume that context switching is infinitely fast, so job j can switch between processors such that it appears to be served by more than one processor at the same time, at total service rate of $\sum_p w_{jp}c_p$. In distributed systems, this means that job migration must be free of costs. Under these assumptions, there is no difference between multiprocessors and distributed systems.

Naturally, we require

$$0 \leq w_{jp} \leq 1, \text{ for all } j, p, \tag{2.17}$$

$$\sum_j w_{jp} \leq 1, \text{ for all } p, \tag{2.18}$$

$$\sum_p w_{jp} \leq 1, \text{ for all } j. \tag{2.19}$$

Requirement (2.17) is trivial. Requirements (2.18) and (2.19) state that a processor cannot spend more than 100% of the time serving jobs, and that a job cannot be served more than 100% of the time. We will prove in Section 4.2 that under MJPPS, jobs $1, \ldots, J$ ($J \geq 1$) can be given service rates $\sigma_1^{\mathrm{J}} \geq \cdots \geq \sigma_J^{\mathrm{J}} > 0$, if and only if, for $j = 1, \ldots, J$,

$$\sum_{k=1}^{j} \sigma_k^{\mathrm{J}} \leq \sum_{p=1}^{\min(k,P)} c_p. \tag{2.20}$$

When we compare this result to the definition of $m^{\mathrm{T}}(t)$ in (2.11) it is obvious that, under MJPPS, the set of jobs present in the system can always be provided with a share of $m^{\mathrm{T}}(t)$. Also, all jobs of group g can be provided with $m_g^{\mathrm{G}}(t)$ (defined in (2.12)), $g = 1, \ldots, G$. However, it is not guaranteed that multiple groups can obtain their shares of $m_g^{\mathrm{G}}(t)$ *simultaneously*!

2.4 Objective I: Delivery of Feasible Group Shares

The first and foremost objective of share scheduling is to provide groups with their *feasible group shares*. This section focuses on the definition of these shares. Our starting point

is the definition of a *constant* share for each group, the *required group share*, in Section 2.4.1. We then state requirements for the definition of the feasible group share in Section 2.4.2, and list the cases that complicate such a definition in Section 2.4.3. We propose our definition of the feasible group share in Section 2.4.4. As it turns out, the feasible group share depends only on the required group share, the number of jobs present of the group, and the average processor capacity. Moreover, on uniprocessors and on multiprocessors, we can always provide groups with their feasible group shares simultaneously. This is not the case on distributed systems.

Subsequently, we introduce a special class of systems called *partitionable* systems in Section 2.4.5. Such systems have nice properties for share scheduling. They are introduced here as an example of classes of systems in which one could use other definitions of the feasible group share than proposed in this thesis. However, the notion of partitionability plays an important role in Chapters 4, 5, and 6 as well.

In Section 2.4.6, we introduce the performance measures for compliance with the feasible group share. Finally, in Section 2.4.7, we end this section with a discussion.

2.4.1 Required Group Shares

The *required group share* r_g^{G} *of group* g is the fraction of the total system capacity c that group g is entitled to. We assume that

$$\sum_g r_g^{\mathrm{G}} = 1. \tag{2.21}$$

The required group shares are assumed to be constant over time and to be known in advance to the system. These shares are the only numbers based on which we want share scheduling to discriminate among the G groups. In other words, when the required group shares of two groups are the same, the system should treat them equally, and when one exceeds the other, the system should give preferential treatment to the group with the largest r_g^{G}.

It is immediately clear that a group cannot always be provided with its required group share, for instance, when there are no jobs of the group present in the system. This is one of the reasons we cannot define the *feasible* group share as a constant. Nevertheless, the required group share will play an important role in the definition of the feasible group share.

2.4.2 Requirements for the Definition of the Feasible Group Share

We state the following requirements for the definition of the feasible group share $f_g^{\mathrm{G}}(t)$, for $g = 1, \ldots, G$:

1. $0 \leq f_g^{\mathrm{G}}(t) \leq m_g^{\mathrm{G}}(t)$.
2. $f_g^{\mathrm{G}}(t)$ only depends on $|J_{g*}(t)|$, r_g^{G}, P, and $c_1, \ldots, c_P$, and is non-decreasing in $|J_{g*}(t)|$ and r_g^{G}.
3. if $|J_{g*}(t)|$ is large enough (and certainly when $|J_{g*}(t)| \geq P$), then $f_g^{\mathrm{G}}(t) = r_g^{\mathrm{G}}$.

The first requirement is obvious: feasible group shares imply a promise to a group, and we should not promise more than the maximum we can provide.

The second requirement is essential: the feasible group share only depends on the number of jobs of the group and *not* on the coincidental presence of jobs of other groups. This independence is one of the key characteristics of share scheduling: groups are promised a share of the system that does not depend on the activity of other groups. The second requirement also states that the feasible group shares of two groups with equal required group shares should have identical definitions. Note that share deviations from the past cannot be used in the definition of the feasible group share.

The third requirement states that when the number of jobs in a group exceeds a threshold, the feasible group share equals the required group share. This threshold may be different for different groups, provided they have different required group shares. Note that definitions of $f_g^{\mathrm{G}}(t)$ in which $f_g^{\mathrm{G}}(t)$ goes to r_g^{G} *asymptotically* are excluded.

From requirements 2 and 3, it follows that

$$f_g^{\mathrm{G}}(t) \leq r_g^{\mathrm{G}}, \tag{2.22}$$

and therefore, when (2.22) is combined with (2.21),

$$\sum_g f_g^{\mathrm{G}}(t) \leq 1. \tag{2.23}$$

This means we never promise more than the total system capacity to all groups together. When (2.22) is combined with requirement 1, we obtain what we call the *maximum feasible group share* $\bar{f}_g^{\mathrm{G}}(t)$:

$$\bar{f}_g^{\mathrm{G}}(t) \triangleq \min(r_g^{\mathrm{G}}, m_g^{\mathrm{G}}(t)), \tag{2.24}$$

which meets requirements 1, 2, and 3. Any definition of $f_g^{\mathrm{G}}(t)$ which meets requirements 1, 2, and 3 satisfies

$$f_g^{\mathrm{G}}(t) \leq \bar{f}_g^{\mathrm{G}}(t). \tag{2.25}$$

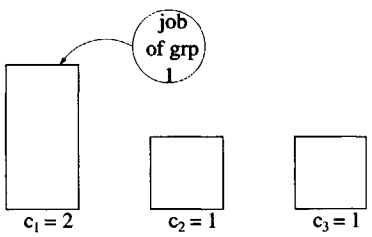
In the next section, we will study $\bar{f}_g^{\mathrm{G}}(t)$ as a candidate for the definition of $f_g^{\mathrm{G}}(t)$.

2.4.3 Complications in the Definition of the Feasible Group Share

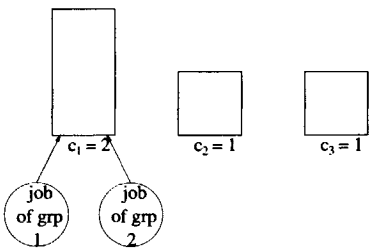
There are four cases that need to be considered when defining the feasible group share $f_g^{\mathrm{G}}(t)$:

1. lack of jobs,
2. inhomogeneity,
3. impossible fit in the absence of job migration, and
4. irrevocability of initial placement due to the absence of job migration.

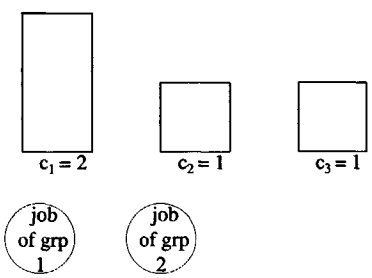
An overview of these cases is given in Figure 2.6.



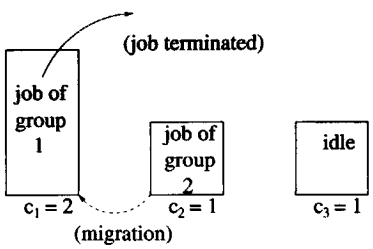
1. Lack of jobs: with only a single job in the system, group 1 can obtain a share of at most 0.5.



2. Inhomogeneity: Although groups 1 and 2 can be provided with a share of 0.4, this cannot be done simultaneously ($r_1^G = r_2^G = 0.4$, $r_3^G = 0.2$).



3. Impossible fit in the absence of job migration: Providing group 1 and 2 each with a share of 0.375 is only possible with job migration.



4. Irrevocability of initial placement due to the absence of job migration: When the job of group 1 terminates, we would need job migration in order to provide group 2 with a share of at least 0.375.

Figure 2.6: The four cases that complicate the definition of the feasible group share.

Case 1: Lack of Jobs

As we already stated, there may not be enough jobs of a group to provide it with a specific share such as its required group share r_g^G. It was already shown that $o_g^G(t) \leq m_g^G(t)$, and we therefore required that $f_g^G(t) \leq m_g^G(t)$ in Section 2.4.2. This case can arise in all systems, whether homogeneous or not, and even when $G = 1$. Therefore, in the discussion of the remaining three cases, we consider $\bar{f}_g^G(t)$ as a possible target for the definition of $f_g^G(t)$.

Case 2: Inhomogeneity

In some systems, while each group separately can always be provided with a share of $\bar{f}_g^G(t)$, this may not be possible for all groups simultaneously, because each group may need the fastest processors in order to have $o_g^G(t) = \bar{f}_g^G(t)$ (cf. (2.10)).

As an example, consider a system with $P = 3$, $c_1 = 2$, $c_2 = c_3 = 1$, with free job migration. There are three groups, and $r_1^G = r_2^G = 0.4$, $r_3^G = 0.2$. When each group has one job in the system, we have $\bar{f}_1^G(t) = \bar{f}_2^G(t) = 0.4$ and $\bar{f}_3^G(t) = 0.2$. The job of group 1 can be provided with a share of at least 0.4 by letting it run on processor 1, since the capacity of this processor is $0.5c$. The same holds for the job of group 2. However, the two jobs cannot be given a share of 0.4 each simultaneously, since $o_1^J(t) + o_2^J(t) \leq 0.75$ by (2.10). Probably the best we can do is let the two jobs of groups 1 and 2 share processors 1 and 2, and let job 3 run on processor 3. Then we have $o_1^J = o_2^J = 0.375$ and $o_3^J = 0.25$. Note that there is no point in decreasing o_3^J to 0.2 (r_3^G) in this example: the released capacity cannot be put to work sensibly for groups 1 and 2. (In fact, doing this would lead to capacity loss, as will be explained in Section 2.5.)

The case outlined above can only occur in heterogeneous systems with $G > 1$ and $P > 2$. For $G > 1$, this is obvious. In a system with $P = 2$, each group can always be provided with $\bar{f}_g^G(t)$, $g = 1, \ldots, G$ at all times. In such systems, we must comply with (2.20) for any possible job set. However, the only requirement that makes sense in (2.20) is

$$o_1^J \leq c_1/c,$$

where job 1 is the job requiring the highest share o_1^J. This requirement is met when a group g has only one job present, because in that case we have $\bar{f}_g^G(t) \leq c_1/c$. When more jobs of a group g are present, we have $\bar{f}_g^G(t) = r_g^G$. But now we can claim a fraction r_g^G on each of the two processors in order to provide group g with r_g^G, putting some of the jobs on processor 1, and some of them on processor 2.

Summarizing, in heterogeneous systems, even when job migration is free, a group cannot always be provided with a share of $\bar{f}_g^G(t)$, unless $P = 2$.

Case 3: Impossible Fit in the Absence of Job Migration

Without job migration, the jobs may not fit on the processors in such a way that $o_g^G(t) = \bar{f}_g^G(t)$, $g = 1, \ldots, G$, even though this would be possible on a multiprocessor.

As an example, consider a system with $P = 3$, $c_1 = 2$, $c_2 = c_3 = 1$, without job migration. There are three groups, $r_1^G = r_2^G = 0.375$, $r_3^G = 0.25$. There is one job of each group present in the system, numbered 1 to 3, each job having the same number of the group to which it belongs. Clearly, cases 1 and 2 do not apply, and $\bar{f}_g^G(t) = r_g^G$, $g = 1, 2, 3$.

Case	$P=1$	$P>1$			
		Job Migration		No Job Migration	
		HOM.	HET.	HOM.	HET.
1. Lack of Jobs	•	•	•	•	•
2. Inhomogeneity			$G>1, P>2$		$G>1, P>2$
3. Impossible Fit				$G>1$	$G>1$
4. Irrevocability				•	•

Table 2.1: Summary of the four cases affecting the definition of the feasible group share. A "•" means the case can arise.

Without job migration, there is no way in which we can assign the jobs to the processors such that the obtained shares equal the r_g^G's. Both job 1 and job 2 need to spend some time on processor 1, because the other processors do not have enough capacity. When there is free job migration, we can use the MJPPS policy described in Section 2.3.2. By setting

$$(w_{jp}) = \begin{pmatrix} 0.5 & 0.5 & 0 \\ 0.5 & 0.5 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

all groups obtain their required shares.

This case applies to all systems without job migration when $G > 1$. It arises at job *arrivals*.

Case 4: Irrevocability of Initial Placement due to Absence of Job Migration

When there is no job migration, the initial placement of a job on a processor is irrevocable. As a result, job departures may leave the system in a state in which it is impossible to provide a group with a specific share, although this could be achieved by rearranging (i.e., by migrating) the jobs among the processors.

As an example, consider the system of case 3, and assume job j has been assigned to processor j, $j = 1, 2, 3$. Then, the job of group 2 obtains a share (0.25) that is smaller than required (0.375). Assume job 1 is the first job to finish. On the departure of job 1, processor 1 is left idle, but since there is no job migration, we cannot put it to use for group 2. The assignment of job 2 to processor 2 is irrevocable. Note that since service times are unknown in advance, we could not have foreseen this situation.

As opposed to case 3, this case may arise at job *departures*, and may do so even when $G = 1$. Also, case 4 can be solved by a one-time redistribution of the jobs across the processors. This is not the case with case 3, where the jobs would have to be migrated continuously. Therefore, if job migration is available but expensive, it would help to solve case 4, but not case 3.

Summary of Cases

In Table 2.1, we summarize the four cases affecting the definition of the feasible group share.

2.4.4 The Definition of the Feasible Group Share

Throughout this thesis, we use the following definition for the feasible group share $f_g^{\mathrm{G}}(t)$:

$$f_g^{\mathrm{G}}(t) \triangleq \min\left(r_g^{\mathrm{G}}, \frac{|J_{g*}(t)|}{P}\right). \tag{2.26}$$

We now show why this choice is reasonable for all types of systems:

- uniprocessors,
- homogeneous systems with job migration,
- heterogeneous systems with job migration, and
- systems without job migration.

Uniprocessors

In uniprocessors, (2.26) reduces to

$$f_g^{\mathrm{G}}(t) = \begin{cases} 0 & \text{when } |J_{g*}(t)| = 0, \\ r_g^{\mathrm{G}} & \text{otherwise.} \end{cases} \tag{2.27}$$

In uniprocessors, only case 1 of Section 2.4.3 can arise. It is easily verified that with the JPPS policy introduced in Section 2.3.1, we can always enforce $o_g^{\mathrm{G}}(t) \geq f_g^{\mathrm{G}}(t)$ for all g and all t. Therefore, (2.27) is reasonable.

Homogeneous Systems with Job Migration

In homogeneous systems with job migration, only case 1 of Section 2.4.3 can arise, as in uniprocessors. One can easily verify that with the MJPPS policy introduced in Section 2.3.2, we can always guarantee that $o_g^{\mathrm{G}}(t) \geq f_g^{\mathrm{G}}(t)$ for all g and all t. Also note that in homogeneous systems, we always have

$$f_g^{\mathrm{G}}(t) = \bar{f}_g^{\mathrm{G}}(t).$$

Heterogeneous Systems with Job Migration

In heterogeneous systems with job migration, cases 1 and 2 of Section 2.4.3 can arise. This means we cannot always provide all groups with their $\bar{f}_g^{\mathrm{G}}(t)$ simultaneously. However, as we show now, we can always guarantee a share of $f_g^{\mathrm{G}}(t)$. Consider an arbitrary heterogeneous system and its homogeneous counterpart with the same total processing capacity and the same number of processors. We call the original heterogeneous system system A, and the homogeneous one system B. By (2.26), given an arbitrary set of jobs, the feasible group shares are equal in both systems. So what we must prove is that for any set of jobs, any set of service rates that can be achieved in system B, can also be achieved in system A. In system A, we have, for $p = 1, \ldots, P$,

$$c_1 + \cdots + c_p \;\geq\; \frac{p}{P}(c_1 + \cdots + c_P). \tag{2.28}$$

Because of (2.28), if (2.20) holds for system B, it certainly holds for system A. This implies we can guarantee $f_g^{\mathrm{G}}(t)$ to each group simultaneously in a heterogeneous system, and therefore (2.26) makes sense.

Systems without Job Migration

In systems without job migration, in general, all four cases of Section 2.4.3 can arise. However, in the definition of $f_g^{\mathrm{G}}(t)$ in (2.26), we have already taken into account cases 1 and 2, so we will concentrate on cases 3 and 4 here. Due to cases 3 and 4, it is generally not possible, in the absence of job migration, to *guarantee* that the system will always deliver an obtained group share equal to r_g^{G} when the number of jobs is large enough. We can only guarantee a minimum service rate of $r_g^{\mathrm{G}} c_P/c$, because in the worst case, all jobs of all groups end up running on the slowest processor. However, defining $f_g^{\mathrm{G}}(t) = r_g^{\mathrm{G}} c_P/c$ does not comply with the requirements given in Section 2.4.2, because $f_g^{\mathrm{G}}(t) < r_g^{\mathrm{G}}$ when $P > 1$, irrespective of $|J_{g*}(t)|$. There appears to be no general solution to this problem. In the remainder of this thesis, we will therefore employ (2.26) in all systems, whether homogeneous or heterogeneous, and whether job migration is present or not. We will have to keep in mind, however, that deviations between the obtained and feasible group shares cannot as a rule be avoided in systems without job migration.

The fact that feasible group shares can always be guaranteed in uniprocessors and multiprocessors, and not in distributed systems without job migration, makes share scheduling in the latter type of systems a challenge.

2.4.5 Partitionable Systems

Our definition of the feasible group share in (2.26) takes into account a number of cases that obstruct the delivery of shares to groups of jobs. However, in so-called ***partitionable systems***, the delivery of shares is actually easier than in generic systems, because the capacity of a subset of processors exactly matches the required service rate of a subset of groups. Note that the classification partitionable depends not only on the processor configuration, but on the definition of the required group shares as well. We will introduce partitionable systems, and present an example in which an alternative, more aggressive, definition of the feasible group share can be used.

Another reason for introducing partitionable systems is that many of the share-scheduling policies introduced in Chapters 4, 5, and 6 use the concept of partitioning the system in order to provide groups with their feasible shares.

Definition 2.1 *A system is called partitionable when a non-empty proper subset* Γ *of* $\{1,\ldots,G\}$ *and a non-empty proper subset* Π *of* $\{1,\ldots,P\}$ *exist such that*

$$\sum_{g\in\Gamma} r_g^{\mathrm{G}} = \sum_{p\in\Pi} \frac{c_p}{c}. \tag{2.29}$$

When a system is partitionable, we can split the system into two or more independent subsystems. These subsystems are smaller than the original one, and we can usually obtain a definition of $f_g^{\mathrm{G}}(t)$ in which a group will have to supply fewer jobs in order to have $f_g^{\mathrm{G}}(t) = r_g^{\mathrm{G}}$.

Definition 2.2 *A system is called fully partitionable when disjoint subsets* $\Pi_1,\ldots,\Pi_G$ *of* $\{1,\ldots,P\}$ *exist such that, for* $g = 1,\ldots,G$,

$$\sum_{p\in\Pi_g} \frac{c_p}{c} = r_g^{\mathrm{G}}. \tag{2.30}$$

In a fully partitionable system, one can assign each group its "own" processor subset.

Definition 2.3 *A system is called fully one-to-one partitionable when for each group g a different processor p_g exists such that, for $g = 1, \ldots, G$,*

$$\frac{c_{p_g}}{c} = r_g^{\mathrm{G}}. \tag{2.31}$$

In a fully one-to-one partitionable system, one can assign each group its "own" processor.

As an example of the advantages of a partitionable system, consider a system with $P = 3$, $c_1 = 2$, $c_2 = c_3 = 1$, $G = 3$, $r_1^{\mathrm{G}} = 0.5$, and $r_2^{\mathrm{G}} = r_3^{\mathrm{G}} = 0.25$. Then, by (2.26),

$$f_1^{\mathrm{G}}(t) = \begin{cases} 0, & \text{when } |J_{1*}(t)| = 0, \\ 0.33, & \text{when } |J_{1*}(t)| = 1, \\ 0.5, & \text{when } |J_{1*}(t)| \geq 2, \end{cases} \tag{2.32}$$

$$f_2^{\mathrm{G}}(t) = \begin{cases} 0, & \text{when } |J_{2*}(t)| = 0, \\ 0.25, & \text{when } |J_{2*}(t)| \geq 1, \end{cases} \tag{2.33}$$

and

$$f_3^{\mathrm{G}}(t) = \begin{cases} 0, & \text{when } |J_{3*}(t)| = 0, \\ 0.25, & \text{when } |J_{3*}(t)| \geq 1. \end{cases} \tag{2.34}$$

However, the system is fully one-to-one partitionable, because $r_1^{\mathrm{G}} = c_1/c$, $r_2^{\mathrm{G}} = c_2/c$, and $r_3^{\mathrm{G}} = c_3/c$. Therefore, we can split the system by dedicating processor p to group p, $p = 1, 2, 3$. Using (2.26) in each subsystem (but still relating the feasible shares to the whole system) gives

$$f_1^{\mathrm{G}}(t) = \begin{cases} 0, & \text{when } |J_{1*}(t)| = 0, \\ 0.5, & \text{when } |J_{1*}(t)| \geq 1, \end{cases} \tag{2.35}$$

$$f_2^{\mathrm{G}}(t) = \begin{cases} 0, & \text{when } |J_{2*}(t)| = 0, \\ 0.25, & \text{when } |J_{2*}(t)| \geq 1, \end{cases} \tag{2.36}$$

$$f_3^{\mathrm{G}}(t) = \begin{cases} 0, & \text{when } |J_{3*}(t)| = 0, \\ 0.25, & \text{when } |J_{3*}(t)| \geq 1. \end{cases} \tag{2.37}$$

The latter definition of the $f_g^{\mathrm{G}}(t)$ is preferable over the definition in the unpartitioned system since in the partitioned system, group 1 only has to supply one job in order to have $f_g^{\mathrm{G}}(t) = r_g^{\mathrm{G}}$, instead of two jobs in the unpartitioned system.

An undesired side effect of partitioning is that it may cause unfairness among groups. For instance, consider the previous example but with only two groups ($G = 2$) and $r_1^{\mathrm{G}} = r_2^{\mathrm{G}} = 0.5$. In this case, $r_1^{\mathrm{G}} = c_1/c$ and $r_2^{\mathrm{G}} = (c_2 + c_3)/c$, and hence the system is still partitionable. However, when we split the system by dedicating processor 1 to group 1 and processors 2 and 3 to group 2, group 1 is obviously preferred over the other, although its required group share is equal to that of group 2. Clearly, this violates requirement 3 of Section 2.4.2.

2.4.6 Performance Measures for the Delivery of Feasible Group Shares

In order to evaluate a policy's ability to provide feasible shares, we will define performance measures for the delivery of feasible group shares. A suitable measure of the deviation between feasible and obtained shares for a group g is the *group-share deviation* $\Delta_g^G(t)$ *of group* g *at time* t, defined as

$$\Delta_g^G(t) \triangleq \frac{f_g^G(t) - o_g^G(t)}{r_g^G}. \tag{2.38}$$

We use relative differences here since we feel that a shortage of 0.1 is worse for a group with $r_g^G = 0.15$, than it is for one with $r_g^G = 0.80$. Clearly, $\Delta_g^G(t)$ should be as small as possible. When $\Delta_g^G(t) < 0$, group g obtains more service from the system than its feasible share.

Since we usually consider stochastic workloads, we define the *expected group-share deviation* Δ_g^G *of group* g as

$$\Delta_g^G \triangleq \lim_{t\to\infty} \mathbf{E}\left[\Delta_g^G(t)\right]. \tag{2.39}$$

In order to have a single performance measure for the delivery of group shares, we define the *maximum expected group-share deviation* Δ^G as

$$\Delta^G \triangleq \max_g \left\{\Delta_g^G\right\}. \tag{2.40}$$

Usually, we will refer to Δ^G as the *group-share deviation.*

2.4.7 Discussion

In the previous sections, we have given the definitions of the feasible group share and of the performance measures for the delivery of the feasible group share to a single group, and to all groups together. Of course, there are alternative definitions that may be more appropriate for special architectures or specific workloads. We proposed such an alternative in Section 2.4.4 for partitionable systems. Also, in heterogeneous systems with $P = 2$, case 2 of the current section does not apply, and one could better define $f_g^G(t) = \bar{f}_g^G(t)$. Clearly, both alternatives work only in specific systems. One could also consider the use of a history mechanism keeping track of share deviations in the (recent) past, and temporarily increase or decrease the feasible group share in order to compensate for these deviations. Finally, since $\Delta_g^G(t)$ is zero when there are no jobs present of group g, it makes sense to only consider busy periods of group g, because otherwise, the idle periods of group g compensate for share deviations during busy periods. Moreover, the user or the users associated with a group will not be interested in the delivery of group shares when there are no jobs present of that group. However, a disadvantage of confining Δ_g^G to busy periods in (2.39) is that it makes the definition of the group-share deviation more complex. Also, we found that it did not have an influence on the qualitative comparisons between different share-scheduling policies.

2.5 Objective II: Minimization of Capacity Loss

Our second objective is to utilize as much of the system capacity as possible. This means setting the target for the total obtained share $o^{\mathrm{T}}(t)$.

2.5.1 Total Feasible Share

The *total feasible share* $f^{\mathrm{T}}(t)$ is defined as

$$f^{\mathrm{T}}(t) \triangleq m^{\mathrm{T}}(t). \tag{2.41}$$

In systems with job migration, we can guarantee that $o^{\mathrm{T}}(t) = f^{\mathrm{T}}(t)$ at any time, by always employing as many of the fastest processors as possible. When a processor becomes idle due to a job departure while a slower processor serves one or more jobs, a job is migrated from the slow to the fast processor. However, in systems without job migration, this is not always possible. For instance, consider a two-processor system with $c_1 > c_2$. When one job is being served by processor 1, another job should be placed on processor 2 to ensure that $o^{\mathrm{T}}(t) = f^{\mathrm{T}}(t)$. When the job on processor 1 terminates, we cannot migrate the other job from processor 2 to processor 1, which would be necessary in order to make $o^{\mathrm{T}}(t) = f^{\mathrm{T}}(t)$. But even if the system is homogeneous, we may have $o^{\mathrm{T}}(t) < f^{\mathrm{T}}(t)$, for instance when a processor goes idle while at another processor two or more jobs are present.

2.5.2 Performance Measures for the Minimization of Capacity Loss

We define the *total share deviation* (also named the *capacity loss*) $\Delta^{\mathrm{T}}(t)$ *at time* t as

$$\Delta^{\mathrm{T}}(t) \triangleq f^{\mathrm{T}}(t) - o^{\mathrm{T}}(t). \tag{2.42}$$

Analogous to Section 2.4, we define our performance measure for minimization of capacity loss, the *expected capacity loss* Δ^{T}, as

$$\Delta^{\mathrm{T}} \triangleq \lim_{t\to\infty} \mathbf{E}\left[\Delta^{\mathrm{T}}(t)\right]. \tag{2.43}$$

The objective of the minimization of capacity loss is identical to load sharing in distributed systems (cf. Section 1.3.3): preventing processors from being unnecessarily idle. Usually, we will refer to Δ^{T} as the *capacity loss*.

2.6 Objective III: Internal Fairness

The final objective of share scheduling in this thesis is to distribute the obtained group share equally among the jobs of a group. We therefore set a target for the obtained job share $o_j^{\mathrm{J}}(t)$.

2.6.1 Feasible Job Share

We define the *feasible job share* $f_g^{\mathrm{J}}(t)$ *for any job* j *of group* g *at time* t as:

$$f_g^{\mathrm{J}}(t) \triangleq \frac{o_g^{\mathrm{G}}(t)}{|J_{g*}(t)|}\ , \text{ when } |J_{g*}(t)| > 0. \tag{2.44}$$

Since we are only interested in the *differences* between the obtained job shares of jobs within a group, and *not* in the actual values of these shares, the feasible job share is simply the obtained (and not the *feasible*) group share divided by the number of jobs present.

2.6.2 Performance Measures for Internal Fairness

We define the *job-share deviation* $\Delta_g^{\mathrm{J}}(t)$ *of group* g *at time* t as

$$\Delta_g^{\mathrm{J}}(t) \triangleq \begin{cases} 0 & \text{when } o_g^{\mathrm{G}}(t) = 0, \\ \max_{j \in J_{g*}(t)} \left\{ \frac{f_g^{\mathrm{J}}(t) - o_j^{\mathrm{J}}(t)}{f_g^{\mathrm{J}}(t)} \right\} & \text{otherwise.} \end{cases} \tag{2.45}$$

The objective of internal fairness is to minimize the job-share deviation of each group. Obviously,

$$0 \leq \Delta_g^{\mathrm{J}}(t) \leq 1, \tag{2.46}$$

because, by (2.45),

$$0 \leq \min_{j \in J_{g*}(t)} \left\{ o_j^{\mathrm{J}}(t) \right\} \leq f_g^{\mathrm{J}}(t).$$

We have $\Delta_g^{\mathrm{J}}(t) = 0$ when none of the jobs of group g is in service. This sometimes happens, for instance, in FCFS queues. We define the *expected job-share deviation* Δ_g^{J} *of group* g as

$$\Delta_g^{\mathrm{J}} \triangleq \lim_{t \to \infty} \mathrm{E}\left[\Delta_g^{\mathrm{J}}(t) \right]. \tag{2.47}$$

Our performance measure for internal fairness is the *maximum expected job-share deviation* Δ^{J}, which we define as

$$\Delta^{\mathrm{J}} \triangleq \max_g \left\{ \Delta_g^{\mathrm{J}} \right\}. \tag{2.48}$$

We will usually refer to Δ^{J} as the *job-share deviation.*

2.7 Summary

In this chapter, we have proposed the following objectives of share scheduling in distributed systems and in multiprocessors:

1. the delivery of the feasible group shares, setting the targets for the obtained group shares $o_g^{\mathrm{G}}(t)$,

2. the minimization of capacity loss, setting the target for the total obtained share $o^{\mathrm{T}}(t)$, and

3. internal fairness, setting the targets for the obtained job shares $o_j^J(t)$.

We have shown that in multiprocessors, whether homogeneous or heterogeneous, all targets can be met by means of the MJPPS policy. This means that distributed systems, in which job migration is absent or at least expensive, are the more interesting systems to study.

It goes without saying that our share-scheduling objectives differ substantially from traditional objectives described in the literature, such as the minimization of the average job response time. Nevertheless, there are some similarities. For instance, the load-sharing and load-balancing policies described in Chapter 1 are prominent candidates for our objective of minimization of capacity loss.

Chapter 3

Share-Scheduling Performance of Common Uniprocessor Scheduling Policies

3.1 Introduction

In Chapter 2 we presented the objectives and defined performance measures for share-scheduling policies. In the present chapter we will evaluate the share-scheduling performance of some important uniprocessor scheduling policies. For some policies an exact analysis can be given, for others we use simulation. Studying uniprocessor scheduling policies is essential for the purpose of this thesis, because in distributed systems, one depends on the local scheduling policy for the delivery of shares to jobs.

All policies considered in this chapter are *work conserving*, i.e., they employ the full processor's capacity to serve one or more jobs present in the queue. In other words, the *total obtained share* $o^{\mathrm{T}}(t)$ equals 1 when there are jobs present in the system, i.e., when $|J(t)| > 0$. The *total feasible share* $f^{\mathrm{T}}(t)$ equals 1 when $|J(t)| > 0$, and is zero otherwise. As a result, the capacity loss $\Delta^{\mathrm{T}}(t)$ is always zero, since it is defined as $f^{\mathrm{T}}(t) - o^{\mathrm{T}}(t)$ in (2.42). Therefore, capacity loss will not be considered in this chapter. Also, we put the processor capacity c equal to 1, without loss of generality. (Recall that the required, obtained, and feasible shares were all defined relative to the total capacity c.)

An overview of our taxonomy of uniprocessor scheduling policies is given in Figure 3.1. The policies we will study are also shown; their descriptions are given later. We distinguish the following three classes of uniprocessor scheduling policies: *non-preemptive*, *preemptive*, and *processor-sharing* policies. In non-preemptive and preemptive policies, only one job is served at a time. In non-preemptive policies, jobs are served until completion, while in preemptive policies, the processor may interrupt the service of a job in favor of another job. Later on, the interrupted job may resume execution at the point where it was interrupted. (In fact, such policies are called *preemptive/resume*. In *preemptive/restart* policies, a job is restarted from the beginning. Such policies are not considered in this thesis.) We assume that the overhead due to preempting and resuming jobs is negligible compared to the service time of jobs.

In processor-sharing policies, a processor serves multiple jobs simultaneously, possibly at different rates. Such policies are often used to model round-robin policies, in which jobs

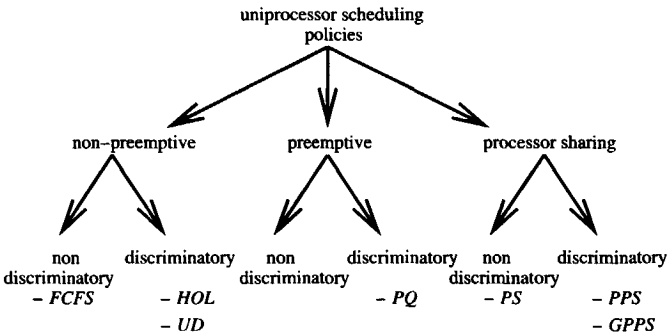


Figure 3.1: A taxonomy of uniprocessor scheduling policies.

receive service one at a time, each for a small amount of time called a *quantum*. When the quantum length is small compared to the service times of jobs and the context-switching overhead is low compared to a quantum, round-robin policies can be approximated by processor-sharing policies.

Another distinction is that between *non-discriminatory* and *discriminatory* policies. Non-discriminatory policies do not take into account the group to which a job belongs while discriminatory policies do. It is obvious that non-discriminatory policies are not expected to perform very well with respect to the delivery of feasible group shares. In this chapter, we study the improvement that can be achieved by using discriminatory policies over non-discriminatory ones.

We will briefly review the workload models of Section 1.2, and introduce some extra notation. We consider both non-permanent and permanent jobs in this chapter. There are $G > 1$ groups, numbered $1, \ldots, G$. The service times of jobs of group g are exponentially distributed with mean $1/\mu_g$. This is a special case of the model introduced in 1.2, where arbitrary service-time distributions were allowed. Non-permanent jobs arrive according to G independent Poisson processes with intensities $\lambda_1, \ldots, \lambda_G$. We define the *total arrival rate* λ as

$$\lambda \stackrel{\Delta}{=} \sum_{g=1}^{G} \lambda_g, \tag{3.1}$$

the *load* ρ_g *due to jobs of group* g as

$$\rho_g = \lambda_g/\mu_g, \tag{3.2}$$

and the *total load* ρ as

$$\rho \stackrel{\Delta}{=} \sum_{g} \lambda_g/\mu_g. \tag{3.3}$$

We define the $|\cdot|$-operator on a vector as the sum of its elements. The (constant) number of permanent jobs of group g is denoted by N_g^P. The number of jobs (both permanent and non-permanent) of group g in the system at time t is denoted by $N_g(t)$, so $N_g(t) = |J_{g*}(t)|$. We define

$$\boldsymbol{N}(t) \stackrel{\Delta}{=} (N_1(t), \ldots, N_G(t)), \tag{3.4}$$

$$\boldsymbol{N}^{\mathrm{P}} \triangleq (N_1^{\mathrm{P}}, \ldots, N_G^{\mathrm{P}}), \tag{3.5}$$

$$N(t) \triangleq \sum_{g=1}^{G} N_g(t), \tag{3.6}$$

$$N^{\mathrm{P}} \triangleq \sum_{g=1}^{G} N_g^{\mathrm{P}}. \tag{3.7}$$

Since we will use stochastic interarrival-time and service-time distributions, we use the notation

$$\mathbf{P}\left[\boldsymbol{X} = \boldsymbol{x}\right]$$

as a shorthand for

$$\lim_{t\to\infty} \mathbf{P}\left[\boldsymbol{X}(t) = \boldsymbol{x}\right],$$

for a real-valued multidimensional stochastic process $\boldsymbol{X}(t)$ and a real vector $\boldsymbol{x}$, provided the limit exists. We define the probability-generating function (pgf) F of the numbers of non-permanent jobs as

$$F(z_1, \ldots, z_G) \triangleq \sum_{\boldsymbol{n}} \mathbf{P}\left[\boldsymbol{N} = \boldsymbol{N}^{\mathrm{P}} + \boldsymbol{n}\right] z_1^{n_1} \cdots z_G^{n_G}, \tag{3.8}$$

with $\boldsymbol{n} = (n_1, \ldots, n_G)$, $n_g \geq 0$, $g = 1, \ldots, G$. Finally, we define

$$n \triangleq |\boldsymbol{n}|, \tag{3.9}$$

for any vector $\boldsymbol{n}$.

We discuss the non-preemptive, preemptive, and processor-sharing policies in Sections 3.2, 3.3, and 3.4, respectively. Our interest is in the delivery of feasible group shares and in internal fairness. Therefore, for each of the policies considered, we will try to derive expressions for Δ^{G} and Δ^{J}. A performance comparison of the policies is given in Section 3.5.

3.2 Non-Preemptive Policies

We discuss three non-preemptive policies: the First-Come First-Served policy, which is non-discriminatory, and the Head-of-the-Line and Up-Down policies, which are discriminatory.

3.2.1 First-Come First-Served

The First-Come First-Served (FCFS) policy, also known as the First-In First-Out (FIFO) policy, is probably the policy most studied in the literature (see, e.g., [1, 16]). Jobs are appended to the tail of the queue upon arrival. When a job terminates, the job at the head of the queue (if any) is served next.

Intuitively, with FCFS, the delivery of feasible group shares is unacceptably sensitive to the interarrival-time and the service-time distributions. In general, a group can obtain a larger share of the system when more jobs of that group are submitted to the system, or when the service times of that group are increased. Also, there is no internal fairness:

only the job currently being served contributes to the obtained group share. The following theorem gives the steady-state distribution of the joint number of jobs in the system, when arrivals are Poisson, and when service times are exponential with *identical means* for all groups. One or more permanent jobs may be present. The theorem is used to derive expressions for our share-scheduling performance measures (Δ_g^{G} and Δ_g^{J}) later in this section.

Theorem 3.1 *In a FCFS uniprocessor system with exponential service times with identical means, if $\rho < 1$, the steady-state distribution $\mathbf{P}\left[\boldsymbol{N} = \boldsymbol{N}^{\mathrm{P}} + \boldsymbol{n}\right]$, $\boldsymbol{n} = (n_1, \ldots, n_G)$, is given by*

$$\mathbf{P}\left[\boldsymbol{N} = \boldsymbol{N}^{\mathrm{P}} + \boldsymbol{n}\right] = (1-\rho)^{N^{\mathrm{P}}+1} \cdot \frac{(N^{\mathrm{P}} + n_1 + \cdots + n_G)!}{N^{\mathrm{P}}! \cdot n_1! \cdots n_G!} \cdot \rho_1^{n_1} \cdots \rho_G^{n_G}. \tag{3.10}$$

The pgf F of the numbers of non-permanent jobs is given by

$$F(z_1, \ldots, z_G) = \left(\frac{1-\rho}{1-\rho_1 z_1 - \cdots - \rho_G z_G}\right)^{N^{\mathrm{P}}+1}. \tag{3.11}$$

PROOF: When there are no permanent jobs, i.e., when $N^{\mathrm{P}} = 0$, the steady-state distribution of the number of jobs in an M/M/1 FCFS queue is (see, e.g., [1, page 263])

$$\mathbf{P}\left[N = n\right] = (1-\rho)\rho^n, \; n = 0, 1, 2, \ldots, \tag{3.12}$$

with pgf

$$G(z) \stackrel{\Delta}{=} \sum_{n=0}^{\infty} \mathbf{P}\left[N = n\right] z^n = \frac{1-\rho}{1-\rho z}. \tag{3.13}$$

It is well known [8], that in the presence in an M/M/1 queue of N^{P} permanent jobs with exponential service times with mean identical to that of the non-permanent jobs, the pgf $H(z)$ of the number of non-permanent jobs is the $(N^{\mathrm{P}}+1)$-fold convolution of that of the queue without permanent jobs, i.e.,

$$H(z) = \left(\frac{1-\rho}{1-\rho z}\right)^{N^{\mathrm{P}}+1}. \tag{3.14}$$

Clearly, this still holds in our case. Since the probability that a non-permanent job belongs to group g equals ρ_g/ρ, we have

$$\mathbf{P}\left[\boldsymbol{N} = \boldsymbol{N}^{\mathrm{P}} + \boldsymbol{n}\right] = \mathbf{P}\left[N = N^{\mathrm{P}} + n\right] \frac{n!}{n_1! \cdots n_G!} \cdot \left(\frac{\rho_1}{\rho}\right)^{n_1} \cdots \left(\frac{\rho_G}{\rho}\right)^{n_G}. \tag{3.15}$$

Hence,

$$\begin{aligned}
F(z_1, \ldots, z_G) &= \sum_{\boldsymbol{n}} \mathbf{P}\left[\boldsymbol{N} = \boldsymbol{N}^{\mathrm{P}} + \boldsymbol{n}\right] z_1^{n_1} \cdots z_G^{n_G} \\
&= \sum_{n=0}^{\infty} \mathbf{P}\left[N = N^{\mathrm{P}} + n\right] \sum_{\sum n_g = n} \frac{n!}{n_1! \cdots n_G!} \cdot \left(\frac{\rho_1 z_1}{\rho}\right)^{n_1} \cdots \left(\frac{\rho_G z_G}{\rho}\right)^{n_G} \\
&= \sum_{n=0}^{\infty} \mathbf{P}\left[N = N^{\mathrm{P}} + n\right] \left(\frac{\rho_1 z_1 + \cdots + \rho_G z_G}{\rho}\right)^n \\
&= H\left(\frac{\rho_1 z_1 + \cdots + \rho_G z_G}{\rho}\right),
\end{aligned}$$

from which (3.11) follows. The probability distribution function (3.10) is the inverse transform of (3.11). Its validity can be checked with (3.8). □

Since the pgf of the joint number of non-permanent jobs is known for FCFS, we can derive expressions for the expected group-share deviations Δ_g^{G}, $g = 1, \ldots, G$.

Theorem 3.2 *In a FCFS uniprocessor system with exponential service times with identical means, if $\rho < 1$, the expected group-share deviation Δ_g^{G} of group g is given by*

$$\Delta_g^{\mathrm{G}} = \begin{cases} 1 - \left(\frac{1-\rho}{1-(\rho-\rho_g)}\right)^{N^{\mathrm{P}}+1} - \frac{\rho_g}{r_g^{\mathrm{G}}}, & N_g^{\mathrm{P}} = 0, \\ 1 - \frac{\rho_g + \frac{N_g^{\mathrm{P}}}{N^{\mathrm{P}}}(1-\rho)}{r_g^{\mathrm{G}}}, & N_g^{\mathrm{P}} > 0. \end{cases} \tag{3.16}$$

PROOF: From the definitions (2.27) and (2.38) we have

$$\begin{aligned} \Delta_g^{\mathrm{G}} &= \lim_{t\to\infty} \mathbf{E}\left[\frac{f_g^{\mathrm{G}}(t) - o_g^{\mathrm{G}}(t)}{r_g^{\mathrm{G}}}\right] \\ &= \frac{1}{r_g^{\mathrm{G}}}\left\{\mathbf{P}\left[N_g > 0\right] r_g^{\mathrm{G}} - \mathbf{E}\left[o_g^{\mathrm{G}}\right]\right\}. \end{aligned} \tag{3.17}$$

From (3.11) we have

$$\mathbf{P}\left[N_g > 0\right] = \begin{cases} 1 - F(1,\ldots,1,0,1,\ldots,1) = 1 - \left(\frac{1-\rho}{1-(\rho-\rho_g)}\right)^{N^{\mathrm{P}}+1}, & N_g^{\mathrm{P}} = 0, \\ 1, & N_g^{\mathrm{P}} > 0. \end{cases} \tag{3.18}$$

Obviously, when a job of group g is being served, $o_g^{\mathrm{G}}(t)$ equals 1. When not, $o_g^{\mathrm{G}}(t)$ equals 0. The server spends a fraction ρ of time on serving non-permanent jobs, and a fraction $(1-\rho)$ on serving permanent jobs, if present. Since the probability that a non-permanent job in service belongs to group g is ρ_g/ρ, the obtained group share of group g due to non-permanent jobs equals ρ_g. The service-time means for the permanent jobs are equal, so the obtained group share of group g due to permanent jobs is $N_g^{\mathrm{P}}/N^{\mathrm{P}} \times (1-\rho)$. Hence,

$$\mathbf{E}\left[o_g^{\mathrm{G}}\right] = \begin{cases} \rho_g, & N_g^{\mathrm{P}} = 0, \\ \rho_g + \frac{N_g^{\mathrm{P}}}{N^{\mathrm{P}}}(1-\rho), & N_g^{\mathrm{P}} > 0. \end{cases} \tag{3.19}$$

Equation 3.16 now follows from substitution of (3.18) and (3.19) into (3.17). □

We will analyze the behavior of FCFS when there are no permanent jobs in the system, i.e., $N^{\mathrm{P}} = 0$. We then have from (3.16) for the expected group-share deviation Δ_g^{G} of group g,

$$\Delta_g^{\mathrm{G}} = \rho_g \left\{\frac{1}{1-(\rho-\rho_g)} - \frac{1}{r_g^{\mathrm{G}}}\right\}, \tag{3.20}$$

and,

$$\Delta_g^{\mathrm{G}} = 0 \Longleftrightarrow \rho_g = r_g^{\mathrm{G}} + (1-\rho). \tag{3.21}$$

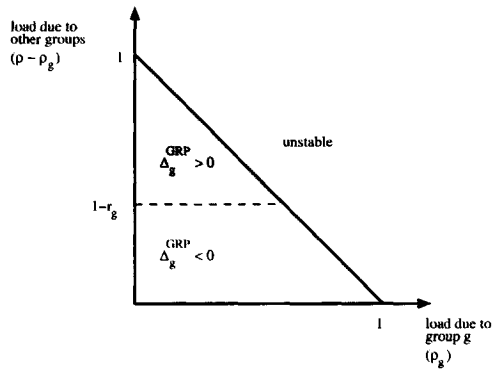


Figure 3.2: The behavior of the expected group-share deviation Δ_g^G for FCFS when there are no permanent jobs.

The result found in (3.20) is somewhat counterintuitive. Compliance with share-scheduling objectives requires a value of Δ_g^G as small as possible (preferably, negative). Equation (3.20) states that whether or not $\Delta_g^G \leq 0$ is *only* affected by the loads $\rho - \rho_g$ due to other groups than g, and *not* by ρ_g. Moreover, if $\Delta_g^G > 0$, increasing ρ_g (while the other group loads remain constant) only has the undesired effect of increasing Δ_g^G even further, despite the fact that according to (3.19), group g's obtained share increases. Only when $\Delta_g^G < 0$, a group can further decrease Δ_g^G by increasing ρ_g. Clearly, FCFS is not a very attractive policy for share scheduling.

When there are no permanent jobs, the light-traffic and heavy-traffic cases follow directly from (3.20). Clearly, it holds that

$$\lim_{\rho \downarrow 0} \Delta_g^G = 0, \ g = 1, \ldots, G. \tag{3.22}$$

Also, when $\rho \uparrow 1$ in such a way that the limits

$$\lim_{\rho \uparrow 1} \rho_g = \bar{\rho}_g, \ g = 1, \ldots, G,$$

exist, we have

$$\lim_{\rho \uparrow 1} \Delta_g^G = 1 - \frac{\bar{\rho}_g}{r_g^G}, \ g = 1, \ldots, G. \tag{3.23}$$

It is easily verified that when $\rho \uparrow 1$, the only way to keep $\Delta_g^G \leq 0$ for all g, is to have $\bar{\rho}_g = r_g^G$. One way of achieving this is by setting $\rho_g = r_g^G \rho$.

An interesting question is under which conditions $\Delta_g^G \leq 0$, $g = 1, \ldots, G$ (and thus $\Delta^G \leq 0$) for fixed $\rho < 1$. (Recall that $\Delta_g^G \leq 0$ means that group g's share is large enough.) In Figure 3.2, we show the regions in which $\Delta_g^G < 0$ and $\Delta_g^G > 0$, respectively. From

(3.20) we find

$$\Delta_g^{\mathrm{G}} \leq 0 \Longleftrightarrow \rho_g \geq r_g^{\mathrm{G}} - (1-\rho),\ g = 1, \ldots, G. \tag{3.24}$$

It is easily found that these requirements can be met for any value of ρ in the interval $(0,1)$, for instance by putting $\rho_g = r_g^{\mathrm{G}}\rho$. We then have

$$\Delta_g^{\mathrm{G}} = -\frac{(1-\rho)\rho(1-r_g^{\mathrm{G}})}{1-\rho(1-r_g^{\mathrm{G}})} \leq 0. \tag{3.25}$$

When $G = 2$, (3.24) is equivalent to $\rho_g \leq r_g^{\mathrm{G}}$, $g = 1, 2$. This condition for $g = 1, \ldots, G$ is only *sufficient* when $G > 2$, as can be seen directly from (3.20). Also note that when

$$\rho \leq 1 - \max_g \left\{ r_g^{\mathrm{G}} \right\}$$

by (3.20) we always have

$$\Delta^{\mathrm{G}} \leq 0$$

for arbitrary values of $\rho_1, \ldots, \rho_G$.

Our final theorem gives an expression for the expected job-share deviation Δ_g^{J}.

Theorem 3.3 *In a FCFS uniprocessor system with exponential service times with identical means, if $\rho < 1$, the expected job-share deviation Δ_g^{J} of group g is given by*

$$\Delta_g^{\mathrm{J}} = \begin{cases} \rho_g \left\{ 1 - \left(\frac{1-\rho}{1-(\rho-\rho_g)} \right)^{N^{\mathrm{P}}+1} \right\}, & N_g^{\mathrm{P}} = 0, \\ \rho_g + \frac{1-\rho}{N^{\mathrm{P}}} \left\{ 1 - \left(\frac{1-\rho}{1-(\rho-\rho_g)} \right)^{N^{\mathrm{P}}} \right\}, & N_g^{\mathrm{P}} = 1, \\ \rho_g + \frac{N_g^{\mathrm{P}}}{N^{\mathrm{P}}}(1-\rho), & N_g^{\mathrm{P}} \geq 2. \end{cases} \tag{3.26}$$

PROOF: From (2.45) we have $\Delta_g^{\mathrm{J}}(t) = 1$ when a job of group g is being served and $N_g(t) \geq 2$, and $\Delta_g^{\mathrm{J}}(t) = 0$ otherwise. In other words,

$$\Delta_g^{\mathrm{J}} = \mathbf{P}\left[\text{job of group } g \text{ in service} | N_g(t) \geq 2\right]. \tag{3.27}$$

In all three cases in (3.26) the server spends a fraction ρ_g of time serving non-permanent jobs of group g, and if $N^{\mathrm{P}} > 0$, a fraction $N_g^{\mathrm{P}}/N^{\mathrm{P}} \times (1-\rho)$ of time serving permanent jobs of group g.

When $N_g^{\mathrm{P}} = 0$, we can only have $\Delta_g^{\mathrm{J}}(t) = 1$ when a non-permanent job of group g is in service while another one is waiting in the queue. In order to derive the probability $\mathbf{P}[\Upsilon]$ that *no* such job is waiting at the start of the service to an arbitrary job of group g, we use the distribution function $F_R(\tau) = \mathbf{P}[R \leq \tau]$ of the response time R of a non-permanent job, for which the following result holds [8], for $\mathrm{Re}\ \gamma \geq 0$,

$$\mathbf{E}\left[e^{-\gamma R}\right] = \int_0^\infty e^{-\gamma\tau} dF_R(\tau) = \left(\frac{\mu - \lambda}{\mu - \lambda + \gamma} \right)^{N^{\mathrm{P}}+1}. \tag{3.28}$$

The waiting time W of a non-permanent job is given by $W = R - S$ where S is its service time, for which the Laplace-Stieltjes transform (LST) is

$$\mathbf{E}\left[e^{-\gamma S}\right] = \frac{\mu}{\mu+\gamma}. \tag{3.29}$$

For the waiting time, we therefore obtain

$$\mathbf{E}\left[e^{-\gamma W}\right] = \left(\frac{\mu-\lambda}{\mu-\lambda+\gamma}\right)^{N^{\mathrm{P}}+1} \cdot \frac{\mu+\gamma}{\mu}. \tag{3.30}$$

Since the probability that no jobs of group g arrive during a period of length τ is $e^{-\lambda_g \tau}$, we have, writing $F_W(\cdot)$ for the distribution function of W,

$$\begin{aligned} \mathbf{P}[\Upsilon] &= \int_0^\infty e^{-\lambda_g \tau} dF_W(\tau) = \mathbf{E}\left[e^{-\lambda_g W}\right] \\ &= \left(\frac{1-\rho}{1-(\rho-\rho_g)}\right)^{N^{\mathrm{P}}+1} \cdot (1+\rho_g). \end{aligned} \tag{3.31}$$

So, with probability $1 - \mathbf{P}[\Upsilon]$, service to a job of group g starts while other jobs of the group are present, and so $\Delta_g^{\mathrm{J}}(t) = 0$ during the entire service period, which is of average length $1/\mu$. With probability $\mathbf{P}[\Upsilon]$, service to a job of group g starts with no other jobs of the group being present. When no jobs of group g arrive until its service completion, $\Delta_g^{\mathrm{J}}(t) = 0$ during its entire service, otherwise $\Delta_g^{\mathrm{J}}(t) = 0$ until the first arrival of a job of group g, and $\Delta_g^{\mathrm{J}}(t) = 1$ after this arrival. As the probability of an arrival of a job of group g during the service time of a job is $\lambda_g/(\lambda_g+\mu)$, and the average residual service time after the arrival is still $1/\mu$, we find

$$\Delta_g^{\mathrm{J}} = \lambda_g \left((1 - \mathbf{P}[\Upsilon]) \cdot \frac{1}{\mu} + \mathbf{P}[\Upsilon] \cdot \frac{\lambda_g}{\lambda_g+\mu} \cdot \frac{1}{\mu} \right), \tag{3.32}$$

from which (3.26) follows for $N_g^{\mathrm{P}} = 0$.

When $N_g^{\mathrm{P}} = 1$ and a non-permanent job of group g is in service, we clearly have $N_g(t) \geq 2$ and therefore, $\Delta_g^{\mathrm{J}}(t) = 1$ a fraction ρ_g of the time. When the single permanent job of group g is in service, we have $\Delta_g^{\mathrm{J}}(t) = 1$ if a non-permanent job is in the queue, and $\Delta_g^{\mathrm{J}}(t) = 0$ otherwise. We first derive an expression for the probability $\mathbf{P}[\Omega]$ that *no* other job of group g is present at the start of this service. The stationary distribution of the number of non-permanent jobs in a gap between two permanent jobs equals that of the M/M/1 queue without permanent jobs [8], and therefore, the probability that there are no jobs of group g in such a gap is given by $(1-\rho)/(1-(\rho-\rho_g))$, cf. (3.11). For the $N^{\mathrm{P}} - 1$ gaps together, this probability is

$$\left(\frac{1-\rho}{1-(\rho-\rho_g)}\right)^{N^{\mathrm{P}}-1}. \tag{3.33}$$

Any (non-permanent) jobs behind the last permanent job have arrived during the service of *all* non-permanent jobs in a gap. The probability that no jobs of group g arrive during the service time of a *single* non-permanent job equals

$$\int_0^\infty e^{-\lambda_g \tau} \mu e^{-\mu\tau} d\tau = \frac{\mu}{\mu+\lambda_g} = \frac{1}{1+\rho_g}. \tag{3.34}$$

We therefore have,

$$\begin{aligned}\mathbf{P}[\Omega] &= \left(\frac{1-\rho}{1-(\rho-\rho_g)}\right)^{N^{\mathrm{P}}-1} \sum_{n=0}^{\infty}(1-\rho)\rho^n\left(\frac{1}{1+\rho_g}\right)^n \\ &= \left(\frac{1-\rho}{1-(\rho-\rho_g)}\right)^{N^{\mathrm{P}}}(1+\rho_g). \end{aligned} \tag{3.35}$$

So, with probability $1-\mathbf{P}[\Omega]$, $\Delta_g^{\mathrm{J}}(t)=1$ during the entire service of the permanent job of group g, and with probability $\mathbf{P}[\Omega]$, service to this job starts with no other jobs of the group being present. Because the average number of times per time unit that the permanent job of group g is executed is equal to $(1-\rho)\mu/N^{\mathrm{P}}$, we find, using the same arguments as for the case $N^{\mathrm{P}}=0$, that

$$\Delta_g^{\mathrm{J}} = \rho_g + \frac{(1-\rho)\mu}{N^{\mathrm{P}}}\left((1-\mathbf{P}[\Omega])\cdot\frac{1}{\mu}+\mathbf{P}[\Omega]\cdot\frac{\lambda_g}{\lambda_g+\mu}\cdot\frac{1}{\mu}\right), \tag{3.36}$$

from which (3.26) follows for $N_g^{\mathrm{P}}=1$.

When $N_g^{\mathrm{P}}\geq 2$, (3.27) reduces to

$$\Delta_g^{\mathrm{J}} = \mathbf{P}[\text{job of group } g \text{ in service}] = \mathbf{E}\left[o_g^{\mathrm{G}}\right],$$

and by (3.19), (3.26) follows for $N_g^{\mathrm{P}}\geq 2$. □

As expected, there is little (if any) internal fairness in FCFS, since always $\Delta^{\mathrm{J}}>0$ when $\rho_g>0$.

It is easy to show that when there are no permanent jobs, the results for FCFS hold for any non-preemptive non-discriminatory policy. This is because for such policies (3.10) holds with $N^{\mathrm{P}}=0$, and the probability that a job of group g is being served when the system is in state $\boldsymbol{n}\neq\mathbf{0}$ equals n_g/n. Examples of such policies are Last-Come First-Served (LCFS) and Random Selection for Service (RSS).

3.2.2 Head-of-the-Line

The Head-of-the-Line (HOL) policy is a discriminatory, non-preemptive policy. In HOL, the jobs in the system are served in the order according to their *priority class*, and a job in service is not preempted when a job of a higher priority class arrives. Jobs within the same priority class are served in FCFS order. In HOL, each priority class is made up of groups with identical required group shares. The higher the required group share corresponding to a class, the higher the priority.

HOL is an example of a priority queue; these are studied in detail in [43]. Unfortunately, deriving closed-form expressions for Δ_g^{G} and Δ_g^{J} seems impossible. Nevertheless, it is easy to show that HOL is not a very promising policy with respect to share scheduling. Consider for instance the case in which some group in the highest priority class has two or more permanent jobs present in the system. Then jobs of groups in lower priority classes will never be served at all. (When only a single permanent job of the highest priority class is present, other classes can be served if the HOL policy takes the scheduling decision upon departure of this job *before* it reenters the system.)

3.2.3 Up-Down

As was shown in the previous section, one of the problems with HOL is that a group with a large required group share is given almost unlimited preferential treatment over groups with smaller required shares. This preferential treatment is independent of the actual values of the required shares, as long as their ordering remains. For instance, when $G = 2$, the behavior of HOL with $r_1^G = 0.55$ and $r_2^G = 0.45$ is identical to its behavior with $r_1^G = 0.95$ and $r_2^G = 0.05$. Apparently, our objective of minimizing Δ^G requires a more subtle approach. This leads to the idea of the Up-Down (UD) policy, a policy similar to the equally named algorithm used in CONDOR [60] to grant fair access to idle workstations.

The Up-Down policy is non-preemptive and discriminatory. As in HOL, each group has a priority, and jobs within a group are served in FCFS order. However, as opposed to HOL, these priorities are dynamic. When a job of a group receives service, the group priority decreases, and when all jobs of a group are waiting for service, the priority increases. When a job terminates, the first job of the group with the highest priority is selected for service. Formally, we define the *group priority* $x_g(t)$ *of group* g *at time* t as

$$x_g(t) \triangleq r_g^G + \int_0^t \left(f_g^G(\tau) - o_g^G(\tau) \right) d\tau, \; g = 1, \ldots, G. \tag{3.37}$$

At $t = 0$, group g has priority r_g^G and therefore the UD policy starts as the HOL policy. Recall that $f_g^G(t) = 0$ when $N_g(t) = 0$ and $f_g^G(t) = r_g^G$ when $N_g(t) > 0$. Obviously, when a job of group g is being served, $o_g^G(t) = 1$ and $x_g(t)$ decreases (we assume $G > 1$, and therefore $r_g^G < 1$, $g = 1, \ldots, G$). On the other hand, when all jobs present of group g are waiting, we have $o_g^G(t) = 0$, and $x_g(t)$ increases. Note that $x_g(t)$ can obtain any real value.

The UD policy was designed with permanent jobs in mind. When HOL is used and the group with the highest priority has more than one permanent job in the system, the other groups will not be served. By dynamically adjusting the group priorities, UD is expected to perform much better than HOL under a workload of permanent jobs.

Although UD is a non-preemptive policy, it shares some similarities with Time-Function Scheduling [32] described in Section 1.5. Both policies dynamically adjust group priorities based on job waiting times (TFS) or past share deviations (UD). However, a major difference between the two is that in TFS, the time functions apply to each job of a group, whereas in UD, the priority function $x_g(t)$ applies to the group as a whole.

3.3 Preemptive Policies

In preemptive policies, the job currently in service may be interrupted in favor of another job. Here, we only consider the Priority Queueing policy, because some analytical results are known for it.

3.3.1 Priority Queueing

Priority Queueing (PQ) is the preemptive/resume variation of the Head-of-the-Line policy described in Section 3.2.2. As with HOL, each job belongs to a priority class. The PQ

policy always serves the first job of the highest-priority class for which jobs are present. Jobs within a single priority class are served in FCFS order. A job in service is interrupted when a job of a higher priority class arrives. It is resumed when there are no more jobs of higher priority classes to be served. We assume that

$$r_1^{\mathrm{G}} > \cdots > r_G^{\mathrm{G}},$$

and that group g_1 has priority over g_2 when $g_1 < g_2$.

We study the performance of PQ in a system with $G = 2$ and no permanent jobs, because analytical results are available for this system. Different means $1/\mu_1$ and $1/\mu_2$ for the service-time distributions are allowed. We define

$$\alpha \triangleq \frac{\mu_2}{\mu_1}. \tag{3.38}$$

Theorem 3.4 *In a PQ uniprocessor system with two groups with exponential service times and no permanent jobs, if $\rho < 1$, the pgf of the numbers of jobs is given by*

$$F(z_1, z_2) = \left(\frac{1 - \rho_1 - \rho_2}{1 - \eta(z_2) - \rho_2 z_2}\right) \cdot \left(\frac{1 - \eta(z_2)}{1 - \eta(z_2) z_1}\right), \tag{3.39}$$

with

$$\eta(z_2) = \frac{1}{2\mu_1}\left(\mu_1 + \lambda_1 + \lambda_2(1 - z_2) - \sqrt{(\mu_1 + \lambda_1 + \lambda_2(1 - z_2))^2 - 4\lambda_1\mu_1}\right). \tag{3.40}$$

PROOF: The expression is given in [33, pages 95–96]. It is obtained by manipulating the expression for the pgf given in [43, page 95] for *arbitrary* service-time distributions. □

Theorem 3.5 *In a PQ uniprocessor system with two groups with exponential service times and no permanent jobs, if $\rho < 1$, we have*

$$\Delta_1^{\mathrm{G}} = -\rho_1 \frac{1 - r_1^{\mathrm{G}}}{r_1^{\mathrm{G}}}, \tag{3.41}$$

$$\Delta_2^{\mathrm{G}} = \frac{\rho_1 + \rho_2 - \eta(0)}{1 - \eta(0)} - \frac{\rho_2}{r_2^{\mathrm{G}}}. \tag{3.42}$$

PROOF: For jobs of group 1, the system behaves as an ordinary FCFS M/M/1 queue with arrival rate λ_1 and service rate μ_1, so we have (3.20) with $\rho = \rho_1$. In order to derive (3.42), we use (2.38):

$$\begin{aligned}\Delta_2^{\mathrm{G}} &= \mathbf{P}[N_1 = 0, N_2 > 0] \frac{r_2^{\mathrm{G}} - 1}{r_2^{\mathrm{G}}} + \mathbf{P}[N_1 > 0, N_2 > 0] \frac{r_2^{\mathrm{G}} - 0}{r_2^{\mathrm{G}}} \\ &= \mathbf{P}[N_2 > 0] - \mathbf{P}[N_1 = 0, N_2 > 0] \frac{1}{r_2^{\mathrm{G}}}.\end{aligned} \tag{3.43}$$

From (3.39) we have

$$\begin{aligned}\mathbf{P}[N_2 > 0] &= 1 - \mathbf{P}[N_2 = 0] \\ &= 1 - F(1, 0) \\ &= \frac{\rho_1 + \rho_2 - \eta(0)}{1 - \eta(0)},\end{aligned} \tag{3.44}$$

and

$$\mathbf{P}\left[N_1 = 0, N_2 > 0\right] = F(0,1) - F(0,0) = \rho_2, \tag{3.45}$$

from which (3.42) follows. □

From (3.41) we find that always $\Delta_1^G < 0$, which is not surprising since the system is always working on jobs of group 1 when they are present. Before solving for $\Delta_2^G = 0$, we show that increasing ρ_1 increases Δ_2^G. This is intuitively clear: increasing ρ_1 decreases the availablity of the server for jobs of group 2, and hence increases $\mathbf{P}\left[N_2 > 0\right]$, which in turn increases Δ_2^G.

Lemma 3.6 *In a PQ uniprocessor system with two groups with exponential service times and no permanent jobs, if $\rho_1 > 0$, $\rho_2 > 0$, and $\rho < 1$, we have*

$$\frac{\partial \Delta_2^G}{\partial \rho_1} > 0. \tag{3.46}$$

PROOF: Rewriting (3.42) as

$$\Delta_2^G = 1 - \frac{1-\rho}{1-\eta(0)} - \frac{\rho_2}{r_2^G}, \tag{3.47}$$

we obtain

$$\frac{\partial \Delta_2^G}{\partial \rho_1} = \frac{1 - \eta(0) - (1-\rho)\frac{\partial \eta(0)}{\partial \rho_1}}{(1-\eta(0))^2}. \tag{3.48}$$

Clearly, the denominator is positive. Using (3.53), we find for the enumerator

$$\begin{aligned} & 1 - \eta(0) - (1-\rho)\frac{\partial \eta(0)}{\partial \rho_1} \\ &= 1 - \frac{1}{2}\left(1 + \rho_1 + \alpha\rho_2 - \sqrt{(1+\rho_1+\alpha\rho_2)^2 - 4\rho_1}\right) \\ &\quad -\frac{1}{2}(1-\rho_1-\rho_2)\left(1 - \frac{\rho_1 + \alpha\rho_2 - 1}{\sqrt{(1+\rho_1+\alpha\rho_2)^2 - 4\rho_1}}\right) \\ &= \frac{((\alpha-1)\eta(0) + (\alpha+1))\,\rho_2}{\sqrt{(1+\rho_1+\alpha\rho_2)^2 - 4\rho_1}}, \end{aligned} \tag{3.49}$$

which is positive for $\alpha \geq 1$, because $\eta(0) > 0$. For $0 < \alpha < 1$, it must hold that

$$\eta(0) < \frac{\alpha+1}{1-\alpha}, \tag{3.50}$$

which is true because $\eta(0) < \rho_1 < 1$ by (3.55). □

We now turn our attention to solving for $\Delta_2^G < (=, >)\ 0$.

Theorem 3.7 *In a PQ uniprocessor system with two groups with exponential service times and no permanent jobs, if $\rho_1 > 0$, $\rho_2 > 0$, and $\rho < 1$, we have*

$$\Delta_2^G < (=, >)\ 0 \iff \rho_1 < (=, >)\ \gamma_\alpha + \left(\frac{r_1^G - \gamma_\alpha}{r_2^G}\right)\rho_2, \tag{3.51}$$

where

$$\gamma_\alpha \triangleq r_1^G - \frac{1}{2}r_2^G\left(\sqrt{(\alpha+1)^2 + 4\frac{r_1^G}{r_2^G}\alpha} - (\alpha+1)\right). \tag{3.52}$$

PROOF: We will prove (3.51) for $\Delta_2^G = 0$. The cases $\Delta_2^G < 0$ and $\Delta_2^G > 0$ then follow directly from Lemma 3.6. From (3.38) and (3.40), we have

$$\begin{aligned}\eta(0) &= \frac{1}{2\mu_1}\left(\mu_1 + \lambda_1 + \lambda_2 - \sqrt{(\mu_1 + \lambda_1 + \lambda_2)^2 - 4\lambda_1\mu_1}\right) \\ &= \frac{1}{2}\left(1 + \rho_1 + \alpha\rho_2 - \sqrt{(1 + \rho_1 + \alpha\rho_2)^2 - 4\rho_1}\right).\end{aligned} \tag{3.53}$$

Clearly, $\eta(0) > 0$, and because

$$\frac{\partial}{\partial\lambda_2}\eta(0) < 0, \tag{3.54}$$

we obtain, by setting $\lambda_2 = 0$ in (3.53),

$$0 < \eta(0) < \rho_1. \tag{3.55}$$

It is therefore obvious from (3.42) that

$$\rho_2 \geq r_2^G \Rightarrow \Delta_2^G < 0. \tag{3.56}$$

Also, rewriting (3.42) as

$$\Delta_2^G = \frac{\left(r_2^G - \frac{r_1^G}{\rho_1}\rho_2\right)\rho_1 - (r_2^G - \rho_2)\eta(0)}{(1 - \eta(0))r_2^G}, \tag{3.57}$$

and using (3.55), we find

$$\rho_1 \geq r_1^G \Rightarrow \Delta_2^G > 0. \tag{3.58}$$

From hereon, we assume that $\rho_2 < r_2^G$ and $\rho_1 < r_1^G$. By setting $\Delta_2^G = 0$ in (3.42) we obtain

$$\eta(0) = \frac{r_2^G\rho_1 - r_1^G\rho_2}{r_2^G - \rho_2}, \tag{3.59}$$

and so, by (3.53),

$$\sqrt{(1 + \rho_1 + \alpha\rho_2)^2 - 4\rho_1} = 1 + \rho_1 + \alpha\rho_2 - \frac{2r_2^G\rho_1 - 2r_1^G\rho_2}{r_2^G - \rho_2}. \tag{3.60}$$

We will check on the right-hand side of (3.60) being positive later. Taking the square on both sides, dividing by ρ_2, and using $\rho = \rho_1 + \rho_2$ and $r_1^G = 1 - r_2^G$, we obtain the following quadratic equation in ρ_1:

$$a_0 + a_1\rho_1 + a_2\rho_1^2 = 0, \tag{3.61}$$

with

$$a_0 = (1 - r_2^G)\left(r_2^G + (\alpha - 1)r_2^G\rho_2 - \alpha\rho_2^2\right), \tag{3.62}$$

$$a_1 = r_2^G\left(-(\alpha + 1)(r_2^G - \rho_2) - 2(1 - r_2^G)\right), \tag{3.63}$$

$$a_2 = r_2^G. \tag{3.64}$$

This equation has two roots, $\bar{\rho}_1^L$ and $\bar{\rho}_1^U$:

$$\bar{\rho}_1^L = \frac{1}{2}(r_2^G - \rho_2)(\alpha + 1) + (1 - r_2^G) - \frac{1}{2}(r_2^G - \rho_2)\sqrt{(\alpha + 1)^2 + 4\frac{r_1^G}{r_2^G}\alpha}, \tag{3.65}$$

$$\bar{\rho}_1^U = \frac{1}{2}(r_2^G - \rho_2)(\alpha + 1) + (1 - r_2^G) + \frac{1}{2}(r_2^G - \rho_2)\sqrt{(\alpha + 1)^2 + 4\frac{r_1^G}{r_2^G}\alpha}. \tag{3.66}$$

We have $\bar{\rho}_1^U > r_1^G$ because $\rho_2 < r_2^G$, so $\bar{\rho}_1^U$ is not a valid solution of $\Delta_2^G = 0$. But $\bar{\rho}_1^L$ is, because $0 < \bar{\rho}_1^L < r_1^G$ and the right-hand side of (3.60) is positive for $\rho_1 = \bar{\rho}_1^L$ and any $\alpha > 0$:

$$\begin{aligned}
&1 + \bar{\rho}_1^L + \alpha\rho_2 - \frac{2r_2^G\rho - 2\rho_2}{r_2^G - \rho_2} \\
&= \frac{-(r_2^G + \rho_2)\rho_1^L + (r_2^G + \rho_2) + (\alpha - 2)r_2^G\rho_2 - \alpha\rho_2^2}{r_2^G - \rho_2} \\
&= \frac{(r_2^G)^2 + (\alpha - 1)r_2^G\rho_2 + \frac{1}{2}((r_2^G)^2 - \rho_2^2)\left(\sqrt{(\alpha + 1)^2 + 4\frac{r_1^G}{r_2^G}\alpha} - (\alpha + 1)\right) - \alpha\rho_2^2}{r_2^G - \rho_2} \\
&> \frac{(r_2^G)^2 - \rho_2^2}{r_2^G - \rho_2} \\
&> 0.
\end{aligned}$$

□

Theorem 3.7 allows us to explain the behavior of Δ_2^G as a function of r_1^G, ρ_1, ρ_2, and α. In Figure 3.3, we show the regions in which $\Delta_2^G < 0$ and $\Delta_2^G > 0$, respectively. Above the line segment starting at $\rho_1 = \gamma_\alpha$, $\rho_2 = 0$ and ending at $\rho_1 = r_1^G$, $\rho_2 = r_2^G$, we have $\Delta_2^G > 0$, below it, we have $\Delta_2^G < 0$. We introduce a separate proposition that explains the relation between Δ_2^G and α.

Proposition 3.8 *In a PQ uniprocessor system with two groups with exponential service times and no permanent jobs, if $\rho_1 > 0$, $\rho_2 > 0$, and $\rho < 1$, we have*

$$0 < \gamma_\alpha < r_1^G, \tag{3.67}$$

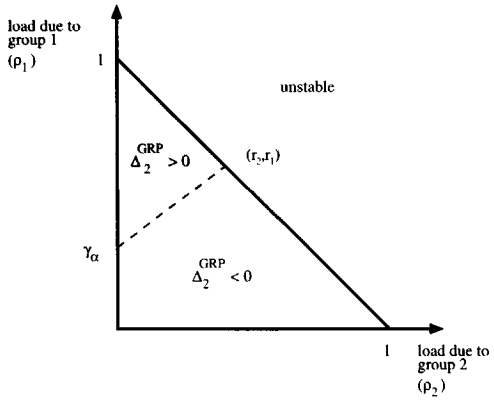


Figure 3.3: The behavior of the expected group-share deviation Δ_2^G for PQ when there are no permanent jobs, with $G = 2$.

$$\frac{\partial \gamma_\alpha}{\partial \alpha} < 0, \tag{3.68}$$

$$\left.\frac{\partial \Delta_2^G}{\partial \alpha}\right|_{\rho_1, \rho_2 \text{ constant}} > 0, \tag{3.69}$$

and furthermore,

$$\lim_{\alpha \downarrow 0} \gamma_\alpha = r_1^G, \tag{3.70}$$

$$\gamma_1 = 1 - \sqrt{r_2^G}, \tag{3.71}$$

$$\lim_{\alpha \to \infty} \gamma_\alpha = 0. \tag{3.72}$$

PROOF: Equation 3.52 can be written as

$$\gamma_\alpha = r_1^G - \frac{1}{2} r_2^G \left(\sqrt{(\alpha+1)^2 + b\alpha} - (\alpha+1) \right),$$

with $b = 4r_1^G / r_2^G > 0$, from which (3.68), (3.70), (3.71), and (3.72) follow directly. Note that for $b > 0$,

$$\lim_{\alpha \to \infty} \sqrt{(\alpha+1)^2 + b\alpha} - (\alpha+1) = \frac{1}{2} b.$$

The lower and upper bounds on γ_α in (3.67) follow from (3.68), (3.70), and (3.72). From (3.53) we have

$$\left.\frac{\partial \eta(0)}{\partial \alpha}\right|_{\rho_1, \rho_2 \text{ constant}} = \frac{1}{2}\rho_2 \left(1 - \frac{1 + \rho_1 + \alpha\rho_2}{\sqrt{(1+\rho_1+\alpha\rho_2)^2 - 4\rho_1}} \right) < 0, \tag{3.73}$$

and by (3.47), (3.69) follows. □

Referring again to Figure 3.3, when $\alpha \to 0$, we have $\gamma_\alpha \to r_1^G$, and the line segment approaches the horizontal line at $\rho_1 = r_1^G$. When $\alpha \to \infty$, we have $\gamma_\alpha \to 0$, and the line segment starts at the origin. Note that the point where the line segment intersects the line $\rho_1 + \rho_2 = 1$ is independent of α.

The relation between Δ_2^G and ρ_1 is best explained by moving upwards along a vertical line in Figure 3.3. Using Lemma 3.6 and Theorem 3.7, we have $\Delta_2^G < 0$ when $\rho_1 = 0$, and it increases as ρ_1 increases. If $\rho_2 < r_2^G$, Δ_2^G eventually becomes zero and positive when ρ_1 increases further.

Because of (3.69), Δ_2^G increases when α increases and ρ_1 and ρ_2 remain constant. This can be explained as follows. The obtained group shares of both groups do not change and neither does $\mathbf{P}[N_1 > 0]$. However, the average service time of jobs of group 2 decreases, and that of jobs of group 1 increases. Therefore, the average length $\mathbf{E}[B_1]$ of a group-1 busy period, given by

$$\mathbf{E}[B_1] = \frac{1/\mu_1}{1-\rho_1}$$

increases. Because of this, $\mathbf{P}[N_2 > 0]$ increases, which in turn increases both the expected feasible group share and the expected group-share deviation of group 2.

The condition $\rho_1 \leq r_1^G$ is *not* sufficient for $\Delta_2^G \leq 0$, as is the case with FCFS with $G = 2$. For instance, when $r_1^G = 0.6$, $r_2^G = 0.4$, $\alpha = 1.0$, $\rho_1 = 0.5 < r_1^G$, and $\rho_2 = 0.1$, we have $\Delta_2^G = 0.053$ by (3.42) and (3.53).

When $\lim_{\rho \uparrow 1} \rho_2 = \bar{\rho}_2$, we find from (3.42):

$$\lim_{\rho \uparrow 1} \Delta_2^G = -\frac{\bar{\rho}_2 - r_2^G}{r_2^G}.$$

So, under heavy traffic, we must require $\bar{\rho}_2 \geq r_2^G$ in order to have $\Delta_2^G \leq 0$. Clearly, this requires $\bar{\rho}_1 \leq r_1^G$. When we compare these results with those for FCFS, we find that progress has been made in the sense that when $\rho \uparrow 1$, ρ_2 can be anywhere between r_2^G and 1 in order to have $\Delta^G \leq 0$, instead of $\bar{\rho}_2 = r_2^G$ for FCFS. However, PQ has the problem that $\Delta_2^G > 0$ when $\rho_1 > r_1^G$.

Clearly, the lower r_2^G, the better the performance of PQ for jobs of group 2. This is intuitively clear since PQ favors jobs of group 1 over those of group 2 in the strongest possible way. This is illustrated in Figure 3.4, where we plot Δ^G as a function of λ_1/λ, with $\rho = 0.9$ and $\mu_1 = \mu_2 = 1.0$. First, in the downslope linear part of each graph we have $\Delta_1^G > \Delta_2^G$, so we have (3.41). When the fraction of jobs of group 1 increases further, we have $\Delta_2^G > \Delta_1^G$, and Δ^G follows (3.42). When $\lambda_1/\lambda \to 1$, we have $\lambda_2 \to 0$, and hence $\Delta_2^G \to 0$.

Our next theorem gives an expression for the expected job-share deviations Δ_1^J and Δ_2^J for groups 1 and 2, respectively.

Theorem 3.9 *In a PQ uniprocessor system with two groups with exponential service times and no permanent jobs, if $\rho_1 > 0$, $\rho_2 > 0$, and $\rho < 1$, we have*

$$\Delta_1^J = \rho_1^2, \tag{3.74}$$

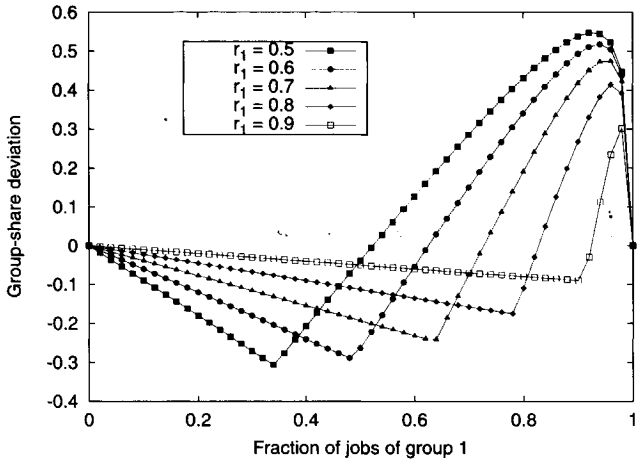


Figure 3.4: The group-share deviation Δ^{G} for Priority Queueing as a function of the fraction λ_1/λ of jobs of group 1 for different values of the required group share r_1^{G} of group 1 ($N^{\mathrm{P}} = 0$, $G = 2$, $\rho = 0.9$, $\mu_1 = \mu_2 = 1.0$).

$$\Delta_2^{\mathrm{J}} = \rho_2 \frac{\rho_1 + \rho_2 - \eta(0)}{1 - \eta(0)}. \tag{3.75}$$

PROOF: For jobs of group 1, the system behaves as an FCFS queue with total load ρ_1 and no permanent jobs, and hence we have (3.26) with $\rho = \rho_1$ and $N^{\mathrm{P}} = 0$ from which (3.74) follows. For group 2 we have by (2.45) $\Delta_2^{\mathrm{J}}(t) = 1$ when $N_1(t) = 0$ and $N_2(t) > 1$. In all other cases $\Delta_2^{\mathrm{J}}(t) = 0$. Therefore,

$$\begin{aligned} \Delta_2^{\mathrm{J}} &= \mathbf{P}\left[N_1 = 0, N_2 > 1\right] \\ &= F(0,1) - F(0,0) - \left.\frac{\partial F(0,z_2)}{\partial z_2}\right|_{z_2=0} \end{aligned} \tag{3.76}$$

From (3.39) we have

$$\left.\frac{\partial F(0,z_2)}{\partial z_2}\right|_{z_2=0} = \frac{(1-\rho_1-\rho_2)\rho_2}{1-\eta(0)}, \tag{3.77}$$

and so, by (3.45), (3.75) follows. □

The analysis for PQ can be extended to cases where one or more permanent jobs are present. When group 1 has permanent jobs in the system, no service is given to group 2, and $\Delta_2^{\mathrm{G}} = 1$, $\Delta_2^{\mathrm{J}} = 0$. When only group 2 has permanent jobs, we have

$$\Delta_2^{\mathrm{G}} = \mathbf{P}\left[N_1 = 0\right] \frac{r_2^{\mathrm{G}} - 1}{r_2^{\mathrm{G}}} + \mathbf{P}\left[N_1 > 0\right] \frac{r_2^{\mathrm{G}} - 0}{r_2^{\mathrm{G}}}$$

$$= (1-\rho_1)\frac{r_2^{\mathrm{G}}-1}{r_2^{\mathrm{G}}}+\rho_1$$
$$= \frac{\rho_1 - r_1^{\mathrm{G}}}{r_2^{\mathrm{G}}},$$

and so $\rho_1 \leq r_1^{\mathrm{G}}$ is a necessary and sufficient condition for $\Delta_2^{\mathrm{G}} \leq 0$.

When the service-time means are equal, the analysis can be extended to $G > 2$. In order to derive expressions for Δ_g^{G}, $g > 2$, one can treat groups $1, 2, \ldots, g-1$ as a single, high-priority group, and apply the formulas for the case with two groups.

3.4 Processor-Sharing Policies

Processor-sharing policies are approximations of round-robin policies with a time slice that is small compared to job service times. Round-robin policies are widely used in computer systems; they allow a processor to be shared among jobs. The main difference between the three processor-sharing policies discussed below is the possibility to control the obtained shares.

3.4.1 Processor Sharing

In Processor Sharing (PS) [51], all jobs are served simultaneously on an equal basis, so the obtained job share $o_j^{\mathrm{J}}(t)$ of job $j \in J(t)$ is given by

$$o_j^{\mathrm{J}}(t) = \frac{1}{N(t)}, \tag{3.78}$$

and the obtained group share $o_g(t)$ of group g is given by

$$o_g^{\mathrm{G}}(t) = \begin{cases} \frac{N_g(t)}{N(t)}, & \text{if } N(t) > 0, \\ 0, & \text{otherwise.} \end{cases} \tag{3.79}$$

Since all jobs present are served at equal rates, the policy is sometimes referred to as Egalitarian Processor Sharing. Clearly, $\Delta^{\mathrm{J}} = 0$ for PS.

The PS policy was introduced by Kleinrock [51], who derived expressions for the mean job response time when arrivals are Poisson and service times are exponential. Coffman, Muntz, and Trotter [15] derived the response-time distribution conditioned on the job service time, again for Poisson arrivals and exponentially distributed service times. Yashkov [88] and Ott [70] extended these results to arbitrary service-time distributions.

We will give closed-form expressions for the group-share deviation Δ_g^{G}, for a workload consisting of a mixture of non-permanent and permanent jobs. Before calculating the group-share deviations, a separate theorem for the probability distribution of the joint numbers of jobs is given.

Theorem 3.10 *In a PS uniprocessor system with exponential service times, if $\rho < 1$, the steady-state distribution $\mathbf{P}\left[\boldsymbol{N} = \boldsymbol{N}^{\mathrm{P}} + \boldsymbol{n}\right]$, $\boldsymbol{n} = (n_1, \ldots, n_G)$, is given by*

$$\mathbf{P}\left[\boldsymbol{N} = \boldsymbol{N}^{\mathrm{P}} + \boldsymbol{n}\right] = (1-\rho)^{N^{\mathrm{P}}+1} \cdot \frac{(N^{\mathrm{P}}+n)!}{N^{\mathrm{P}}! \cdot n_1! \cdots n_G!} \cdot \rho_1^{n_1} \cdots \rho_G^{n_G}, \tag{3.80}$$

for arbitrary $\boldsymbol{n} = (n_1, \ldots, n_G)$, $n_g \geq 0$, $g = 1, \ldots, G$. *The pgf of the numbers of non-permanent jobs is given by:*

$$F(z_1, \ldots, z_G) = \left(\frac{1-\rho}{1-\rho_1 z_1 - \cdots - \rho_G z_G} \right)^{N^{\mathrm{P}}+1}. \tag{3.81}$$

Furthermore, we have for $n = 0, 1, 2, \ldots,$

$$\mathbf{P}\left[N = N^{\mathrm{P}} + n\right] = (1-\rho)^{N^{\mathrm{P}}+1} \binom{N^{\mathrm{P}}+n}{n} \rho^n, \tag{3.82}$$

and if $N^{\mathrm{P}} = 0$,

$$\mathbf{P}\left[N = n\right] = (1-\rho)\rho^n. \tag{3.83}$$

PROOF: Equation 3.80 can be readily checked by substituting it into Kolmogorov's forward equation for the Markov process $\boldsymbol{N}(t)$:

$$\left(\lambda + \sum_{g=1}^{G} \frac{n_g \mu_g}{N^{\mathrm{P}}+n}\right) \mathbf{P}\left[\boldsymbol{N} = \boldsymbol{N}^{\mathbf{P}} + \boldsymbol{n}\right] = \tag{3.84}$$
$$\sum_{g=1}^{G} \delta^{n_g} \lambda_g \mathbf{P}\left[\boldsymbol{N} = \boldsymbol{N}^{\mathbf{P}} + \boldsymbol{n} - \boldsymbol{e}_g\right] + \sum_{g=1}^{G} \frac{(n_g+1)\mu_g}{N^{\mathrm{P}}+n+1} \mathbf{P}\left[\boldsymbol{N} = \boldsymbol{N}^{\mathbf{P}} + \boldsymbol{n} + \boldsymbol{e}_g\right],$$

where $\boldsymbol{e}_g$ is the unit vector in dimension g, and

$$\delta^{n_g} \triangleq \begin{cases} 0, & \text{when } n_g = 0, \\ 1, & \text{when } n_g > 0. \end{cases}$$

Also,

$$\sum_{\boldsymbol{n}} \mathbf{P}\left[\boldsymbol{N} = \boldsymbol{N}^{\mathbf{P}} + \boldsymbol{n}\right] = 1,$$

which completes the proof of (3.80). The pgf follows from its definition in (3.8) and from (3.80):

$$\begin{aligned} F(z_1, \ldots, z_G) &= \sum_{\boldsymbol{n}} \mathbf{P}\left[\boldsymbol{N} = \boldsymbol{N}^{\mathbf{P}} + \boldsymbol{n}\right] z_1^{n_1} \cdots z_G^{n_G} \\ &= (1-\rho)^{N^{\mathrm{P}}+1} \sum_{\boldsymbol{n}} \frac{(N^{\mathrm{P}}+n)!}{N^{\mathrm{P}}! \cdot n_1! \cdots n_G!} (\rho_1 z_1)^{n_1} \cdots (\rho_G z_G)^{n_G} \\ &= (1-\rho)^{N^{\mathrm{P}}+1} \sum_{n=0}^{\infty} \frac{(N^{\mathrm{P}}+n)!}{N^{\mathrm{P}}! \cdot n!} (\rho_1 z_1 + \cdots + \rho_G z_G)^n \\ &= \left(\frac{1-\rho}{1-\rho_1 z_1 - \cdots - \rho_G z_G} \right)^{N^{\mathrm{P}}+1}, \end{aligned}$$

which proves (3.81). Equations 3.82 and 3.83 follow directly from (3.80). □

Note that by (3.10), the stationary distribution of the joint number of jobs present for PS is identical to that for FCFS with identical means. Fortunately, (3.80) has a nice structure, allowing us to calculate the group-share deviations Δ_g^{G}.

Theorem 3.11 *In a PS uniprocessor system with exponential service times, if $\rho < 1$, the expected group-share deviation Δ_g^{G} of group g is given by*

$$\Delta_g^{\mathrm{G}} = \begin{cases} 1 - \left(\frac{1-\rho}{1-(\rho-\rho_g)}\right)^{N^{\mathrm{P}}+1} - \frac{\rho_g}{r_g^{\mathrm{G}}}, & N_g^{\mathrm{P}} = 0, \\ 1 - \frac{\rho_g + \frac{N_g^{\mathrm{P}}}{N^{\mathrm{P}}}(1-\rho)}{r_g^{\mathrm{G}}}, & N_g^{\mathrm{P}} > 0. \end{cases} \tag{3.85}$$

PROOF: From the definitions (2.27) and (2.38) we have

$$\begin{aligned} \Delta_g^{\mathrm{G}} &= \lim_{t\to\infty} \mathbf{E}\left[\frac{f_g^{\mathrm{G}}(t) - o_g^{\mathrm{G}}(t)}{r_g^{\mathrm{G}}}\right] \\ &= \frac{1}{r_g^{\mathrm{G}}}\left\{\mathbf{P}\left[N_g > 0\right] r_g^{\mathrm{G}} - \mathbf{E}\left[q_g^{\mathrm{G}}\right]\right\}. \end{aligned} \tag{3.86}$$

From (3.81) we have

$$\mathbf{P}\left[N_g > 0\right] = \begin{cases} 1 - F(1,\ldots,1,0,1,\ldots,1) = 1 - \left(\frac{1-\rho}{1-(\rho-\rho_g)}\right)^{N^{\mathrm{P}}+1}, & N_g^{\mathrm{P}} = 0, \\ 1, & N_g^{\mathrm{P}} > 0. \end{cases} \tag{3.87}$$

From (3.79) and (3.80) we have

$$\mathbf{E}\left[o_g^{\mathrm{G}}\right] = \begin{cases} \rho_g, & N_g^{\mathrm{P}} = 0, \\ \rho_g + \frac{N_g^{\mathrm{P}}}{N^{\mathrm{P}}}(1-\rho), & N_g^{\mathrm{P}} > 0. \end{cases} \tag{3.88}$$

Equation 3.85 now follows from substitution of (3.87) and (3.88) into (3.86). □

Clearly, the performance of PS with respect to minimizing Δ^{G} is identical to that of FCFS. Therefore, we will not discuss it further.

3.4.2 Priority Processor Sharing

In Priority Processor Sharing (PPS) [51], each group g has a *weight* w_g ($w_g > 0$). Each job receives service at a rate proportional to the weight of its group, so, when $N_g(t) > 0$, we have for a job of group g,

$$o_j^{\mathrm{J}}(t) \triangleq \frac{w_g}{\sum_h N_h(t) w_h}, \tag{3.89}$$

and therefore,

$$o_g^{\mathrm{G}}(t) \triangleq \frac{N_g(t) w_g}{\sum_h N_h(t) w_h}. \tag{3.90}$$

As in PS, a group can always increase its obtained share by putting more jobs on the processor (unless it is the only active group; then $o_g^{\mathrm{G}}(t) = 1$). We put $w_g = r_g^{\mathrm{G}}$, $g = 1,\ldots,G$.

The PPS policy was introduced by Kleinrock [51], who derived expressions for the expected job response time for Poisson arrivals and exponential service-time distributions. The arrival rates and the expected service times are allowed to be different for each

group. The approach taken was to consider the analogous discrete-time system, and letting the quantum size approach zero. Unfortunately, the result found was incorrect, as pointed out by, amongst others, Fayolle et al. [30]. They derived expressions for the expected job response times conditioned on the required service time for general service-time distributions, and expressions for the unconditional job response times. The (joint) queue-length distribution remained an open issue until 1993, when it was finally solved by Rege and Sengupta [72], for Poisson arrivals (different rates are allowed) and exponential service-time distributions (different means are allowed). Unfortunately, the form of the solution is rather complex and deriving expressions for the expected group-share deviations Δ_g^{G}, $g = 1, \ldots, G$, seems impossible. As in PS, the job-share deviation Δ^{J} is obviously zero.

In the more recent literature, the policy is often referred to as (Kleinrock's) *Discriminatory Processor Sharing* [72].

3.4.3 Group-Priority Processor Sharing

In Group-Priority Processor Sharing (GPPS), as in PPS, each group has a weight w_g ($w_g > 0$). We put $w_g = r_g^{\mathrm{G}}$, $g = 1, \ldots, G$. If $N(t) > 0$, the obtained group share of group g is given by

$$o_g^{\mathrm{G}}(t) \triangleq \frac{w_g}{\sum_{u|N_u>0} w_u}, \tag{3.91}$$

which is shared equally among all jobs of group g present. In GPPS, $o_g^{\mathrm{G}}(t)$ cannot be increased by offering more jobs to the processor (except of course when the group is not active on it), which is a principal advantage of GPPS over PS and PPS. Moreover, the obtained group share always satisfies

$$o_g^{\mathrm{G}}(t) \geq \frac{w_g}{\sum_u w_u}, \tag{3.92}$$

with strict inequality when one or more groups other than g are inactive. The obtained group share can only decrease when another group becomes active.

Clearly, GPPS is the ideal uniprocessor share-scheduling policy. Since $f_g^{\mathrm{G}}(t) = r_g^{\mathrm{G}}$ for all active groups g and $o_g^{\mathrm{G}}(t) \geq r_g^{\mathrm{G}}$, we conclude that the obtained share is always greater than or equal to the feasible share. As PS and PPS, GPPS performs perfectly with respect to internal fairness, since all jobs of a group always obtain equal shares.

To the best of our knowledge, the GPPS policy has not been described previously in the literature. Unfortunately, finding the joint queue-length distribution, let alone finding expressions for Δ_g^{G}, seems impossible.

3.5 Performance Evaluation

In this section, we evaluate the performance of the local scheduling policies described. The two performance measures of interest are the group-share deviation Δ^{G} and the job-share deviation Δ^{J}. We consider the following two types of workloads:

- Only non-permanent jobs.

- A combination of permanent and non-permanent jobs.

We study the behavior of Δ^G and Δ^J for a system with only two groups, i.e., $G = 2$. For most policies, one can extrapolate the results to systems with $G > 2$, for instance, by treating a number of groups as one single group. The arrival rates of the groups can be different; the mean service rates are set to $\mu = \mu_1 = \mu_2 = 1.0$. This allows us to use the analytical results for FCFS.

The main purpose of this section is to show that of the non-preemptive and preemptive policies, only Up-Down performs reasonably, but not always perfectly, with respect to the delivery of the feasible group shares. For the three other policies, viz. First-Come First-Served, Head-of-the-Line, and Priority Queueing, we will show that even in this simple setting, problems arise with specific types of workload. Of the processor-sharing policies, we have already shown that Group-Priority Processor Sharing is the ideal share-scheduling policy. We will show that neither PS nor PPS can match GPPS's performance.

In the sequel, all results for FCFS, PQ, and PS are obtained analytically. For the other policies we use simulations, described in more detail in Appendix A. In some specific cases, one or more of the other policies behave exactly as FCFS or PS (for instance, PPS when the required group shares are equal), and the appropriate analytical results are used.

3.5.1 Only non-permanent jobs

First, we consider a workload consisting of only non-permanent jobs. The workload due to the non-permanent jobs is constant, we put $\rho = 0.8$. For this system, we consider the following two cases:

1. the required group shares are equal, while the workload is *not* distributed evenly among the groups, and
2. the required group shares are *not* equal, while the workload is distributed evenly among the groups.

Clearly, both cases present challenges to a share-scheduling policy.

Case 1

In case 1, we have $r_1^G = r_2^G = 0.5$, and we let the ratio ρ_1/ρ vary from 0 to 0.5. (Because of the symmetry, it makes no sense to let ρ_1/ρ run to 1.)

In Figure 3.5, we show the group-share deviation Δ^G as a function of the fraction ρ_1/ρ. Clearly, since $r_1^G = r_2^G$ both HOL and PQ behave as FCFS, and PPS behaves as PS. Also, the group-share deviation for PS is equal to that for FCFS since $\mu_1 = \mu_2$. Therefore, the single graph for FCFS, HOL, PQ, PS, and PPS is obtained analytically by (3.20). Defining

$$x \triangleq \rho_1/\rho,$$

we therefore have by (3.20), for all policies except UD and GPPS,

$$\Delta_1^G = \frac{4x(3-8x)}{5(1+4x)}, \tag{3.93}$$

$$\Delta_2^G = \frac{4(1-x)(3-8(1-x))}{5(1+4(1-x))}. \tag{3.94}$$

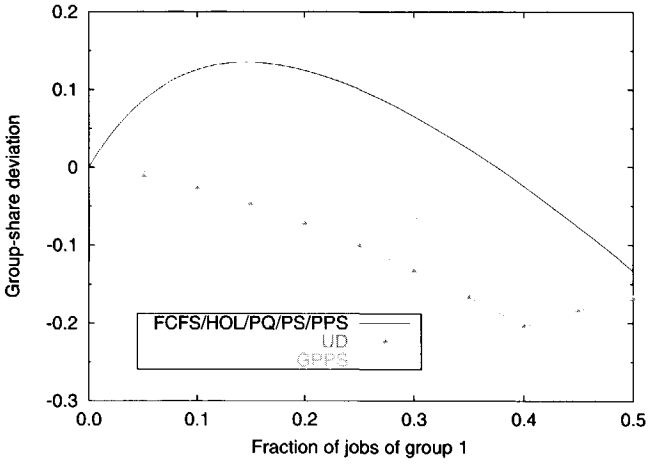


Figure 3.5: The group-share deviation Δ^{G} in a uniprocessor system with two groups, as a function of the fraction ρ_1/ρ of jobs of group 1 ($r_1^{\mathrm{G}} = r_2^{\mathrm{G}} = 0.5$, $\rho = 0.8$, $N^{\mathrm{P}} = 0$).

We have $\Delta_1^{\mathrm{G}} > \Delta_2^{\mathrm{G}}$ when $0 \leq x < 0.5$, and $\Delta_1^{\mathrm{G}} = \Delta_2^{\mathrm{G}}$ when $x = 0.5$, and hence $\Delta^{\mathrm{G}} = \Delta_1^{\mathrm{G}}$.

From the plot it is clear that FCFS and PS, as expected, perform poorly with respect to the delivery of feasible group shares. By (3.93), when $\rho_1/\rho < 3/8 = 0.375$, we have $\Delta_1^{\mathrm{G}} > 0$. Only UD and GPPS perform acceptably, for both policies $\Delta^{\mathrm{G}} < 0$. The performance improvement of UD over GPPS is remarkable, but is caused by the fact the UD actively compensates for feasible group-share deviations from the past, whereas GPPS does not. For GPPS, $\Delta_1^{\mathrm{G}} > \Delta_2^{\mathrm{G}}$, which is because group 2 profits from the idle periods of group 1 (obtaining a share of 1, instead of 0.5) more often than the other way around.

For UD, $\Delta_1^{\mathrm{G}} > \Delta_2^{\mathrm{G}}$ for $x \leq 0.4$, but $\Delta_1^{\mathrm{G}} = \Delta_2^{\mathrm{G}}$ for $x = 0.45$ and (naturally) $x = 0.5$. In order to explain this phenomenon, consider Figure 3.6, where we show both Δ_1^{G} and Δ_2^{G} for UD. For low values of ρ_1/ρ, most of the workload consists of jobs of group 2, and UD behaves as FCFS for these jobs. Hence, by (3.94), $\Delta_2^{\mathrm{G}} \to -0.8$ as $\rho_1 \downarrow 0$. Also, the difference between $x_1(t)$ and $x_2(t)$, which are defined in (3.37), grows without bounds. This means, loosely speaking, that UD is not able to treat the two groups equally. For jobs of group 1, UD behaves as HOL (with group 1 being the high-priority group) for low values of ρ_1/ρ. As the fraction of jobs of group 1 increases, jobs of group 2 have to wait more and more for jobs of group 1 passing by, and Δ_2^{G} increases almost linearly. Only when $\rho_1/\rho \approx 0.4$ and beyond, UD can keep the group priorities $x_1(t)$ and $x_2(t)$ (and, in this case, also Δ_1^{G} and Δ_2^{G}) within each other's range. Now Δ_1^{G} starts to increase, because UD no longer automatically selects jobs of group 1 for service when they are present, i.e., it no longer acts as HOL for group 1.

In Figure 3.7, we show the job-share deviation Δ^{J} as a function of the fraction ρ_1/ρ. Again, the results for FCFS, HOL, PQ, PS, PPS, and GPPS are obtained analytically. By (3.26) we have for FCFS, HOL, and PQ, with $x = \rho_1/\rho$,

$$\Delta_1^{\mathrm{J}} = \frac{16x^2}{5 + 20x}, \tag{3.95}$$

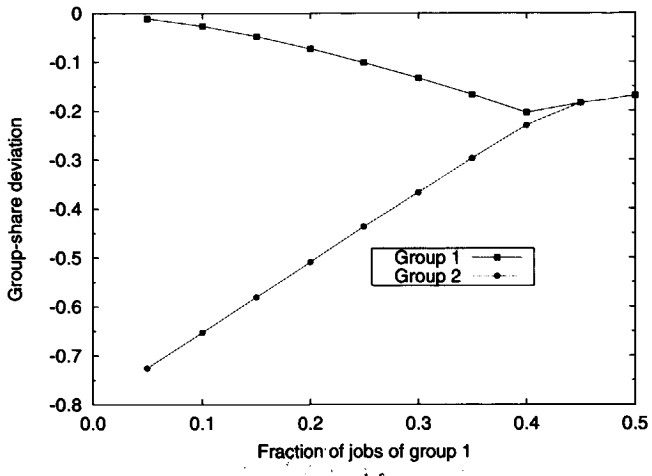


Figure 3.6: The group-share deviations Δ_1^G and Δ_2^G for Up-Down in a uniprocessor system with two groups, as a function of the fraction ρ_1/ρ of jobs of group 1 ($r_1^G = r_2^G = 0.5$, $\rho = 0.8$, $N^P = 0$).

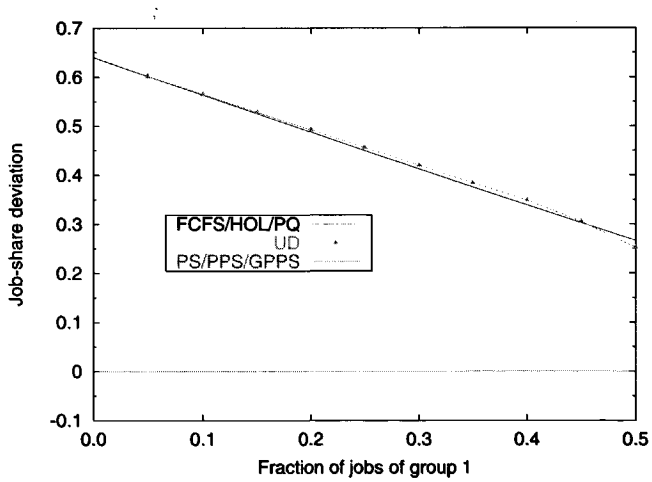


Figure 3.7: The job-share deviation Δ^J in a uniprocessor system with two groups, as a function of the fraction ρ_1/ρ of jobs of group 1 ($r_1^G = r_2^G = 0.5$, $\rho = 0.8$, $N^P = 0$).

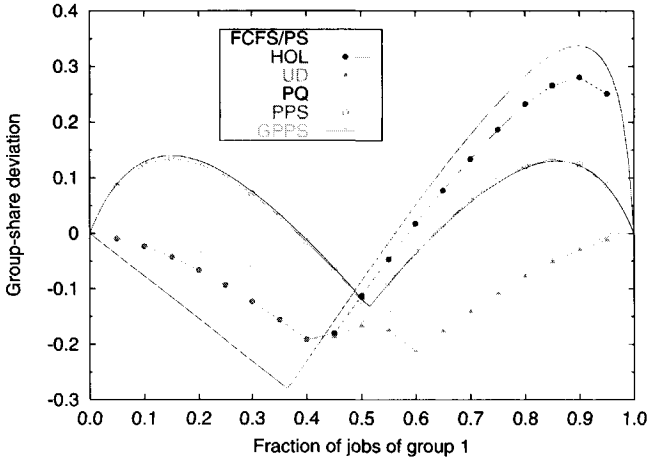


Figure 3.8: The group-share deviation Δ^G in a uniprocessor system with two groups, as a function of the fraction ρ_1/ρ of jobs of group 1 ($r_1^G = 0.51$, $r_2^G = 0.49$, $\rho = 0.8$, $N^P = 0$).

$$\Delta_2^J = \frac{16(1-x)^2}{5+20(1-x)}, \tag{3.96}$$

and $\Delta_2^G \geq \Delta_1^G$ for $0 \leq x \leq 0.5$, hence $\Delta^J = \Delta_2^J$. For the processor-sharing policies, $\Delta^J = 0$. Clearly, no non-preemptive or preemptive policy can outperform the processor-sharing policies with respect to internal fairness.

The job-share deviation for UD is practically equal to that for FCFS. Also for UD $\Delta_2^J > \Delta_1^J$ for $0 \leq x < 0.5$, simply because group 2 is present more often than group 1. Both UD and FCFS are non-preemptive, and the expected job-share deviation of group g equals the probability that at least one job of group g is waiting while another job of group g is being served. For group 2, this probability is hardly affected when UD instead of FCFS is used.

It is interesting to study what happens when r_1^G is increased slightly, such that HOL and PQ will give preferential treatment to group 1. We therefore put $r_1^G = 0.51$ and $r_2^G = 0.49$. This time, ρ_1/ρ varies from 0 to 1. In Figure 3.8, we show Δ^G as a function of ρ_1/ρ. Clearly, by (3.16), Δ^G for FCFS and PS differs only slightly compared to Figure 3.5. The same holds for UD, PPS, and GPPS, which makes sense, since the behavior of these policies does not change abruptly when r_1^G is increased slightly.

The change in r_1^G and r_2^G clearly has a large impact on the performance of HOL and PQ, which exhibit large values of Δ^G when the fraction of arrivals of group 1 is larger than approximately 0.6. This is caused by the jobs of group 2; these are fully dependent on the idle periods of group 1 in order to obtain service. The performance of HOL is somewhat better than that of PQ here, because with HOL, jobs of group 2 are not preempted once taken into service. From these results it is clear that neither HOL nor PQ is a good share-scheduling policy, even in the absence of permanent jobs.

In this case, the Up-Down policy performs best of all policies. It is evident from Figure

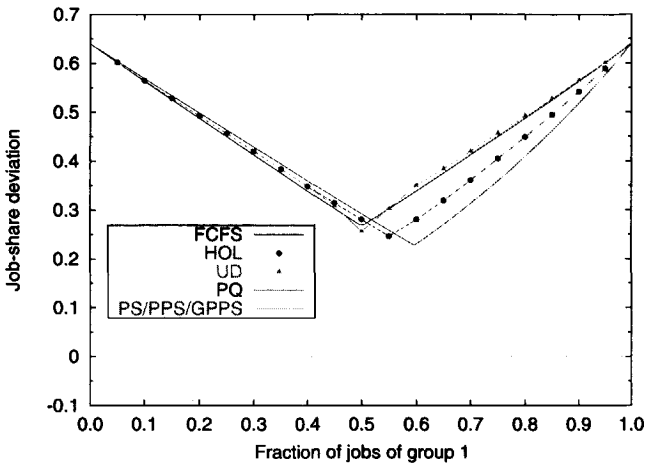


Figure 3.9: The job-share deviation Δ^J in a uniprocessor system with two groups, as a function of the fraction ρ_1/ρ of jobs of group 1 ($r_1^G = 0.51$, $r_2^G = 0.49$, $\rho = 0.8$, $N^P = 0$).

3.8 that UD behaves as HOL for low values of ρ_1/ρ, with group 1 being the high priority group, as noted earlier. For high values of ρ_1/ρ, UD gives unlimited preferential treatment to group 2.

Finally, from Figure 3.8 it is clear that the behavior of PPS is much closer to that of PS than to that of GPPS, and that PPS suffers from the same problem as PS does: the group-share deviation strongly depends on the distribution of the workload among the groups. We therefore conclude that just like PS, PPS is not a good share-scheduling policy either.

In Figure 3.9, we show the job-share deviation Δ^J as a function of the fraction ρ_1/ρ. Again, Δ^J for UD is practically equal to that for FCFS. When $\rho_1 > \rho_2$, Δ^J for HOL is somewhat smaller than that for FCFS and UD. For PQ, it is even smaller than that for HOL. This is caused by the fact that when ρ_1 increases, ρ_2 decreases, and the service to jobs of group 2 decreases, and there can only be a job-share deviation for jobs of group 2 when a job of that group is being served. For values of $\rho_1/\rho > 0.5$, Δ^J is dominated by jobs of group 1. Clearly, because jobs of group 1 are given preferential treatment over jobs of group 2, we have a lower value of $\mathbf{P}[N_1 > 1]$ for HOL and PQ than for FCFS. This results in a lower value of Δ^J.

Case 2

In case 2, we put $\rho_1 = \rho_2 = 0.4$, and we let r_1^G vary from 0.5 to 1. In this case, the load is distributed evenly among the groups, and the challenge to the scheduling policy is to provide groups with different shares.

In Figure 3.10, we show the group-share deviation Δ^G as a function of the required group share of group 1. Again, FCFS and PS perform poorly, since they cannot achieve the required level of discrimination among the groups. For these policies, we have by

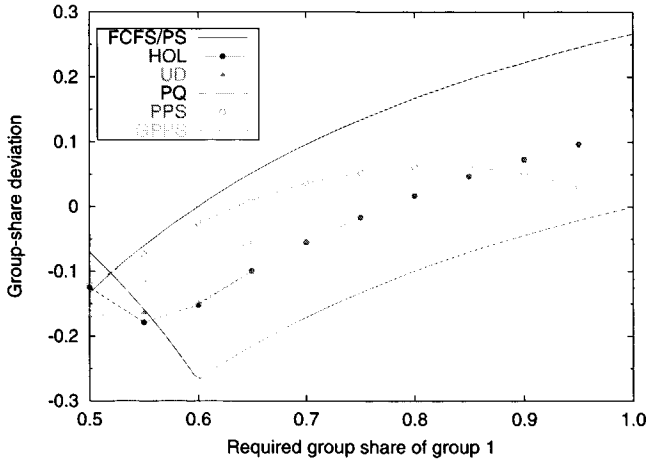


Figure 3.10: The group-share deviation Δ^G in a uniprocessor system with two groups, as a function of the required group share r_1^G of group 1 ($\rho_1 = \rho_2 = 0.4$, $N^P = 0$).

(3.20),

$$\Delta_1^G = \frac{2(5r_1^G - 3)}{15r_1^G}, \tag{3.97}$$

$$\Delta_2^G = \frac{2(5(1 - r_1^G) - 3)}{15(1 - r_1^G)}, \tag{3.98}$$

and we have $\Delta_1^G > \Delta_2^G$ for $r_1^G > 0.5$. Also, $\Delta^G > 0$ for $r_1^G > 0.6$. PPS definitely performs better than PS, but still gives poor results for larger values of r_1^G. In this case, only GPPS performs sufficiently well. PQ performs excellent; we have $\Delta^G < 0$ for all values of r_1^G. Note that in this figure, PQ still gives preferential treatment to jobs of group 1 when $r_1^G = 1/2$. Starting at $r_1^G = 1/2$, when r_1^G increases, we first have $\Delta_2^G > \Delta_1^G$, and $\Delta^G = \Delta_2^G$. As r_1^G increases further, Δ_2^G decreases (because r_2^G decreases) and Δ_1^G increases. When $r_1^G = 0.6$, we have $\Delta_1^G = \Delta_2^G$, and when r_1^G increases further, we have $\Delta_1^G > \Delta_2^G$. Obviously, Δ_1^G approaches zero when r_1^G approaches unity.

However, the most striking result in Figure 3.10 is the performance of UD, which is poorer than expected. We have $\Delta_1^G > \Delta_2^G$ for $r_1^G > 0.5$. For $r_1^G > 0.55$, UD behaves as HOL. The reason for this is that UD starts to give unlimited preferential treatment to jobs of group 1, because of its high required share, for reasons explained already in case 1. The negative share deviations during service periods to jobs of group 1 do not compensate for the positive share deviations during the periods during which such jobs await the completion of jobs of group 2. We therefore conclude that when a group has a high required share but only a low or moderate load, UD is not a good share-scheduling policy. On the other hand, it is questionable that any non-preemptive policy can do better. The only strategy that could lead to success is to start denying service to group 2 once the priority of group 1 reaches a certain threshold. In other words, to await the

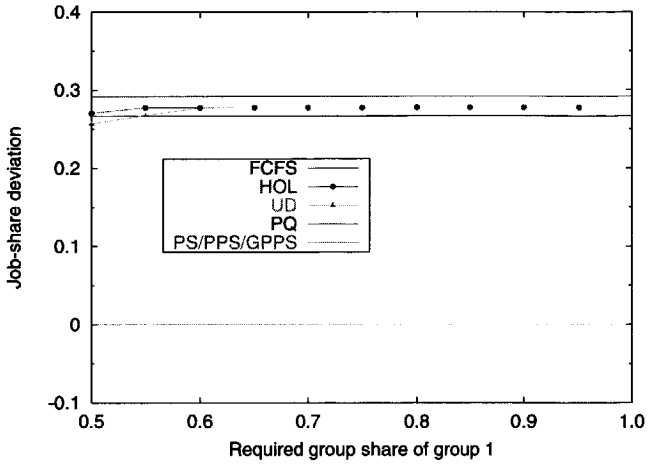


Figure 3.11: The job-share deviation Δ^{J} in a uniprocessor system with two groups, as a function of the required group share r_1^{G} of group 1 ($\rho_1 = \rho_2 = 0.4$, $N^{\mathrm{P}} = 0$).

arrival of jobs of group 1. However, this would lead to capacity loss, so we do not consider it any further.

In Figure 3.11, we show the job-share deviation Δ^{J} as a function of the required group share of group 1. We already explained why Δ^{J} is lower for HOL and PQ than for FCFS. It should be clear that the behavior of FCFS, HOL and PQ does not change in the interval $(0.5, 1)$, which explains why Δ^{J} is constant.

From the results of a workload consisting of only non-permanent jobs, we conclude that all policies except GPPS have problems providing feasible group shares over a wide range of workload parameters and of required shares.

3.5.2 Permanent and non-permanent jobs

We now consider a workload consisting of a mixture of permanent and non-permanent jobs. We study a system with $G = 2$, $\rho_1 = \rho_2 = 0.25$, and we let r_1^{G} vary between 0.5 and 1. We have lowered the total load ρ from 0.8 to 0.5, because in the presence of permanent jobs, some of the policies (notably HOL, UD, PQ, and GPPS) no longer provide enough service to the low-priority group, which causes the queue length to grow without bounds. For instance, with GPPS, group g has its own virtual processor with relative speed r_g^{G} (due to the presence of permanent jobs), and the queue length grows without bounds when $\rho_g \geq r_g^{\mathrm{G}}$.

First, we look at a system with $N_1^{\mathrm{P}} = N_2^{\mathrm{P}} = 1$, and then at a system with $N_1^{\mathrm{P}} = N_2^{\mathrm{P}} = 2$.

In Figure 3.12, we show Δ^{G} as a function of the required group share r_1^{G} of group 1, which runs from 0.5 to 1, when each group has one permanent job in the system. In Figure 3.13, we show Δ^{J}. Recall that after the departure of a permanent job, HOL and UD take their scheduling decision *before* the permanent job reenters the queue. From the plot it is

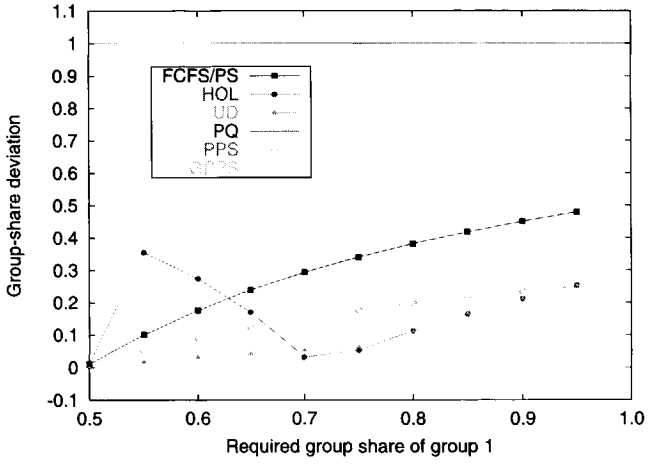


Figure 3.12: The group-share deviation Δ^{G} in a uniprocessor system with two groups, as a function of the required group share r_1^{G} of group 1 ($\rho_1 = \rho_2 = 0.25$, $N_1^{\mathrm{P}} = N_2^{\mathrm{P}} = 1$).

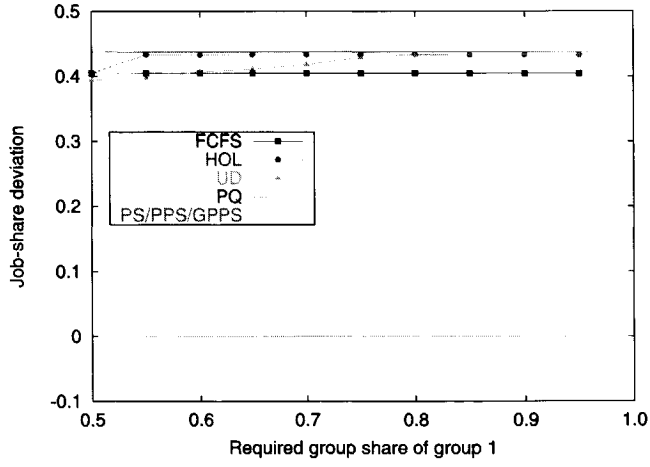


Figure 3.13: The job-share deviation Δ^{J} in a uniprocessor system with two groups, as a function of the required group share r_1^{G} of group 1 ($\rho_1 = \rho_2 = 0.25$, $N_1^{\mathrm{P}} = N_2^{\mathrm{P}} = 1$).

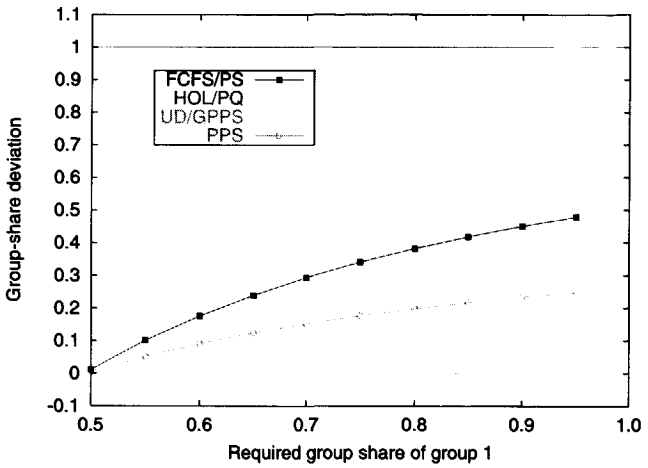


Figure 3.14: The group-share deviation Δ^{G} in a uniprocessor system with two groups, as a function of the required group share r_1^{G} of group 1 ($\rho_1 = \rho_2 = 0.25$, $N_1^{\mathrm{P}} = N_2^{\mathrm{P}} = 2$).

clear that the presence of the permanent jobs has a dramatic effect on the performance of PQ. With PQ, only jobs of group 1 are served, and hence $\Delta^{\mathrm{G}} = \Delta_2^{\mathrm{G}} = 1$. With HOL, jobs of group 2 can still be taken into service when the permanent job of group 1 terminates and no other jobs of group 1 are present. Therefore, the group-share deviation with HOL is much better than that with PQ. However, none of the policies FCFS, HOL, UD, PQ, PS, and PPS perform adequately. As was the case with a workload of only non-permanent jobs, UD behaves as HOL for high values of r_1^{G}. UD still cannot compensate for the share deviation for group 1 when it waits for the completion of a job of group 2 in service. Since a job of group 1 is always present, the feasible share of group 1 is higher compared to the case without permanent jobs. Not surprisingly, we have $\Delta^{\mathrm{G}} = 0$ with GPPS, which again shows it is the only policy that performs well under all circumstances.

We now increase the number of permanent jobs to 2 per both groups. In Figure 3.14, we show Δ^{G} as a function of r_1^{G}, and in Figure 3.15, we show Δ^{J}. Now HOL no longer serves jobs of group 2 either, hence $\Delta^{\mathrm{G}} = 1$ for both PQ and HOL. For UD, the presence of the additional permanent jobs clearly has merits: the group-share deviation is zero, not to our surprise. When a job terminates, UD can always choose between a job from group 1 and a job from group 2 to be served next, and in the long run, both groups obtain exactly their required shares.

We have $\Delta^{\mathrm{J}} = 1$ for PQ and HOL, because there is always a job of group 1 in service, and at least one job of that same group waiting in the queue. For FCFS, the two groups share the processor equally in the long run, and for both groups there is always at least one job present. Because $\Delta_g^{\mathrm{J}}(t) = 0$ when a job of a group other than g is being served, we have $\Delta_1^{\mathrm{J}} = \Delta_2^{\mathrm{J}} = 0.5$, and hence $\Delta^{\mathrm{J}} = 0.5$. The job-share deviation for UD is between that for FCFS and for HOL/PQ.

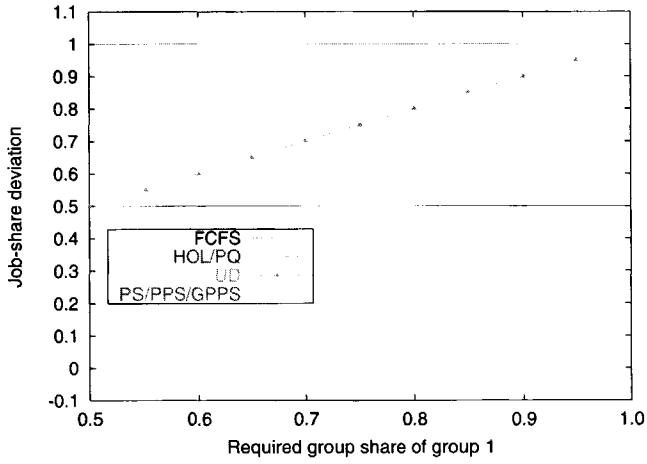


Figure 3.15: The job-share deviation Δ^{J} in a uniprocessor system with two groups, as a function of the required group share r_1^{G} of group 1 ($\rho_1 = \rho_2 = 0.25$, $N_1^{\mathrm{P}} = N_2^{\mathrm{P}} = 2$).

3.6 Summary and Conclusions

In this chapter we have studied the performance of a number of common scheduling policies for uniprocessors with respect to share scheduling. The two performance measures used, the group-share deviation Δ^{G} and the job-share deviation Δ^{J}, are effective measures for this.

We have studied non-preemptive (FCFS, HOL, UD), preemptive (PQ), and processor-sharing (PS, PPS, and GPPS) policies. Of these policies, FCFS and PS are non-discriminatory, the others are discriminatory. Clearly, the non-discriminatory policies can never be satisfying policies for share scheduling, since they do not consider which job belongs to which group.

The HOL and PQ policies are the simplest of the discriminatory policies. Unfortunately, both are totally useless for share scheduling when the workload includes permanent jobs of a group with a high priority (i.e., with a high required group share), because such jobs exclude lower-priority groups from service. The UD policy keeps a history of past share deviations in order to dynamically adjust group priorities. It performs reasonably well, especially in the presence of permanent jobs. However, problems arise with UD when a group with a high required group share only offers low load. Then, UD cannot compensate for share deviations from the past for that group. The resulting behavior is to give unlimited preferential treatment to the group, like HOL does, but still this is not enough to keep the group-share deviation smaller than zero.

Although PPS is a discriminatory policy, its performance in share scheduling is inadequate because it suffers from the same problem as PS: the obtained share of a group always increases as the number of jobs present of that group increases. From the simulation results it was clear that both PS (treating all jobs equally) and PPS perform poorly under heavy workloads in cases where the distribution of the workload among the

groups does not match the required group shares. This is not the case for the GPPS policy, which is undoubtedly the perfect share-scheduling policy. It guarantees non-positive group-share deviation (which is good) at all times, and no job-share deviation. Its implementation is not much more difficult than that of the better-known PPS policy, although the performance of GPPS is much better.

Even though we have studied cases with two groups only, we revealed fundamental problems with all policies except GPPS, which we believe also hold for systems with more groups. For FCFS, PS, and PQ, one can aggregate multiple groups and treat them as a single group in order to calculate the performance measures (for FCFS and PQ, the μ_g's have to be equal). With HOL, the problem is twofold: the almost unlimited preferential treatment of a group over a lower priority group, and the queueing of high-priority groups in order to wait for the completion of a low-priority job. Neither of these problems will disappear when more groups are added. For UD and PPS, the problems are closely related to those of HOL and PS, respectively. Finally, it was already shown that GPPS's performance is excellent irrespective of the number of groups, arrival rates, services times and the presence of permanent jobs.

The PS and GPPS policies discussed in the present chapter will return in Chapters 5 and 6, where we will use them as local scheduling policies in distributed systems.

Chapter 4

Static Share Scheduling in Multiprocessors and Distributed Systems

4.1 Introduction

In the previous chapter we concluded that on uniprocessors, only the Group-Priority Processor Sharing policy is able to provide groups with their feasible group shares at all times. In the current and subsequent chapters, we study systems with more than one processor. Clearly, the opportunity for decisions on which processor to serve which jobs provides us with additional means for compliance with the feasible group shares. However, the use of multiple processors also introduces the possibility of capacity loss: processors may be unnecessarily idle, given the number of jobs present in the system.

In the present chapter we discuss so-called *static* policies: The scheduling decisions are taken independently of the current state of the system. We study two different models. In Section 4.2, we consider a deterministic scheduling policy for multiprocessors (where we assume job migration is free). A fixed set of jobs must be scheduled in such a way that predefined job service rates are met. In our main result, Theorem 4.1, we state the necessary and sufficient conditions for meeting these service rates. This result has been used extensively in Chapter 2 in stating the objectives of share scheduling, and is therefore of fundamental importance for this thesis. We also introduce a scheduling policy called Multiprocessor Group-Priority Processor Sharing (MGPPS). The policy guarantees zero capacity loss and compliance with the feasible group shares at all times.

In Sections 4.3, 4.4, and 4.5, we study static probabilistic policies for share scheduling in distributed systems without job migration. We describe the system model, workload model, and related literature in Section 4.3. In the model, arriving jobs are routed to processors according to routing probabilities. This method of routing is known as *random splitting*. The local scheduling policy is Processor Sharing (which we described in Section 3.4.1). Subsequently, we try to find the optimal routing probabilities for minimization of the capacity loss in Section 4.4, and for minimization of the group-share deviation in Section 4.5. The main result is that both objectives are very hard to meet in general with random splitting. Also, in many cases, the objectives of minimization of capacity loss and of compliance with the feasible group shares cannot be met simultaneously.

This motivates the study of dynamic policies in Chapters 5 and 6. Because we cannot solve the problem of minimizing the group-share deviation in general, we introduce and compare two heuristic policies: Horizontal Partitioning in Random Splitting and Vertical Partitioning in Random Splitting. We will use the fundamental ideas behind these two policies in Chapters 5 and 6 as well.

In Section 4.6, we present our conclusions. The most important ones are that

1. the availability of free job migration takes away every challenge for share scheduling, because at all times, the feasible group shares can be met with zero capacity loss (using MGPPS);

2. in systems without job migration, random splitting induces a high capacity loss at the very least, and in many cases an unacceptably high group-share deviation as well.

4.2 Deterministic Share Scheduling with Free Job Migration

We consider a multiprocessor system with free job migration. The processors may have different capacities. In Section 4.2.1, we derive the feasibility constraints on providing prespecified shares to a fixed set of jobs. Subsequently, in Section 4.2.2 we introduce the Multiprocessor Group-Priority Processor-Sharing (MGPPS) policy, which extends the GPPS policy (see Section 3.4.3) to multiprocessor systems.

4.2.1 Feasibility of Providing Prespecified Shares

In this section, we derive necessary and sufficient conditions for the allocation of a fixed set of jobs on a multiprocessor in such a way that these jobs are served at prespecified rates. There is a set $\mathcal{P}$ of P processors, numbered $1, \ldots, P$, with capacities $c_1 \geq \cdots \geq c_P > 0$, and a set $\mathcal{J}$ of J jobs, numbered $1, \ldots, J$. Jobs can migrate from one processor to another without cost. A set $\mathcal{F} = \{ f_{jp} \mid j \in \mathcal{J}, p \in \mathcal{P} \}$ of functions on $[0, 1)$ is called a *schedule of* $\mathcal{J}$ *on* $\mathcal{P}$ when for all $t \in [0, 1)$ it holds that

$$f_{jp}(t) = 0 \text{ or } 1, \tag{4.1}$$

$$\sum_{p=1}^{P} f_{jp}(t) = 0 \text{ or } 1, \tag{4.2}$$

$$\sum_{j=1}^{J} f_{jp}(t) = 0 \text{ or } 1. \tag{4.3}$$

When $f_{jp}(t) = 1$, job j is served by processor p at time t. Condition (4.2) states that job j can be served by at most one processor at a time, while condition (4.3) states that processor p can serve at most one job at a time. Note that when $\mathcal{F}$ is a schedule of $\mathcal{J}$ on $\mathcal{P}$, $\mathcal{F}' = \{f_{jp} | j \in \mathcal{J}', p \in \mathcal{P}'\}$ with $\mathcal{J}' \subset \mathcal{J}$ and $\mathcal{P}' \subset \mathcal{P}$ is a schedule of $\mathcal{J}'$ on $\mathcal{P}'$.

The *average service rate* σ_j of job $j \in \mathcal{J}$ during the interval $[0, 1)$ is defined as

$$\sigma_j \triangleq \sum_{p=1}^{P} \left\{ \int_0^1 f_{jp}(t) dt \right\} c_p. \tag{4.4}$$

In this chapter, a schedule is always normalized to the interval $[0,1)$. It is trivial to transform a schedule on $[0,1)$ into a schedule on an arbitrary real interval $[a,b)$ such that the average service rates in both schedules are equal. The choice of the actual interval is therefore irrelevant. By choosing it small enough, one can approximate the processor-sharing policies for uniprocessors discussed in Chapter 3.

An obvious question is what service rates σ_j can be achieved. Without loss of generality, we assume $\sigma_1 \geq \cdots \geq \sigma_J > 0$. The following theorem provides necessary and sufficient conditions for the existence of a schedule with prespecified average service rates. Such a schedule is constructed in the course of the proof. The importance of this theorem lies in the fact that we have used it in Section 2.3.2 in stating the objectives of share scheduling.

Theorem 4.1 *The conditions*

$$\sum_{k=1}^{j} \sigma_k \leq \sum_{p=1}^{\min(j,P)} c_p, \quad j = 1, \ldots, J, \tag{4.5}$$

are necessary and sufficient for the existence of a schedule $\mathcal{F} = \{ f_{jp} \mid j \in \mathcal{J}, p \in \mathcal{P} \}$ *such that job* j *has average service rate* σ_j, $j = 1, \ldots, J$.

PROOF: Suppose such a schedule $\mathcal{F}$ exists. Using (4.4), we can rewrite the left-hand side of (4.5) as

$$\sum_{k=1}^{j} \sigma_k = \sum_{p=1}^{P} \beta_{jp} c_p, \tag{4.6}$$

with

$$\beta_{jp} \triangleq \sum_{k=1}^{j} \int_0^1 f_{kp}(t)dt. \tag{4.7}$$

Because of (4.1) and (4.3), we have $0 \leq \beta_{jp} \leq 1$. From (4.1) and (4.2), we find $\sum_{p=1}^{P} \beta_{jp} \leq j$. Hence, $\sum_{p=1}^{P} \beta_{jp} \leq \min(j,P)$. Combining this with (4.6) and with $c_1 \geq \cdots \geq c_P$, we find that (4.5) must hold.

We will first prove that (4.5) is sufficient for two trivial cases: the case where $J = 1$ and the case where $\sigma_1 \leq c_P$. For all other cases, we then prove that a schedule $\mathcal{F}$ of $\mathcal{J}$ on $\mathcal{P}$ with average service rates $\sigma_1, \ldots, \sigma_J$ exists, when we can find a schedule $\mathcal{F}'$ for a reduced system with $P-1$ processors and $J-1$ jobs. By repeatedly performing this reduction, we end up with a trivial scheduling problem with only one job or with $\sigma_1 \leq c_P$ (or both). The latter case includes the case with $P = 1$, since $\sigma_1 \leq c_1$ by (4.5). We first deal with the two bottom cases, and then with the reduction step.

Case 1: $J = 1$

When $J = 1$, a possible schedule is given by (cf. Figure 4.1):

$$\begin{aligned} f_{11}(t) &= \begin{cases} 1, & 0 \leq t < \sigma_1/c_1, \\ 0, & \sigma_1/c_1 \leq t < 1, \end{cases} \\ f_{1p}(t) &= 0, \ 0 \leq t < 1, \ p = 2, \ldots, P. \end{aligned} \tag{4.8}$$

Obviously, (4.1), (4.2), and (4.3) are satisfied, and the average service rate of job 1 equals σ_1.

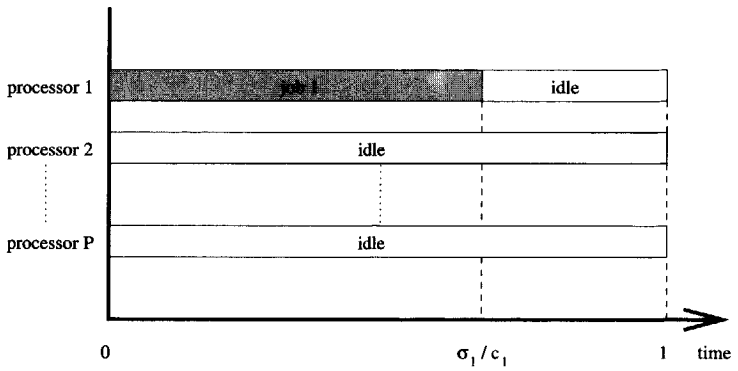


Figure 4.1: A possible schedule when $J = 1$.

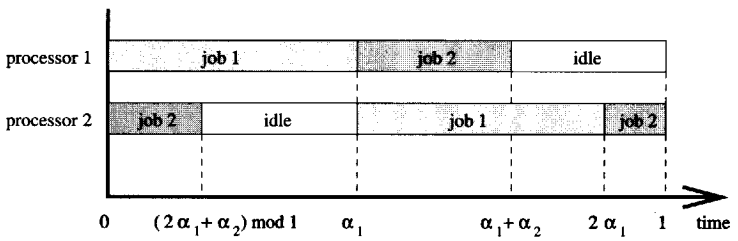


Figure 4.2: A possible schedule when $\sigma_1 \leq c_P$ (in this example $P = 2$ and $J = 2$).

Case 2: $\sigma_1 \leq c_P$

When $\sigma_1 \leq c_P$, we define, for each $j \in \mathcal{J}$,

$$\alpha_j \stackrel{\triangle}{=} \frac{\sigma_j}{\sum_{p=1}^{P} c_p}. \tag{4.9}$$

The idea of the schedule $\mathcal{F}$ is to put job j a fraction α_j of the time on each processor in turn. For notational convenience, we put $t_0 \stackrel{\triangle}{=} 0$, $t_j \stackrel{\triangle}{=} \alpha_1 + \cdots + \alpha_j$ for $j \in \mathcal{J}$, and we define $x \oplus y \stackrel{\triangle}{=} (x + y) \bmod 1$ and $x \ominus y \stackrel{\triangle}{=} (x - y) \bmod 1$ for $x, y \in [0, 1)$. The schedule is constructed by assigning to processor 1 job 1 from $t = t_0$ to $t = \alpha_1$, then job 2 from $t = \alpha_1$ to $t = \alpha_1 + \alpha_2$, and so on. The schedule for a processor $p > 1$ is constructed by shifting the schedule for processor $p - 1$ by an amount α_1 to the right. This results in the following schedule (cf. Figure 4.2), for $j \in \mathcal{J}$,

$$\begin{aligned} f_{j1}(t) &= \begin{cases} 1, & t_{j-1} \leq t < t_j, \\ 0, & \text{otherwise}, \end{cases} \\ f_{jp}(t) &= f_{j,p-1}(t \ominus \alpha_1), \ 0 \leq t < 1, \ p = 2, \ldots, P. \end{aligned} \tag{4.10}$$

It is easily verified that (4.1) and (4.3) hold. Also, clearly, the average service rate of job j equals σ_j, $j \in \mathcal{J}$. We still need to prove that (4.2) holds, i.e., that the schedules

of an arbitrary job j on two arbitrary, distinct processors p_1 and p_2 are non-overlapping (i.e., $f_{jp_1}(t) + f_{jp_2}(t) = 0$ or 1). It is obvious from Figure 4.2 that this holds for $j = 1$, since job 1 is scheduled on each processor in turn, starting on processor 1 at $t = 0$, and ending at processor P at $t = P\alpha_1 < 1$ by (4.9) and $\sigma_1 \leq c_P$. Now consider an arbitrary job j. Job j runs on processor 1 during the interval

$$[\alpha_1 + \cdots + \alpha_{j-1}, \alpha_1 + \cdots + \alpha_j), \tag{4.11}$$

and on processor p during

$$[(p\alpha_1 + \alpha_2 + \cdots + \alpha_{j-1}) \bmod 1, (p\alpha_1 + \alpha_2 + \cdots + \alpha_j) \bmod 1). \tag{4.12}$$

These intervals are non-overlapping because

$$p\alpha_1 + \alpha_2 + \cdots + \alpha_{j-1} \geq \alpha_1 + \cdots + \alpha_j,$$

and

$$p\alpha_1 + \alpha_2 + \cdots + \alpha_j \leq \alpha_1 + \cdots + \alpha_{j-1} + 1.$$

By now, we have proven that the schedules of job j on processor 1 and an arbitrary other processor are non-overlapping. But this must also hold for two arbitrary processors, because the schedule on processor p is simply the schedule of processor 1 shifted (modulo 1) an amount $p\alpha_1$ to the right. Therefore, we can shift all schedules an amount $p\alpha_1$ to the left (modulo 1), renumber the processors such that processor p becomes processor 1, processor $(p + 1) \bmod P$ becomes processor 2, etc., and apply the preceeding arguments.

The reduction step

When $J > 1$ and $\sigma_1 > c_P$, we proceed as follows. We schedule job 1 on processors q and $q + 1$ with q uniquely determined by

$$c_q \geq \sigma_1 > c_{q+1}. \tag{4.13}$$

Because $\sigma_1 \leq c_1$ by (4.5), such a processor q always exists. Also, $q < P$ since $\sigma_1 > c_P$.

In the *reduced system*, job 1 has been removed and processors q and $q+1$ are replaced with a single new processor with capacity $c_q + c_{q+1} - \sigma_1$. We use accents to distinguish the processors in the reduced system from those in the original system (e.g., c'_p is the capacity of processor p in the reduced system). In the reduced system, the processors are renumbered from 1 onwards, the job numbering remains the same $(2, \ldots, J)$. Obviously, because of (4.13), we have $c'_1 \geq \cdots \geq c'_{P-1}$. Because of (4.5) we have, for $j = 1, \ldots, q-1$,

$$\sum_{k=2}^{j+1} \sigma_k \leq \sum_{k=1}^{j} \sigma_k \leq \sum_{p=1}^{j} c_p = \sum_{p=1}^{j} c'_p, \tag{4.14}$$

and for $j = q, \ldots, J-1$,

$$\sum_{k=2}^{j+1} \sigma_k \leq \sum_{p=1}^{q-1} c_p + c_q + c_{q+1} - \sigma_1 + \sum_{p=q+2}^{\min(j+1,P)} c_p = \sum_{p=1}^{\min(j,P-1)} c'_p. \tag{4.15}$$

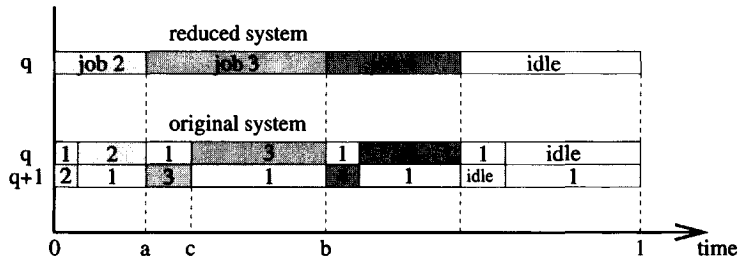


Figure 4.3: The transformation of the schedule on processor q in the reduced system into schedules for processors q and $q+1$ in the original system.

In other words, relations (4.5) hold for the reduced system. Therefore, we can repeat reducing the system until either $J = 1$ or $\sigma_1 \leq c_P$, for which schedules were found earlier in cases 1 and 2, respectively.

What remains to be proven is that a schedule $\mathcal{F}$ with average service rates $\sigma_1, \ldots, \sigma_J$ can be constructed from a schedule $\mathcal{F}' \triangleq \{f'_{jp} | j = 2, \ldots, J,\ p = 1, \ldots, P-1\}$ with average service rates $\sigma_2, \ldots, \sigma_J$ for the reduced system. Suppose such a schedule $\mathcal{F}'$ exists for the reduced system. First of all, the schedule $\mathcal{F}'$ on every processor but q in the reduced system is used unchanged in the schedule $\mathcal{F}$ on the corresponding processor in the original system, and job 1 is not served on any of these processors. Because $\mathcal{F}'$ is a schedule, (4.1) holds for $\mathcal{F}$ for $j = 1, \ldots, J$ and $p = 1, \ldots, q-1, q+2, \ldots, P$, and (4.3) holds for $\mathcal{F}$ for $p = 1, \ldots, q-1, q+2, \ldots, P$. The schedule on processor q in the reduced system, i.e., f'_{jq}, $j = 2, \ldots, J$, is transformed into a schedule on processors q and $q+1$ in the original system, i.e., f_{jq} and $f_{j(q+1)}$, $j = 1, \ldots, J$, as is shown in Figure 4.3. In the new schedule $\mathcal{F}$, job 1 is served only by processors q and $q+1$. The transformation is as follows. Let $[a, b)$ be a maximal subinterval of $[0, 1)$ on which the f'_{jq}, $j = 2, \ldots, J$ are constant. We now put

$$f_{1q}(t) = \begin{cases} 1, & a \leq t < c, \\ 0, & c \leq t < b, \end{cases} \tag{4.16}$$

$$f_{1(q+1)}(t) = \begin{cases} 0, & a \leq t < c, \\ 1, & c \leq t < b, \end{cases} \tag{4.17}$$

and, for $j = 2, \ldots, J$,

$$f_{jq}(t) = \begin{cases} 0, & a \leq t < c, \\ f'_{jq}(t), & c \leq t < b, \end{cases} \tag{4.18}$$

$$f_{j(q+1)}(t) = \begin{cases} f'_{jq}(t), & a \leq t < c, \\ 0, & c \leq t < b, \end{cases} \tag{4.19}$$

where

$$c \triangleq a + \frac{\sigma_1 - c_{q+1}}{c_q - c_{q+1}}(b - a). \tag{4.20}$$

From (4.13), we have $a < c \leq b$. It is immediately obvious that for $\mathcal{F}$, (4.1) holds for $j = 1, \ldots, J$ and $p = q, q+1$, and (4.3) holds for $p = q, q+1$. Also, (4.2) holds for $j = 1$ by

(4.16) and (4.17), and for $j = 2, \ldots, J$ by (4.18) and (4.19) and because $\mathcal{F}'$ is a schedule. At this point, we have proven that $\mathcal{F}$ is a schedule. We still need to prove that the service rates $\sigma_1, \ldots, \sigma_J$ are obtained in $\mathcal{F}$. Clearly, the average service rate of job 1 in schedule $\mathcal{F}$ is

$$\left(\frac{\sigma_1 - c_{q+1}}{c_q - c_{q+1}}\right) c_q + \left(1 - \frac{\sigma_1 - c_{q+1}}{c_q - c_{q+1}}\right) c_{q+1} = \sigma_1. \tag{4.21}$$

For jobs $j = 2, \ldots, J$, we only need to show that the average service rate obtained on processor q in $\mathcal{F}'$ is equal to that obtained on processors q and $q+1$ in $\mathcal{F}$. Referring to Figure 4.3 and (4.20), this is evident from

$$\left(1 - \frac{\sigma_1 - c_{q+1}}{c_q - c_{q+1}}\right) c_q + \left(\frac{\sigma_1 - c_{q+1}}{c_q - c_{q+1}}\right) c_{q+1} = c'_q. \tag{4.22}$$

This concludes the proof of Theorem 4.1. □

4.2.2 The Multiprocessor Group-Priority Processor-Sharing Policy

In this section, we introduce Multiprocessor Group-Priority Processor Sharing (MGPPS), a scheduling policy for multiprocessors that provides each group with at least its feasible group share with zero capacity loss. The MGPPS policy is a special case of MJPPS (see Section 2.3.2), and an extension of GPPS (see Section 3.4.3) to multiprocessors.

The problem we must solve is to calculate the so-called *group weights* w_{gp}, $g = 1, \ldots, G$, $p = 1, \ldots, P$, given the required group shares $r_1^{\mathrm{G}}, \ldots, r_G^{\mathrm{G}}$, the numbers of jobs $N_1, \ldots, N_G$, and the processor capacities $c_1 \geq \cdots \geq c_P > 0$. The group weight w_{gp} denotes the fraction of time processor p spends on serving jobs of group g. The service to group g on processor p is equally distributed among the jobs of group g.

Obviously, in order to be able to construct a schedule that satisfies (4.1), (4.2), and (4.3), the group weights w_{gp} must satisfy the following conditions:

$$0 \leq w_{gp} \leq 1, \quad g = 1, \ldots, G, \; p = 1, \ldots, P, \tag{4.23}$$

$$\sum_p w_{gp} \leq N_g, \quad g = 1, \ldots, G, \tag{4.24}$$

$$\sum_g w_{gp} \leq 1, \quad p = 1, \ldots, P. \tag{4.25}$$

We define $N \triangleq \sum_g N_g$. For $\Delta^{\mathrm{T}} = 0$ we must have

$$\sum_g \sum_p w_{gp} c_p = \sum_{p=1}^{\min(N,P)} c_p, \tag{4.26}$$

where the left-hand side is the (average) total obtained share o^{T}, because we assume that $\sum_p c_p = 1$, and the right-hand side is the (constant) total feasible share f^{T} by (2.11) and (2.41). For $\Delta^{\mathrm{G}} \leq 0$ we must have, for $g = 1, \ldots, G$,

$$\sum_p w_{gp} c_p \geq \min(r_g^{\mathrm{G}}, N_g/P), \tag{4.27}$$

```
(01) foreach g ∈ {1,...,G} do
(02)   begin
(03)     M_g := N_g // max allocatable to g
(04)     foreach p ∈ {1,...,P} do w_gp := 0
(05)   end
(06)
(07) for p := 1 step 1 until P do
(08)   begin
(09)     B := 0 // allocated so far on p
(10)     while B < 1 do
(11)       begin
(12)         G := {g | M_g > 0}
(13)         if G = ∅ then done
(14)         r_G := Σ_{g∈G} r_g^G
(15)         b := min_{g∈G} {M_g r_G / r_g^G} // next allocation on p
(16)         b := min (b, 1 − B)
(17)         foreach g ∈ G do
(18)           begin
(19)             w_gp += r_g^G b / r_G
(20)             M_g −= r_g^G b / r_G
(21)           end
(22)         B += b
(23)       end
(24)   end
```

Figure 4.4: The procedure for calculating the w_{gp}'s in MGPPS.

where the left-hand side is the (average) obtained group share o_g^{G}, and the right-hand side is the (constant) feasible group share f_g^{G} by (2.26).

The procedure for calculating the w_{gp}'s is shown in Figure 4.2.2. The underlying idea is to allocate processor time to groups in descending order of the processor capacities, until all processor time has been allocated (in which case we fall through line 24) or until there are no jobs left (the done statement in line 13). For each group, in line 3 we initialize the variable M_g, which holds the remaining processor time that can be allocated to the entire group. The for-loop in line 7 considers the processors in descending order of the capacities. The variable B initialized in line 9 holds the fraction of time already allocated on processor p. We use this variable to make sure that we do not allocate more than 100% of the time on processor p. We can only allocate processor time to groups that (still) have $M_g > 0$; these groups are selected in lines 12 through 14. However, because M_g can become zero for some group g anywhere in the process, we need a second loop starting at line 10, and the statements in lines 15 and 16 to ascertain that $M_g \geq 0$ and $B <= 1$. In line 19, we allocate processor time to groups at rates proportional to the required group shares.

It is easily verified that the procedure outlined in Figure 4.2.2 satisfies the conditions (4.23), (4.24), (4.25), and (4.26), in other words, we can construct a schedule with $\Delta^{\mathrm{T}} = 0$.

In order to see that it also satisfies (4.27), first observe that in the worst case for group g, all other groups are present on all processors, with $w_{hp} \geq r_h^{\mathrm{G}}$, $h \neq g$, $p = 1, \ldots, P$. We will show that in this case $o_g^{\mathrm{G}} \geq f_g^{\mathrm{G}}$ and thus $\Delta_g^{\mathrm{G}} \leq 0$. From the procedure in Figure 4.2.2 it follows that

$$w_{gp} = \begin{cases} r_g^{\mathrm{G}}, & p = 1, \ldots, \min(Q_g, P), \\ N_g - r_g^{\mathrm{G}} Q_g, & p = Q_g + 1 \text{ and } Q_g < P, \\ 0, & p = Q_g + 2, \ldots, P, \end{cases} \tag{4.28}$$

with

$$Q_g \triangleq \lfloor N_g / r_g^{\mathrm{G}} \rfloor. \tag{4.29}$$

If $Q_g < P$, we have $r_g^{\mathrm{G}} > N_g/P$ and therefore $f_g^{\mathrm{G}} = N_g/P$ by (2.26). In this case,

$$o_g^{\mathrm{G}} = \sum_{p=1}^{P} w_{gp} c_p \geq \sum_{p=1}^{P} \frac{N_g}{P} c_p = \frac{N_g}{P}, \tag{4.30}$$

because $c_1 \geq \cdots \geq c_P > 0$, $w_{g1} \geq \cdots \geq w_{gP}$, and $\sum_p w_{gp} = N_g$. So $o_g^{\mathrm{G}} \geq f_g^{\mathrm{G}}$ and hence $\Delta_g^{\mathrm{G}} \leq 0$. If $Q_g \geq P$, we have $r_g^{\mathrm{G}} \geq N_g/P$, and $f_g^{\mathrm{G}} = r_g^{\mathrm{G}}$. Now group g simply obtains a share of r_g^{G} on each processor, and thus $o_g^{\mathrm{G}} = r_g^{\mathrm{G}}$ and $\Delta_g^{\mathrm{G}} = 0$. As a result, MGPPS guarantees compliance with the feasible group shares with zero capacity loss.

4.3 Probabilistic Share Scheduling

An important global static scheduling policy is *random splitting* [84], in which arriving jobs are sent to processors according to fixed probabilities. Clearly, random splitting is a static policy, because the scheduling decisions do not depend on the current state of the system. Random splitting is the subject of the remainder of the present chapter. In Section 4.3.1 we introduce the system and workload models. Because there is a substantial amount of related work on random splitting, we discuss a selection of the relevant literature in Section 4.3.2. Partially based upon this literature survey, in Section 4.3.3 we state our research questions for random splitting. In Section 4.4, we concentrate on the minimization of the capacity loss Δ^{T}, given the total load ρ, and in Section 4.5, we study the minimization of the group-share deviation Δ^{G}. Because the problems we study are relatively new, we will restrict ourselves in this section to Poisson arrivals and exponential service times.

4.3.1 System and Workload Models

In our model, arriving jobs are routed probabilistically to one of the P processors, where they are served until completion; job migration is not allowed. Unless noted otherwise, the local scheduling policy on each processor is Processor Sharing (PS). The capacity of processor p is c_p, $p = 1, \ldots, P$, and we assume that $c_1 \geq \cdots \geq c_P > 0$, and that $\sum_p c_p = 1$.

An arriving job of group g is sent to processor p with probability π_{gp}, $g = 1, \ldots, G$, $p = 1, \ldots, P$. The routing probabilities π_{gp} are constant. Obviously, for $g = 1, \ldots, G$,

$$\sum_{p=1}^{P} \pi_{gp} = 1. \tag{4.31}$$

Jobs arrive according to G independent Poisson processes with rates $\lambda_1, \ldots, \lambda_G$. There are no permanent jobs. The normalized service times (w.r.t. a processor of capacity 1) of jobs of group g are independent random variables with an exponential distribution and mean $1/\mu_g$. We define, for $g = 1, \ldots G$, $p = 1, \ldots, P$, the *load* ρ_{gp} *on processor* p *due to group* g as

$$\rho_{gp} \triangleq \frac{\pi_{gp}\lambda_g}{c_p\mu_g}, \tag{4.32}$$

the *load* ρ_{*p} *on processor* p as

$$\rho_{*p} \triangleq \sum_{g=1}^{G} \rho_{gp}, \tag{4.33}$$

the *group load* ρ_{g*} *of group* g as

$$\rho_{g*} \triangleq \frac{\lambda_g}{\mu_g} = \sum_{p=1}^{P} c_p\rho_{gp}, \tag{4.34}$$

and the *total load* ρ as

$$\rho \triangleq \sum_{g=1}^{G} \rho_{g*}. \tag{4.35}$$

We assume that ρ and ρ_{g*}, $g = 1, \ldots, G$ are given, and that $\rho < 1$.

We denote the $G \times P$-matrix with the numbers of jobs of the individual groups on the individual processors at time t by $\boldsymbol{N}(t)$. The $G \times 1$-vector holding the numbers of jobs of the individual groups on processor p at time t is denoted by $\boldsymbol{N}_{*p}(t)$. The $1 \times P$-vector holding the numbers of jobs of group g on the individual processors at time t is denoted by $\boldsymbol{N}_{g*}(t)$. Finally, $\boldsymbol{N}_{*,p_1\cdots p_2}(t)$ denotes the $G \times (p_2 - p_1 + 1)$-matrix with the number of jobs of the individual groups on processors p_1 through p_2 at time t. As in Chapter 3, we omit the argument t when referring to the stationary distribution of a random variable, so for a $G \times P$-matrix $\boldsymbol{n}$,

$$\mathbf{P}\left[\boldsymbol{N} = \boldsymbol{n}\right] \triangleq \lim_{t\to\infty} \mathbf{P}\left[\boldsymbol{N}(t) = \boldsymbol{n}\right].$$

Also, normal-typeface letters refer to the sum of the elements of the corresponding vector or matrix, for instance

$$\begin{aligned} N(t) &\triangleq |\boldsymbol{N}(t)|, \\ n &\triangleq |\boldsymbol{n}|. \end{aligned}$$

One of the nice features of random splitting is that the arrival streams at the processors are still Poisson. If the π_{gp}'s are chosen such that $\rho_{*p} < 1$, $p = 1, \ldots, P$, then the joint queue lengths have an equilibrium distribution with a product form. This distribution is given by, using (3.80),

$$\mathbf{P}\left[\boldsymbol{N} = \boldsymbol{n}\right] = \prod_{p=1}^{P} (1 - \rho_{*p}) \left((n_{1p} + \cdots + n_{Gp})!\right) \prod_{g=1}^{G} \frac{\rho_{gp}^{n_{gp}}}{n_{gp}!}, \tag{4.36}$$

where $\boldsymbol{n} \triangleq (n_{gp})$ (a $G \times P$-matrix). For the equilibrium distribution of the total number of jobs, we find by (3.82), if $\rho_{*p} < 1$, $p = 1, \ldots, P$, for $n = 0, 1, 2, \ldots$,

$$\mathbf{P}[N = n] = \sum_{\substack{n_1=0 \\ n_1+\cdots+n_P=n}}^{n} \cdots \sum_{n_P=0}^{n} \prod_{p=1}^{P} (1 - \rho_{*p}) \rho_{*p}^{n_p}. \tag{4.37}$$

However, in some of the following sections, we will also consider cases in which we allow $\rho_{*p} \geq 1$. Even though a stationary probability distribution for the joint queue length does not exist then, it may still exist for Δ^{T} or Δ^{G}.

4.3.2 Related Literature

Our overview of related literature on random splitting consists of four parts. First, we discuss the wait-while-idle probability and the probability of load-balancing success, and show that these are high with random splitting, indicating the potential performance benefits of using dynamic policies over random splitting. Second, we compare random splitting to dynamic policies. Third, we describe single-class optimization problems with random splitting. Fourth, we discuss multiclass problems. Unless noted otherwise, the local scheduling discipline in the models described in this section is FCFS.

The literature overview presented here is not complete. Our main purpose is to compare the objectives of minimization of the capacity loss and of the group-share deviation to minimization of response and waiting times with random splitting.

The Wait-While-Idle Probability and the Probability of Load-Balancing Success

Capacity loss is closely related to what Livny and Melman [61] call the *wait-while-idle probability* $\mathbf{P}_{\mathrm{wwi}}$, the probability that one or more processors are idle while at least one processor has two or more jobs present in its queue. For the homogeneous case with identical loads ρ at the P processors (for instance, through random splitting with equal probabilities for the processors), we have[1]

$$\begin{aligned} \mathbf{P}_{\mathrm{wwi}} &= \sum_{p=1}^{P} \binom{P}{p} (1-\rho)^p \left(\rho^{P-p} - [(1-\rho)\rho]^{P-p} \right) \\ &= 1 - \rho^P - (1-\rho)^P \left((1+\rho)^P - \rho^P \right), \end{aligned} \tag{4.38}$$

and for $0 < \rho < 1$,

$$\lim_{P \to \infty} \mathbf{P}_{\mathrm{wwi}} = 1. \tag{4.39}$$

In Figure 4.5 we show $\mathbf{P}_{\mathrm{wwi}}$ as a function of ρ for various values of P (after [61]), which indicates the potential benefits of load-sharing and load-balancing policies for the reduction of the wait-while-idle time.

Rommel [73] extends the idea of $\mathbf{P}_{\mathrm{wwi}}$ to that of the *probability of load-balancing success*, $\mathbf{P}_{\mathrm{lbs}}(n, m)$, $0 \leq n < m$, which is the probability that at least one of the processors

[1] We believe there is an error in the expression in [61].

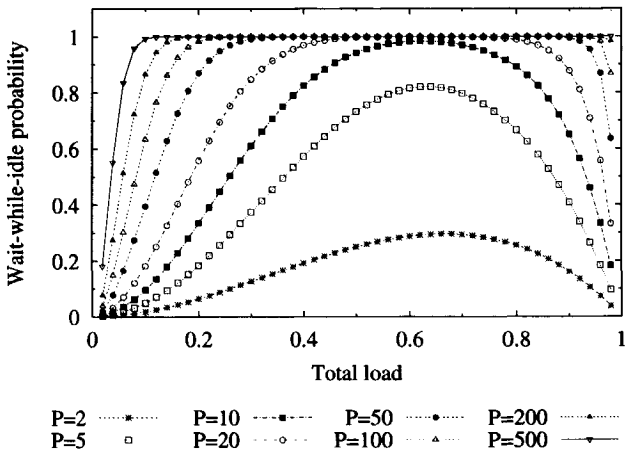


Figure 4.5: The wait-while-idle probability $\mathbf{P}_{\text{wwi}}$ as a function of the total load ρ for various values of the number P of processors (after [61]).

has n or fewer jobs present, while at least one other processor has m or more jobs present. Clearly $\mathbf{P}_{\text{wwi}} = \mathbf{P}_{\text{lbs}}(0,2)$. For a P-processor homogeneous system, we have

$$\begin{aligned}\mathbf{P}_{\text{lbs}}(n,m) \\ &= 1 - \mathbf{P}\left[N_{*p} < m,\ p = 1,\ldots,P\right] - \mathbf{P}\left[N_{*p} > n,\ p = 1,\ldots,P\right] \\ &\quad + \mathbf{P}\left[n < N_{*p} < m,\ p = 1,\ldots,P\right]. \end{aligned} \tag{4.40}$$

If the loads on the processors are all equal to $\rho < 1$, we have

$$\mathbf{P}_{\text{lbs}}(n,m) = 1 - (1-\rho^m)^P - \rho^{(n+1)P} + \left(\rho^{n+1} - \rho^m\right)^P, \tag{4.41}$$

and, for $0 < \rho < 1$,

$$\lim_{P\to\infty} \mathbf{P}_{\text{lbs}}(n,m) = 1. \tag{4.42}$$

Whereas $\mathbf{P}_{\text{wwi}}$ focuses on processors being unnecessarily idle in the system, $\mathbf{P}_{\text{lbs}}(n,m)$ more generally exposes unbalances in the system. However, both are less meaningful in heterogeneous systems.

Comparisons between Random Splitting and Dynamic Policies

Random splitting is often compared to dynamic policies in order to indicate the potential performance benefits of the latter. For the homogeneous case, Wang and Morris [84] compare random splitting (and the dual policy, *random service*) with equal probabilities for all queues with cyclic splitting (which is also static) and dynamic policies such as Join Shortest Queue (JSQ). In general, the dynamic policies and cyclic splitting outperform random splitting with respect to $Q_A(\rho)$, for which we gave the definition in Section 1.3.3.

Eager et al. [24] study random splitting with equal probabilities for all queues as a location policy in load balancing. Jobs arrive at the individual processors according to independent Poisson processes. At each processor, a (dynamic) transfer policy decides whether to process an arriving job locally, or to send it to another processor; a job is eligible for transfer to another processor if the (local) queue length equals or exceeds a fixed threshold T. The combination of this simple transfer policy and random splitting as a location policy substantially improves performance, in terms of mean job response time, over a system without load balancing.

Chow and Kohler [13] compare three dynamic policies (see Section 1.3.6) with the *proportional branching (PB) policy*, a random-splitting policy in which the fraction of jobs sent to each processor is proportional the processor's capacity. Arrivals are Poisson with rate λ, the service time distribution is exponential with mean $1/\mu$ on a processor with unit capacity, and the local scheduling discipline is FCFS. Hence, the mean job response time $\mathbf{E}[R]$ with PB is (with $\sum_p c_p = 1$)

$$\mathbf{E}[R] = \sum_{p=1}^{P} c_p \left(\frac{1/(c_p\mu)}{1-\rho} \right) = \frac{P}{\mu - \lambda}. \tag{4.43}$$

For a two-processor system, the authors show that the mean job response time is substantially reduced with the dynamic policies instead of with PB. As we will see below, in heterogenous systems with random splitting, proportional branching does not minimize the mean job response time.

Single-Class Random-Splitting Optimization Problems

There are many articles that focus on random-splitting optimization problems. In this section, we concentrate on the minimization of the mean job waiting time and the mean job response time. In the descriptions of the articles (and in those in the next section on multi-class optimization problems), we mostly stick to the original notation.

Buzen and Chen [9] consider a heterogeneous system. Jobs arrive according to a Poisson process with rate λ, and job service times are i.i.d. with mean $1/\mu_p$ and variance σ_p^2 on processor p, $p = 1, \ldots, P$. The service-time distributions may be of different type on each processor. We assume that $\mu_1 \geq \cdots \geq \mu_P$. The arrival rate at processor p is λ_p. The objective is to minimize the mean job response time, given by the Pollaczek-Khinchine fomula [1]:

$$\mathbf{E}[R] = \sum_{p=1}^{P} \frac{\lambda_p}{\lambda} \left(\frac{1}{\mu_p} + \frac{(\lambda_p/\mu_p)^2 + \lambda_p^2\sigma_p^2}{2\lambda_p(1-\lambda_p/\mu_p)} \right), \tag{4.44}$$

subject to

$$0 \leq \lambda_p < \mu_p, \qquad p = 1, \ldots, P, \tag{4.45}$$

$$\sum_{p=1}^{P} \lambda_p = \lambda. \tag{4.46}$$

On each processor, any scheduling discipline for which the Pollaczek-Khinchine formula holds is allowed, such as FCFS and PS. One can solve the optimization problem by defining a Lagrangean function and solving for the roots of its gradient with respect to $\lambda_1, \ldots, \lambda_P$.

Due to the nice mathematical structure of the objective function (4.44) (the p'th term only depends on λ_p, μ_p, and σ_p^2, not on the figures for any of the other processors), this approach yields a closed-form expression for the λ_p's. As it turns out, the optimal arrival rates at the processors are, for $p = 1, \ldots, P$,

$$\lambda_p = \mu_p \left(1 - \sqrt{\frac{\mu_p^2 \sigma_p^2 + 1}{\delta \mu_p + \mu_p^2 \sigma_p^2 - 1}} \right), \tag{4.47}$$

with δ chosen such that $\sum_p \lambda_p = \lambda$. However, if one or more of the λ_p's is smaller than zero in the solution, the minimum is outside the feasible region, and we must put $\lambda_P = 0$ (i.e., on the slowest processor), and solve the optimization problem for the remaining $P - 1$ processors. When the service times are exponentially distributed, we have $\sigma_p^2 = 1/\mu_p^2$, and

$$\lambda_p = \mu_p \left(1 - \sqrt{\frac{1}{\delta' \mu_p}} \right), \tag{4.48}$$

with $\delta' = \delta/2$ chosen such that $\sum_p \lambda_p = \lambda$. For this case, Ni and Hwang [69] provide an algorithm that determines the value of δ' and the number of processors that have $\lambda_p > 0$.

Combé and Boxma [19] compare random splitting to another static policy, *pattern allocation*, which assigns jobs to processors following a fixed pattern, e.g., according to an infinite sequence a_n, $n = 1, 2, \ldots$, where a_n describes where to send the n'th arriving job. Usually, this sequence is periodic and completely described by a finite-length *pattern*. Jobs arrive according to a Poisson process with rate λ, and must be irrevocably assigned to one of P FCFS processors. Service times are independent with the service time of a job assigned to processor p having a general distribution $B_p(\cdot)$ with first and second moments β_p and $\beta_p^{(2)}$, respectively. The objective is to minimize

$$\sum_{p=1}^{P} f_p C_p \mathbf{E}[W_p], \tag{4.49}$$

where $\mathbf{E}[W_p]$ is the mean waiting time at processor p, C_p is the cost factor for waiting at queue p, and f_p are additional, load-dependent, weight factors. For random splitting, minimization of (4.49) in some cases allows an analytical solution, dependent on the C_p's and f_p's. For instance, denoting the arrival rate at processor p by λ_p, with $f_p = \lambda_p/\lambda$ the optimal arrival rates are

$$\lambda_p = \frac{1}{\beta_p} \left(1 - \left(\sqrt{1 + \frac{2\beta_p \delta}{C_p \beta_p^{(2)}}} \right)^{-1} \right), \quad p = 1, \ldots, P, \tag{4.50}$$

with δ chosen such that $\sum_p \lambda_p = \lambda$. If in addition $C_p = 1$, $p = 1, \ldots, P$, the objective function (4.49) becomes the overall mean job waiting time. In that case, with exponential service-time distributions the optimal arrival rate at processor p is, since $\beta_p^{(2)} = 2\beta_p^2$,

$$\lambda_p = \frac{1}{\beta_p} \left(1 - \sqrt{\frac{\beta_p}{\beta_p + \delta}} \right). \tag{4.51}$$

Under pattern allocation, the arrival process at each queue is a so-called *Markov Arrival Process (MAP)*, in which arrivals occur at certain transitions in some underlying continuous-time Markov process with finite state space. The MAP arrivals at the individual processors are more regular than the Poisson arrivals in the case of random splitting, and therefore, a MAP/G/1 queue has a lower mean waiting time than an M/G/1 queue with the same mean arrival rate. However, in many cases, finding the optimal pattern presents formidable mathematical problems. In this thesis, we will not consider pattern allocation.

Multiclass Random-Splitting Optimization Problems

Ni and Hwang [69] extend the model of Buzen and Chen [9] described earlier by classifying jobs into m classes. Jobs arrive according to independent per-class Poisson processes. The service times of jobs are exponentially distributed with mean $1/\mu_p$ when the job is sent to processor p. Each processor can only serve jobs from prespecified job classes. The authors present a recursive algorithm to determine the optimal routing probabilities for this case.

Borst [7] studies a model in which G independent Poisson arrival streams, corresponding to G job classes, must be assigned through random splitting to P processors. The processors may have different speeds. The arrival rate of jobs of class g is λ_g, and service times on a processor of unit capacity are i.i.d. with first and second means β_g and $\beta_g^{(2)}$, respectively. For stability[2], $\sum_g \lambda_g \beta_g < 1$ must hold. The processors are not allowed to discriminate among the groups, and hence the expected waiting time of a job sent to a particular processor does not depend on the group to which the job belongs, and is given by the well-known result for the M/G/1 queue.

The objective is to minimize the *mean total waiting cost per unit of time*, defined as

$$\sum_{g=1}^{G} C_g \lambda_g \mathbf{E}[W_g], \tag{4.52}$$

where C_g is the waiting cost per unit of time for group-g jobs, and $\mathbf{E}[W_g]$ is the mean job waiting time of jobs of group g. Note that when $G = 1$, the problem studied by Buzen and Chen appears. The following ergodicity conditions must hold:

$$\sum_{g=1}^{G} \pi_{gp} \lambda_g \beta_g < c_p, \ p = 1, \ldots, P. \tag{4.53}$$

The objective function is non-linear and not convex in the π_{gp}'s, so obtaining the global optimum is non-trivial. Borst [7] provides details on the structure of the solution, and addresses the fundamental question whether with the optimal assignment matrix (π_{gp}), the traffic mix at each processor follows the traffic mix presented to the total system. This is *not* the case in general. In fact, in the optimal case, most of the routing probabilities are either 0 or 1.

Sethuraman and Squillante [75] consider a multiclass random-splitting model similar to that of Borst, but with HOL as the local scheduling policy. We will use the same

[2]In our description, we assume without loss of generality that $\sum_p c_p = 1$. In [7], this assumption is not made.

notation. The objective is to minimize a linear cost function in the per-class mean job response times, i.e.,

$$\sum_{g=1}^{G} C_g \frac{\lambda_g}{\lambda} \mathbf{E}[R_g], \tag{4.54}$$

$C_g > 0$, where $\mathbf{E}[R_g]$ is the expected response time of jobs of group g. Without loss of generality, it can be assumed that

$$\frac{C_1}{\beta_1} \geq \frac{C_2}{\beta_2} \cdots \geq \frac{C_G}{\beta_G}. \tag{4.55}$$

Locally, group g has (non-preemptive) priority over group h if $g < h$. HOL minimizes the objective function locally [77]. The mean job response time $\mathbf{E}[R_{gp}]$ of jobs of group g on processor p is [52]

$$\mathbf{E}[R_{gp}] = \frac{\sum_{g=1}^{G} \lambda_g \beta_g^2 / c_p^2}{2(1 - \sum_{h<g} \rho_{hp})(1 - \sum_{h \leq g} \rho_{hp})} + \frac{\beta_g}{c_p}, \tag{4.56}$$

and the mean job response time of group-g jobs is

$$\mathbf{E}[R_g] = \sum_{p=1}^{P} \pi_{gp} R_{gp}. \tag{4.57}$$

The authors prove that any local minimum in the interior of the feasible domain is a global minimum.

4.3.3 Research Questions

Our main interest in the remainder of this chapter is to what extent the capacity loss and the group-share deviation can be controlled with the routing matrix (π_{gp}). In Section 4.4 we study the problem of minimizing Δ^{T}, in Section 4.5, we study Δ^{G}. In addtion, we will answer the following research questions:

1. How does the minimization of the capacity loss compare to the minimization of (1) the wait-while-idle probability $\mathbf{P}_{\mathrm{wwi}}$, (2) the probability of load-balancing success $\mathbf{P}_{\mathrm{lbs}}(n, m)$, (3) the mean job response time, and (4) the mean job waiting time? To our knowledge, the problem of minimization of the capacity loss in a random-splitting system has not been addressed before in the literature.

2. Does the potential improvement of compliance with the feasible group shares in a distributed system outweigh the increase in capacity loss compared to a uniprocessor PS system of the same capacity? As we have shown in Chapter 3, PS is not very suitable for share scheduling on uniprocessors since it cannot discriminate among the jobs of different groups. Clearly, such a form of control *is* present in the random-splitting model: by means of setting (π_{gp}), one can give preference to one group over another. Unfortunately, the use of multiple processors comes at the cost of capacity loss: processors can be idle while others serve more than one job, or some processor can be idle while a lower-capacity processor is busy.

3. In the optimal case with respect to minimizing the group-share deviation, is the traffic mix at the processors homogeneous or heterogeneous? This question is the same as in [7]. We have strong reasons to believe that in our case, the optimal traffic mix at the processors is heterogeneous. We cannot provide a rigorous proof of this, but compare two heuristic splitting strategies in Section 4.5: Horizontal Partitioning in Random Splitting (HPRS), which is completely homogeneous, and Vertical Partitioning in Random Splitting (VPRS), which is heterogeneous.

The question as to whether dynamic policies outperform static policies with respect to the minimization of capacity loss and group-share deviation will be deferred until Chapters 5 and 6.

4.4 Probabilistic Share Scheduling: Minimization of the Capacity Loss

In this section, our objective is to minimize the capacity loss Δ^{T} with random splitting. In Section 4.4.1, we state the problem and some of its fundamental mathematical properties. In Section 4.4.2, we first consider homogeneous systems. As it turns out, balancing the load ρ among the processors minimizes the capacity loss in homogeneous systems. Subsequently, in Section 4.4.3 we study the minimization of the capacity loss in heterogeneous systems. For the two-processor case, we give a closed-form expression for the workload distribution that minimizes the capacity loss. For an arbitrary number of processors, such an expression could not be found.

4.4.1 Problem Statement

Our objective is to find the routing matrix (π_{gp}) that minimizes the capacity loss Δ^{T}. By Equation 2.42 we have $\Delta^{\mathrm{T}}(t) = f^{\mathrm{T}}(t) - o^{\mathrm{T}}(t)$. From (2.7) it is clear that the set of processors busy at time t and their capacities completely determine $o^{\mathrm{T}}(t)$. Similarly, by (2.11) and (2.41) the total number $N(t)$ of jobs present at time t completely determines $f^{\mathrm{T}}(t)$. As a consequence, Δ^{T} is completely determined by ρ_{*p}, c_p, $p = 1, \ldots, P$. We first introduce a proposition that explains this relationship between Δ^{T} and the ρ_{*p}'s and c_p's. Subsequently, in Problem 4.3 we state the problem of minimizing Δ^{T} in mathematical terms.

Proposition 4.2 *In a random-splitting system, if $\rho < 1$ and the π_{gp}'s ($g = 1, \ldots, G$, $p = 1, \ldots, P$) are chosen such that $\rho_{*p} < 1$, $p = 1, \ldots, P$, we have for the capacity loss Δ^{T}:*

$$\Delta^{\mathrm{T}} = 1 - \rho - \sum_{n=0}^{P-1} (c_{n+1} + \cdots + c_P) \sum_{\substack{n_1=0 \\ n_1+\cdots+n_P=n}}^{n} \cdots \sum_{n_P=0}^{n} \prod_{p=1}^{P} (1 - \rho_{*p}) \rho_{*p}^{n_p}. \tag{4.58}$$

If $\rho_{*p} \geq 1$ for some p, the capacity loss Δ^{T} is not minimal.

PROOF: By (3.83), we have for $p = 1, \ldots, P$, $n = 0, 1, 2, \ldots$, if $\rho_{*p} < 1$,

$$\mathbf{P}\left[N_{*p} = n\right] = (1 - \rho_{*p}) \rho_{*p}^{n}, \tag{4.59}$$

and obviously

$$\mathbf{P}\left[N_{*p} > 0\right] = \rho_{*p}. \tag{4.60}$$

From the definition of the total obtained share $o^{\mathrm{T}}(t)$ in (2.7) it follows that a busy or an idle processor (say p) accounts for an amount of c_p or 0 of $o^{\mathrm{T}}(t)$, respectively. Combined with (4.60), we have, as long as $\rho_{*p} < 1$, $p = 1, \ldots, P$,

$$\mathbf{E}\left[o^{\mathrm{T}}\right] = \sum_{p=1}^{P} c_p \mathbf{P}\left[N_{*p} > 0\right] = \rho, \tag{4.61}$$

so the total obtained share is then independent of the π_{gp}'s. For the expected total feasible share $\mathbf{E}\left[f^{\mathrm{T}}\right]$ we have from (2.41),

$$\begin{aligned}
\mathbf{E}\left[f^{\mathrm{T}}\right] &= \sum_{p=1}^{P-1} (c_1 + \cdots + c_p)\mathbf{P}\left[N = p\right] + \mathbf{P}\left[N \geq P\right] \\
&= \sum_{p=1}^{P} c_p \mathbf{P}\left[N \geq p\right] \qquad (4.62) \\
&= 1 - \sum_{p=1}^{P} c_p \sum_{n=0}^{p-1} \mathbf{P}\left[N = n\right] \\
&= 1 - \sum_{n=0}^{P-1} (c_{n+1} + \cdots + c_P)\mathbf{P}\left[N = n\right]. \qquad (4.63)
\end{aligned}$$

The capacity loss $\Delta^{\mathrm{T}} = \mathbf{E}\left[f^{\mathrm{T}}\right] - \mathbf{E}\left[o^{\mathrm{T}}\right]$ (by (2.42)) follows from substitution of the expression (4.37) for $\mathbf{P}\left[N = n\right]$ into (4.63), and combining it with (4.61). This proves (4.58).

If $\rho_{*p} \geq 1$ for some p, the queue of processor p grows without a bound, and we have $\mathbf{E}\left[f^{\mathrm{T}}\right] = 1$ by (2.41), $\mathbf{E}\left[o^{\mathrm{T}}\right] \leq \rho$ by (2.7), and hence $\Delta^{\mathrm{T}} = \mathbf{E}\left[f^{\mathrm{T}}\right] - \mathbf{E}\left[o^{\mathrm{T}}\right] \geq 1 - \rho$ by (2.42). In fact,

$$\mathbf{E}\left[o^{\mathrm{T}}\right] = \rho - \sum_{p=1}^{P} c_p \max(0, \rho_{*p} - 1). \tag{4.64}$$

However, if we distribute the load evenly among the processors such that $\rho_{*p} = \rho < 1$, $p = 1, \ldots, P$, we have $\mathbf{E}\left[f^{\mathrm{T}}\right] < 1$ and $\mathbf{E}\left[o^{\mathrm{T}}\right] = \rho$, and so the resulting capacity loss is smaller than that of the unstable system. □

From Proposition 4.2, it follows that only the loads on the processors determine Δ^{T}; it does not matter to which groups the load belongs. Therefore, in order to minimize the capacity loss, we only need to know the total load ρ. We now describe our optimization problem.

Problem 4.3 *In a random-splitting system, given $c_1, \ldots, c_P$, ρ, with $c_1 \geq \cdots \geq c_P > 0$, $\sum_{p=1}^{P} c_p = 1$, $0 < \rho < 1$, minimize Δ^{T} as given in (4.58), for ρ_{*p}, $p = 1, \ldots, P$, subject to*

$$0 \leq \rho_{*p} < 1, \quad p = 1, \ldots, P, \tag{4.65}$$

$$\sum_{p=1}^{P} c_p \rho_{*p} = \rho. \tag{4.66}$$

We denote the minimum capacity loss of Problem 4.3 by $\Delta^{\mathrm{T,MIN}}$, i.e.,

$$\Delta^{\mathrm{T,MIN}} \triangleq \min\{\Delta^{\mathrm{T}}\}. \tag{4.67}$$

An important question is whether the solution in terms of the ρ_{*p}'s is unique. We will give the answer in the next sections. Since we assume that the ρ_{g*}'s are given, we still need to show how to construct the matrix (π_{gp}) from the ρ_{*p}'s found. From (4.32), (4.33), and (4.34), we must solve for π_{gp}, $g = 1, \ldots, G$, $p = 1, \ldots, P$,

$$\sum_{g=1}^{G} \pi_{gp}\rho_{g*} = c_p\rho_{*p}, \ p = 1, \ldots P, \tag{4.68}$$

subject to the requirements that π_{gp} is a probability and that (4.31) must hold:

$$\begin{aligned} 0 \leq \pi_{gp} \leq 1, &\qquad g = 1, \ldots, G, \ p = 1, \ldots, P, \\ \sum_{p=1}^{P} \pi_{gp} = 1, &\qquad g = 1, \ldots, G. \end{aligned}$$

Clearly, in general there is no unique solution. A possible solution is

$$\pi_{gp} = \frac{c_p\rho_{*p}}{\rho}.$$

We study Problem 4.3 for homogeneous systems in Section 4.4.2, and for heterogeneous systems in Section 4.4.3.

4.4.2 Homogeneous Systems

In order to solve Problem 4.3 for homogeneous systems, we first consider a two-processor system. Subsequently, we extend our results to systems with more than two processors. Our main result is Theorem 4.5, which states that in homogeneous systems, balancing the load among the processors (i.e., setting $\rho_{*1} = \cdots = \rho_{*P} = \rho$) minimizes the capacity loss. We show by Theorem 4.6 that as the number of processors increases, the minimal capacity loss approaches a peak of 0.5 at $\rho = 0.5$. Finally, Theorem 4.7 states that balancing the load also minimizes the probability $\mathbf{P}_{\mathrm{lbs}}(n, m)$ of load-balancing success, $n = 0, 1, 2, \ldots$, $m = 1, 2, 3, \ldots$, $n < m$.

In a homogeneous two-processor random-splitting system ($P = 2$, $c_1 = c_2 = 0.5$), we have, by (2.42), (4.36), and (4.66),

$$\begin{aligned} \Delta^{\mathrm{T}} &= \frac{1}{2}\mathbf{P}\left[N_{*1} = 0\right]\mathbf{P}\left[N_{*2} > 1\right] + \frac{1}{2}\mathbf{P}\left[N_{*1} > 1\right]\mathbf{P}\left[N_{*2} = 0\right] \\ &= \frac{1}{2}(1 - \rho_{*1})\rho_{*2}^2 + \frac{1}{2}\rho_{*1}^2(1 - \rho_{*2}) \\ &= (1 + \rho)(\rho_{*1} - \rho)^2 + \rho^2(1 - \rho). \end{aligned} \tag{4.69}$$

It follows that Δ^{T} obtains a minimum when $\rho_{*1} = \rho$. In other words, for fixed $\rho < 1$, Δ^{T} is minimized when the loads on both processors are equal. The value of this minimum is

$$\Delta^{\mathrm{T,MIN}} = \rho^2(1 - \rho). \tag{4.70}$$

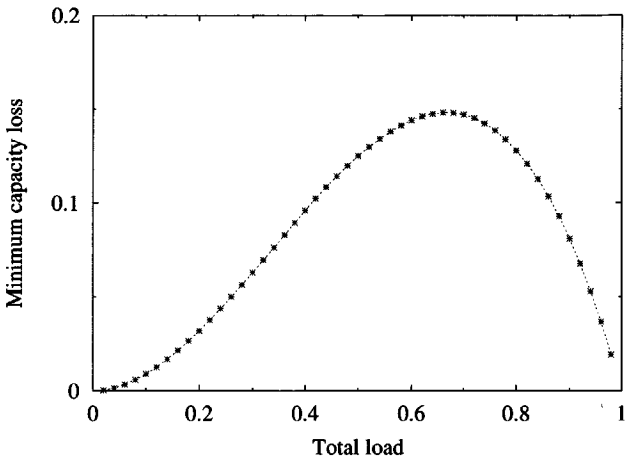


Figure 4.6: The minimum capacity loss $\Delta^{\mathrm{T,MIN}}$ as a function of the total load ρ in a homogeneous two-processor system.

Because we also have for $P = 2$

$$\Delta^{\mathrm{T}} = \frac{1}{2}\mathbf{P}_{\mathbf{wwi}},$$

balancing the load minimizes $\mathbf{P}_{\mathbf{wwi}}$ as well.

In Figure 4.6, we show $\Delta^{\mathrm{T,MIN}}$ as a function of ρ. The minimum capacity loss $\Delta^{\mathrm{T,MIN}}$ has a maximum of $4/27 \approx 14.8\%$ at $\rho = 2/3$. Clearly, in a homogeneous two-processor system, capacity loss is not much of a problem when the load is either close to 0 (and processors are rarely busy) or close to 1 (and processors are rarely idle).

For $P > 2$, the expression for Δ^{T} quickly becomes very complex. However, balancing the load among the processors still minimizes Δ^{T}. In order to prove this, we need the following lemma, stating that in a homogeneous two-processor system, the probability that the number of jobs exceeds some threshold is minimized when the processor loads are equal.

Lemma 4.4 *In a homogeneous two-processor random-splitting system with a single Poisson arrival process with $\rho < 1$, for every $n \geq 1$, the probability $\mathbf{P}[N \geq n]$ has a unique, global minimum when the loads on the two processors are equal.*

PROOF: We write $\rho_i \triangleq \rho_{*i}$, $i = 1, 2$, and

$$f_n(\rho_1) \triangleq \mathbf{P}[N \geq n]. \tag{4.71}$$

The probability that an M/M/1 queue with load ρ_i has exactly or at least n jobs is equal to $(1 - \rho_i)\rho_i^n$ and ρ_i^n, respectively. So we have

$$f_n(\rho_1) = \rho_1^n + \sum_{i=1}^{n}(1 - \rho_1)\rho_1^{n-i}\rho_2^i. \tag{4.72}$$

Here, the first term is for the states with at least n jobs in queue 1, and the summation is over all states with exactly $n-i$ jobs in queue 1 and at least i jobs in queue 2, for $i = 1, \ldots, n$. Using $\rho_2 = 2\rho - \rho_1$, we have

$$\begin{aligned} f_n(\rho_1) &= \rho_1^n + (1-\rho_1)\sum_{i=1}^{n} \rho_1^{n-i}(2\rho-\rho_1)^i \\ &= \rho_1^{n+1} + (1-\rho_1)\sum_{i=0}^{n} \rho_1^{n-i}(2\rho-\rho_1)^i, \end{aligned} \tag{4.73}$$

which can be rewritten as

$$\begin{aligned} f_n(\rho_1) &= (\rho + (\rho_1-\rho))^{n+1} \\ &\quad +(1-\rho-(\rho_1-\rho))\sum_{i=0}^{n}(\rho+(\rho_1-\rho))^{n-i}(\rho-(\rho_1-\rho))^i \\ &= \sum_{i=0}^{n+1} a_{n,i}(\rho_1-\rho)^i. \end{aligned} \tag{4.74}$$

We will prove that for i odd $a_{n,i} = 0$, and for i even $a_{n,i} > 0$, which implies that $f_n(\rho_1)$ has a unique minimum in $\rho_1 = \rho$.

Defining $h_n(x)$ by

$$h_n(x) = \sum_{i=0}^{n}(\rho+x)^{n-i}(\rho-x)^i,$$

we can write

$$f_n(\rho_1) = (\rho+(\rho_1-\rho))^{n+1} + (1-\rho)h_n(\rho_1-\rho) - (\rho_1-\rho)h_n(\rho_1-\rho). \tag{4.75}$$

Let

$$h_n(x) = \sum_{i=0}^{n} b_{n,i}x^i.$$

We will prove that

$$b_{n,i} = \begin{cases} 0, & i \text{ odd}, \\ \left(\begin{array}{c} n+1 \\ i+1 \end{array}\right)\rho^{n-i}, & i \text{ even}. \end{cases} \tag{4.76}$$

Of course,

$$b_{n,0} = (n+1)\rho^n,$$

which satisfies (4.76). For $i > 0$ we will prove (4.76) by induction to n. Because $h_1(x) = 2\rho$, (4.76) holds for $n = 1$. The following relation between the functions $h_n(x)$ holds:

$$h_{n+1}(x) = (\rho - x)h_n(x) + (\rho+x)^{n+1}. \tag{4.77}$$

By (4.77), we have

$$b_{n+1,i} = \rho b_{n,i} - b_{n,i-1} + \left(\begin{array}{c} n+1 \\ i \end{array}\right)\rho^{n+1-i},\ i = 1, \ldots, n.$$

Using the induction hypothesis (4.76) for $b_{n,i}$ and $b_{n,i-1}$, we find for i odd,

$$b_{n+1,i} = -\binom{n+1}{i}\rho^{n+1-i} + \binom{n+1}{i}\rho^{n+1-i} = 0,$$

and for i even,

$$b_{n+1,i} = \rho\binom{n+1}{i+1}\rho^{n-i} + \binom{n+1}{i}\rho^{n+1-i} = \binom{n+2}{i+1}\rho^{n+1-i},$$

which proves (4.76).

Now by (4.75) and (4.76), we find, for i odd,

$$\begin{aligned} a_{n,i} &= (1-\rho)b_{n,i} - b_{n,i-1} + \binom{n+1}{i}\rho^{n+1-i} \\ &= 0 - \binom{n+1}{i}\rho^{n+1-i} + \binom{n+1}{i}\rho^{n+1-i} \\ &= 0, \end{aligned} \tag{4.78}$$

for i even, $i < n+1$,

$$\begin{aligned} a_{n,i} &= (1-\rho)b_{n,i} - b_{n,i-1} + \binom{n+1}{i}\rho^{n+1-i} \\ &= (1-\rho)\binom{n+1}{i+1}\rho^{n-i} - 0 + \binom{n+1}{i}\rho^{n+1-i} \\ &> 0, \end{aligned} \tag{4.79}$$

and, for n odd,

$$a_{n,n+1} = 0 - 0 + \binom{n+1}{n+1}\rho^0 = 1 > 0. \tag{4.80}$$

This concludes the proof of Lemma 4.4. □

We now show that in homogeneous systems, balancing the load among the processors minimizes Δ^{T}.

Theorem 4.5 *In a homogeneous random-splitting system, balancing the load among the processors, i.e., setting the routing matrix (π_{gp}) such that $\rho_{*1} = \cdots = \rho_{*P} = \rho$, minimizes the expected capacity loss Δ^{T}. Furthermore, we have*

$$\Delta^{\mathrm{T,MIN}} = 1 - \rho - \frac{1}{P}\sum_{n=0}^{P-1}(P-n)\binom{P+n-1}{n}(1-\rho)^P\rho^n. \tag{4.81}$$

PROOF: From (4.69) and (4.70) it is clear that the theorem holds for $P = 2$. Consider a system with $P > 2$. We will show that Δ^{T} obtains a global minimum when all processor loads ρ_{*p}, $p = 1, \ldots, P$, are equal. It is enough to show that $\rho_{*1} = \rho_{*2}$.

Clearly, minimizing Δ^{T} is equivalent to minimizing $\mathbf{E}\left[f^{\mathrm{T}}\right]$, for which we rewrite (4.62) for homogeneous systems:

$$\mathbf{E}\left[f^{\mathrm{T}}\right] = \frac{1}{P}\sum_{n=1}^{P}\mathbf{P}\left[N \geq n\right]. \tag{4.82}$$

If we separate the jobs on processors 1 and 2 from the jobs on the other processors, we obtain

$$\begin{aligned} P\mathbf{E}\left[f^{\mathrm{T}}\right] &= \sum_{n=1}^{P}\mathbf{P}\left[N_{*,3\cdots P} \geq n\right] + \sum_{n=1}^{P}\sum_{m=0}^{n-1}\mathbf{P}\left[N_{*,3\cdots P} = m\right]\mathbf{P}\left[N_{*,1\cdots 2} \geq n-m\right] \\ &= \sum_{n=1}^{P}\mathbf{P}\left[N_{*,3\cdots P} \geq n\right] + \sum_{m=0}^{P-1}\mathbf{P}\left[N_{*,3\cdots P} = m\right]\sum_{n=1}^{P-m}\mathbf{P}\left[N_{*,1\cdots 2} \geq n\right]. \end{aligned} \tag{4.83}$$

Because of Lemma 4.4, for every $n \geq 1$, $\mathbf{P}\left[N_{*,1\cdots 2} \geq n\right]$ has a unique, global minimum for $\rho_{*1} = \rho_{*2}$. Therefore, setting $\rho_{*1} = \rho_{*2}$ minimizes $\mathbf{E}\left[f^{\mathrm{T}}\right]$ and Δ^{T}.

Rewriting (4.58) for a homogeneous system, we have in general

$$\Delta^{\mathrm{T}} = 1 - \rho - \frac{1}{P}\sum_{n=0}^{P-1}(P-n)\mathbf{P}\left[N = n\right]. \tag{4.84}$$

Obviously, with $\rho_{*1} = \cdots = \rho_{*P} = \rho$, the probability of an arbitrary state in which $N = n$ is $(1-\rho)^P\rho^n$, and since there are

$$\binom{P+n-1}{n}$$

distinct ways of putting n indistinguishable jobs onto P distinguishable processors, we obtain

$$\mathbf{P}\left[N = n\right] = \binom{P+n-1}{n}(1-\rho)^P\rho^n. \tag{4.85}$$

So, $\mathbf{P}\left[N = n\right]$ has a negative binomial distribution. Equation 4.81 follows from substitution of (4.85) into (4.84). □

Equation 4.81 is hard to analyze. In Figure 4.7 we show the minimum capacity loss $\Delta^{\mathrm{T,MIN}}$ as a function of the total load ρ for various values of the number P of processors. Clearly, capacity loss is an increasing problem with probabilistic scheduling when the number of processors increases. The following theorem describes the behavior of $\Delta^{\mathrm{T,MIN}}$ for large P.

Theorem 4.6 *In a homogeneous random-splitting system, it holds that*

$$\lim_{P\to\infty}\Delta^{\mathrm{T,MIN}} = \begin{cases} \frac{\rho^2}{1-\rho} & 0 < \rho \leq \frac{1}{2}, \\ 1-\rho & \frac{1}{2} < \rho < 1. \end{cases} \tag{4.86}$$

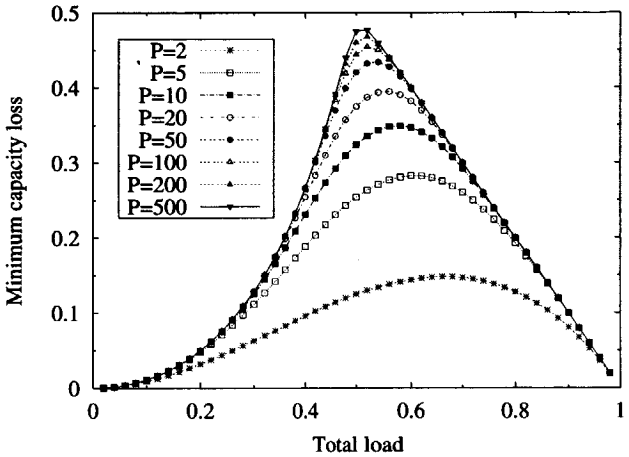


Figure 4.7: The minimum capacity loss $\Delta^{\mathrm{T,MIN}}$ as a function of the total load ρ for various values of the number P of processors.

PROOF: By (4.58) we must prove that

$$\lim_{P\to\infty} \mathbf{E}\left[f^{\mathrm{T}}\right] = \begin{cases} \frac{\rho}{1-\rho} & 0 < \rho \leq \frac{1}{2}, \\ 1 & \frac{1}{2} < \rho < 1. \end{cases}$$

The mean number of jobs in an M/M/1 queue is equal to $\rho/(1-\rho)$, which is less than or exceeds 1 for $\rho < 0.5$ and $\rho \geq 0.5$, respectively. For increasing values of P, the distribution of the numbers of jobs on the processors will increasingly well reflect the (discrete) distribution of the number of jobs in an M/M/1 queue, and so for $\rho \leq 0.5$, $\mathbf{E}\left[f^{\mathrm{T}}\right]$ approaches $\rho/(1-\rho)$, and for $\rho \geq 0.5$, $\mathbf{E}\left[f^{\mathrm{T}}\right]$ approaches 1. □

Obviously, in homogeneous systems with random splitting, capacity loss is a serious problem. With many processors, half the system's capacity is lost when $\rho = 1/2$. At the same time, the $\mathbf{P}_{\mathrm{wwi}}$ and $\mathbf{P}_{\mathrm{lbs}}(n,m)$ are close to one. We believe the notion of capacity loss is more meaningful than $\mathbf{P}_{\mathrm{wwi}}$, because Δ^{T} states exactly *how much* of the system capacity is lost, whereas $\mathbf{P}_{\mathrm{wwi}}$ only indicates that capacity is being lost.

The following theorem states that balancing the load among the processors also minimizes the wait-while-idle probability and the probability of load-balancing success.

Theorem 4.7 *In a homogeneous random-splitting system with total load ρ, $0 < \rho < 1$, the wait-while-idle probability $\mathbf{P}_{\mathrm{wwi}}$ and, for every $n = 0, 1, \ldots$, $m = n+1, n+2, \ldots$, the probability of load-balancing success $\mathbf{P}_{\mathrm{lbs}}(n,m)$ have a unique, global minimum when the loads on the processors are equal, i.e., when $\rho_{*1} = \cdots = \rho_{*P} = \rho$.*

PROOF: We use the shorthand notations N_p and ρ_p for N_{*p} and ρ_{*p}, respectively, $p = 1, \ldots, P$. Clearly, $\mathbf{P}_{\mathrm{wwi}} = \mathbf{P}_{\mathrm{lbs}}(0,2)$, so we only consider $\mathbf{P}_{\mathrm{lbs}}(n,m)$, $n = 0, 1, \ldots$, $m =$

$n+1, n+2, \ldots$. By (4.40), if $\rho_p > 1$ for one or more p, we have

$$\mathbf{P}_{\text{lbs}}(n,m) = 1 - \mathbf{P}\left[N_p > n,\ p = 1, \ldots, P\right],$$

which strictly decreases in the loads on the processors with stable queues (i.e., with a load smaller than one), of which at least one must exist. As a result, $\mathbf{P}_{\text{lbs}}(n,m)$ cannot be minimal, and we therefore restrict ourselves to $0 \leq \rho_p \leq 1$, $p = 1, \ldots, P$. Then, by (4.37), (4.40) can be rewritten as

$$\mathbf{P}_{\text{lbs}}(n,m) = 1 - \prod_{p=1}^{P}\left(1-\rho_p^m\right) - \prod_{p=1}^{P}\rho_p^{n+1} + \prod_{p=1}^{P}\left(\rho_p^{n+1} - \rho_p^m\right), \tag{4.87}$$

which still holds if one or more of the ρ_p's are one. We first prove the theorem for $P = 2$, then we generalize to $P \geq 2$.

For $P = 2$, we write $\rho_1 = \rho + \varepsilon$ and $\rho_2 = \rho - \varepsilon$, $\varepsilon \geq 0$. Due to the symmetry, we do not consider $\varepsilon < 0$. Furthermore, we must have $\varepsilon \leq \min(\rho, 1-\rho)$. We have from (4.87),

$$\begin{aligned}
\mathbf{P}_{\text{lbs}}(n,n+1) &= \rho_1^{n+1} + \rho_2^{n+1} - 2\left(\rho_1\rho_2\right)^{n+1} \\
&= (\rho+\varepsilon)^{n+1} + (\rho-\varepsilon)^{n+1} - 2\left(\rho^2-\varepsilon^2\right)^{n+1} \\
&= 2\sum_{\substack{i=0 \\ i \text{ even}}}^{n+1}\binom{n+1}{i}\rho^{n+1-i}\varepsilon^i - 2\left(\rho^2-\varepsilon^2\right)^{n+1},
\end{aligned} \tag{4.88}$$

which is strictly increasing in ε, $0 \leq \varepsilon < \min(\rho, 1-\rho)$. Also from (4.87),

$$\begin{aligned}
\mathbf{P}_{\text{lbs}}(n,m+1) &= (\rho-\varepsilon)\left(1-\rho_1^{n+1}\right)\rho_2^m + (\rho+\varepsilon)\rho_1^m\left(1-\rho_2^{n+1}\right) \\
&= \rho\mathbf{P}_{\text{lbs}}(n,m) + \varepsilon\left(\rho_1^m\left(1-\rho_2^{n+1}\right) - \left(1-\rho_1^{n+1}\right)\rho_2^m\right),
\end{aligned} \tag{4.89}$$

in which the second term is strictly increasing in ε for $0 \leq \varepsilon < \min(\rho, 1-\rho)$. So, $\mathbf{P}_{\text{lbs}}(n,m+1)$ is strictly increasing in ε if $\mathbf{P}_{\text{lbs}}(n,m)$ is strictly increasing in ε. By induction to m, and because the induction hypothesis holds for $\mathbf{P}_{\text{lbs}}(n,n+1)$, it holds that $\mathbf{P}_{\text{lbs}}(n,m)$, $n = 0, 1, \ldots$, $m = n+1, n+2, \ldots$, is strictly increasing in ε, $0 \leq \varepsilon < \min(\rho, 1-\rho)$, and therefore has a unique, global minimum for $\varepsilon = 0$. This completes the proof for $P = 2$.

For $P > 2$, we will prove that with any uneven distribution of the total load ρ, the $\mathbf{P}_{\text{lbs}}(n,m)$ cannot be minimal. Consider such an uneven distribution of ρ. We may assume that $0 < \rho_1 \leq 1$ and $0 \leq \rho_2 < \rho_1$ because two such processors always exist. We define $\tilde{\rho} \triangleq (\rho_1 + \rho_2)/2$, and we write $\rho_1 = \tilde{\rho} + \varepsilon$ and $\rho_2 = \tilde{\rho} - \varepsilon$. We have $0 < \tilde{\rho} < 1$, and $\varepsilon > 0$.

Rewriting (4.87) in terms of $\tilde{\rho}$ and ε, we obtain,

$$\mathbf{P}_{\text{lbs}}(n,m) = 1 - af_1(\varepsilon) - bf_2(\varepsilon) + cf_3(\varepsilon), \tag{4.90}$$

with

$$a = \prod_{p=3}^{P}\left(1-\rho_p^m\right),\ b = \prod_{p=3}^{P}\rho_p^{n+1},\ c = \prod_{p=3}^{P}\left(\rho_p^{n+1} - \rho_p^m\right), \tag{4.91}$$

and

$$\begin{aligned} f_1(\varepsilon) &= \mathbf{P}\left[N_1 < m,\ N_2 < m\right] \\ &= 1 - (\tilde{\rho}+\varepsilon)^m - (\tilde{\rho}-\varepsilon)^m + \left(\tilde{\rho}^2 - \varepsilon^2\right)^m, \end{aligned} \tag{4.92}$$

$$f_2(\varepsilon) = \mathbf{P}\left[N_1 > n,\ N_2 > n\right] = \left(\tilde{\rho}^2 - \varepsilon^2\right)^{n+1}, \tag{4.93}$$

$$f_3(\varepsilon) = \mathbf{P}\left[n < N_1 < m,\ n < N_2 < m\right]. \tag{4.94}$$

We have $0 \leq a \leq 1$, $0 \leq b \leq 1$, and $0 \leq c \leq \min(a,b) < 1$. First, we deal with the special case $a = b = 0$ (and so, $c = 0$), which arises only when $\rho_p = 0$ and $\rho_q = 1$ for one or more p and one or more q, $p, q = 3, \ldots, P$. It is obvious from (4.90) that irrespective of ε, we then have $\mathbf{P}_{\text{lbs}}(n,m) = 1$ (clearly, load-balancing success is guaranteed on processors p and q). However, this can never be optimal because for the balanced case, $\mathbf{P}_{\text{lbs}}(n,m) < 1$ by (4.41). Therefore, in the sequel we assume that $a > 0$ or $b > 0$ (or both).

It is enough to prove that $\mathbf{P}_{\text{lbs}}(n,m)$ is strictly increasing in ε for $0 \leq \varepsilon < \min(\tilde{\rho}, 1-\tilde{\rho})$. If $c = 0$, this is obvious from (4.90) because $f_1(\varepsilon)$ and $f_2(\varepsilon)$ are strictly decreasing in ε, and $a, b \geq 0$ with at least one of them being strictly positive. If $c > 0$, we have from (4.90),

$$\begin{aligned} \frac{\partial \mathbf{P}_{\text{lbs}}(n,m)}{\partial \varepsilon} &= -a\frac{\partial f_1(\varepsilon)}{\partial \varepsilon} - b\frac{\partial f_2(\varepsilon)}{\partial \varepsilon} + c\frac{\partial f_3(\varepsilon)}{\partial \varepsilon} \\ &\geq c\left(-\frac{\partial f_1(\varepsilon)}{\partial \varepsilon} - \frac{\partial f_2(\varepsilon)}{\partial \varepsilon} + \frac{\partial f_3(\varepsilon)}{\partial \varepsilon}\right), \end{aligned} \tag{4.95}$$

so it suffices to prove that

$$f(\varepsilon) \triangleq 1 - f_1(\varepsilon) - f_2(\varepsilon) + f_3(\varepsilon) \tag{4.96}$$

is strictly increasing in ε. But, $f(\varepsilon)$ is exactly the $\mathbf{P}_{\text{lbs}}(n,m)$ in a two-processor system with total load $\tilde{\rho}$, $0 < \tilde{\rho} < 1$, for which we already proved this. □

For homogeneous systems, we conclude that for any number of processors, balancing the total load $\rho < 1$ among the processors minimizes the capacity loss, the mean job response time, the mean job waiting time, the wait-while-idle probability and the probability of load-balancing success. In the next section, we proceed with the minimization of capacity loss in heterogeneous systems.

4.4.3 Heterogeneous Systems

We now turn our attention to capacity loss in heterogeneous systems. Again, we study the two-processor system first, and then consider systems with more than two processors. Recall that $c_1 \geq \cdots \geq c_P$, and $\sum_p c_p = 1$.

One might suspect that in order to minimize Δ^{T}, one should set $\rho_{*p} = \rho$, $p = 1, \ldots, P$, as in homogeneous systems, in other words, that one should use what is called Proportional Branching (PB). However, this is not the case. As it turns out, we must have $\rho_{*1} > \rho_{*2}$ when $c_1 > c_2$, and in some cases, we even have $\rho_{*2} = 0$.

For a two-processor system, we first show that the objective function Δ^{T}, is not convex in general in ρ_{*1} and ρ_{*2}. The importance of this result is that a local minimum of the

objective function Δ^{T} on the feasible region (which is convex) may not be the global minimum. Subsequently, we present the solution to Problem 4.3, and we compare this to the capacity loss obtained with Proportional Branching. Finally, we compare the minimization of the capacity loss to the minimization of the mean job waiting time and to that of the mean job response time.

Lemma 4.8 *In a two-processor random-splitting system with* $1/2 \leq c_1 < 1$ *and loads* $\rho_{*1}, \rho_{*2} < 1$ *on processors* 1 *and* 2, *respectively, we have*

$$\Delta^{\mathrm{T}} = (1-\rho_{*1})\rho_{*2}^2 c_1 + \rho_{*1}^2(1-\rho_{*2})c_2 + (1-\rho_{*1})(1-\rho_{*2})\rho_{*2}(c_1-c_2), \tag{4.97}$$

which is not convex in general in ρ_{*1} *and* ρ_{*2}.

PROOF: There are three cases of capacity loss:

1. Processor 1 is idle while processor 2 is serving two or more jobs. In this case, the capacity loss $\Delta^{\mathrm{T}} = c_1$.

2. Processor 1 is serving two or more jobs while processor 2 is idle. In this case, the capacity loss $\Delta^{\mathrm{T}} = c_2$.

3. Processor 1 is idle while processor 2 is serving a single job. In this case, the capacity loss $\Delta^{\mathrm{T}} = c_1 - c_2$. (This case is specific to heterogeneous systems.)

From this (4.97) follows.

The objective function (4.97) is not convex in general in ρ_{*1} and ρ_{*2} because the Hessian matrix

$$\begin{aligned} H_{\Delta^{\mathrm{T}}} &\triangleq \begin{bmatrix} \frac{\partial^2 \Delta^{\mathrm{T}}}{\partial \rho_{*1}^2} & \frac{\partial^2 \Delta^{\mathrm{T}}}{\partial \rho_{*1} \partial \rho_{*2}} \\ \frac{\partial^2 \Delta^{\mathrm{T}}}{\partial \rho_{*2} \partial \rho_{*1}} & \frac{\partial^2 \Delta^{\mathrm{T}}}{\partial \rho_{*2}^2} \end{bmatrix} \\ &= \begin{bmatrix} 2(1-\rho_{*2})c_2 & -2(\rho_{*1}+\rho_{*2})c_2 - (c_1-c_2) \\ -2(\rho_{*1}+\rho_{*2})c_2 - (c_1-c_2) & 2(1-\rho_{*1})c_2 \end{bmatrix} \end{aligned} \tag{4.98}$$

is not always positive semidefinite. For instance, putting $\rho_{*1} = \rho_{*2} = 1/2$ yields

$$H_{\Delta^{\mathrm{T}}} = \begin{bmatrix} c_2 & -1 \\ -1 & c_2 \end{bmatrix}, \tag{4.99}$$

which is not positive semidefinite. For that, all principal submatrices of $H_{\Delta^{\mathrm{T}}}$ must have non-negative determinants, i.e., $c_2 \geq 0$, which is true, and $\det H_{\Delta^{\mathrm{T}}} = c_2^2 - 1 \geq 0$, which is obviously *not* true. □

The following theorem solves Problem 4.3 for the two-processor case.

Theorem 4.9 *In a two-processor heterogeneous random-splitting system with* $1/2 < c_1 < 1$ *and fixed total load* ρ, $0 < \rho < 1$, Δ^{T} *is minimized when*

$$\rho_{*1} = \min\left(\hat{\rho}_1, \frac{\rho}{c_1}\right), \tag{4.100}$$

with

$$\hat{\rho}_1 \triangleq \frac{1}{A}(B - \sqrt{C}), \tag{4.101}$$

$$A \triangleq 3c_1(2c_1 - 1), \tag{4.102}$$

$$B \triangleq (3c_1 - 1)\rho + 4c_1^2 - 3c_1 + 1, \tag{4.103}$$

$$C \triangleq (3c_1^2 - 3c_1 + 1)\rho^2 + (-8c_1^2 + 9c_1 - 2)\rho + 4c_1^4 - 12c_1^3 + 14c_1^2 - 6c_1 + 1. \tag{4.104}$$

PROOF: For notational convenience, we write ρ_i and N_i instead of ρ_{*i} and N_{*i}, respectively, $i = 1, 2$. We rewrite (4.97), using $c_2 = 1 - c_1$ and $\rho_2 = (\rho - c_1\rho_1)/(1 - c_1)$, as

$$\Delta^{\mathrm{T}} = a_0 + a_1\rho_1 + a_2\rho_1^2 + a_3\rho_1^3, \tag{4.105}$$

where

$$a_0 = \frac{\rho^2 + (2c_1 - 1)\rho}{1 - c_1}, \tag{4.106}$$

$$a_1 = -\frac{\rho^2 + (4c_1 - 1)\rho + (2c_1 - 1)c_1}{1 - c_1}, \tag{4.107}$$

$$a_2 = \frac{(3c_1 - 1)\rho + 4c_1^2 - 3c_1 + 1}{1 - c_1}, \tag{4.108}$$

$$a_3 = -\frac{c_1(2c_1 - 1)}{1 - c_1}. \tag{4.109}$$

Since $1/2 < c_1 < 1$, we have $a_3 < 0$. By taking the derivative of Δ^{T} with respect to ρ_1 and solving for its roots, we obtain the two local optima of Δ^{T}, one of which is $\hat{\rho}_1$ defined in (4.101). The other root is

$$\bar{\rho}_1 \triangleq \frac{1}{A}(B + \sqrt{C}). \tag{4.110}$$

We have, omitting the details of the calculations,

$$0 < \rho < \hat{\rho}_1 < 1 < \bar{\rho}_1, \tag{4.111}$$

and because $a_3 < 0$, $\hat{\rho}_1$ is a local minimum, and $\bar{\rho}_1$ a local maximum. However, the minimum $\rho_1 = \hat{\rho}_1$ must be in the feasible region, in other words, it must meet (4.65) (it already meets (4.66), obviously). For processor 1, this is clear from (4.111). For processor 2, we must have $0 \leq \hat{\rho}_2 < 1$, with $\hat{\rho}_2 = (\rho - c_1\hat{\rho}_1)/(1 - c_1)$, in other words,

$$\frac{\rho}{c_1} - \frac{1 - c_1}{c_1} < \hat{\rho}_1 \leq \frac{\rho}{c_1}. \tag{4.112}$$

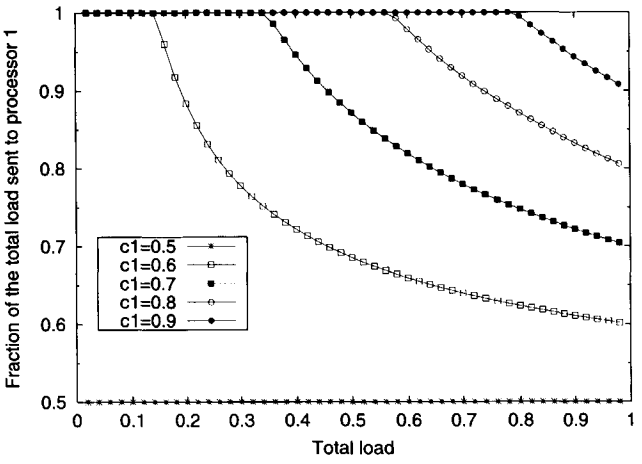


Figure 4.8: The optimal fraction of the total load sent to processor 1 as a function of the total load ρ for different values of c_1 ($P = 2$).

The lower bound is not a problem, because $\rho/c_1 - (1-c_1)/c_1 < \rho$ and by (4.111). But the upper bound is, because depending on c_1 and ρ, we can have $\hat{\rho}_1 > \rho/c_1$. If so, by (4.111) Δ^{T} is a strictly decreasing function of ρ_1 between $\rho_1 = 0$ and $\rho_1 = \rho/c_1$, and minimizing Δ^{T} implies setting ρ_1 as high as possible, i.e., $\rho_1 = \rho/c_1$. If $\hat{\rho}_1 \leq \rho/c_1$, Δ^{T} obtains a minimum at $\rho_1 = \hat{\rho}_1$. From this (4.100) follows. □

In Figures 4.8 and 4.9 we show the optimal fraction $c_1\rho_{*1}/\rho$ of the total load sent to processor 1 as a function of ρ for different values of c_1, and as a function of c_1 for different values of ρ, respectively. For small values of the total load ρ (the exact threshold depends on c_1), all jobs are sent to the fastest processor, i.e., processor 1. As the total load approaches one, the fractions of jobs sent to the two processors match the processors' capacities, as dictated by (4.65).

In Figures 4.10 and 4.11 we show $\Delta^{\mathrm{T,MIN}}$ as a function of ρ for different values of c_1, and as a function of c_1 for different values of ρ, respectively. From these plots, it is clear that heterogeneous two-processor systems do not in general exhibit a lower minimum capacity loss than homogeneous two-processor systems. For $c_1 < 0.75$, the minimum capacity loss $\Delta^{\mathrm{T,MIN}}$ as a function of ρ is almost identical to that for the homogeneous system. Only when c_1 is increased further beyond $c_1 = 0.75$, the capacity loss decreases significantly. From Figure 4.11, we conclude that for low to moderate values of ρ ($\rho < 0.75$), the minimum capacity loss $\Delta^{\mathrm{T,MIN}}$ even increases when c_1 increases from 0.5 to approximately 0.75, and then drops sharply to zero when c_1 increases further to one (i.e., all system capacity is concentrated in processor 1). For high values of ρ ($\rho > 0.75$), $\Delta^{\mathrm{T,MIN}}$ stays flat when c_1 increases from 0.5, and drops to zero beyond a certain point. However, this point moves to higher values of c_1 as ρ increases, e.g., $c_1 \approx 0.9$ for $\rho = 0.9$. As c_1 approaches one, $\Delta^{\mathrm{T,MIN}}$ goes to zero because the system more and more turns into a uniprocessor

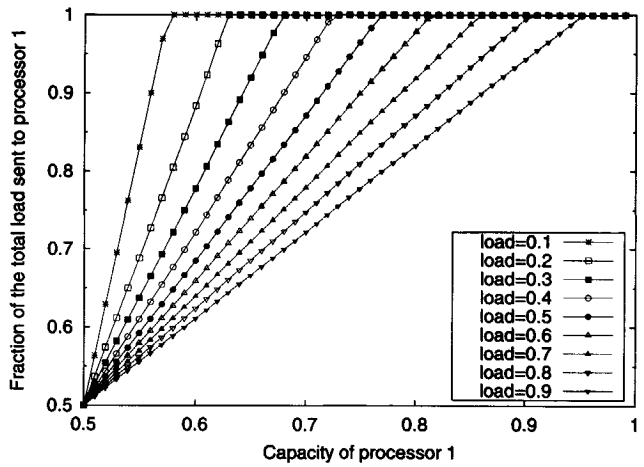


Figure 4.9: The optimal fraction of the total load sent to processor 1 as a function of the capacity c_1 of processor 1 for different values of ρ ($P = 2$).

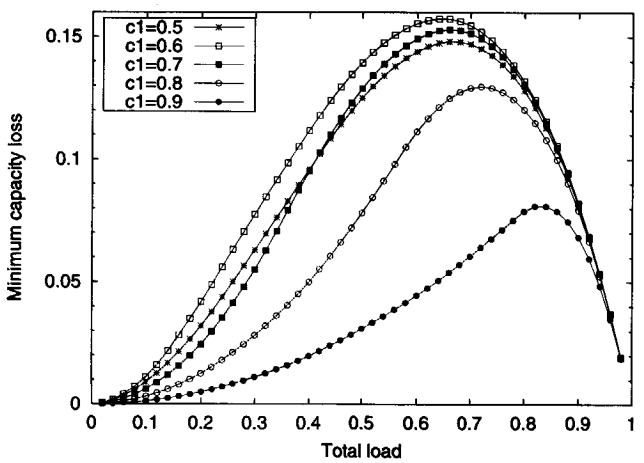


Figure 4.10: The minimum capacity loss $\Delta^{\mathrm{T,MIN}}$ as a function of the total load ρ for different values of c_1 ($P = 2$).

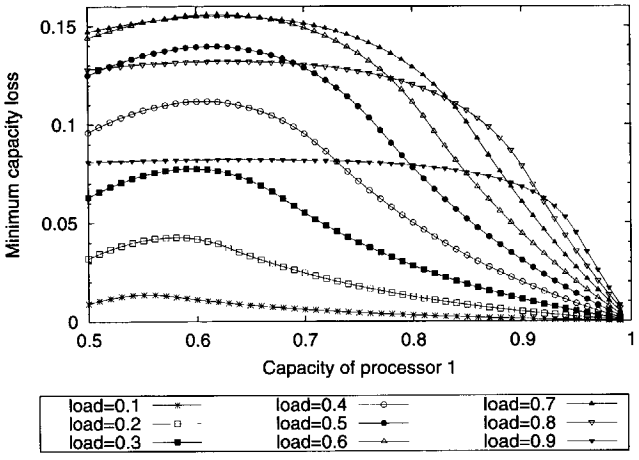


Figure 4.11: The minimum capacity loss $\Delta^{\mathrm{T,MIN}}$ as a function of the capacity c_1 of processor 1 for different values of the total load ρ ($P = 2$).

system. This follows directly from (4.100), (4.97), and (4.111):

$$\lim_{c_1 \to 1} \Delta^{\mathrm{T,MIN}} = 0. \tag{4.113}$$

With Proportional Branching, $\rho_{*1} = \rho_{*2} = \rho$, and the capacity loss $\Delta^{\mathrm{T,PB}}$ is, by (4.97),

$$\Delta^{\mathrm{T,PB}} = (1-\rho)\rho^2 c_1 + (1-\rho)\rho^2 c_2 + (1-\rho)^2\rho(c_1 - c_2) = (1-\rho)\rho^2 + (1-\rho)^2\rho(c_1 - c_2). \tag{4.114}$$

The term $(1-\rho)\rho^2$ is equal to the minimum capacity loss in a two-processor homogeneous system, by (4.70). We show in Figures 4.12 and 4.13 the excess capacity loss $\Delta^{\mathrm{T,PB}} - \Delta^{\mathrm{T,MIN}}$ as a function of the total load ρ and as a function of the capacity c_1 of processor 1, respectively. Clearly, PB is not a good choice for a low capacity loss in heterogeneous systems when $c_1 > 0.7$. With PB, one can expect an additional capacity loss of as much as 0.25 compared to the optimal routing policy. Moreover, we have by (4.114),

$$\lim_{c_1 \to 1} \Delta^{\mathrm{T,PB}} = (1-\rho)\rho^2 + (1-\rho)^2\rho = (1-\rho)\rho, \tag{4.115}$$

which is symmetric around $\rho = 1/2$.

In order to compare the minimization of the capacity loss to the minimization of the mean job waiting (response) time, we consider a two-processor heterogeneous system with $G = 1$, and we calculate the optimal fraction $\pi_{*1} = c_1\rho_{*1}/\rho$ of the load ρ sent to processor 1 for three different objectives:

- minimization of the mean job waiting time;
- minimization of the mean job response time;
- minimization of the capacity loss.

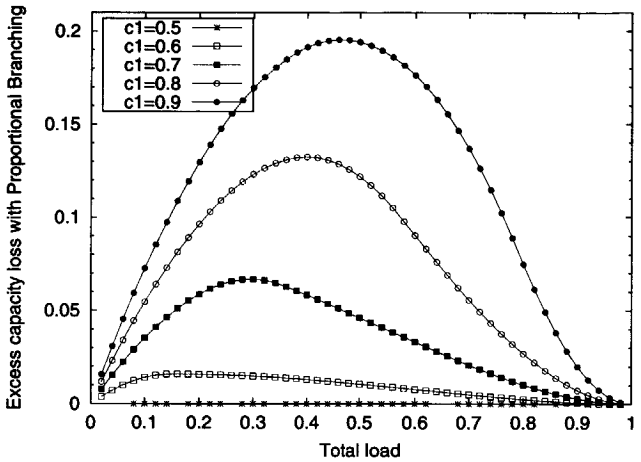


Figure 4.12: The excess capacity loss $\Delta^{T,PB} - \Delta^{T,MIN}$ for Proportional Branching as a function of the total load ρ for different values of c_1 ($P = 2$).

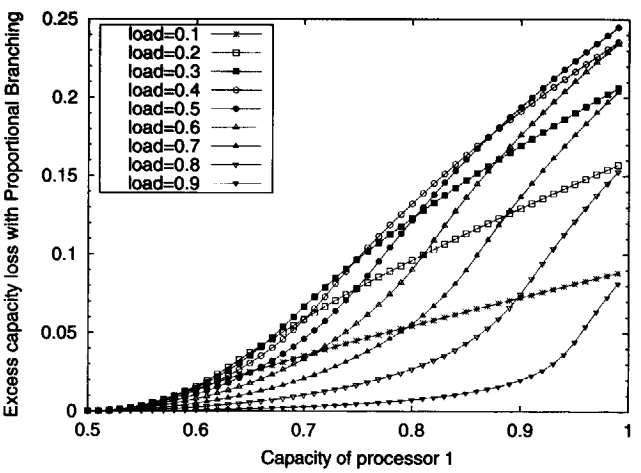


Figure 4.13: The excess capacity loss $\Delta^{T,PB} - \Delta^{T,MIN}$ for Proportional Branching as a function of the capacity c_1 of processor 1 for different values of the total load ρ ($P = 2$).

For the mean job *waiting-time* objective, we have from (4.51),

$$\rho_{*p} = 1 - \sqrt{\frac{1}{1+\delta_w c_p}},\ p = 1, 2, \tag{4.116}$$

with δ_w chosen such that $c_1\rho_{*1} + c_2\rho_{*2} = \rho$. For the mean *response-time* objective we have from (4.48),

$$\rho_{*p} = 1 - \sqrt{\frac{1}{\delta_r c_p}},\ p = 1, 2, \tag{4.117}$$

with δ_r chosen such that $c_1\rho_{*1} + c_2\rho_{*2} = \rho$. However, if this results in $\rho_{*2} < 0$, we must put $\rho_{*1} = \rho/c_1$ and $\rho_{*2} = 0$. For the capacity-loss objective, we have, by (4.100),

$$\rho_{*1} = \min\left(\hat{\rho}_1,\ \rho/c_1\right), \tag{4.118}$$

with $\hat{\rho}_1$ given by (4.101), and $\rho_{*2} = (\rho - c_1\rho_{*1})/c_2$. We show the optimal fraction of the total load sent to processor 1 in Figures 4.14, 4.15, and 4.16, for a low ($\rho = 0.1$), a moderate ($\rho = 0.5$), and a high total load ($\rho = 0.9$), respectively. When $\rho = 0.9$ the fraction of the total load sent to processor 1 for minimum waiting time is practically equal to that for minimum response time. Clearly, for all three objectives, if $c_1 > c_2$ we have $\rho_{*1} > \rho$ in the optimum. Especially with a low total load, for minimization of the mean job response time one needs to send a substantially larger fraction of the total load to the fastest processor than for minimization of the mean job waiting time. This is intuitively clear, because the response time of a job depends more on the capacity of a processor it runs on than the waiting time does, since the latter does not include the service time of the job. For minimization of the capacity loss, an even larger fraction of the total load must be sent to the fastest processor than for the minimization of the mean job response time.

In heterogeneous systems with more than two processors, the expression for Δ^{T} becomes progressively more difficult as the number of processors increases. Both the number of terms in the objective function Δ^{T} in (4.58) as well as the exponents of the ρ_{*p}'s increase in P. Therefore, deriving closed-form expression for the ρ_{*p}'s such that Δ^{T} is minimized seems impossible.

When $P > 2$, it is *not* true in general that in the optimum, the loads ρ_{*i} and ρ_{*j} of two arbitrary processors i and j follow that of a two-processor system with processor capacities $c_i/(c_i + c_j)$ and $c_j/(c_i + c_j)$ and total load $(c_i\rho_{*i} + c_j\rho_{*j})/(c_i + c_j)$. For instance, with $P = 3$, $c_1 = 0.4$, $c_2 = c_3 = 0.3$, and $\rho = 0.1$, the optimal loads are $\rho_{*1} = 0.188$ and $\rho_{*2} = \rho_{*3} = 0.041$ by numerical computation. However, if we consider only processors 1 and 2 with an equivalent total load of $(0.188 \times 0.4 + 0.041 \times 0.3)/(0.4 + 0.3) = 0.125$ and rescaled capacities $0.4/0.7 = 0.571$ and $0.3/0.7 = 0.429$, the optimal loads become $\rho'_{*1} = 0.197$ and $\rho'_{*2} = 0.030$.

A closed-form expression for the routing probabilities that minimize the capacity loss in heterogeneous systems with $P \geq 3$ remains an open issue of theoretical interest.

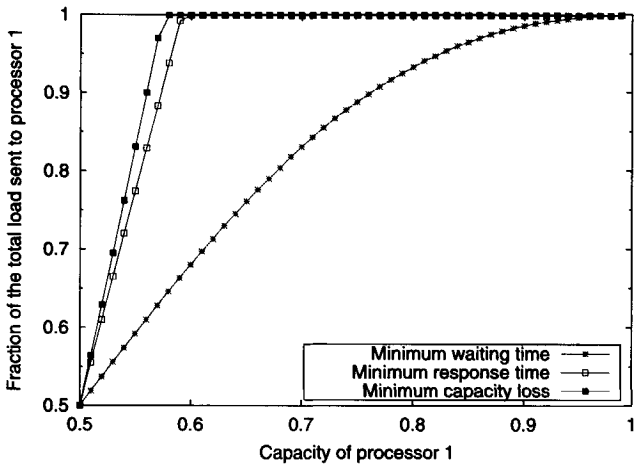


Figure 4.14: The optimal fraction of the total load sent to processor 1 as a function of the capacity c_1 of processor 1 in a two-processor heterogeneous system under low load ($\rho = 0.1$).

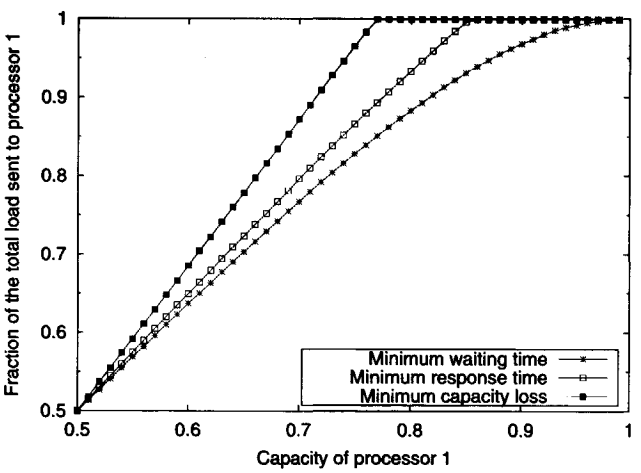


Figure 4.15: The optimal fraction of the total load sent to processor 1 as a function of the capacity c_1 of processor 1 in a two-processor heterogeneous system under moderate load ($\rho = 0.5$).

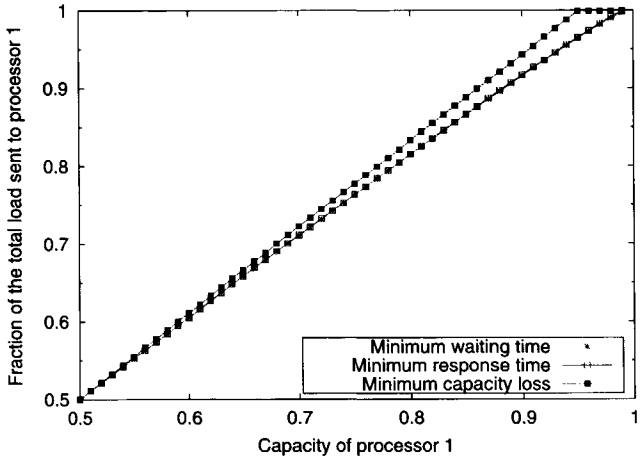


Figure 4.16: The optimal fraction of the total load sent to processor 1 as a function of the capacity c_1 of processor 1 in a two-processor heterogeneous system under high load ($\rho = 0.9$).

4.5 Probabilistic Share Scheduling: Compliance with the Feasible Group Shares

We study the problem of finding the routing matrix (π_{gp}) that minimizes Δ^{G} in a random-splitting system. In Section 4.5.1, we formulate the problem in mathematical terms. Because we cannot solve the problem in general, we introduce and compare two heuristic policies. In Horizontal Partitioning in Random Splitting (HPRS) introduced in Section 4.5.2, every processor get the same, homogeneous, traffic mix, identical to the traffic mix presented to the system as a whole. In Vertical Partitioning in Random Splitting (VPRS) introduced in Section 4.5.3, the traffic mixes at the processors are heterogeneous; the policy dedicates entire processors to groups. In Section 4.5.4, we compare HPRS and VPRS. We show that VPRS has the potential to outperform HPRS with respect to the minimization of the group-share deviation. The fundamental ideas behind HPRS and VPRS will return in Chapters 5 and 6, where we introduce global dynamic policies for share scheduling.

4.5.1 Problem Statement

In this section, our objective is to minimize the group-share deviation Δ^{G}, in other words,

$$\min_{(\pi_{gp})} \max_{g} \{\Delta_g^{\mathrm{G}}\}. \tag{4.119}$$

The group loads $\rho_{1*}, \ldots, \rho_{G*}$ (defined in (4.34)) are given, with $0 \leq \rho_{g*} < 1$, $g = 1, \ldots, G$, and $\rho = \sum_g \rho_{g*} < 1$.

In this section, we must explicitly allow $\rho_{*p} \geq 1$, because in the optimum, we do not necessarily have $\rho_{*p} < 1$ for all p. On a processor p with $\rho_{*p} < 1$ we have $\mathbf{E}\left[o_{gp}^{\mathrm{G}}\right] = c_p \rho_{gp}$ by (2.5) and (3.88); on a processor p with $\rho_{*p} \geq 1$ we have $\mathbf{E}\left[o_{gp}^{\mathrm{G}}\right] = c_p \rho_{gp}/\rho_{*p}$ because the processor spends ρ_{gp}/ρ_{*p} of time serving jobs of group g. (Recall that by (2.5), $o_{gp}^{\mathrm{G}}(t)$ is defined relative to the *total* system capacity, not to the capacity of processor p.) Hence, by (2.6), the expected obtained group share $\mathbf{E}\left[o_g^{\mathrm{G}}\right]$ of group g is given by

$$\begin{aligned} \mathbf{E}\left[o_g^{\mathrm{G}}\right] &= \sum_{p,\rho_{*p}<1} c_p \rho_{gp} + \sum_{p,\rho_{*p}\geq 1} \frac{c_p \rho_{gp}}{\rho_{*p}} \\ &= \sum_{p=1}^{P} \frac{c_p \rho_{gp}}{\max(1, \rho_{*p})}. \end{aligned} \tag{4.120}$$

If $\pi_{gp} > 0$ on any processor p with $\rho_{*p} \geq 1$, the number of jobs of group g grows without bounds, and we have $\mathbf{E}\left[f_g^{\mathrm{G}}\right] = r_g^{\mathrm{G}}$ by (2.26). Otherwise, the stationary probability distribution $\mathbf{P}\left[N_{g*} = n\right]$ exists, and

$$\mathbf{E}\left[f_g^{\mathrm{G}}\right] = \sum_{n=0}^{\infty} \min(r_g^{\mathrm{G}}, n/P) \mathbf{P}\left[N_{g*} = n\right]. \tag{4.121}$$

Using (3.80), we have if $\rho_{*p} < 1$, for $n_g = 0, 1, 2, \ldots$, $g = 1, \ldots, G$,

$$\mathbf{P}\left[N_{1p} = n_1, \ldots, N_{Gp} = n_G\right] = (1 - \rho_{*p}) \cdot \frac{(n_1 + \cdots + n_G)!}{n_1! \cdots n_G!} \cdot \rho_{1p}^{n_1} \cdots \rho_{Gp}^{n_G}, \tag{4.122}$$

and therefore,

$$\mathbf{P}\left[N_{gp} = n\right] = \frac{1 - \rho_{*p}}{1 - (\rho_{*p} - \rho_{gp})} \left(\frac{\rho_{gp}}{1 - (\rho_{*p} - \rho_{gp})} \right)^n. \tag{4.123}$$

So if $\rho_{*p} < 1$ on every processor p with $\rho_{gp} > 0$, we have

$$\mathbf{P}\left[N_{g*} = n\right] = \sum_{\substack{n_1=0 \\ n_1+\cdots+n_P=n}}^{n} \cdots \sum_{n_P=0}^{n} \prod_{p,\rho_{gp}>0} \frac{1 - \rho_{*p}}{1 - (\rho_{*p} - \rho_{gp})} \left(\frac{\rho_{gp}}{1 - (\rho_{*p} - \rho_{gp})} \right)^{n_p}. \tag{4.124}$$

Finally, by (2.39) we have

$$\Delta_g^{\mathrm{G}} \triangleq \frac{\mathbf{E}\left[f_g^{\mathrm{G}}\right] - \mathbf{E}\left[o_g^{\mathrm{G}}\right]}{r_g^{\mathrm{G}}}, \tag{4.125}$$

so our optimization problem is the following.

Problem 4.10 *In a random-splitting system, given* $c_1, \ldots, c_P$, $\rho_{1*}, \ldots, \rho_{G*}$, *with* $c_1 \geq \cdots \geq c_P > 0$, $\sum_{p=1}^{P} c_p = 1$, $\rho_{g*} \geq 0$, $g = 1, \ldots, G$, *and* $\sum_{g=1}^{G} \rho_{g*} < 1$, *minimize for the routing matrix* (π_{gp})

$$\Delta^{\mathrm{G}} = \max_g \frac{1}{r_g^{\mathrm{G}}} \left\{ \mathbf{E}\left[f_g^{\mathrm{G}}\right] - \sum_{p=1}^{P} \frac{c_p \rho_{gp}}{\max(1, \rho_{*p})} \right\}, \tag{4.126}$$

where, for $g = 1, \ldots, G$,

$$f_g^{\mathrm{G}} = \begin{cases} \sum_{n=0}^{\infty} \min(r_g^{\mathrm{G}}, n/P)\mathbf{P}\left[N_{g*} = n\right], & \text{if } \rho_{*p} < 1 \text{ on all } p \text{ with } \rho_{gp} > 0, \\ r_g^{\mathrm{G}}, & \text{otherwise,} \end{cases} \tag{4.127}$$

and $\mathbf{P}\left[N_{g*} = n\right]$ *as given in (4.124), subject to*

$$\rho_{gp} \geq 0, \quad g = 1, \ldots, G, \quad p = 1, \ldots, P, \tag{4.128}$$

$$\sum_{p=1}^{P} \rho_{gp} c_p = \rho_{g*}, \quad g = 1, \ldots, G. \tag{4.129}$$

We denote the minimum group-share deviation of Problem 4.10 by $\Delta^{\mathrm{G,MIN}}$, i.e.,

$$\Delta^{\mathrm{G,MIN}} \triangleq \min\{\Delta^{\mathrm{G}}\}. \tag{4.130}$$

Just as Problem 4.3, we have formulated Problem 4.10 in terms of the group loads instead of the π_{gp}'s. Ignoring for the moment the problem of the uniqueness of a solution in terms of the ρ_{gp}'s, we can easily calculate the (unique) π_{gp}'s from the ρ_{gp}'s found by using (4.32) and (4.34):

$$\pi_{gp} = \frac{c_p \rho_{gp}}{\rho_{g*}}, \quad g = 1, \ldots, G, \quad p = 1, \ldots, P. \tag{4.131}$$

By (4.128) and (4.129) it is then clear that $\pi_{gp} \geq 0$ for $g = 1, \ldots, G$, $p = 1, \ldots, P$, and $\sum_p \pi_{gp} = 1$, for $g = 1, \ldots, G$.

Unfortunately, we are not able to provide a general solution to Problem 4.10. We proceed by introducing two heuristic policies: HPRS and VPRS. In Section 4.5.4 we compare the two.

4.5.2 Horizontal Partitioning in Random Splitting

The idea behind HPRS is to have the same load on each processor and to present each of them the same traffic mix as that presented to the system as a whole. This will spread the jobs of the different groups among the processors as much as possible. In HPRS, we put

$$\pi_{gp} = c_p, \tag{4.132}$$

and therefore,

$$\rho_{gp} = \rho_{g*}, \quad g = 1, \ldots, G, \quad p = 1, \ldots, P, \tag{4.133}$$

$$\rho_{*p} = \rho, \quad p = 1, \ldots, P. \tag{4.134}$$

In homogeneous systems, we clearly have $\rho_{*1} = \cdots = \rho_{*P} = \rho < 1$ with HPRS, and therefore, by Theorem 4.5 the capacity loss Δ^{T} is minimal. In heterogeneous systems, HPRS amounts to proportional branching, for which we already know that it does not minimize the capacity loss. The following theorem gives expressions for the capacity loss Δ^{T} and the expected group-share deviation Δ_g^{G} of group g with HPRS.

Theorem 4.11 *In a P-processor random-splitting system with HPRS,* $c_1 \geq \cdots \geq c_P > 0$, $\sum_p c_p = 1$, $0 \leq \rho < 1$, $\rho_{g*} \geq 0$, $g = 1, \ldots, G$, *and* $\sum_g \rho_{g*} = \rho$, *we have*

$$\Delta^{\mathrm{T}} = 1 - \rho - \sum_{n=0}^{P-1} (c_{n+1} + \cdots + c_P) \binom{P+n-1}{n} (1-\rho)^P \rho^n, \tag{4.135}$$

and, for $g = 1, \ldots, G$,

$$\Delta_g^{\mathrm{G}} = 1 - \sum_{n_g=0}^{\lfloor r_g^{\mathrm{G}} P \rfloor} \left(1 - \frac{n_g}{r_g^{\mathrm{G}} P}\right) \binom{P+n_g-1}{n_g} (1-\tilde{\rho}_g)^P (\tilde{\rho}_g)^{n_g} - \frac{\rho_{g*}}{r_g^{\mathrm{G}}}, \tag{4.136}$$

with

$$\tilde{\rho}_g \triangleq \frac{\rho_{g*}}{1 - (\rho - \rho_{g*})}. \tag{4.137}$$

PROOF: Equation (4.135) follows directly from substitution of (4.134) into (4.58). Noting that $\rho_{*p} < 1$ for all p, (4.136) and (4.137) follow from substitution of (4.133) and (4.134) into (4.120), (4.124), (4.125), and (4.127). □

With random-splitting, HPRS is a natural extension of PS to systems with more than one processor: each processor receives the same traffic mix. Equation (4.136) still holds for $P = 1$. Furthermore, the behavior of HPRS with respect to Δ_g^{G} is independent of the heterogeneity of the system; only the number P of processors is relevant. As in a uniprocessor PS system, Δ_g^{G} only depends on $\rho - \rho_{g*}$, the exact composition of the load due to other groups than group g is irrelevant.

Because HPRS treats jobs of different groups equally, we do not expect an improvement of the compliance with the feasible group shares compared to a uniprocessor PS system. However, an interesting combination is HPRS with GPPS (introduced in Section 3.4.3) as the local scheduling policy. From Chapter 3 we already know that GPPS is the ideal local share-scheduling policy. Unfortunately, the stationary probability distribution $\mathbf{P}[N_g = n]$ for a uniprocessor GPPS system is not available, so a direct calculation of the Δ_g^{G}'s for HPRS/GPPS is impossible.

In Section 4.5.4, we study the performance of HPRS in more detail, including the performance of HPRS/GPPS, and we compare it to another policy introduced next, VPRS. The fundamental idea behind HPRS, namely, to spread the jobs of every group among all the processors and to rely on the local scheduling policy for the delivery of the feasible group shares is the basis for Horizontal Partitioning, a dynamic global share-scheduling policy introduced and evaluated in Chapter 5.

4.5.3 Vertical Partitioning in Random Splitting

In VPRS, each group is served exclusively by its own subset holding exactly as many processors as dictated by its required group share. In order to avoid problems with sharing a processor among multiple groups and subsets with processors of different speeds, we assume that the system is homogeneous and fully partitionable, which means that $r_g^{\mathrm{G}} P$ is integral for each group g. We define the *subset* Ω_g *of group* g as

$$\Omega_g \triangleq \{1 + \sum_{h=1}^{g-1} r_g^{\mathrm{G}} P, \ldots, \sum_{h=1}^{g} r_g^{\mathrm{G}} P\}. \tag{4.138}$$

In VPRS, we put

$$\pi_{gp} = \begin{cases} 1/|\Omega_g|, & g = 1, \ldots, G,\ p \in \Omega_g, \\ 0, & g = 1, \ldots, G,\ p \notin \Omega_g. \end{cases} \tag{4.139}$$

We first prove that in VPRS, load-balancing within Ω_g minimizes Δ_g^{G}, $g = 1, \ldots, G$. Then we give an expression for Δ_g^{G} with VPRS. Subsequently, we prove for a very simple case that VPRS is the optimal assignment. Finally, we prove that, even in fully partitionable systems, VPRS is not always optimal. In the next section, we show that VPRS is superior to HPRS in many cases.

Theorem 4.12 *If in a homogeneous system group g runs exclusively on P_g processors, setting $\pi_{gp} = 1/P_g$ on these processors minimizes Δ_g^{G}. In other words, the group load ρ_{g*} should be balanced among the processors serving group g.*

PROOF: It is clear that minimizing Δ_g^{G} in this case is equivalent to minimizing Δ^{T} in a homogeneous P_g-processor system with total load $\tilde{\rho} = P/P_g \times \rho_{g*}$. If $\tilde{\rho} < 1$, we have Theorem 4.5. Otherwise, balancing the load maximizes $\mathbf{E}\left[o_g^{\mathrm{G}}\right]$ (although the maximum is not unique in general) and we always have $\mathbf{E}\left[f_g^{\mathrm{G}}\right] = r_g^{\mathrm{G}}$. By (2.38), the proof is complete. □

Theorem 4.13 *In a P-processor homogeneous fully partitionable random-splitting system with VPRS, $\rho_{g*} \geq 0$, $g = 1, \ldots, G$, and $\sum_g \rho_{g*} = \rho < 1$, we have, if $\rho_g < r_g^{\mathrm{G}}$, $g = 1, \ldots, G$,*

$$\Delta^{\mathrm{T}} = 1 - \rho - \sum_{n=0}^{P-1} (1 - \frac{n}{P}) \sum_{\substack{n_1=0 \\ n_1+\cdots+n_G=n}}^{n} \cdots \sum_{n_G=0}^{n} \prod_{g=1}^{G} \binom{r_g^{\mathrm{G}} P + n_g - 1}{n_g} \left(1 - \frac{\rho_g}{r_g^{\mathrm{G}}}\right)^{r_g^{\mathrm{G}} P} \left(\frac{\rho_g}{r_g^{\mathrm{G}}}\right)^{n_g}, \tag{4.140}$$

and otherwise

$$\Delta^{\mathrm{T}} = 1 - \rho + \sum_g \max(0, \rho_{g*} - r_g^{\mathrm{G}}). \tag{4.141}$$

For the expected feasible group-share deviation Δ_g^{G} for group g we have, if $\rho_{g*} < r_g^{\mathrm{G}}$,

$$\Delta_g^{\mathrm{G}} = 1 - \sum_{n_g=0}^{r_g^{\mathrm{G}} P - 1} \left(1 - \frac{n_g}{r_g^{\mathrm{G}} P}\right) \binom{r_g^{\mathrm{G}} P + n_g - 1}{n_g} \left(1 - \frac{\rho_{g*}}{r_g^{\mathrm{G}}}\right)^{r_g^{\mathrm{G}} P} \left(\frac{\rho_{g*}}{r_g^{\mathrm{G}}}\right)^{n_g} - \frac{\rho_{g*}}{r_g^{\mathrm{G}}}, \tag{4.142}$$

and if $\rho_{g*} \geq r_g^{\mathrm{G}}$,

$$\Delta_g^{\mathrm{G}} = 0. \tag{4.143}$$

If $r_g^{\mathrm{G}} P = 1$, we also have $\Delta_g^{\mathrm{G}} = 0$.

PROOF: Clearly, we have from (4.138) and (4.139),

$$\rho_{gp} = \begin{cases} \rho_{g*}/r_g^{\mathrm{G}}, & g = 1, \ldots, G,\ p \in \Omega_g, \\ 0, & g = 1, \ldots, G,\ p \notin \Omega_g, \end{cases} \tag{4.144}$$

and

$$\rho_{*p} = \rho_{gp}, \text{ for } p \in \Omega_g. \tag{4.145}$$

Noting that $\rho_{*p} < 1$ if $\rho_{g*} < r_g^{\mathrm{G}}$, $g = 1, \ldots, G$, (4.140) follows directly from substitution of (4.145) into (4.58) with $c_p = 1/P$, $p = 1, \ldots, P$. When $\rho_{g*} \geq r_g^{\mathrm{G}}$, we have $\rho_{*p} \geq 1$ for $p \in \Omega_g$, and as a consequence, $\mathbf{E}\left[f^{\mathrm{T}}\right] = 1$. In that case, we have (4.64) for the total obtained share $\mathbf{E}\left[o^{\mathrm{T}}\right]$. Equation (4.141) now follows from (2.42).

Equations (4.142) and (4.143) follow from substitution of (4.138) and (4.139) into (4.120), (4.124), (4.125), and (4.127), again treating the cases $\rho_{g*} < r_g^{\mathrm{G}}$ and $\rho_{g*} \geq r_g^{\mathrm{G}}$ separately. □

We now prove that in a homogeneous two-processor fully one-to-one partitionable system, VPRS minimizes Δ^{G}.

Theorem 4.14 *In a homogeneous two-processor fully one-to-one partitionable random-splitting system with $\rho < 1$, VPRS minimizes Δ^{G}. Furthermore,*

$$\Delta^{\mathrm{G,MIN}} = 0. \tag{4.146}$$

This minimum is unique.

PROOF: It is clear that with VPRS we have $\Delta^{\mathrm{G}} = 0$, because each group always obtains its feasible group share of 1/2 whenever present. We will prove that with any other allocation, $\Delta_g^{\mathrm{G}} > 0$ for at least one group g. If a group is not present on both processors, then, unless we have VPRS, it is present on one processor which it must share with the other group and, clearly, $\Delta^{\mathrm{G}} > 0$. Hence, we assume that both groups are present on each processor. Because $\rho < 1$, there is at least one group g with $\rho_{g*} < 1/2$. Because this group has $\Delta_g^{\mathrm{G}} > 0$ if it is present on an unstable processor, we must have $\rho_{*1}, \rho_{*2} < 1$.

We assume that $\rho_{1*} \leq \rho_{2*}$. We will prove that $\Delta_1^{\mathrm{G}} > 0$. We have $\mathbf{E}\left[o_1^{\mathrm{G}}\right] = \rho_{1*}$ by (4.120) and (4.129). Also, $\mathbf{E}\left[f_1^{\mathrm{G}}\right] = 1/2 \times \mathbf{P}\left[N_{1*} > 0\right]$ by (4.127) and by $r_1^{\mathrm{G}} = 1/P = 1/2$. Substituting into (4.125) and using (4.124), we must prove that

$$\left(\frac{1 - \rho_{11} - \rho_{21}}{1 - \rho_{21}}\right) \cdot \left(\frac{1 - \rho_{12} - \rho_{22}}{1 - \rho_{22}}\right) < 1 - \rho_{11} - \rho_{12}. \tag{4.147}$$

If $\rho_{11} + \rho_{21} > (<) \rho_{12} + \rho_{22}$, we can increase the left-hand side of (4.147) by moving load due to group 1 from processor 1 (2) to processor 2 (1), until either $\rho_{11} + \rho_{21} = \rho_{12} + \rho_{22}$ (i.e., the loads on the processors are equal) or $\rho_{11} = 0$ ($\rho_{12} = 0$). In the latter case, $\Delta_1^{\mathrm{G}} \geq 0$, and certainly $\Delta_1^{\mathrm{G}} > 0$ in the original allocation. If $\rho_{11} + \rho_{21} = \rho_{12} + \rho_{22}$, we can increase the left-hand side of (4.147) by concentrating the load due to group 1 on a single processor, while keeping $\rho_{11} + \rho_{21} = \rho_{12} + \rho_{22}$, because doing so minimizes the denominator. We end up with $\Delta_1^{\mathrm{G}} \geq 0$, and certainly $\Delta_1^{\mathrm{G}} > 0$ in the original allocation. □

The result of Theorem 4.14 is rather strong. No matter how unbalanced the total load ρ might be among the groups, each group should have its own, dedicated, processor in this case. Clearly, a system with minimal Δ^G can have a substantial capacity loss Δ^T.

We have strong reasons to believe that in fully one-to-one partitionable homogeneous systems with more than two processors, VPRS is still optimal. We believe the optimum is unique (apart from permutations of the groups) for all groups with $\rho_{g*} < 1/P$. Unfortunately, so far, we have not been able to prove this. However, VPRS is not optimal in general, as we show by means of the following counterexample.

Theorem 4.15 *In a homogeneous fully partitionable system with $r_g^G P > 1$ for at least one group, VPRS does not minimize Δ^G in general.*

PROOF: We give one counterexample. Consider a homogeneous system with $P = 4$, $G = 2$, $r_1^G = r_2^G = 0.5$, $\rho_{1*} = 0.3$, and $\rho_{2*} = 0.2$. Clearly, the system is fully partitionable. With VPRS, we have $\rho_{11} = \rho_{12} = 0.6$, $\rho_{13} = \rho_{14} = 0$, and $\rho_{21} = \rho_{22} = 0$, $\rho_{23} = \rho_{24} = 0.4$. As a result, $\Delta_1^G = 0.144$, $\Delta_2^G = 0.096$, and $\Delta^G = \max(0.144, 0.096) = 0.144$. However, a better assignment is $\rho_{11} = \rho_{12} = 0.55$, $\rho_{13} = 0.1$, $\rho_{14} = 0$, and $\rho_{21} = \rho_{22} = 0$, $\rho_{23} = 0.35$, $\rho_{24} = 0.45$, which yields $\Delta^G = \Delta_1^G = \Delta_2^G = 0.12$. In other words, compared to VPRS, in the latter assignment a fraction of the jobs of group 1 is sent to the subset of group 2. □

Even though VPRS is not optimal in general, we believe that the assignment obtained with VPRS is "close" to the optimal assignment. Therefore, VPRS stands model for the Static Vertical Partitioning (SVP) policy introduced in Chapter 5. Contrary to what its name suggests, SVP is a dynamic policy and it uses the same fundamental idea as VPRS does, which is to reserve entire processors to groups.

4.5.4 Performance Evaluation

In this section, we compare HPRS to VPRS, and we study the performance improvement of using GPPS instead of PS as the local scheduling policy (clearly, with VPRS the local scheduling policy is irrelevant). We restrict ourselves to homogeneous fully-partitionable systems and we consider the following three cases:

1. the required group shares are equal and the group loads are equal;
2. the required group shares are equal while the group loads are not;
3. the required group shares are not equal while the group loads are.

Case 1: Equal Required Group Shares and Equal Group Loads

In the first case, we consider a system with two groups, $r_1^G = r_2^G = 0.5$, and $\rho_{1*} = \rho_{2*}$. In Figure 4.17 we plot Δ^G as a function of ρ for various numbers of processors. The local scheduling policy is PS. Of course, in this case, the capacity loss is identical for both HPRS and VPRS, since the loads on the processors are equal. Because of this, the capacity loss is as shown in Figure 4.7.

It is clear that the group-share deviation Δ^G closely follows the capacity loss Δ^T with HPRS. HPRS treats both groups equally (as is required by the r_g^G's), as does the local

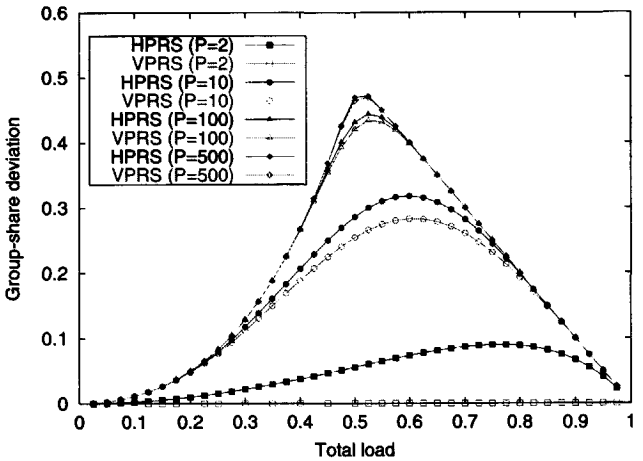


Figure 4.17: The group-share deviation Δ^G as a function of the total load ρ in a homogeneous system under HPRS/PS or VPRS ($G = 2$, $r_1^G = r_2^G = 0.5$, $\rho_{1*} = \rho_{2*}$).

scheduling policy, PS, so most of the group-share deviation is due to the capacity loss. With the same number of processors, VPRS performs better than HPRS, because with the former policy, the group-share deviation equals the capacity loss in a $P/2$-processor system. However, as P increases, the capacity loss in a $P/2$-processor system becomes practically equal to that in a P-processor system, by Theorem 4.6. With 100 or more processors, the difference between HPRS and VPRS is negligible.

In Figure 4.18 we show the effects of using GPPS over PS as the local scheduling policy with HPRS for the first case. Using GPPS only yields significant improvement when the number of processors is small. This supports the observation made earlier that the biggest problem with HPRS for this case is that of the high capacity loss, not that of a bad local scheduling policy.

Case 2: Equal Required Group Shares and Unequal Group Loads

In our second case, we vary the distribution of the load ρ among the groups, but we keep the required group shares equal. We show Δ_1^G and Δ_2^G as functions of ρ_{1*}/ρ, with $\rho = 0.8$ in Figure 4.19 for $P = 10$, and in Figure 4.20 for $P = 500$. The local scheduling policy for both figures is PS.

It is not surprising that HPRS is not capable of providing groups with their feasible share under all distributions of the workload among the groups; in fact, HPRS does not even attempt this with PS as the local policy. VPRS performs much better with uneven distributions of the workload. This was to be expected as well: with VPRS the two groups do not interfere, and the group-share deviation is caused only by the capacity loss "within the subsets". As the load becomes highly unbalanced, given the high total load ρ of 0.8, one subset will be flooded with jobs, causing the group-share deviation of that group to drop to zero. At the same time, the other subset will serve hardly any jobs, in which case

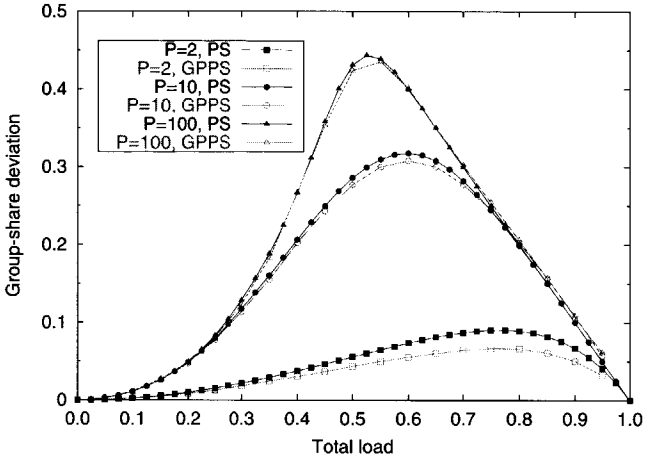


Figure 4.18: The group-share deviation Δ^{G} as a function of the total load ρ in a homogeneous system under HPRS/PS or HPRS/GPPS ($G = 2$, $r_1^{\mathrm{G}} = r_2^{\mathrm{G}} = 0.5$, $\rho_{1*} = \rho_{2*}$).

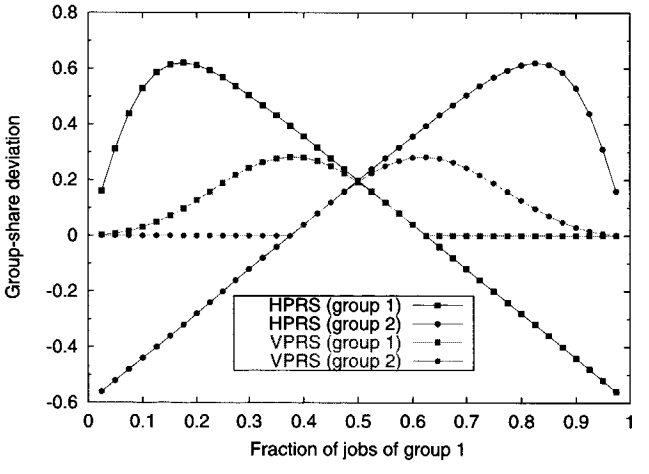


Figure 4.19: The group-share deviations Δ_1^{G} and Δ_2^{G} as functions of the fraction ρ_{1*}/ρ of jobs of group 1 in a homogeneous system under HPRS/PS or VPRS ($P = 10$, $G = 2$, $r_1^{\mathrm{G}} = r_2^{\mathrm{G}} = 0.5$, $\rho = 0.8$).

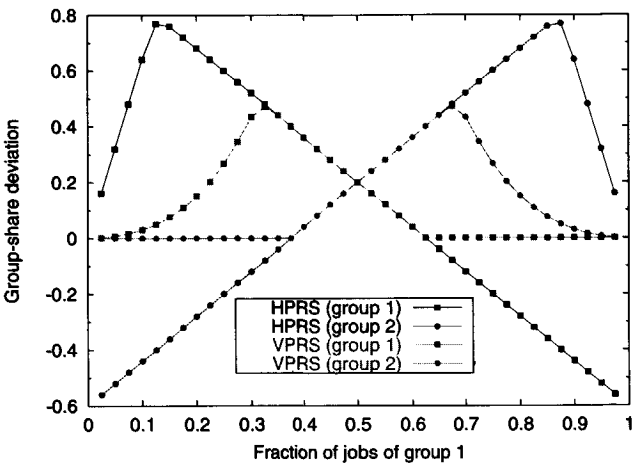


Figure 4.20: The group-share deviations Δ_1^G and Δ_2^G as functions of the fraction ρ_{1*}/ρ of jobs of group 1 in a homogeneous system under HPRS/PS or VPRS ($P = 500$, $G = 2$, $r_1^G = r_2^G = 0.5$, $\rho = 0.8$).

the group-share deviation of that group drops to zero as well.

However, one should note that even with VPRS, the group-share deviation can be quite high, upto 0.5 with many processors (see Figure 4.20). Clearly, this is because up to half the capacity of a subset can be lost (see Figure 4.7). In our case with $P = 500$, the maximum is reached when the group load is 0.25 (and hence the subset experiences an equivalent load of 0.5), in other words, when the fraction of jobs of group 1 is $0.25/0.8 = 0.3125$, in which case the maximum is reached for group 1, or $1 - 0.3125 = 0.6875$, in which case the maximum is reached for group 2.

In Figure 4.21 we show the effect of using GPPS as the local scheduling policy with HPRS for the second case. Compared to the first case, in the second case the use of GPPS instead of PS has more benefits, even with 100 processors. However, still, HPRS/GPPS performs worse than VPRS.

Case 3: Unequal Required Group Shares and Equal Group Loads

Instead of changing the distribution of the load among the groups, in the third case we keep this distribution constant ($\rho_{*1} = \rho_{*2} = 0.4$), and study the performance of the policies when the required group shares are unequal. In Figure 4.22 we show Δ_1^G and Δ_2^G as functions of $|\Omega_1| = r_1^G P$, for a system with $P = 100$ and PS as the local scheduling policy on all processors. At all times, we keep the system fully partitionable. However, for readability, we have connected the points in the graphs with lines. From the figure, we conclude that VPRS performs better than HPRS. However, as in the previous case, the group-share deviation with VPRS can be as high as 0.5, which we consider unacceptably high.

In Figure 4.23 we show the effect of using GPPS as the local scheduling policy with

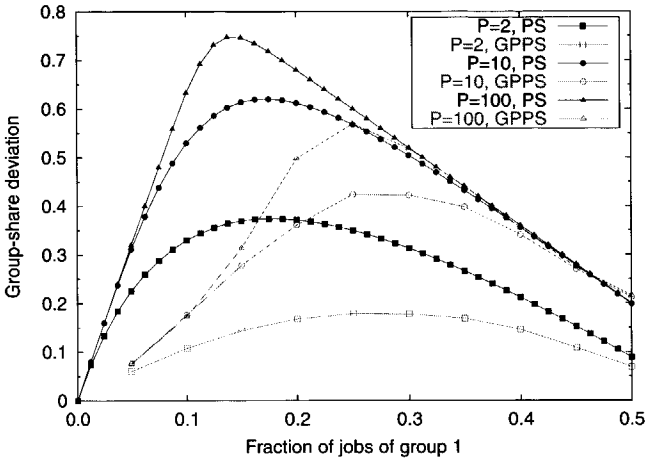


Figure 4.21: The group-share deviation Δ^{G} as a function of the fraction ρ_{1*}/ρ of jobs of group 1 in a homogeneous system under HPRS/PS or HPRS/GPPS ($G = 2$, $r_1^{\mathrm{G}} = r_2^{\mathrm{G}} = 0.5$, $\rho = 0.8$).

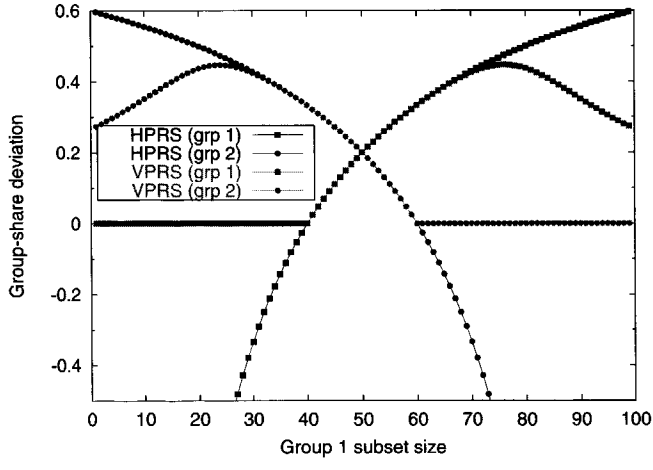


Figure 4.22: The group-share deviations Δ_1^{G} and Δ_2^{G} as functions of the subset size $|\Omega_1|$ of group 1 in a homogeneous system under HPRS/PS or VPRS ($P = 100$, $G = 2$, $\rho_{1*} = \rho_{2*} = 0.4$).

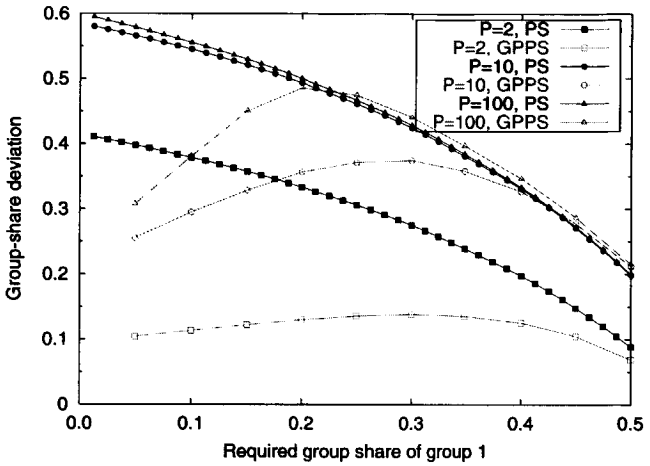


Figure 4.23: The group-share deviation Δ^{G} as a function of the required group share r_1^{G} of group 1 in a homogeneous system under HPRS/PS or HPRS/GPPS ($G = 2$, $\rho_{1*} = \rho_{2*} = 0.4$).

HPRS for the third case. Clearly, with a modest number of processors and a high level of discrimination required among the groups, the use of GPPS improves performance considerably. However, with a large number of processors (e.g. 100), the improvement is minor. The same holds for the case where the required group shares are almost equal.

Conclusions

Based on the performance comparison between HPRS and VPRS, we state the following conclusions:

1. in general, both policies suffer from high capacity loss (in the optimal case, upto 0.5) and a high group-share deviation;
2. unless the number of processors is very high, HPRS is superior to VPRS with respect to minimization of the capacity loss, while it is inferior with respect to compliance with the feasible group shares;
3. although HPRS benefits from using GPPS instead of PS as the local scheduling policy, the improvement is not enough to outperform VPRS.

4.6 Conclusions

In this chapter, we have considered three scheduling problems. In the first problem (Section 4.2), a fixed set of jobs must be scheduled on a heterogenous set of processors in such a way that prespecified job service rates are met. Job migration is free. We have provided necessary and sufficient conditions for the existence of a schedule, and we

have shown how to construct such a schedule if it exists. This makes share scheduling in systems with free job migration an easy exercise, because at all times, the feasible group shares can be met with zero capacity loss, for instance with the Multiprocessor Group-Priority Processor-Sharing policy introduced in Section 4.2.2.

The other scheduling problems involve random splitting, for which we defined the model in Section 4.3. In the second problem (Section 4.4), we try to minimize the capacity loss Δ^{T}, in the third problem (Section 4.5), the group-share deviation Δ^{G}. We have shown that in a homogeneous system with PS as the local scheduling policy, balancing the load among the processors minimizes the capacity loss when arrivals are Poisson and service times have exponential distributions with possibly different means for each group. We have also provided the optimal solution for the heterogeneous two-processor system, but could not extend this result to arbitrary numbers of processors. For the minimization of the group-share deviation, we have shown that it is better to allocate entire processors to groups (as is done in Vertical Partitioning in Random Splitting, VPRS), than to spread evenly the jobs of each group among all processors (as is done in Horizontal Partitioning in Random Splitting, HPRS).

Using random splitting for share scheduling suffers from severe problems. First of all, the system suffers from capacity loss. Moreover, even the optimal routing probabilities are not sufficient for acceptably low group-share deviation, except in the rare case of fully one-to-one partitionable systems. Finally, minimization of capacity loss and compliance with the feasible group shares are sometimes conflicting objectives. In the next two chapters we will study whether the performance improves when dynamic policies are used. In particular, we will explore in greater detail the approach of partitioning the system. To that end, we will propose dynamic policies based upon the ideas behind HPRS and VPRS.

Chapter 5

Dynamic Central Share Scheduling

5.1 Introduction

In general, policies for (share) scheduling in distributed systems consist of a *global* component, which decides which job runs on which processor, and of a *local* component, which decides how to schedule jobs locally. In this chapter, we study policies whose global components are *dynamic* and *central*. A dynamic policy uses information about the actual state of the system (for instance, the current numbers of jobs present on the processors), as opposed to the *static* policies described in Chapter 4, which use only static information. The potential advantages of dynamic policies over their static counterparts, such as better adaptation to workload fluctuations and increased flexibility, have been pointed out by many authors [11, 24, 25, 84]. A central policy is a policy run by a single process. In other words, there is a single decision point [11]. Policies with multiple decision points are called *distributed* policies. In such policies, decisions are made at different locations, and information is exchanged between these locations. We study distributed policies in Chapter 6.

In our model, the decisions for global scheduling are only taken upon job arrivals and departures, the scheduling *events*, and the system state (in terms of the numbers of jobs present on the processors) does not change in between. Also, jobs are assumed to have very long service times, on the order of minutes to hours. If the number of processors is not too large (say, smaller than 100), the average time between scheduling events is on the order of minutes or seconds at the worst. As a result, we may safely assume that the information required by the global component of the policy to arrive at scheduling decisions is available instantaneously. Clearly, the cost of maintaining this information centrally is negligible. Hence, we assume that the scheduling policy always has complete and current knowledge of the state of the system.

Our objectives in this chapter are

1. to compare combinations of global and local policies with respect to compliance with the feasible group shares and to minimization of capacity loss, and to answer the fundamental question to what extent the choices of the global and local policy affect the performance;

2. to compare dynamic global policies to the static policies HPRS and VPRS introduced in Chapter 4;

3. to study the benefits of job migration.

The performance measures we use have been defined in Chapter 2: the group-share deviation Δ^G and the capacity loss Δ^T. We do not consider the job-share deviation Δ^J in this chapter.

Our main conclusion is that for a workload consisting of non-permanent jobs, in practically all cases the global dynamic policies outperform the static policies of Chapter 4. This is in agreement with the results reported in the literature on global scheduling policies for load sharing and load balancing. With the availability of job migration, capacity loss is zero with all global dynamic policies, and compliance with the feasible group shares is much better than in systems without job migration. However, as the number of processors becomes very large (say, 100 processors with just a few groups), a combination of simple global and local policies (such as Join Shortest Queue with Processor Sharing) suffices for low capacity loss and good compliance with the feasible group shares, even if job migration is absent.

In Section 5.2 we describe our system model, followed by an overview of related literature in Section 5.3. In Section 5.4, we define two important state properties, *system balance* and *group balance*. We propose four global dynamic policies in Section 5.5. Deriving analytical expressions for the performance measures Δ^G and Δ^T is infeasible. Therefore, we use simulation for a performance comparison of the policies. We present the simulation results in Sections 5.6 and 5.7 for systems without and with job migration, respectively. We summarize our conclusions in Section 5.8.

5.2 System Model

In this section we fill in some of the parameters of the distributed-system model outlined in Section 1.1.3. The number of parameters of the full model is too large for simulation studies in which all parameters are varied over their allowed ranges, so we make some restrictions. We first describe the model assumptions, then we give an outline of the lifecycle of a job.

5.2.1 Model Assumptions

In this chapter, we only consider homogeneous systems, i.e., we put $c_1 = \cdots = c_P = 1/P$ and $c = 1$ (irrespective of P). Some of the policies we will propose would become rather complex when defined for heterogeneous systems. Also, distributed systems used for cluster computing often consist of machines whose speeds are (almost) equal.

Second, we assume that all processors run the same local scheduling policy, which is either Processor Sharing (PS) or Group-Priority Processor Sharing (GPPS). These policies, which have been described in Chapter 3, represent two extremes of the processor-sharing class of local scheduling policies: PS allows no control over the obtained shares of the jobs running on a processor, while GPSS is considered the ideal policy with perfect control for share scheduling. We do not consider any of the non-preemptive and preemptive policies described in Chapter 3, because most contemporary general-purpose operating systems use some form of round-robin scheduling.

Third, we assume that job migration is free of cost, but we allow it only at arrivals and departures epochs. When a job arrives, the global policy assigns it to a processor and is allowed to migrate a *single* other job. The global policy is also allowed to migrate a single job when a job departs. The choices *whether* to migrate a job and if so, *which* job to migrate are up to the global policy. In distributed systems, job migration is usually expensive in terms of network load and network delay. On the other hand, transmission speeds have grown substantially in the recent years, from 10 Mbps around 1990 to 1 Gbps around 2000. Even a job with a large address space of say 100 MegaBytes can be migrated in about a second on a 1 Gbps network. Hence, if we restrict migrations to occur at arrivals and departures, we can safely assume their costs to be negligible.

5.2.2 Lifecycle of a Job

We summarize the lifecycle of a job:

- The job arrives at the central scheduler.
- The scheduler takes the initial-placement decision for the job, and sends it to the destination processor. The scheduler can also migrate another job at the same instant.
- The destination processor serves the job until completion, or until the job is selected for migration to another processor.
- If the job is selected for migration, it is transferred across the network to the new destination processor. However, transferring the job takes place in zero time.
- When the job terminates, it leaves the system.
- The job-migration component of the global scheduler is invoked, which may cause the migration of a single job.
- If the original job was permanent, a new job arrives immediately.

We will study a more realistic model of the network in Chapter 6.

5.3 Related Literature

Numerous authors compare static scheduling policies to dynamic ones. As a rule, relatively simple dynamic policies outperform static policies such as random splitting and cyclic splitting with respect to minimization of mean job response time.

Chow and Kohler [13] compare dynamic policies with proportional branching (defined in Section 4.4.3) in a heterogeneous system. All three dynamic policies considered perform significantly better than the static random-splitting policy. We refer to Section 1.3.6 for more detailed descriptions of the policies and of the system model.

Livny and Melman [61] show the potential of reducing mean job response time of load-balancing algorithms in a homogeneous distributed system with a broadcast-type contention-based communications network (e.g., an Ethernet). They compare three load-balancing algorithms to a set of independently operating M/M/1 queues, each with load

ρ. The local scheduling policy on the processors is FCFS, and jobs may be transferred between processors as long as their execution has not yet started. Because the effects of communication delays play an important aspect in their study, we defer a more detailed description until Section 6.6, where we will study the use of stale and incomplete information on the system state by scheduling policies.

Wang and Morris [84] compare dynamic global scheduling policies with random splitting and cyclic splitting/service disciplines in homogeneous systems without job migration. We already discussed their article in Sections 1.3.3, 1.3.6, and 4.3.2, and refer to those sections for more details. Their focus is on the distinction between source-initiative and server-initiative policies, and on the amount of information needed by a global policy. Without job migration, all policies in our model are source initiative. With job migration, our policies are both source initiative (upon arrivals) and server initiative (upon departures). Our main interest is in the potential performance improvements of dynamic policies over static ones. We will find that with share-scheduling objectives, just as in [84] for minimization of mean job response time, even very simple dynamic policies outperform (static) random-splitting policies.

In this chapter, we cover some fundamental approaches and algorithms studied in [84], such as "server partition" (the SVP policy introduced in Section 5.5.3), "random splitting" (the VPRS and HPRS policies studied in Chapter 4), and "Join Shortest Queue". Noteworthy approaches we do not consider are "cyclic splitting" (sources send arriving jobs to servers in a prespecified sequence) and "cyclic service" (servers take jobs from sources in a prespecified sequence). Although the latter two approaches are very interesting for share scheduling, finding the optimal sequence is far more complicated with share-scheduling objectives than with load-balancing objectives.

Eager, Lazowska, and Zahorjan [24, 25] show the performance benefits of simple load-balancing policies over systems without load balancing (mostly modeled as a set of M/M/1 queues with identical loads). We will describe their work in more detail in Section 6.6, because the effects of communications delays and of using stale and incomplete information play an important role.

Leland and Ott [59] study initial-placement and job-migration policies in homogeneous distributed systems. They conclude that

1. In their computing environment, the service time of long-lived jobs does *not* have an exponential distribution, nor a distribution with an exponential tail. Instead, the service time of such jobs shows a decreasing hazard rate $\eta(\cdot)$, which is defined as follows for a distribution function $F(\cdot)$:

$$\eta(x) \triangleq -\frac{d}{dx}\log(1 - F(x)). \tag{5.1}$$

   Note that a random variable X with exponential distribution and mean $\mathbf{E}[X]$ has constant hazard rate $\eta(x) = 1/\mathbf{E}[X]$. Intuitively, the decreasing hazard rate of the service-time distribution means that $\mathbf{P}[X > x + h | X > x]$, $h > 0$, increases in x. In other words, the distribution is not "memoryless", as is the case with an exponential distribution.

2. Initial placement and job migration both provide significant reductions in response-time ratios of large jobs over systems in which jobs are served on the processor to which they were submitted.

3. Simple heuristic schemes suffice to achieve the aforementioned performance improvements.

Krueger and Livny [56] compare various combinations of global and local scheduling policies using a variety of performance indices, such as the job waiting time, the job wait ratio, and the correlation between job wait ratio and job service time and between job waiting time and job service time. The basic model is a homogeneous $P\times$(M/H/1) queueing system, i.e., P independent servers with Poisson arrivals and hyperexponential service-time distributions. As in [59], the authors argue that service-time demands are ill-described with an exponential distribution. Jobs arrive at the processors according to Poisson processes with different rates. All scheduling mechanisms (initial placement, job migration, preemption) are free of cost. An important result from their study is that for some performance indices, notably the waiting-time ratio, the choice of the local scheduling policy is very important. In order to reduce the mean and variance of the waiting-time ratio, PS should be used instead of FCFS as the local policy. With respect to the global policies, they argue that the narrow objective of load sharing (preventing processors from being unnecessarily idle) only yields reasonable performance in systems with homogeneous arrival rates at the nodes and a small coefficient of variation of the service-time distribution. In other cases, load balancing (balancing the total number of jobs among the processors) is necessary.

Our conclusion from this literature study is that for many response-time related objectives, simple dynamic policies provide significant performance improvement over static policies such as random splitting, cyclic splitting and server partition. However, in many of the cases studied, the choice of the local scheduling policy seems of minor importance, because the objective function is only influenced by the number of processors busy and the total number of jobs present (as is capacity loss). Since compliance with the feasible group shares deals explicitly with the distribution of the total system capacity over groups of jobs, we expect that the local scheduling policy will play a more prominent role in our case.

5.4 System Balance and Group Balance

It should be clear from the preceding discussion that the system state, in terms of jobs present on processors, can only change upon an arrival or a departure of a job. Most global scheduling policies include actions to somehow spread jobs among the processors (examples of these are load-balancing policies). The following state properties, *system balance* and *group balance*, are used to specify that the system is in some kind of balance. The definitions only apply to homogeneous systems.

Definition 5.1 *The system is said to be in system balance at time t if, for all $p, q = 1, \ldots, P$, the numbers of jobs on processors p and q do not differ by more than one, i.e.,* $||J_{*p}(t)| - |J_{*q}(t)|| \leq 1$.

In other words, system balance is achieved if the jobs are optimally spread among the processors. This has a number of advantages. Clearly, system balance is a sufficient condition for zero capacity loss. Also, when PS is used, jobs are treated as fairly as

possible: no migration can decrease the maximal difference in service rates of two arbitrary jobs. Finally, when the expected residual service times of all jobs are equal, the amount of work in the system is optimally spread. Intuitively, this helps reducing capacity loss in the future (i.e., during the remaining lifetimes of these jobs).

Definition 5.2 *The system is said to be in group balance for group g, $g = 1, \ldots, G$, at time t if, for all $p, q = 1, \ldots, P$, the numbers of jobs of group g on processors p and q do not differ by more than one, i.e., $||J_{gp}(t)| - |J_{gq}(t)|| \leq 1$. The system is said to be in group balance when it is in group balance for all groups.*

When the system is in group balance for group g, the jobs of group g are optimally spread among the processors. A system can be in system balance but *not* in group balance, and vice versa.

5.5 Global Scheduling Policies

We distinguish two components of a global scheduling policy in distributed systems, the *initial-placement* policy and the *migration* policy. The initial-placement policy decides which processor is to serve an arriving job. The migration policy decides when to migrate which job to which processor. In the next sections we propose four global policies for share scheduling in distributed systems: Join Shortest Queue, Horizontal Partitioning, Static Vertical Partitioning, and Dynamic Vertical Partitioning. For each policy, we describe the initial-placement and migration components of the global policy. In every policy, ties (for instance, due to equal queue lengths at some processors) are broken with equal probabilities. Although the policies can be generalized to heterogeneous systems, we have chosen to describe them for homogeneous systems only.

If GPPS is the local scheduling policy, we must set the *group weights* on the individual processors. (The meaning of the group weights has been explained in Section 3.4.3.) With all global policies, the group weights are set only once, and remain constant thereafter. We denote the weight of group g on processor p by w_{gp}.

5.5.1 Join Shortest Queue

The Join Shortest Queue (JSQ) policy is a typical load-balancing policy. The only objective of JSQ is to obtain and preserve system balance. The policy ignores the group to which a job belongs, and therefore, relies on the local scheduling policy for compliance with the feasible group shares. If GPPS is the local scheduling policy, we put $w_{gp} = r_g^{\mathrm{G}}$, $g = 1, \ldots, G$, $p = 1, \ldots, P$.

The initial-placement policy of JSQ for an arriving job is as follows:

I1 Choose the processor p for which $|J_{*p}(t)|$ is minimal.

There is no job migration upon arrivals. If job migration is available, the job-migration policy of JSQ upon departure of a job from processor q is as follows:

M1 Choose a random job on the processor p for which $|J_{*p}(t)|$ is maximal, and migrate the job from p to q if $|J_{*p}(t)| \geq |J_{*q}(t)| + 2$.

These initial-placement and job-migration policies have the following consequences.

1. JSQ maintains system balance upon arrivals. Therefore, JSQ also maintains zero capacity loss upon arrivals.

2. If job migration is available, JSQ preserves system balance (and zero capacity loss) upon departures, but if job migration is not available, departing jobs may violate system balance and zero capacity loss.

3. Because of the previous two remarks, JSQ as a whole preserves system balance and zero capacity loss if job migration is available.

4. In rule M1, if the system is in system balance before the departure of the job from processor q, we can never have $|J_{*p}(t)| > |J_{*q}(t)| + 2$ after the departure but right before the possible migration. Still, we use the present formulation because it is more robust: Should the system ever enter a state in which there is no system balance, the current rule still works appropriately.

The JSQ initial-placement policy has received much attention in the literature. Some authors [54, page 9] refer to it as the Shortest Queue Policy. In homogeneous systems without job migration, the JSQ policy minimizes the expected number of jobs in homogeneous systems for Poisson arrivals and exponential service times [85]. Chow and Kohler derive the mean job response time for the two-processor case in [12]. So far, a closed-form expression for the joint queue-length distribution for JSQ has not been reported in the literature. Cohen studies this distribution for the homogeneous two-processor case in [17], and for the heterogeneous two-processor case in [18]. Nelson and Philips propose an approximation for the mean job response time for general interarrival and service-time distributions in [68]. Only the first and second moments of these distributions are used. The approximation is based upon the notion that for sufficiently low and sufficiently high values of the total load, the JSQ system behaves as a G/G/P system (P is the number of servers). In a G/G/P system, a processor is never idle while another processor serves two or more jobs (or has a job waiting in its queue).

In order to illustrate this, in Figure 5.1, we show the probability of queueing in a homogeneous M/M/P system as a function of the total load ρ for various values of the number P of processors. As the load increases from 0 upwards, the probability of queueing is almost zero upto a certain point. If ρ increases further, the probability of queueing quickly increases to unity. However, the aforementioned point moves towards $\rho = 1$ as the number of processors increases. The important consequence of this is that the difference between JSQ with and without job migration, as well as the effect of the local scheduling policy, will diminish as the number of processors increases.

If arrivals are Poisson at rates $\lambda_1, \ldots, \lambda_G$, $\sum_g \lambda_g = \lambda$, and the service-time distribution on a single processor serving no other jobs is exponential with mean P/μ irrespective of the group to which a job belongs, JSQ with job migration amounts to an M/M/P system with PS on every processor and total capacity $c = 1$. Then [1], if $\rho \triangleq \lambda/(c\mu) < 1$, for $n = 0, 1, 2, \ldots$,

$$\mathbf{P}[N = n] = \begin{cases} \frac{P^n \rho^n}{n!} \mathbf{P}[N = 0], & n = 0, 1, \ldots, P, \\ \frac{P^P \rho^n}{P!} \mathbf{P}[N = 0], & n = P+1, P+2, \ldots, \end{cases} \tag{5.2}$$

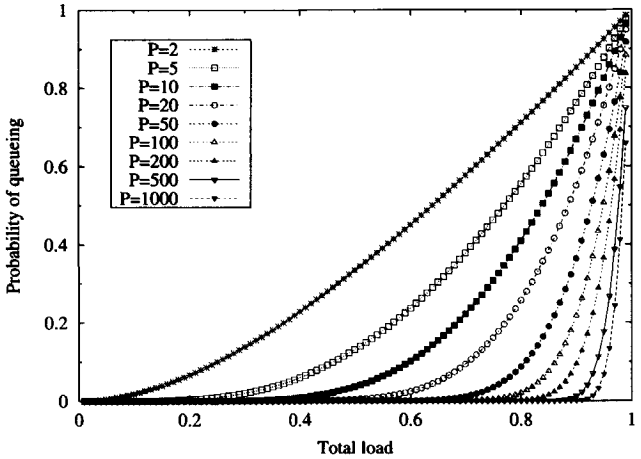


Figure 5.1: The probability of queueing in a homogeneous M/M/P system as a function of the total load ρ for various values of the number P of processors.

with $\mathbf{P}[N=0]$ such that $\sum_n \mathbf{P}[N=n]=1$, i.e.,

$$p_0 \triangleq \mathbf{P}[N=0] = \left(\sum_{n=0}^{P} \frac{P^n \rho^n}{n!} + \frac{P^P \rho^{P+1}}{P!(1-\rho)}\right)^{-1}. \tag{5.3}$$

Clearly, the probability that a job belongs to group g is $\lambda_g/\lambda = \rho_{g*}/\rho$, so for $n=0,1,\ldots$, $m=0,1,\ldots,n$,

$$\mathbf{P}[N_{g*}=m|N=n] = \binom{n}{m}\left(\frac{\rho_{g*}}{\rho}\right)^m \left(1-\frac{\rho_{g*}}{\rho}\right)^{n-m}. \tag{5.4}$$

By (2.38) and (2.39),

$$\Delta_g^{\mathrm{G}} = \frac{\mathbf{E}\left[f_g^{\mathrm{G}}\right] - \mathbf{E}\left[o_g^{\mathrm{G}}\right]}{r_g^{\mathrm{G}}} = \frac{\mathbf{E}\left[f_g^{\mathrm{G}}\right] - \rho_{g*}}{r_g^{\mathrm{G}}}, \tag{5.5}$$

so we can calculate Δ_g^{G} from (5.5) and $\mathbf{E}\left[f_g^{\mathrm{G}}\right]$. By (2.41), $\mathbf{E}\left[f_g^{\mathrm{G}}\right]$ depends only on the stationary probability distribution function of $N_{g*}(t)$. Hence, using (2.41), (5.2), and (5.4), and defining, for $g=1,\ldots,G$,

$$\kappa_g \triangleq \lfloor r_g^{\mathrm{G}} P \rfloor, \tag{5.6}$$

we have

$$\mathbf{E}\left[f_g^{\mathrm{G}}\right] = \sum_{n=0}^{\infty}\sum_{m=0}^{n} \min(r_g^{\mathrm{G}}, m/P)\mathbf{P}[N_{g*}=m|N=n]\,\mathbf{P}[N=n]$$

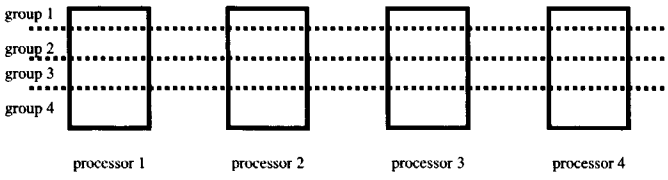


Figure 5.2: Horizontal Partitioning.

$$
\begin{aligned}
&= r_g^{\mathrm{G}} - p_0 \sum_{n=0}^{P} \frac{P^n}{n!} \sum_{m=0}^{\kappa_g} \left(r_g^{\mathrm{G}} - \frac{m}{P}\right) \binom{n}{m} \rho_{g*}^m (\rho - \rho_{g*})^{n-m} \\
&\quad - \frac{P^P p_0}{P!} \sum_{m=0}^{\kappa_g} \left(r_g^{\mathrm{G}} - \frac{m}{P}\right) \left(\frac{\rho_{g*}}{\rho - \rho_{g*}}\right)^m \sum_{n=P+1}^{\infty} \binom{n}{m} (\rho - \rho_{g*})^n \\
&= r_g^{\mathrm{G}} - p_0 \sum_{n=0}^{P} \frac{P^n}{n!} \sum_{m=0}^{\kappa_g} \left(r_g^{\mathrm{G}} - \frac{m}{P}\right) \binom{n}{m} \rho_{g*}^m (\rho - \rho_{g*})^{n-m} \\
&\quad - \frac{P^P p_0}{P!(1-(\rho-\rho_{g*}))} \sum_{m=0}^{\kappa_g} \left(r_g^{\mathrm{G}} - \frac{m}{P}\right) \left(\frac{\rho_{g*}}{(\rho-\rho_{g*})(1-(\rho-\rho_{g*}))}\right)^m \\
&\quad + \frac{P^P p_0}{P!} \sum_{m=0}^{\kappa_g} \sum_{n=m}^{P} \left(r_g^{\mathrm{G}} - \frac{m}{P}\right) \binom{n}{m} \rho_{g*}^m (\rho - \rho_{g*})^{n-m}, \qquad (5.7)
\end{aligned}
$$

where we have used that for $n = 0, 1, 2, \ldots$, $|a| < 1$,

$$\sum_{i=0}^{\infty} \binom{n+i}{n} a^i = \left(\frac{1}{1-a}\right)^{n+1}. \qquad (5.8)$$

By (5.7), $\mathbf{E}\left[f_g^{\mathrm{G}}\right]$ (and Δ_g^{G}) can be calculated with a finite number of additions and multiplications.

We will study JSQ's performance with and without job migration, and compare it to the other policies in Sections 5.6 and 5.7.

5.5.2 Horizontal Partitioning

The Horizontal Partitioning (HP) policy is to spread the jobs of each group among the processors, and use the local scheduling policy to discriminate among the groups, as depicted in Figure 5.2. Hence, the foremost objective of HP is to obtain and preserve group balance. HP is the natural extension of HPRS described in Section 4.5.2 to the class of dynamic policies. If GPPS is the local scheduling policy, we put $w_{gp} = r_g^{\mathrm{G}}$, $g = 1, \ldots, G$, $p = 1, \ldots, P$.

The initial-placement policy of HP for an arriving job j of group g is as follows:

I1 Choose the processor p for which $|J_{gp}(t)|$ is minimal. If more than one such processor exists, choose among them the processor p for which $|J_{*p}(t)|$ is minimal.

There is no job migration upon arrivals. If job migration is available, the job-migration policy of HP upon departure of a job j of group g from processor q is as follows:

M1 Choose a random job of group g from the processor p for which $|J_{gp}(t)|$ is maximal and migrate the job from p to q if $|J_{gp}(t)| \geq |J_{gq}(t)| + 2$. If more than one such processor exists, choose among them the processor p for which $|J_{*p}(t)|$ is maximal.

M2 If no processor was found in rule M1, choose the processor p for which $|J_{*p}(t)|$ is maximal. If $|J_{*p}(t)| \geq |J_{*q}(t)| + 2$, select at random a group h for which $|J_{hp}(t)| \geq |J_{hq}(t)| + 1$ and migrate a random job of group h from p to q.

These initial-placement and job-migration policies have the following consequences.

1. HP maintains zero capacity loss upon arrivals, because the initial-placement policy always prefers an idle processor over a busy one.

2. HP maintains group balance upon arrivals.

3. If job migration is available, HP preserves group balance and zero capacity loss upon departures, but if job migration is not available, departing jobs may violate group balance and zero capacity loss.

4. Because of the previous remarks, HP as a whole preserves group balance and zero capacity loss if job migration is available.

5. HP does *not* preserve system balance in general. For instance, consider a two-processor system with three groups. Suppose a job of group 1 arrives at some time t, and assume that $|J_{11}(t)| = 0$, $|J_{12}(t)| = 1$, $|J_{21}(t)| = 1$, $|J_{22}(t)| = 0$, $|J_{31}(t)| = 1$, and $|J_{32}(t)| = 0$, so the system is in both system balance and in group balance. However, HP picks processor 1, violating system balance.

6. In rule M2, if the system is in group balance before the departure of the job from processor q, we can never have $|J_{hp}(t)| > |J_{hq}(t)| + 2$ for any group h. We use the present formulation because of its robustness.

We introduced the original version of HP in [46]; more details on the ideas behind the policy are given in [45]. An important difference between the current version and the original one is that the original version did not have a migration policy. Also, the initial-placement policy depended upon the local scheduling policy, because decisions were based upon the share an arriving job could obtain on a processor, instead of upon the number of jobs present. We introduced a revised version of HP that included job migration in [48], and we found that the use of job migration substantially improved the compliance with the feasible group shares.

Obviously, the performance of HP with respect to minimizing Δ^{G} will depend on the local scheduling policy. When we use a good local policy such as GPPS, we expect a significant performance improvement with HP over JSQ under high loads when the groups have enough jobs to be present at every processor. However, we expect that, even with GPPS, the policy does not perform very well when some group g has a low group load, while other groups have a high group load. In that case, group g will be present on only a few processors and will obtain only a small share on these processors due to the presence of the other groups. Recall that as long as $|J_{g*}(t)| < r_g^{\mathrm{G}} P$, each job of group g should obtain a processor on its own, by (2.26). When PS is used as the local policy,

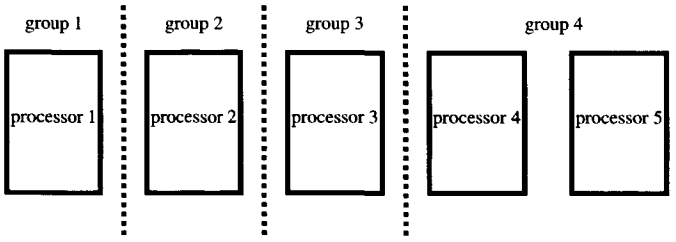


Figure 5.3: Vertical Partitioning in a fully partitionable system.

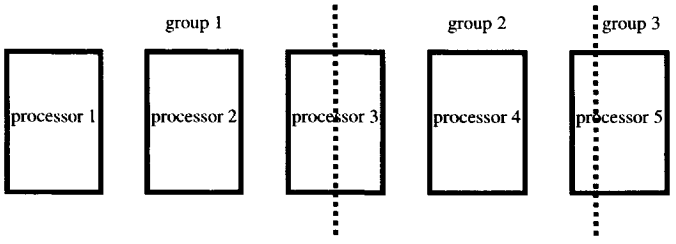


Figure 5.4: Vertical Partitioning in a non-partitionable system.

we expect that the performance of HP will strongly depend on the distribution of the workload across the groups in relation to their required group shares.

We will study HP's performance with and without job migration, and compare it to the other policies in Sections 5.6 and 5.7.

5.5.3 Static Vertical Partitioning

A natural heuristic approach to share scheduling is to dedicate subsets of the processors to groups in such a way that the number of processors in each subset is proportional to the required share of the corresponding group. We call this idea *vertical partitioning* (VP). Clearly, a perfect match is only possible in fully partitionable systems (see Section 2.4.5), as shown in Figure 5.3. However, the idea can be extended to generic systems if we allow processors to be assigned in parts. This is illustrated in Figure 5.4. On a processor assigned to multiple groups (such as processor 3 in Figure 5.4), the local scheduling policy must see to it that these groups, when present on the processor, are served at the proper rates. Therefore, a good local policy is still preferable in systems that are not fully partitionable.

At most two processors in a group's subset must be shared with other groups. Therefore the interference from other groups diminishes as the number of processors in the subset increases, and so does the effect of the local policy. VP is therefore expected to perform extremely well when the number of processors in each subset is large.

The idea of vertical partitioning is promising, because contrary to JSQ and HP, VP assigns entire processors to groups, instead of a part of *each* processor. As a result, jobs

of different groups will interfere with each other much less in VP than in JSQ and HP. For instance, in a system with $r_g^G = 1/P$ for all g (a *fully one-to-one partitionable* system, as introduced in Definition 2.3), VP dedicates a single processor to each group, and we always have $\Delta^G = 0$, irrespective of the local policy used. In the same system, this cannot be obtained with JSQ or HP.

However, the advantages of VP come at a price. The major problem with VP is that of capacity loss. If jobs of a group are served only by processors in the group's subset, processors in other subsets may become unnecessarily idle. When job migration is available, this problem is easily solved by allowing jobs to run in subsets of other groups, and migrate them when needed.

We introduce two variants of VP: Static Vertical Partitioning (SVP) and Dynamic Vertical Partitioning (DVP). In SVP, partitioning is done beforehand as part of the initialization of the system. Contrary to SVP, in DVP partitioning is done dynamically, depending on the current state (such as the number of jobs of each group running on each processor). We describe the DVP policy in Section 5.5.4.

In SVP, the process of partitioning consists of calculating the so-called *partitioning matrix* (X_{gp}), $0 \leq X_{gp} \leq 1$, $g = 1, \ldots, G$, $p = 1, \ldots, P$. When $X_{gp} = 0$, jobs of group g are not served on processor p (we will see later on that this is only partially true), and when $X_{gp} = 1$, processor p is dedicated to group g. When $0 < X_{gp} < 1$, group g shares processor p with other groups. From the preceding discussion, it is clear that we want the partitioning matrix to obey

$$\frac{1}{P} \sum_{p=1}^{P} X_{gp} = r_g^G, \ g = 1, \ldots, G. \tag{5.9}$$

We define the *subset Ω_g of processors for group g*, for $g = 1, \ldots, G$, as

$$\Omega_g \triangleq \{p | X_{gp} > 0\}. \tag{5.10}$$

We will first turn our attention to the construction of the partitioning matrix (X_{gp}). Then we will describe the initial-placement and job-migration policies.

Partitioning Algorithm

In order to derive a suitable partitioning matrix, we note that the number of processors p in Ω_g with $0 < X_{gp} < 1$ should be as low as possible, and the number of processors with $X_{gp} = 1$ should be as high as possible. This has two advantages. First, the interference between jobs of different groups (which arises on processors that are not dedicated to a single group) is kept low. Second, a group can attain its required share with as few jobs in the system as possible.

The partitioning algorithm for SVP consists of two steps:

1. Assign processors to combinations of groups in case the system is partitionable.
2. Assign processors within each combination of groups in case the system is not partitionable.

We define the *required subset size ω_g of group g* (not necessarily integral) as

$$\omega_g \triangleq r_g^G P. \tag{5.11}$$

Initially, we put $X_{gp} = 0$ for all groups g and for all processors p.

Step 1: Assign processors to combinations of groups. When the system is partitionable, we first partition the set of groups into as many subsets as possible such that the sum of the ω_g's in each subset is integral. To each subset, we assign a number of processors equal to the sum of the required subset sizes of the groups in it. In case there is only one group, say g, in a subset, we are finished for that subset, and we set $X_{gp} = 1$ on the processors assigned. The second step of the algorithm is used on each of the remaining subsets and the processors assigned to it. Before invoking step 2 of the algorithm, we rescale the required shares of the groups in the subset to sum to one. Clearly, such a subset is not partitionable.

Determining whether or not a system is partitionable (and finding the optimal partition) is far from trivial. For example, if $P = 2$, we must determine whether there exists a subset $\mathcal{A}$ of $\{1, \ldots, G\}$ with $0 < |\mathcal{A}| < G$ such that $2\sum_{g \in \mathcal{A}} r_g^{\mathrm{G}} = 1$. This is the classical knapsack problem with real-valued item sizes $2r_1^{\mathrm{G}}, \ldots, 2r_G^{\mathrm{G}}$ and a knapsack of size one. The knapsack problem is NP-complete.

Step 2: Assign processors within each combination of groups. When the system is not partitionable, we first assign $\lfloor \omega_g \rfloor$ entire processors to each group g, i.e., we put $X_{gp} = 1$ on each processor p assigned to group g.

Because the system is not partitionable, some processors remain unassigned. We define the *remaining required subset size* ω'_g *of group* g as

$$\omega'_g \triangleq \omega_g - \lfloor \omega_g \rfloor. \tag{5.12}$$

Clearly, $0 < \omega'_g < 1$. The remaining required subset size of a group specifies the fraction of the capacity of a single processor still to be assigned to group g.

Next, we assign the groups in an arbitrary sequence to the processors not yet allocated to a group. Normally, this means setting $X_{gp} = \omega'_g$, where g and p are the group and the processor currently being considered, respectively. However if the current processor does not have enough capacity left to serve a group g at (local) rate ω'_g, the group is split across this processor and the next one.

It is easily verified that the partitioning algorithm guarantees that for each group g, $|\Omega_g| \leq \lfloor \omega_g \rfloor + 2$, and that $|\Omega_g| = \omega_g$ when ω_g is integral.

Partitioning Example. As an example of the partitioning algorithm, consider a system with $P = 5$, $G = 6$, $r_1^{\mathrm{G}} P = 3/2$, $r_2^{\mathrm{G}} P = 1$, $r_3^{\mathrm{G}} P = 2/3$, $r_4^{\mathrm{G}} P = 2/3$, $r_5^{\mathrm{G}} P = 2/3$, and $r_6^{\mathrm{G}} P = 1/2$. Initially, we set $X_{gp} = 0$, $g = 1, \ldots, G$, $p = 1, \ldots, P$. Clearly, the system is partitionable, and therefore, in the first step of the algorithm, we find the following three subsets of groups: $\mathcal{G}_1 = \{2\}$ which get processor 1, $\mathcal{G}_2 = \{1, 6\}$ which gets processors 2 and 3, and $\mathcal{G}_3 = \{3, 4, 5\}$ which gets processors 4 and 5. Because $\mathcal{G}_1$ consists of only a single group, we set $X_{21} = 1$, and we are finished for group 2. In the second step of the algorithm we consider $\mathcal{G}_2$ and $\mathcal{G}_3$ in sequence. For $\mathcal{G}_2$, we assign processor 2 to group 1, since $\lfloor \omega_1 \rfloor = 1$. We find for the remaining required subset sizes for the groups in $\mathcal{G}_2$,

$$\begin{aligned} \omega'_1 &= 3/2 - 1 = 1/2, \\ \omega'_6 &= 1/2. \end{aligned}$$

The algorithm finishes for $\mathcal{G}_2$ by assigning processor 3 to groups 1 and 6 such that each gets half a processor, i.e., $X_{13} = 1/2$ and $X_{63} = 1/2$. For $\mathcal{G}_3$, we have $\lfloor \omega_g \rfloor < 1$, $g \in \mathcal{G}_3$, and we cannot assign entire processors. Therefore,

$$\begin{aligned} \omega'_3 &= 2/3, \\ \omega'_4 &= 2/3, \\ \omega'_5 &= 2/3. \end{aligned}$$

We assign group 3 to processor 4, and we set $X_{34} = 2/3$. However, the remaining capacity on processor 4 ($1 - 2/3 = 1/3$) does not suffice for group 4, since $\omega'_4 > 1/3$. Hence we split group 4 across processors 4 and 5, and we set $X_{44} = 1/3$, $X_{45} = 2/3 - 1/3 = 1/3$. Finally, we set $X_{55} = \omega'_5 = 2/3$, which is exactly the remaining capacity on processor 5.

The resulting partitioning matrix X becomes

$$X = \begin{pmatrix} 0 & 1 & 1/2 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 2/3 & 0 \\ 0 & 0 & 0 & 1/3 & 1/3 \\ 0 & 0 & 0 & 0 & 2/3 \\ 0 & 0 & 1/2 & 0 & 0 \end{pmatrix}. \tag{5.13}$$

Initial-placement and Migration Policies

In SVP, when job migration is not available, jobs of group g are only served by processors in Ω_g. When job migration *is* available, we allow jobs to be served by processors not in the group's subset, in order to eliminate capacity loss. We call such jobs *guest* jobs. By virtue of rule I3 below, guest jobs may run alongside non-guest jobs on the same processor, although SVP tries to avoid this as much as possible. If GPPS is the local scheduling policy, we put

$$w_{gp} = \begin{cases} X_{gp}, & g = 1, \ldots, G,\ p \in \Omega_g, \\ r_g^{\mathrm{G}}, & g = 1, \ldots, G,\ p \notin \Omega_g. \end{cases} \tag{5.14}$$

The initial-placement policy of SVP for an arriving job j of group g is as follows:

I1 Choose the idle processor $p \in \Omega_g$ for which X_{gp} is maximal.

I2 If no processor was found in rule I1 and if job migration is available, choose the processor $p \in \Omega_g$ serving a guest job for which X_{gp} is maximal. Migrate the guest job and put j on p. The decision as to where the guest job is to be migrated to is taken by running the initial-placement policy for it, without rule I2 in order to avoid multiple migrations.

I3 If no processor was found in rule I2 and if job migration is available, choose an idle processor $p \notin \Omega_g$ at random. Hence, job j becomes a guest job.

I4 If no processor was found in rule I3, choose the processor $p \in \Omega_g$ for which $(|J_{gp}(t)| + 1)/X_{gp}$ is minimal.

Note that a single job can be migrated upon arrival of another job. If job migration is available, the job-migration policy of SVP upon departure of a job j of group g from *an otherwise idle* processor q is as follows:

M1 First we try to move a guest job to q. Choose the processor p running a guest job alongside other (non-guest) jobs, and migrate the guest job from p to q, possibly making it a guest job again. If more than one such processor exists, choose the processor p for which $|J_{*p}(t)|$ is maximal.

M2 If no processor was found in rule M1, we try to do load sharing in one of the subsets Ω_g to which q belongs. Consider the pairs (g,p) of groups g with $q \in \Omega_g$ and processors p with $|J_{*p}(t)| \geq 2$ and $|J_{gp}(t)| \geq 1$. If no such pair exists, proceed with rule M3. Otherwise, select among these pairs the pair (h,p') for which $|J_{hp'}(t)|/X_{hp'} - 1/X_{hq}$ is maximal, and migrate a random job of group h from p' to q.

M3 If no processor was found in rule M2, we try to introduce a guest job on processor q. Consider the pairs (g,p) of groups g and processors p with $p \in \Omega_g$ and with $|J_{*p}(t)| \geq 2$. (This means we do not consider guest jobs for migration, because by virtue of rule M1, any guest jobs are running alone now.) If no such pair exists, proceed with rule M4. Otherwise, select among these pairs the pair (h,p') for which $|J_{hp'}(t)|/X_{hp'}$ is maximal, and migrate a random job of group h from p' to q.

M4 If no processor was found in rule M3, we try to remigrate a guest job belonging to some group g for which $q \in \Omega_g$. If there is more that one candidate, choose the one for which X_{gq} is maximal.

These initial-placement and job-migration policies have the following consequences.

1. If job migration is available, SVP maintains zero capacity loss upon arrivals. This is not the case in general in systems without job migration, because in that case, SVP never sends jobs of group g to a processor with $X_{gp} = 0$. Also, with or without job migration, SVP does not in general preserve system balance or group balance upon arrivals.

2. If job migration is available, SVP maintains zero capacity loss upon departures.

3. Because of the previous remarks, SVP as a whole preserves zero capacity loss if job migration is available.

4. If job migration is not available and if the system is fully partitionable (i.e., $r_g^G P$ is integral, $g = 1, \ldots, G$), the choice of PS or GPPS as the local scheduling policy is irrelevant, because jobs of different groups never meet on a processor. If the system is not (fully) partitionable, jobs of different groups can meet on certain processors (i.e., processors p for which $X_{gp} < 1$, $g = 1, \ldots, G$). If GPPS is the local scheduling policy, the shares obtained on such a processor match the X_{gp}'s, and we expect good compliance with the feasible group shares. If PS is the local scheduling policy, this match is absent in general, and compliance will be worse than with GPPS. This effect, however, diminishes as the sizes $|\Omega_g|$ of the subsets to which the

processor belongs increase. We therefore expect that in non-partitionable systems, the difference between SVP/PS and SVP/GPPS vanishes as P becomes large.

5. If job migration is not available and if the system is fully partitionable, SVP behaves identically to JSQ in each subset. This implies that Δ_g^{G}, $g = 1, \ldots, G$, with SVP is equal to Δ^{T} in a homogeneous $r_g^{\mathrm{G}}P$-processor system with total load $\rho_{g*}/r_g^{\mathrm{G}}$ and JSQ as the global scheduling policy.

6. At most one guest job can be present on a processor. A job can become a guest job by rule I3 or by rule M3.

7. If $G > 2$ and if job migration is available, a group can both have guest jobs in other subsets and host guest jobs at the same time. An example scenario is the following. For simplicity, we assume that the system is fully partitionable. However, this assumption is not essential. Suppose all processors except some $p \in \Omega_h$ serve exactly one non-guest job and that group g has an outstanding guest job on p. This state can be entered, for instance, when $r_g^{\mathrm{G}}P$ jobs of each group j, $j = 1, \ldots, G$, $j \neq h$, arrive at an empty system, followed by $r_h^{\mathrm{G}}P - 1$ jobs of group h and a single job of group g. Obviously, $g \neq h$. Suppose an additional job of some group i arrives, $i \neq g$, $i \neq h$. By rule I4, the job is put on one of the processors in Ω_i, and this processor now serves two jobs of group i. When a job on one of the processors in Ω_g terminates, that processor becomes idle, and one of the two group-i jobs sharing a processor is sent to the idle processor, by rule M3 (note that rules M1 and M2 do not apply). So, in the end, group g serves a guest job of group i and has an outstanding guest job on processor p. This situation can only arise when $G > 2$.

8. If $G > 2$ and job migration is available, a guest job can run alongside non-guest jobs on one processor. Starting at the end state of the scenario outlined in the previous item, suppose an additional job of group h arrives. Then, by rule I2, the guest job of group g must migrate and the newly arrived job is put on processor p. The former guest job of group g is put on the processor running the guest job of group i, by rule I4. Hence, guest jobs can run alongside non-guest jobs if $G > 2$ in systems with job migration. However, SVP puts much effort into canceling this undesired situation through rule M1.

9. If we interchange rules M3 and M4, we solve the problem of guest jobs running alongside non-guest jobs. However, this introduces capacity loss. We have not been able to change the policy in such a way that when $G > 2$ there is never capacity loss, and guest jobs never run alongside non-guest jobs.

We will study SVP's performance with and without job migration, and compare it to the other policies in Sections 5.6 and 5.7.

5.5.4 Dynamic Vertical Partitioning

As in SVP, the purpose of Dynamic Vertical Partitioning (DVP) is to assign entire processors to groups. However, contrary to SVP, in DVP, partitioning is done at runtime, depending on the current state. When a job of group g arrives and the corresponding

obtained group share is too large, DVP tries to decrease the subset of group g (the set of processors used to serve jobs of group g) by trying to free one of the processors for use by other groups (rule I2 below). The strategy is to drop the processor running the smallest number of jobs of group g as an eligible destination, hoping that the presence of group g on that processor comes to an end. If the obtained share is large enough (rule I3 below), the arriving job will be placed on a processor in the current subset. If the obtained share of a group g is too small (rule I4 below), the policy is to increase the subset by trying to assign the arriving job to a processor on which group g is not yet present. If job migration is available, DVP always prefers an idle processor over a busy one, in order to decrease the capacity loss (rule I1 below). If GPPS is the local scheduling policy, we put $w_{gp} = r_g^{\mathrm{G}}$, $g = 1, \ldots, G$, $p = 1, \ldots, P$.

The initial-placement policy of DVP for an arriving job j of group g is as follows:

I1 If job migration is available, choose an idle processor at random.

I2 If no processor was found in rule I1 and if $o_g^{\mathrm{G}}(t) \geq r_g^{\mathrm{G}} + \frac{1}{P}$, drop as a candidate the processor q (or one of them at random) with the smallest non-zero value of $|J_{gq}(t)|$. Among the remaining processors, choose the processor p with the smallest non-zero value of $|J_{gp}(t)|$. If more than one such processor exists, choose among them the processor p for which $|J_{*p}(t)|$ is minimal.

I3 If no processor was found in rule I1 and if $r_g^{\mathrm{G}} \leq o_g^{\mathrm{G}}(t) < r_g^{\mathrm{G}} + \frac{1}{P}$, choose the processor p with $|J_{gp}(t)| > 0$ for which $|J_{gp}(t)|$ is minimal. If more than one such processor exists, choose among them the processor p for which $|J_{*p}(t)|$ is minimal.

I4 If no processor was found in rule I1 and if $o_g^{\mathrm{G}}(t) < r_g^{\mathrm{G}}$, choose the processor p for which $|J_{gp}(t)|$ is minimal. If more than one such processor exists, choose among them the processor p for which $|J_{*p}(t)|$ is minimal.

There is no job migration upon arrivals. If job migration is available, the job-migration policy of DVP upon departure of a job j of group g from *an otherwise idle* processor q is as follows:

M1 Consider the pairs (g, p) of groups g and processors p with $o_g^{\mathrm{G}}(t) < f_g^{\mathrm{G}}(t)$, $|J_{gp}(t)| \geq 1$ and $|J_{*p}(t)| \geq 2$. If no such pair exists, proceed with rule M2. Choose the processor p' for which $|J_{*p'}(t)|$ is maximal, and select the group h present on processor p for which $|J_{hp'}(t)|$ is maximal. Migrate a random job of group h from p' to q.

M2 Same as rule M1 above, but for groups for which $f_g^{\mathrm{G}}(t) \leq o_g^{\mathrm{G}}(t) < r_g^{\mathrm{G}} + \frac{1}{P}$. If no groups were found, proceed with rule M3.

M3 Same as rule M1 above, but for groups for which $o_g^{\mathrm{G}}(t) \geq r_g^{\mathrm{G}} + \frac{1}{P}$.

These initial-placement and job-migration policies have the following consequences.

1. If job migration is available, DVP maintains zero capacity loss upon arrivals.
2. If job migration is available, DVP maintains zero capacity loss upon departures.
3. Because of the previous remarks, DVP as a whole preserves zero capacity loss if job migration is available.

4. If job migration is not available and the system is fully partitionable (i.e., $r_g^G P$ is integral, $g = 1, \ldots, G$), DVP is identical to SVP. In that case, just as with SVP, the performance of DVP is independent of whether PS or GPPS is the local scheduling policy.

We will study DVP's performance with and without job migration, and compare it to the other policies in Sections 5.6 and 5.7.

5.6 Performance Evaluation: Systems without Job Migration

In this section we evaluate the performance of the dynamic global policies in systems without job migration, with either PS or GPPS as the local scheduling policy. Clearly, a performance comparison between these two local policies is interesting, because PS allows no control whatsoever over the obtained shares of the jobs running on a processor, while GPSS is considered to be the perfect local share-scheduling policy. We also compare the global dynamic policies to the static policies HPRS and VPRS introduced in Chapter 4. We will study the policies in systems *with* job migration in Section 5.7.

Because the number of parameters is rather large, a study in which all parameters are varied over their allowed ranges is infeasible. We already chose to consider homogeneous systems only. Although it is possible to rewrite the policies for heterogeneous systems in a meaningful way, doing so would make our present study overly complicated. At this stage, our primary interest is in the fundamental differences between a typical load-balancing policy (JSQ), a horizontal-partitioning policy (HP), and two vertical-partitioning policies (SVP and DVP), and on the dependence on the local scheduling policy for good compliance with the feasible group shares.

Another simplification in this (and the next) section is that we only consider workloads consisting of non-permanent jobs. Clearly, such workloads are more challenging than workloads with permanent jobs, because the primary problems of share scheduling, capacity loss and deviations from the feasible group shares, arise with non-permanent jobs when the total load is not too low and not too high. With enough permanent jobs present in the system, three out of four global policies (HP, SVP, and DVP) perform excellent with GPPS as the local policy. For HP, we must have $N_g \geq P$, $g = 1, \ldots, G$; for SVP and DVP, we only need $N_g \geq r_g^G P$, $g = 1, \ldots, G$. For JSQ, the compliance with the feasible group shares strongly depends on the distribution of the permanent jobs among the processors.

Finally, we restrict ourselves to systems with $G = 2$. We believe our major results given below also hold in systems with $G > 2$ with only minor exceptions, but we defer a proper motivation for this belief until we have presented the results for $G = 2$.

Jobs arrive according to independent per-group Poisson processes with intensities λ_1 and λ_2. Job service times are independent random variables with an exponential distribution and mean service time (on a unit-capacity processor) $\mathbf{E}[S] = 1.0$. We define the group load ρ_{g*} of group g as

$$\rho_{g*} \triangleq \frac{\lambda_g \mathbf{E}[S]}{c}, \quad g = 1, \ldots, G, \tag{5.15}$$

and the total load ρ as

$$\rho \triangleq \rho_{1*} + \rho_{2*}. \tag{5.16}$$

We consider the following three cases of required group shares and workload distributions among the groups:

1. the required group shares are equal and the workload is distributed evenly among the groups;
2. the required group shares are equal, while the workload is *not* distributed evenly among the groups;
3. the required group shares are *not* equal, while the workload is distributed evenly among the groups.

The latter two cases have been studied in Chapter 3 as well. Unless otherwise noted, the results in the following sections are from simulation. We refer to Appendix A for more details on the simulation techniques used.

5.6.1 Case 1: Equal Required Group Shares and Equal Group Loads

In case 1, we have $r_1^{\mathrm{G}} = r_2^{\mathrm{G}} = 0.5$ and $\rho_{1*} = \rho_{2*}$, and therefore, $\Delta^{\mathrm{G}} = \Delta_1^{\mathrm{G}} = \Delta_2^{\mathrm{G}}$. Because the distribution of the total load ρ matches the required group shares, we expect little problems with all policy combinations. This very simple case allows us to study the fundamental differences between the global policies. We start our analysis with a two-processor system. Subsequently, we consider a ten-processor system. Finally, we study the effects of the number of processors.

For a two-processor system, we show Δ^{T} and Δ^{G} as functions of ρ in Figures 5.5, and 5.6, respectively. In the plots, we also show the results for HPRS and VPRS introduced in Chapter 4. We already know from Theorem 4.5 that in homogeneous systems, HPRS minimizes Δ^{T} in the class of random-splitting policies. In this specific case, the loads ρ_{*1} and ρ_{*2} on processors 1 and 2 are equal under both HPRS and VPRS, so the capacity loss Δ^{T} under HPRS is equal to that under VPRS. Also, irrespective of the local scheduling policy, the capacity loss Δ^{T} and the group-share deviation Δ^{G} under SVP and DVP are equal to those under VPRS, because the three policies send all jobs of group 1 to one processor, and all jobs of group 2 to the other. The plots for JSQ and HP are from simulations, for HPRS from (4.81) and (4.136), and for VPRS, SVP, and DVP from (4.81) and (4.142). With all global policies, the choice of the local policy has no effect on the capacity loss when $P = 2$. This is obvious for HPRS, VPRS, JSQ, SVP, and DVP. For HP, we found no differences with JSQ with respect to capacity loss in this system.

Figure 5.5 immediately reveals the problem of high capacity loss with SVP and DVP in systems without job migration. Both JSQ and HP yield a much lower capacity loss than HPRS, VPRS, SVP, and DVP.

With respect to compliance with the feasible group shares, we conclude from Figure 5.6 that all global policies except HPRS perform well. For VPRS, SVP, and DVP we have $\Delta^{\mathrm{G}} = 0$, irrespective of the local policy. JSQ and HP perform even better, and benefit somewhat from using GPPS over PS. More interesting is the fact that in combination

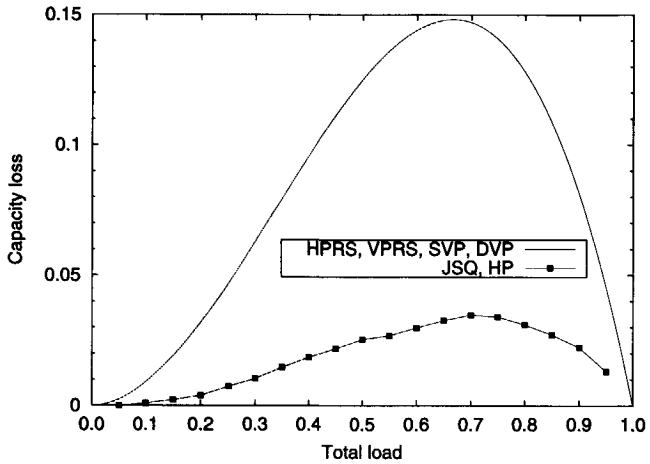


Figure 5.5: The capacity loss (Δ^{T}) in a homogeneous two-processor system without job migration and with two groups, as a function of the total load ρ ($r_1^{\mathrm{G}} = r_2^{\mathrm{G}} = 0.5$, $\rho_{1*} = \rho_{2*}$).

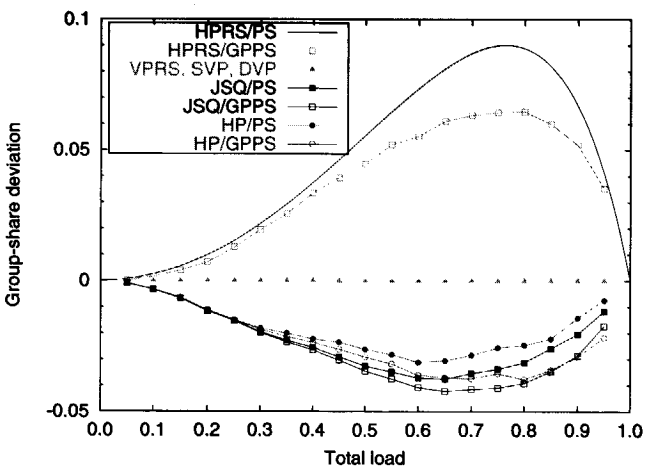


Figure 5.6: The group-share deviation (Δ^{G}) in a homogeneous two-processor system without job migration and with two groups, as a function of the total load ρ ($r_1^{\mathrm{G}} = r_2^{\mathrm{G}} = 0.5$, $\rho_{1*} = \rho_{2*}$).

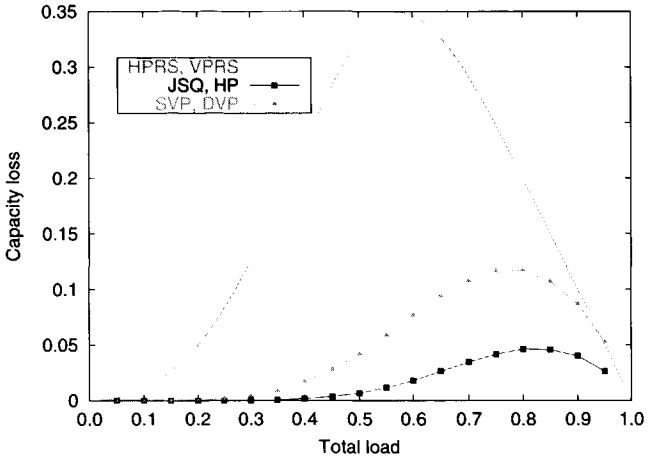


Figure 5.7: The capacity loss (Δ^{T}) in a homogeneous ten-processor system without job migration and with two groups, as a function of the total load ρ ($r_1^{\mathrm{G}} = r_2^{\mathrm{G}} = 0.5$, $\rho_{1*} = \rho_{2*}$).

with the same local policy, JSQ performs better than HP. Apparently, spreading all jobs over the processors as in JSQ is a better strategy than spreading the jobs of every group separately as in HP.

For a ten-processor system, we show Δ and Δ^{G} as functions of ρ in Figures 5.7 and 5.8, respectively. In the latter figure, we have left out the results for HPRS and VPRS, because Δ^{G} for these policies is much larger than that for the dynamic policies (as can be seen in Figure 4.17). As in the two-processor system, capacity loss with SVP and DVP is substantially higher than with JSQ and HP. However, as opposed to the two-processor case, in the ten-processor case the capacity loss for SVP and DVP is far lower than that for HPRS and VPRS. Within each subset serving jobs of one group, both SVP and DVP spread the jobs well among the processors. One can think of SVP and DVP as employing JSQ "in each subset". Again, there are no differences in capacity loss between JSQ and HP and between SVP and DVP, nor between PS and GPPS with any global policy.

With respect to minimizing Δ^{G} (see Figure 5.8), the performance of SVP and DVP is still worse than that of JSQ and HP. The explanation for this is that processors in a subset (Ω_1 or Ω_2) are sometimes idle, while the entire subset contains one or more processors serving at least two jobs. In other words, there is capacity loss *within* a subset. Because SVP and DVP cannot compensate for this by giving a group a larger share than required at other times, as JSQ and HP do, the latter two policies perform substantially better. Still, for SVP and DVP we have $\Delta^{\mathrm{G}} < 0.05$, which we still consider reasonable. As in the two-processor case, JSQ performs better than HP.

In order to study the validity of our results obtained thus far for different values of the number P of processors, in Figures 5.9 and 5.10 we show Δ^{T} and Δ^{G} as functions of P for a system without job migration with $\rho_{1*} = \rho_{2*} = 0.4$, and $r_1^{\mathrm{G}} = r_2^{\mathrm{G}} = 0.5$. (The lines between the points are added for readability.) From the results found with $P = 2$ and with $P = 10$, we conclude that a total load $\rho = 0.8$ presents a significant challenge

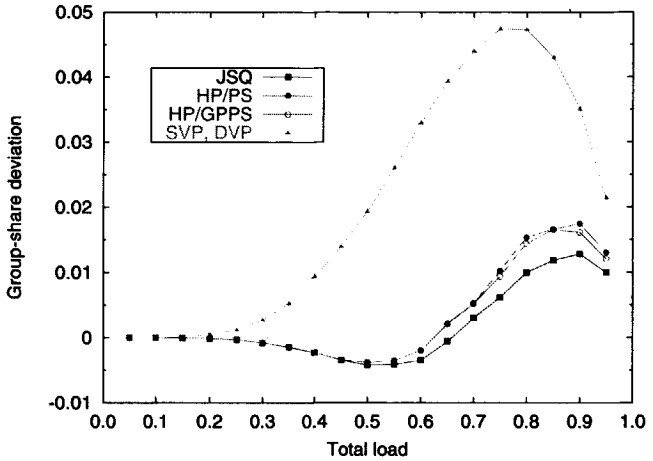


Figure 5.8: The group-share deviation (Δ^G) in a homogeneous ten-processor system without job migration and with two groups, as a function of the total load ρ ($r_1^G = r_2^G = 0.5$, $\rho_{1*} = \rho_{2*}$).

to all the policy combinations.

Again, both Δ^T and Δ^G are lower with the dynamic policies than with VPRS and HPRS (see Figure 4.17). For JSQ and HP, Δ^T is hardly affected by the number of processors, and for $P > 5$, decreases (slowly) in P. For P odd, the system is not (fully) partitionable, and the two groups share a single processor in SVP and DVP. This has a positive effect on Δ^T, because the processor being shared participates in the load-balancing efforts in two subsets. The effect also explains the difference in Δ^G between P even and P odd in Figure 5.10: in a non-partitionable system (P odd), a group sometimes obtains more than its required share (when the processor being shared is not used by the other group). This never happens in a partitionable system (P even). However, the positive effect diminishes as P increases. Remarkably, the same effect shows up with JSQ and HP, albeit with a smaller amplitude than with SVP and DVP.

An interesting question is what happens to Δ^T for large P in a system with JSQ as the global policy. In Figure 5.11, we show Δ^T for JSQ as a function of ρ for different values of P. Comparing this figure to Figure 4.7, we conclude that JSQ dramatically outperforms random splitting with respect to minimizing capacity loss. Moreover, increasing the number of processors has a positive effect on Δ^T, as opposed to the case in a random-splitting system. As ρ increases, Δ^T first remains very close to zero, then reaches a maximum, and finally drops to zero again. As P increases beyond $P = 7$, the maximum value of Δ^T decreases, and the value of ρ at which this maximum is obtained increases as well. For $P = 100$, Δ^T is practically zero for $\rho \leq 0.8$, then reaches its maximum value of 0.026 at $\rho = 0.9$. Because SVP and DVP behave as JSQ in a subset in homogeneous, fully partitionable systems, it is also clear that SVP and DVP outperform VPRS with respect to minimizing Δ_g^G for groups with $r_g^G P \geq 2$. Figure 5.11 is of fundamental importance and supports our claim that even without job migration, the dynamic policies outperform

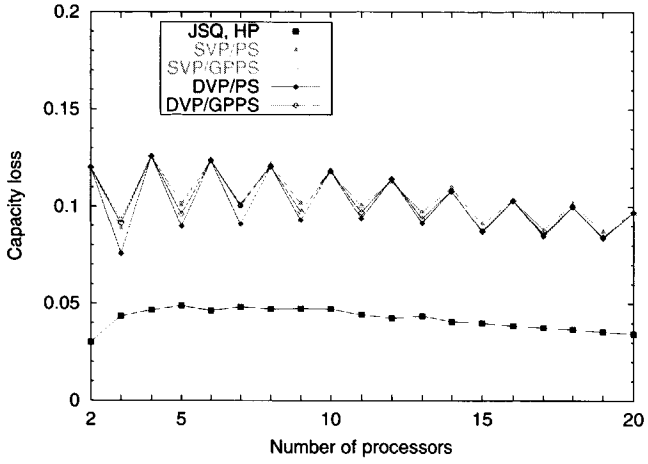


Figure 5.9: The capacity loss (Δ^{T}) in a homogeneous system without job migration and with two groups, as a function of the number P of processors ($r_1^{\mathrm{G}} = r_2^{\mathrm{G}} = 0.5$, $\rho_{1*} = \rho_{2*} = 0.4$).

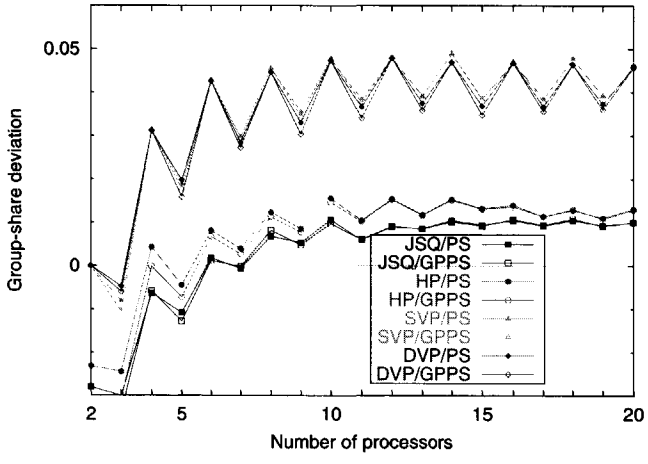


Figure 5.10: The group-share deviation (Δ^{G}) in a homogeneous system without job migration and with two groups, as a function of the number P of processors ($r_1^{\mathrm{G}} = r_2^{\mathrm{G}} = 0.5$, $\rho_{1*} = \rho_{2*} = 0.4$).

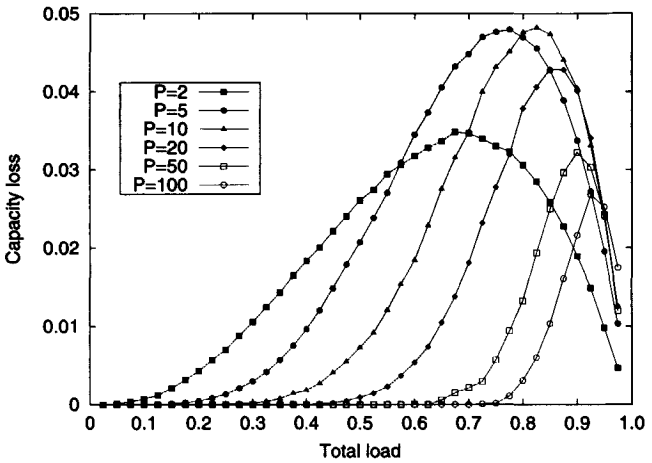


Figure 5.11: The capacity loss (Δ^{T}) in a homogeneous system without job migration and JSQ as the global scheduling policy as a function of the total load ρ for various values of the number P of processors.

the static random-splitting policies. Another conclusion is that for increasing values of P beyond $r_g^{\mathrm{G}} P = 7$, the group-share deviation Δ_g^{G} of group g will decrease under SVP and DVP.

5.6.2 Case 2: Equal Required Group Shares and Unequal Group Loads

In case 2, we put $P = 10$, $r_1^{\mathrm{G}} = r_2^{\mathrm{G}} = 0.5$, $\rho = 0.8$, and we change the distribution of the workload among the two groups, i.e., we vary ρ_{1*}/ρ between 0 and 0.5. So, the challenge for the share-scheduling policy is to equally distribute the system capacity under an uneven distribution of the workload among the groups. We show Δ^{T} and Δ^{G} in Figures 5.12 and 5.13, respectively.

From Figure 5.12, we argue that SVP and DVP cannot effectively restrain the capacity loss over a wide range of workload distributions over the groups. In this specific case, half the system capacity remains unused when $\rho_{1*} = 0$ and $\rho_{2*} = 0.8$, even though the number of jobs in the subset of group 2 grows unboundedly. Hence, half the system capacity is lost and therefore $\Delta^{\mathrm{T}} = 0.5$. In general, in fully partitionable systems, the worst-case distribution of the load ρ among the groups is when $\rho_{G*} = \rho$, causing queues of the (few) processors allocated to group G to grow without bounds when $\rho \geq 1/r_G^{\mathrm{G}}$, while the other processors remain idle (recall that $r_1^{\mathrm{G}} \geq \cdots \geq r_G^{\mathrm{G}}$). The worst-case capacity loss Δ^{T} for SVP and DVP is therefore $1 - r_G^{\mathrm{G}}$ ($= 0.5$ in this specific case). Clearly, both SVP and DVP perform unacceptably with respect to minimizing capacity loss in distributed systems, for which sharing of resources (including processors) is considered a prime objective. The capacity loss with JSQ and HP is much lower than that for SVP and DVP. Moreover, Δ^{T} does not depend on the distribution of the workload among the groups, which is obvious

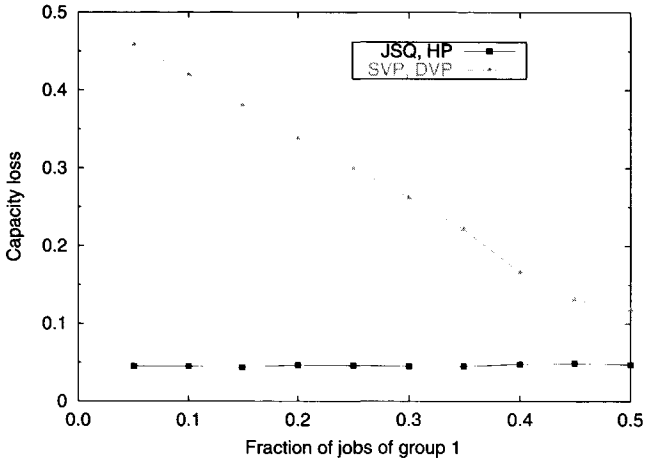


Figure 5.12: The capacity loss (Δ^{T}) in a homogeneous ten-processor system without job migration and with two groups, as a function of the fraction ρ_{1*}/ρ of jobs of group 1 ($r_1^{\mathrm{G}} = r_2^{\mathrm{G}} = 0.5$, $\rho = 0.8$).

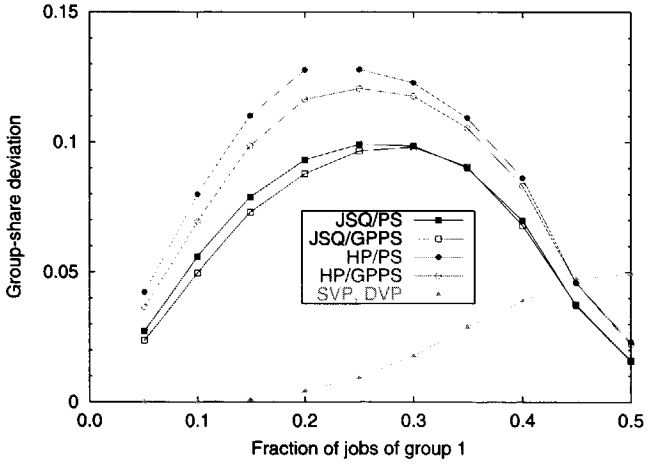


Figure 5.13: The group-share deviation Δ^{G} in a homogeneous ten-processor system without job migration and with two groups, as a function of the fraction ρ_{1*}/ρ of jobs of group 1 ($r_1^{\mathrm{G}} = r_2^{\mathrm{G}} = 0.5$, $\rho = 0.8$).

for JSQ.

With respect to Δ^G (see Figure 5.13), the situation is almost contrary to Δ^T. Clearly, JSQ and HP are unable to perform well under all workload distributions. Even with GPPS, the maximum value of Δ^G is approximately 0.12. As in case 1, the performance of HP is worse than that of JSQ, and neither policy benefits from the use of GPPS over PS. Unless the load is distributed almost equally among the two groups, SVP and DVP perform better than JSQ and HP, and their performance improves as the difference $\rho_{2*} - \rho_{1*}$ increases, in other words, as the load becomes more unbalanced between the two groups. In that case, the internal capacity loss mentioned earlier decreases. Also, because the system is fully partitionable, the choice of the local policy is irrelevant with SVP and DVP.

It is remarkable that, despite the low capacity loss for JSQ and HP in case 2, their performance with respect to Δ^G is so poor and almost independent of the local policy. This is because the two groups regularly meet on a single processor, which increases Δ^G, but hardly with three or more jobs, which explains why GPPS yields hardly any improvement over PS. When $r_1^G = r_2^G$, the effects of using GPPS as the local policy and HP as the global policy, become noticeable at very high loads only.

5.6.3 Case 3: Unequal Required Group Shares and Equal Group Loads

In case 3, we study the effect of having unequal required group shares while the group loads are equal ($\rho_{1*} = \rho_{2*} = 0.4$). The challenge is to provide different fractions of the system capacity to each group, while load is distributed evenly. We put $P = 10$. We show Δ^T and Δ^G in Figures 5.14 and 5.15, respectively.

Clearly, changing the required group shares does not affect the capacity loss for JSQ and HP. Both policies maintain a low capacity loss. The graphs of Δ^T for SVP and DVP have bizarre shapes, which is caused by the dependence of $|\Omega_1|$ and $|\Omega_2|$ on r_1^G. Clearly,

$$\begin{aligned} |\Omega_1| &= \lceil r_1^G P \rceil, \\ |\Omega_2| &= \lceil r_2^G P \rceil. \end{aligned}$$

So, when $n < r_g^G P < n+1$, $n = 0, \ldots, P-1$, n processors are assigned to group 1 and $P - n - 1$ processors are assigned to group 2, and the two groups share one processor, possibly at different fractions, as prescribed by the X_{gp}'s. When $r_g^G P$ is integral, say n, n processors are assigned to group 1 and $P - n$ processors are assigned to group 2. Concentrating on SVP first, when r_1^G increases only a little, and $|\Omega_1|$ and $|\Omega_2|$ do not change, the situation with respect to minimizing capacity loss does not change very much. Only when a processor is added to Ω_1, the capacity loss decreases. It is easy to verify that in that case, the total load ρ becomes more balanced among the processors.

For DVP/GPPS, the capacity loss is much lower than for SVP and DVP/PS when $r_1^G P$ is not integral, in other words, when the system is not fully partitionable. It is obvious from the definitions of SVP and DVP that when $r_1^G P$ is integral, the behavior of both policies is identical and does not depend on the local policy: at most $r_1^G P$ processors are serving jobs of group 1, at most $r_2^G P$ processors are serving jobs of group 2, and the two groups never meet on a single processor. Also, the initial-placement policies for SVP and DVP employ JSQ in a subset to spread the jobs among the processors.

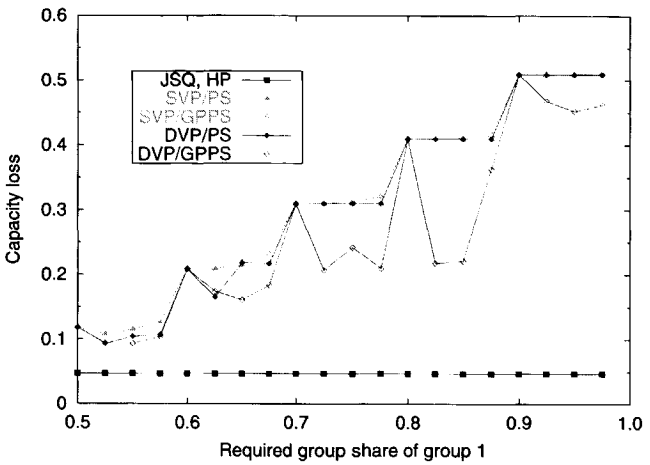


Figure 5.14: The capacity loss (Δ^{T}) in a homogeneous ten-processor system without job migration and with two groups, as a function of the required group share r_1^{G} of group 1 ($\rho_{1*} = \rho_{2*} = 0.4$).

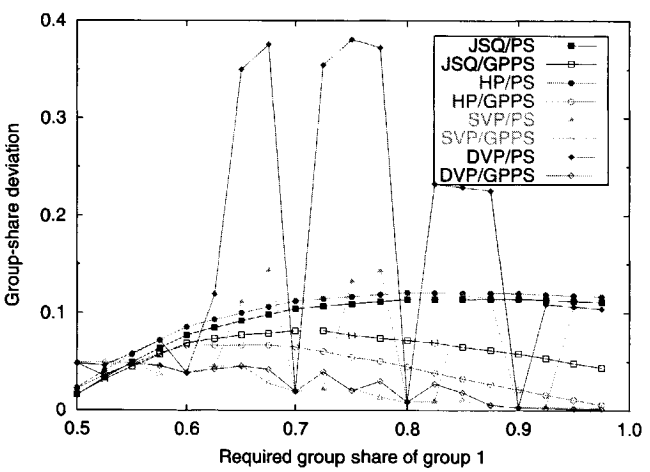


Figure 5.15: The group-share deviation (Δ^{G}) in a homogeneous ten-processor system without job migration and with two groups, as a function of the required group share r_1^{G} of group 1 ($\rho_{1*} = \rho_{2*} = 0.4$).

In order to explain the difference between DVP/PS and DVP/GPPS when $r_1^G P$ is not integral, consider the case where $r_1^G = 0.85$ and $r_2^G = 0.15$. Since $r_2^G P = 1.5$, as long as there is no interference with jobs of group 1, group 2 will be present on 2 processors only, and because $\rho_{2*} P = 4$, the queues on these processors will grow without bounds. For group 1, the load $\rho_{1*} = 0.4$, which is small compared to the 8 remaining processors allocated to group 1. However, from time to time, all processors are busy, 2 processors with large queues containing jobs of group 2 only, and 8 processors each serving one job of group 1. Suppose another job of group 1 arrives. Since $o_1^G(t) = 0.8 < r_1^G$, DVP, by virtue of its rule I3, will assign the job to one of the two processors serving jobs of group 2. When the local scheduling policy is PS, this job will get a very small obtained share because the queue length due to jobs of group 2 is so large, and the obtained share of group 2 hardly decreases. Therefore, jobs of group 2 that arrive subsequently will still be assigned to one of the two processors serving group 2 almost exclusively. Finally, the "alien" job(s) of group 1 will terminate, and conditions are back to normal. Obviously, capacity loss in this case is almost identical to that of SVP. However, when the local scheduling policy is GPPS, the "alien" job of group 1 has a significant effect on the obtained group share of group 2 because it gets a fraction 0.85 of the processor. Hence, $o_2^G(t)$ drops sharply, and subsequent arriving jobs of group 2 (they come in many), will penetrate the subset of processors originally serving jobs of group 1 only. However, such jobs also get only a fraction 0.15 of the capacity of a single processor. In the end, group 2 is present on many processors. Therefore, with GPPS, jobs of group 2 are better spread among the processors than with PS, and the capacity loss is significantly smaller.

The effect described above also explains to a large extent the problems SVP/PS and DVP/PS have with respect to compliance with the feasible group shares. From Figure 5.15, it is clear that these problems arise when the system is not fully partitionable and PS is the local policy. In that case, jobs of different groups cannot always be separated because some processors will be in more than one subset. In this specific case, for most values of r_1^G, the queue of the processor being shared with group 2 grows without bounds due to the high load of group 2, and jobs of group 1 receive hardly any service on this processor. With GPPS, the jobs present on such processors are still served at their proper rates (as prescribed by the X_{gp}'s). SVP/GPPS performs a little bit better than DVP/GPPS, but both policies perform adequately.

It is also clear from Figure 5.15 that JSQ and HP, whether in combination with PS or with GPPS, do not suffer from the fact that the system is not fully partitionable. Both policies benefit greatly from using GPPS over PS, unless the required group shares are close to or equal. When not, HP/GPPS performs better than JSQ/GPPS. This should not surprise us. HP spreads the jobs of each group among the processors, and therefore, benefits from using a good local policy more than JSQ.

5.6.4 Conclusions and Discussion

For systems without job migration, we state the following conclusions:

1. It is hard to achieve both low capacity loss and compliance with the feasible group shares. Aiming at minimization of capacity loss, as both JSQ and HP do, is not the best strategy for share scheduling, for which we consider compliance with the

feasible group shares more important. On the other hand, vertical partitioning reduces the group-share deviation, but increases capacity loss.

2. The dynamic policies perform substantially better than the static policies of Chapter 4, HPRS and VPRS, both with respect to minimization of the capacity loss, as to the minimization of the group-share deviation.

3. SVP and DVP are preferable to JSQ and HP with respect to compliance with the feasible group shares. JSQ performs somewhat better than HP for case-1 (unless the load is very high and GPPS is used as the local scheduling policy) and case-2 workloads. HP outperforms JSQ only under a case-3 workload (when a high level of discrimination is needed among the groups) and with GPPS as the local scheduling policy.

4. JSQ and HP are superior to SVP and DVP with respect to minimization of capacity loss.

5. In general, using GPPS instead of PS as the local scheduling policy improves the compliance with the feasible group shares. The improvements are much larger with JSQ and HP than with SVP and DVP, especially when the system is fully partitionable. In the latter case, the choice of the local scheduling policy is irrelevant with SVP and DVP. However, one can gain more by improving the global policy from JSQ or HP to SVP or DVP, than by improving the local policy from PS to GPPS. In other words, JSQ and HP cannot benefit enough from using GPPS instead of PS as the local policy.

6. As the number of processors becomes very large (say, 100 processors with just a few groups), a combination of simple global and local policies (such as Join Shortest Queue with Processor Sharing) suffices for low capacity loss and good compliance with the feasible group shares. One could argue that the average number of processors per group is likely to increase in the coming years, considering the decreasing cost per processor, and that therefore it makes little sense to invest in special-purpose share-scheduling policies. On the other hand, the number of independent tasks per job will increase too, given the advances in parallel programming. (Recall that in this thesis we do not consider multi-task jobs.)

Of course, we greatly simplified the general model outlined in Section 5.2 by considering systems with two groups only. However, we believe our results still hold for $G > 2$. For JSQ this is obvious, because JSQ does not consider to which group a job belongs. Therefore, the fundamental problems with JSQ with case-2 or case-3 loads still exist when $G > 2$. For HP, the problems with case-1 and case-2 workloads still exist when $G > 2$: unless the total load is very high, HP cannot benefit from using GPPS over PS. The behavior of Δ^G with SVP and DVP is largely dominated by the subset size $|\Omega_g|$, the group load ρ_{g*}, and whether or not the system is fully partitionable. It is easily verified that the excellent behavior of SVP/GPPS and DVP/GPPS, and the poor behavior of SVP/PS and DVP/PS in non-fully-partitionable systems, still hold when $G > 2$. At the same time, the problem of potentially high capacity loss with both policies remains.

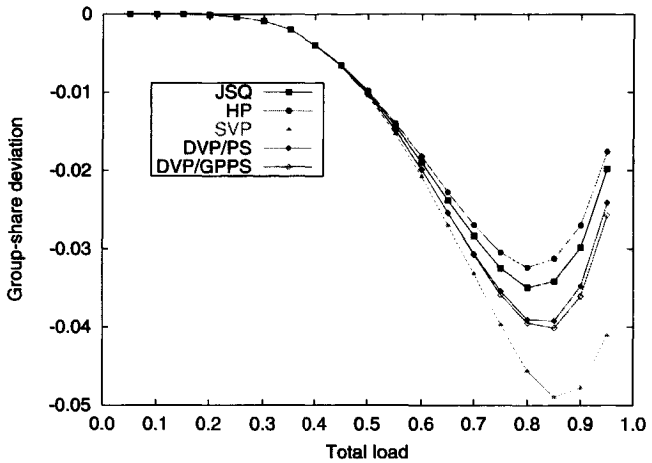


Figure 5.16: The group-share deviation (Δ^G) in a homogeneous ten-processor system with job migration and with two groups, as a function of the total load ρ ($r_1^G = r_2^G = 0.5$, $\rho_{1*} = \rho_{2*}$).

5.7 Performance Evaluation: Systems with Job Migration

In this section, we consider the three cases of Section 5.6 in turn for systems with job migration. Except for the availability of job migration in this section, the model is identical to that described in Section 5.6. Since we already argued that capacity loss is zero for each policy, we will not discuss it any further. However, we must stress that zero capacity loss is most beneficiary to SVP and DVP, for which high capacity loss is the biggest problem in systems without job migration. This makes SVP and DVP even more favorable than JSQ and HP. All results in this section are obtained through simulation, except for JSQ/PS, for which we used the analytical results of Section 5.5.1.

5.7.1 Case 1: Equal Required Group Shares and Equal Group Loads

In case 1, we have $r_1^G = r_2^G = 0.5$ and $\rho_{1*} = \rho_{2*}$. We put $P = 10$. In Figure 5.16, we show Δ^G as a function of the total load ρ. It is clear that in case 1, all policy combinations perform well when job migration is available. We have $\Delta^G < 0$ for every combination. However, the performance improvement for SVP and DVP is so large compared to the case without job migration (see Figure 5.8), that both policies now outperform JSQ and HP. The reason is that in SVP and DVP, a group can now effectively use the unused system capacity of other groups. In SVP, this happens through guest jobs; in DVP, through its initial-placement rule I1.

In Figure 5.17 we show Δ^G as a function of P with $\rho_{1*} = \rho_{2*} = 0.4$. When job

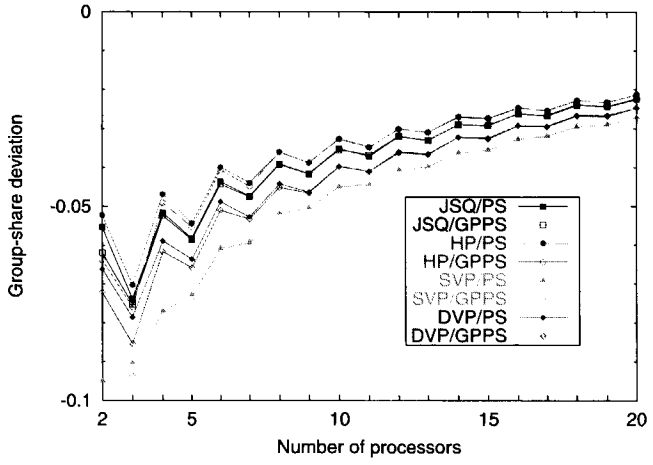


Figure 5.17: The group-share deviation (Δ^{G}) in a homogeneous system with job migration and with two groups, as a function of the number P of processors ($r_1^{\mathrm{G}} = r_2^{\mathrm{G}} = 0.5$, $\rho_{1*} = \rho_{2*} = 0.4$).

migration is available, the differences in performance between the four policies generally diminish as the number of processors increases, as does the dependency of the performance of SVP and DVP on whether or not the system is partitionable. It is obvious that the advantage of the availability of job migration diminishes as the number of processors increases.

For a case-1 workload, we conclude that the introduction of job migration is most beneficiary to SVP and DVP, both with respect to Δ^{T} and Δ^{G}. On the other hand, the distribution of the workload does not yet challenge our policies.

5.7.2 Case 2: Equal Required Group Shares and Unequal Group Loads

In case 2, we have $r_1^{\mathrm{G}} = r_2^{\mathrm{G}} = 0.5$ and $\rho = 0.8$. As in case 1, we put $P = 10$. In Figure 5.18, we show Δ^{G} as a function of the fraction ρ_{1*} of jobs of group 1. If we compare this figure to Figure 5.13, it is clear that with job migration, all policies perform better than without job migration. For JSQ and HP, the improvement is approximately 50%. The performance of HP is (still) disappointing, JSQ performs better. However, SVP and DVP perform excellent, and benefit from the use of job migration right where needed: when the total load is (almost) balanced among the groups. SVP performs better than DVP because of the immediate remigration of guest jobs.

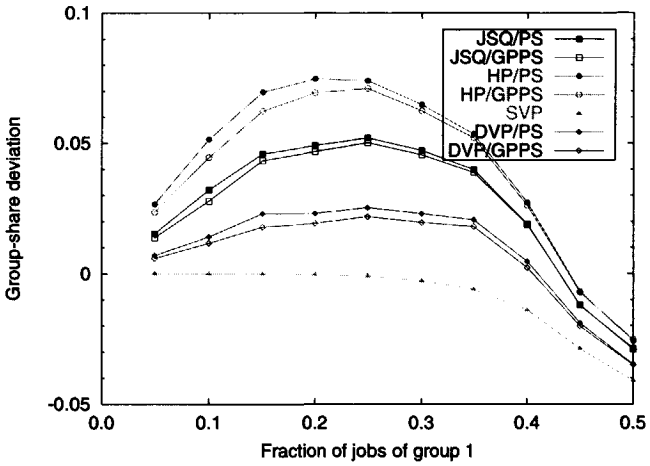


Figure 5.18: The group-share deviation (Δ^G) in a homogeneous ten-processor system with job migration and with two groups, as a function of the fraction ρ_{1*}/ρ of jobs of group 1 ($r_1^G = r_2^G = 0.5$, $\rho = 0.8$).

5.7.3 Case 3: Unequal Required Group Shares and Equal Group Loads

In case 3, we have $\rho_{1*} = \rho_{2*} = 0.4$. Again, we put $P = 10$. In Figure 5.19, we show Δ^G as a function of the required group share r_1^G of group 1. It is clear that for a case-3 workload, all policies except SVP benefit from using GPPS as the local scheduling policy instead of PS. The performance of SVP is relatively independent of the local scheduling policy, even if the system is not partitionable. Clearly, this is because of all global policies, SVP takes most effort to separate jobs of different groups. We conclude again that SVP outperforms all the other policies, and that DVP needs GPPS for compliance with the feasible group shares. Comparing Figures 5.19 and 5.15, we see that all policies perform better with job migration enabled but the improvement is especially notable for SVP/PS and DVP/PS.

5.7.4 Conclusions and Discussion

For systems with job migration, we state the following conclusions:

1. except for systems with many processors, the availability of job migration substantially improves the compliance with the feasible group shares compared to systems without job migration;

2. the capacity loss with all policies is zero, which takes away the principal disadvantage of SVP and DVP in systems without job migration;

3. SVP and DVP/GPPS (still) outperform JSQ and HP;

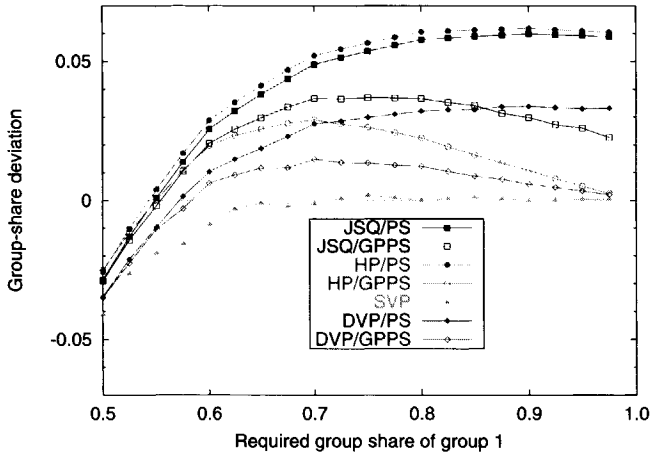


Figure 5.19: The group-share deviation (Δ^G) in a homogeneous ten-processor system with job migration and with two groups, as a function of the required group share r_1^G of group 1 ($\rho_{1*} = \rho_{2*} = 0.4$).

4. except for SVP, all policies benefit from using GPPS instead of PS as the local scheduling policy, especially under a case-3 workload (equal loads, unequal required group share);

5. the benefits of job migration diminish as the number of processors increases.

5.8 Conclusions

In this chapter, we proposed and evaluated four dynamic central global policies for share scheduling: JSQ, HP, SVP, and DVP, with PS and GPPS as the local scheduling policies. For all four policies, we defined an initial-placement policy and a migration policy. We only allowed a single migration upon an arrival and upon a departure of a job. By means of simulations, we evaluated the share-scheduling performance of these policies with respect to (1) their ability to provide groups with their feasible group shares, and (2) their ability to minimize the capacity loss.

Whether or not job migration is available, all the policy combinations outperform the static policies HPRS and VPRS of Chapter 4. When job migration is not available, only SVP/GPPS and DVP/GPPS perform adequately with respect to minimizing Δ^G under all workload conditions. When the system is fully partitionable, however, GPPS has no advantages over PS with either SVP or DVP. The major problem with SVP and DVP is the potentially high capacity loss. This makes them unsuitable for share scheduling when a low capacity loss is important and job migration is not available. For JSQ and HP, the capacity loss is much lower. So, in systems without job migration, maintaining a low capacity loss appears to be in conflict with compliance with the feasible group shares.

When job migration is available, the capacity loss is zero for all policies. The use of job migration improves the compliance with the feasible group shares for all policy combinations. Also, the simulation results indicate that HP does not in general perform better than JSQ. Finally, with job migration, SVP/GPPS performs best, closely followed by DVP/GPPS, and with the same local policy, SVP and DVP are undoubtedly superior to JSQ and HP.

Chapter 6

Dynamic Distributed Share Scheduling

6.1 Introduction

In this chapter we consider dynamic distributed policies for share scheduling. As was the case with the dynamic central policies studied in Chapter 5, the policies described in this chapter use information only available at runtime to make scheduling decisions. However, with distributed policies there is no central scheduling authority, i.e., no single processor making all the decisions. Instead, some or all of the processors cooperate in making the decisions. Therefore, an essential aspect of a distributed policy is the way in which information between the processors is exchanged.

There are a number of reasons to study distributed policies alongside central policies. First and most important, the use of a central processor to make all the scheduling decisions creates a single point of failure, which may not be acceptable for some applications. Second, the central processor can easily become a performance bottleneck, especially in large distributed systems. (In this thesis, though, we do not consider this to be a potential problem. Even with 1,000 processors, given the long service times of jobs, the number of scheduling decisions per second will be on the order of tens per second. This should not pose any problems for a contemporary workstation.) Third, in large systems, the exchange of information between a remote site and the central processor may take a significant amount of time. In a distributed policy, such a remote site can focus on sites in its vicinity. Finally, finding a global optimal decision may no longer be feasible in large systems.

The purpose of the present chapter is to study distributed versions of the policies introduced in Chapter 5, viz., JSQ, HP, SVP, and DVP, under more realistic system assumptions, and to find the dependence of the group-share deviation Δ^{G} and the capacity loss Δ^{T} on the amount and the quality of the information used. For this, we must extend our system and workload models of Chapter 5, which we do in Sections 6.2 and 6.3, respectively. The most important change in the system model is the explicit inclusion of a finite-capacity network, introducing delays in the information exchange and in job migrations. The only change in the workload model is that jobs no longer arrive at a single central scheduler, but instead at the individual processors.

In order to arrive at scheduling decisions, the processors exchange information across

the network through a mechanism called *probing*, and according to the global *information policy*. We describe this mechanism and two information policies in Section 6.4. The information gathered by the information policy is used to make the scheduling decisions. These decisions are taken according to the global *scheduling policy*. We describe the global scheduling policies in Section 6.5. Four out of five of these are straightforward adaptations of the global policies for the central model studied in Chapter 5. Although the philosophies behind the scheduling policies remain the same as those in Chapter 5, the policies must be adapted to using partial information on the system state. This may result in system states that were not anticipated in the original policies, simply because such states could not arise in the central model. In Section 6.5 we also introduce a fifth policy, called *Isolation*, which uses no information at all (it is static), and is only used as a reference policy in order to measure the effectiveness of the dynamic policies. In Section 6.6 we describe relevant literature. We are especially interested in results concerning the use of partial system information in dynamic scheduling policies and in the trade-off between gathering more state information at the cost of higher network delays. In Section 6.7 we evaluate the performance of the global scheduling and information policies. Finally, we conclude in Section 6.8.

Our main conclusion is that the policies of Chapter 5 still perform well under realistic assumptions on delays in information gathering and in job migrations. The observed network utilization due to information exchanges and job migrations stays well below 5% in a ten-processor system. We confirm the observation made in the literature [25, 84] that compared to using no information at all, using only a fraction of the available information on the system state (say, information on 30% of the processors) substantially improves the performance. Using more information yields only marginal benefits. However, the scalability of the information policies considered is a major point of concern. Therefore, for systems with more than say a hundred processors, one should consider splitting up the system in loosely-coupled subsystems, each with its own network segment, in order to keep the network traffic under control.

6.2 System Model

In our system model, there are P identical processors connected through a network. At every processor, jobs can arrive from outside the system. We first describe the processor and network models, then we outline the lifecycle of a job.

6.2.1 Processor Model

A processor consists of the following components:

- The *local server* with GPPS scheduling policy (see Section 3.4.3), serving jobs in its queue, the *server queue*.
- The *schedule queue* holding jobs that arrived at the processor but that still await initial placement.
- The *global scheduling policy* deciding on the placement of jobs in the schedule queue, and on the migration of jobs from or to the server queue.

- The *state information* holding complete and up-to-date information on the local state, and, possibly stale, information on zero or more of the other processors. The global scheduling policy uses the state information for its scheduling decisions.
- The *information policy* maintaining the state information. This policy sends out requests for information to other processors, processes the replies, and replies to requests from other processors. The policy also removes outdated information.

6.2.2 Network Model

We assume that a broadcast-type network with CSMA/CD (Carrier-Sense Multiple Access with Collision Detection), such as Ethernet (e.g. [62, 78]), connects the processors, and we model it as a single-server queue with Processor Sharing (see Section 3.4.1). We define the *network utilization* as the steady-state probability that the network server is busy, and we denote it by ρ_{net}. The server has unit capacity. Our network model captures the important feature of Ethernet that simultaneous traffic flows share the medium on an equal basis. It neglects collisions, but these can be accounted for in the network service time.

The network serves:

- job placements,
- job migrations,
- probe requests,
- probe replies, and
- job-migration requests.

We will describe these in turn. A *job placement* is the transfer of a job from the schedule queue of the source processor to the server queue of the destination processor. The job's service has not yet begun, but the job is taken into service immediately on the destination processor. A *job migration* is the transfer of a running job between two server queues. A *probe request* is a request for information from one processor to one or more other processors. We assume that a *multicasting* mechanism is available, such that multiple requests originating from the same processor can be combined into a single request on the network. This assumption is realistic with Ethernet. A *probe reply* is a reply to a probe request. The probe replies to a single multicast request are sent separately. Finally, a *job-migration request* is a request to the destination processor to transfer a job to the processor originating the request. Unless there are no eligible jobs to migrate, a job-migration request is always granted, and the transfer of the job starts immediately. Among the eligible jobs, one of them is selected at random. A migration request can be restricted to one group, in which case only the jobs of that group in the server queue are eligible for migration. If no group is specified, all jobs in the server queue are eligible for migration.

Except for job migrations, the size of the data transferred for each of these network clients is constant, and among the different clients, of the same order of magnitude. Hence, we assume that the time to serve them on the network is constant. We refer to

this time as the *probe service time*, and we denote it by $S^{\mathcal{P}}$. We estimate the size of each of these clients to be approximately 1,000 bytes, so on a 100 Mbps (Mega-bits per second) network it takes roughly 100 μs to process them. The mean job service time is 1.0 (as in the previous chapters), which corresponds to (at least) 100 seconds in real time. (Recall that we only consider compute-intensive workloads consisting of jobs with service times in the order of minutes to hours.) Therefore we put $S^{\mathcal{P}} = 10^{-4}/10^2 = 10^{-6}$.

It is obvious that job migrations require substantially larger service times than the other clients, and that assuming a constant network service time for them is not realistic. Unfortunately, little is known on the distribution of the size of the address spaces of compute-intensive jobs, but it is clear that this size is limited by the virtual address space of the computer system, so the observed variance should not be too high. We assume, rather arbitrarily, that the (network) service times for job migrations are exponentially distributed, and we denote their mean by $S^{\mathcal{M}}$, the *mean migration service time*. We put $S^{\mathcal{M}} = 0.01$. This corresponds to a 10^7-bytes average virtual-address space. (We modeled this after the virtual-address spaces needed by the simulation jobs described in this thesis.)

6.2.3 Lifecycle of a Job

For clarity, we summarize the lifecycle of a job:

- The job arrives at a processor and is put in the schedule queue.
- The job waits in the schedule queue for a placement decision by the processor's scheduling policy. While the job waits, it contributes to the total feasible share $f^{\mathrm{T}}(t)$ and to the feasible group share $f_g^{\mathrm{G}}(t)$, but its obtained job share is zero.
- The job is either placed on the processor it arrived on, in which case it joins the local server queue, or it is placed on a remote processor, in which case the job is transferred across the network (with network service time $S^{\mathcal{P}}$) to the destination processor, where it immediately joins the server queue.
- If the job is selected for migration, it is transferred across the network (with mean network service time $S^{\mathcal{M}}$) to the destination processor, where it immediately joins the server queue to resume execution. During the transfer, the job contributes to the applicable feasible shares, but does not obtain any share from the system.
- When the job terminates, it leaves the system.

6.3 Workload Model

Our workload model is identical to that of Chapter 5, with only one difference: each job arrives at one of the processors, instead of at a central scheduler. Following the notation introduced in Chapter 4, we denote the probability that an arbitrary job of group g arrives at processor p as π_{gp}. We consider two cases:

- *Horizontal-Partitioning (HP) arrival preference*: $\pi_{gp} = 1/P$,

- *Vertical-Partitioning (VP) arrival preference*: $\pi_{gp} = X_{gp}/(r_g^G P)$,

where (X_{gp}) is the partitioning matrix constructed according to the SVP partitioning algorithm described in Section 5.5.3. The motivation behind the arrival preferences is as follows. With the HP arrival preference, jobs have no processor preference, and the processors all receive identical traffic mixes. This situation arises, for instance, when jobs are submitted to the system from private workstations elsewhere. Spreading the jobs among the processors is then a meaningful strategy to balance the load. The VP arrival preference arises when each group "owns" a subsystem (the group's subset Ω_g), and when these subsystems are connected in order to share the computing power. This model resembles that of the CONDOR system [60], and we believe it is of more practical interest than the HP arrival preference. A fundamental concept in CONDOR is that each workstation has an owner with exclusive rights to that workstation when needed. Of course, there are many alternative possibilities of arrival preference, such as a single, dedicated processor for each group, or even for all groups together. However, the two cases outlined above are clearly the two fundamental ones.

With VP arrival preference, we actually combine the static VPRS policy (see Section 4.5.3) with one of the dynamic policies. From Chapter 4 we know that for not too many processors, VPRS yields a substantial performance benefit over HPRS. Therefore, an interesting question is whether with the dynamic policies, the VP arrival preference still improves the share-scheduling performance compared to the HP arrival preference.

6.4 Information Policies

In the system model, we make a strict separation between information gathering and scheduling. We consider two information policies: *on-demand probing* and *periodic probing*. Both policies update the local state information. However, the state information is also updated as a result of a scheduling decision taken by the scheduling policy.

6.4.1 On-Demand Probing

In *on-demand probing*, a processor only probes upon arrivals and departures at that processor. At each arrival and departure, the processor selects a random subset, called the *probe set*, of the other processors. The probe set has fixed size which we call the *probe limit* $L^{\mathcal{P}}$. An arriving job must wait in the schedule queue until all processors in the probe set have replied to the probe request, so the scheduling policy can take a placement decision. The same holds upon a departure for the migration component of the global scheduling policy.

6.4.2 Periodic Probing

In *periodic probing*, every processor probes periodically. The *mean probe period* is denoted by $\tau^{\mathcal{P}}$; this is the mean time between two probe requests from a processor. The probe period has a uniform distribution between $0.9 \times \tau^{\mathcal{P}}$ and $1.1 \times \tau^{\mathcal{P}}$. We found that the use of a constant probe period often leads to undesirable synchronization effects, for instance, causing multiple processors to assign a job to an idle processor in case of Join Shortest

Queue (see Section 6.5.2). Adding a stochastic element to the probe period reduces this effect. Just as in on-demand probing, the information policy probes a random subset of size $L^{\mathcal{P}}$.

We will compare on-demand probing and periodic probing in Section 6.7. Periodic probing has the advantage that jobs do not have to wait for information to become available. Also, the network load due to periodic probing is independent of the workload, and may be smaller than with on-demand probing with comparable performance. It has the disadvantage that stale information is used for most of the scheduling decisions.

6.4.3 The Choice of the Probe Set with On-Demand Probing

With on-demand probing, we consider two ways to construct the probe set upon arrivals:

- *HP probing preference:* each processor has equal probability to be in the probe set.
- *VP probing preference:* upon arrival of a group g, the processors in Ω_g are considered first. Each of them has probability proportional to X_{gp} to be included in the probe set. Only if $L^{\mathcal{P}} > |\Omega_g|$ ($|\Omega_g| - 1$ if the job arrived on a processor in Ω_g), we consider the processors not in Ω_g. The latter processors then have equal probabilities of being included into the probe set.

Note that upon departures the probing preference is always HP. This is because upon departures it is not clear which group should be preferred with VP probing preference. The same is true for periodic probing.

We expect that the choice of the probe set with on-demand probing has a significant impact on the performance—much more than the arrival preference, especially with low values of the probe limit. Most, although not all, of the policies described in the next section only consider processors in the probe set for initial placement. So, for low values of the probe limit and VP probing preference, most jobs will be placed onto a processor in the subset of the group to which the job belongs, and any policy will behave very similar to Static Vertical Partitioning. From Chapter 5 we know that SVP in general performs much better than JSQ, but is also more complicated. An interesting question is therefore to what extent the performance of JSQ can be improved with VP probing preference.

6.5 Global Scheduling Policies

In this section we extend the policies JSQ, HP, SVP, and DVP of Chapter 5 to use partial state information. We do not rename the policies, because the policies remain unchanged when complete and up-to-date system information is available. As in Chapter 5, each policy consists of an initial-placement component and a job-migration component. We refer to Section 5.5 for more information on the fundamental ideas behind the policies and on the rationales of the policy rules. In this section, we keep their descriptions concise. First, we introduce another policy, which uses no state information at all.

6.5.1 Isolation

The Isolation policy is static; it places a job for service on the processor on which the job arrives. There are no job migrations. Isolation amounts to HPRS (see Section 4.5.2) if

the arrival preference is HP, and to VPRS (see Section 4.5.3) if the arrival preference is VP. We use Isolation as a reference for the performance of the dynamic policies.

6.5.2 Join Shortest Queue

The Join Shortest Queue (JSQ) policy only considers the processors for which information is available for initial-placement and for job migration. Other than that, the policy description (including the setting of the group priorities w_{gp} on the servers) given in Section 5.5.1 applies, and will not be repeated here. For initial-placement, if the processor on which the job arrives is among the candidates with the smallest number of jobs in their queues, the policy prefers that processor over any others.

6.5.3 Horizontal Partitioning

For a description of the Horizontal Partitioning (HP) policy, we refer to Section 5.5.2. The changes in policy operation in the distributed system compared to that in the central system are analogous to the changes for JSQ described in Section 6.5.2.

6.5.4 Static Vertical Partitioning

Of the four central policies described in Section 5.5, the Static Vertical Partitioning (SVP) policy requires the most changes because the original policy depends very much on the fact that certain states cannot arise in the central model. The partitioning algorithm and the setting of the group weights w_{gp}, $g = 1, \ldots, G$, $p = 1, \ldots, P$ remain unchanged. Clearly, the partitioning algorithm uses only static information.

Rules I1 to I4 of the initial-placement policy remain unchanged, except that only processors in the probe set and the processor on which the job arrived are considered for initial placement. In rule I2, we must anticipate for the presence of more than one guest job on a processor. In that case, we pick a guest job at random for migration to its "own" subset. Just as in JSQ and HP, should a tie occur that includes the processor on which the job arrived, then this processor is preferred over the others.

However, there are cases in which the four rules of the original initial-placement policy do not suffice for a decision. This situation arises when no information is available on any of the processors in Ω_g (this implies the job arrived on a processor *not* in Ω_g). For this case, we must add a fifth rule to the initial-placement policy:

I5 If no processor was found in rule I4, choose a random processor $p \in \Omega_g$ with probability proportional to X_{gp}.

SVP is the only policy that puts jobs on processors for which it has no information available.

Some minor changes are required for the migration component of SVP. SVP only considers processors in the probe set as eligible sources for job migration. In rule M1, the set of eligible source processors consists of all such processors that run more than one job, among which at least one is a guest job. This is a change from the central version in which there was no need to anticipate the presence of multiple guest jobs on a single processor. The criterion for selection of the source processor remains unchanged, but should this

processor run multiple guest jobs, then we select one such job at random for migration. Rules M2 to M4 require no changes.

6.5.5 Dynamic Vertical Partitioning

The changes required for the Dynamic Vertical Partitioning (DVP) policy described in Section 5.5.4, are analogous to the changes to JSQ and HP described earlier. The only major change is that DVP is no longer able to calculate the obtained and feasible group shares. Instead, it must make estimates of these using the information available: the processors in the probe set and the processor on which a job arrived or departed. Since these calculations are rather straightforward, we will not discuss them in detail.

We introduced a predecessor of DVP as the "Nice" policy in [47]. The two policies share the idea of making an estimate of the current obtained group share of a group. However, the Nice policy considered only the two cases of a too low obtained group share and a too high one, and was found to exhibit oscillatory behavior due to this deficit. Another difference is that the Nice policy did not include a migration component.

6.6 Related Literature

In this section we describe some results from the literature on dynamic scheduling policies using partial state information.

Livny and Melman [61] consider a homogeneous distributed system with a broadcast communication system using CSMA/CD and compare three load-balancing policies. The communication system has a finite transmission capacity. Jobs arrive according to independent Poisson processes of equal rates, and job service times are exponentially distributed. Locally, the scheduling discipline is FCFS and jobs whose service has not yet started can be transferred to another processor. A scheduling policy consists of a *control law element*, which decides when to transfer a waiting job and whereto, and an *information policy element*, which decides how information is exchanged between the processors. Of particular importance in this study is the transmission dilemma: exchanging more information between the processors is likely to increase the quality of the decisions taken, but also increases the utilization of the communication channel which may cause high transfer delays and the exchange of obsolete information.

In the first policy, the *state broadcast algorithm (STB)*, each processor broadcasts every change in the state of its queue to the other processors. Each processor maintains an estimate of the current system state vector, and transfers a job to another processor according to a balance criterion. In the second policy, the *broadcast idle algorithm (BID)*, each processor broadcasts its status when it turns idle. In response, processors that have jobs waiting broadcast a reservation message after a state-dependent period (processors with many waiting jobs wait only a short period). In the third policy, the *poll when idle algorithm (PID)*, a processors starts polling other nodes (up to a maximum number) when it turns idle. If the processor probed has jobs waiting, one such job is transferred to the initiating processor, provided it is still idle.

The authors compare the three policies with simulation. As the number of processors increases, the performance of all policies improves up to a policy-specific point, beyond

which adding a processor actually worsens the performance due to the increased communication latencies. This point also depends on other parameters such as the ratio of the communication and processing speeds, but it is first reached with the STB policy, and, generally, the communication activity with STB is considerably higher than with the other two policies. For higher numbers of processors, the PID and BID policies perform better than STB. The authors conclude that both the control law and the information policy should depend on the expected transmission delays.

The articles by Wang and Morris [84] and Eager, Lazowska, and Zahorjan [25] both address the required level of complexity of a load-sharing policy. We will not discuss their articles in detail here, but refer to Sections 1.3.6 and 5.3 for more details. The interesting conclusion from both articles is that by using a small amount of system information, a substantial improvement of the performance can be obtained. The resulting performance (in terms of mean response time [25] and the Q-factor [84] defined in (1.3)) is close to what is theoretically achievable. Unlike in our model, the overhead of information exchange in [25, 84] is modeled as processing costs on the sources and servers, rather than as communication latency.

In another article [24], Eager, Lazowska, and Zahorjan compare source-initiative and server-initiative policies. Again, the overhead is modeled by processing costs only. If a job is transferred from one processor to another, this incurs a processing cost at the originating processor only; this cost is exponentially distributed with mean C and overlaps with the processing of the transferred job on the receiver. The source-initiative policy, *Sender*, is a combination of a threshold transfer policy and a threshold location policy (see Section 1.3.6). In the first server-initiative policy, *Receiver*, a server probes at most L^P (the probe limit) processors when a departing job leaves fewer jobs behind than some threshold T. If the probed processor has more than T jobs in its queue, and is not in the process of transferring a job, a job is tranferred to the server (and probing ends immediately). Clearly, Receiver is the dual policy of Sender. In the second server-initiative policy, *Reservation*, only newly arriving jobs can be transferred, in order to avoid the costs of job migration with local scheduling policies that take jobs into service immediately, such as Processor Sharing. The Reservation policy operates identically to Receiver, except that once an eligible source has been found, the sender "reserves" the next job to arrive at the source to be transferred immediately and irrevocably to the server. However, if another reservation is already pending at the source, the server must continue probing other processors.

The authors compare the performance of the three policies in a homogeneous system with Poisson arrival processes with equal rates at each processor, and service times are exponentially distributed with mean S. By assuming that the states of the processors' queues are stochastically independent, an approximation of the mean job response time is obtained, which is independent of the number of processors. In terms of the mean job response time, the Sender and Receiver policies perform substantially better than a system without load sharing. However, the performance of Reservation is disappointing. The Sender policy is preferable at low loads, where many servers are below threshold, whereas the Receiver policy is preferable at high loads, where many sources are above threshold. Both policies perform well relatively independent of the probe limit, and with low values of the threshold T ($T = 1$ or $T = 2$, for instance, will do in most practical cases). Also, values of C below $0.05S$ hardly degrade the mean response time, but the

latter increases rapidly when C increases beyond $0.25S$.

De Jongh [45] describes three global scheduling policies that use probing viz., Shortest, Distributed Horizontal Partitioning, and Nice, and compares their performance with either PS or GPPS as the local scheduling policy on all processors. In the system model there is no probe delay and job migration is not allowed, but other than that, the model is identical to that described in Section 6.2. Shortest assigns an arriving job to the processor probed with the smallest number of jobs in its queue. Distributed Horizontal Partitioning assigns an arriving job of group g to the processor probed such that the obtained share of group g increases most, but it always prefers a processor not serving jobs of group g over one that does. For the initial placement of a job of group g, Nice makes an estimate of the obtained group share of group g by probing $L^{\mathcal{P}}$ processors. If the (estimated) share is higher or equal than the required share, Nice places the job on the processor probed where group g already obtains the highest share, hoping to steal as little share as possible from the other groups. Otherwise, Nice places the job on the processor probed such that the obtained share of group g increases most. It was found that D-HP/GPPS and Nice/GPPS are the most promising combinations for share scheduling. However, the performance of Nice substantially degrades when the probe limit it smaller than half the number of processors, while D-HP is much less sensitive to the probe limit.

6.7 Performance Evaluation

In this section, we evaluate the performance of the distributed versions of the scheduling policies JSQ, HP, SVP, and DVP. All results are from simulations. Unless noted otherwise, we assume that

- probing is on-demand with HP probing preference;
- job service times are exponentially distributed, with mean 1.0 on a unit-capacity processor;
- $G = 2$, $N^{\mathrm{P}} = 0$, arrivals are Poisson at rates λ_1 and λ_2 with HP arrival preference, $\rho = 0.8$;
- $P = 10$, and $c_1 = \cdots = c_P = 1/P$ (system is homogeneous);
- $S^{\mathcal{P}} = 10^{-6}$, $S^{\mathcal{M}} = 10^{-2}$.

We already motivated most of these parameter values in Section 5.6; the values for $S^{\mathcal{P}}$ and $S^{\mathcal{M}}$ were calculated in Section 6.2.2.

We consider the following five cases for the distribution of the workload among the groups:

1. the required group shares are equal and the workload is distributed evenly among the groups: $r_1^{\mathrm{G}} = r_2^{\mathrm{G}} = 0.5$, and $\rho_{1*} = \rho_{*2} = 0.4$;
2. the required group shares are equal while the workload is not distributed evenly among the groups: $r_1^{\mathrm{G}} = r_2^{\mathrm{G}} = 0.5$, and $\rho_{1*} = 0.2$, $\rho_{2*} = 0.6$;

3. the required group shares are not equal, while the workload is distributed evenly among the groups: $r_1^G = 0.75$, $r_2^G = 0.25$, and $\rho_{1*} = \rho_{2*} = 0.4$;

4. the group loads are proportional to the required group shares: $r_1^G = 0.75$, $r_2^G = 0.25$, and $\rho_{1*} = 0.6, \rho_{2*} = 0.2$;

5. the group loads are inversely proportional to the required group shares: $r_1^G = 0.75$, $r_2^G = 0.25$, and $\rho_{1*} = 0.2, \rho_{2*} = 0.6$.

These five cases represent a wide range of distributions of the workload among the two groups. In a ten-processor system, the first two cases correspond to a fully partitionable system, the other cases to a non-partitionable system. The first three cases are identical to those studied in Sections 5.6 and 5.7. All simulation experiments described in the next sections were carried out for all five cases of workload distribution.

As in Chapter 5, the number of parameters in the model is very large. We therefore proceed using the following strategy. First, in Section 6.7.1 we study the effects of the probe limit $L^{\mathcal{P}}$ in a ten-processor system with on-demand probing. All other parameters of the model are fixed to their values given earlier. In Section 6.7.2 we study the effects of the mean probe period $\tau^{\mathcal{P}}$ with periodic probing, and we show that there is little benefit in using periodic probing over on-demand probing. Hence, in all remaining sections we restrict ourselves to on-demand probing. We fix $L^{\mathcal{P}}$ to a reasonable value, and we study the effects of the probe service time $S^{\mathcal{P}}$ and the mean migration service time $S^{\mathcal{M}}$ in Sections 6.7.3 and 6.7.4, respectively. In Section 6.7.5 we study the effect of the number of processors. This is a very important study, because so far, we have only studied ten-processor systems and have not yet been concerned with the scalability of the system, and the validity of our results in systems with many processors. In Section 6.7.6 we study the effect of the service-time distribution. It is well-known that an exponential distribution is not a good model for the service times of compute-intensive jobs. Finally, in Section 6.7.7, we study the effects of the arrival and probing preferences.

6.7.1 Effect of the Probe Limit

First, we study the effects of the probe limit $L^{\mathcal{P}}$. In Figures 6.1 and 6.2 we show Δ^T and Δ^G, respectively, for a ten-processor system with on-demand probing and a case-1 workload. From Chapter 5, we know that with a case-1 workload the differences between the policies are small. With any policy but SVP, when $L^{\mathcal{P}} = 0$ all jobs are served on the processor on which they arrive (which is equivalent to the Isolation policy) and there cannot be any job migrations. In that case, the system behaves as a random-splitting system with HPRS/GPPS. With SVP this is not true because of its rule I5 (see Section 6.5.4), which allows the placement of jobs on processors that have not been probed. When $L^{\mathcal{P}} = 9$, the policies amount to their respective central counterparts of Chapter 5.

From the figures it is clear that probing just a small number of processors significantly decreases Δ^T and Δ^G for all policies compared to Isolation ($L^{\mathcal{P}} = 0$). Both Δ^T and Δ^G are strictly decreasing functions of $L^{\mathcal{P}}$. For Δ^G, our most important performance measure, there is little gain in increasing the probe limit beyond $L^{\mathcal{P}} = 2$. For the capacity loss, there is still substantial benefit in increasing the probe limit from $L^{\mathcal{P}} = 2$ to $L^{\mathcal{P}} = 3$. From simulation results not shown here, we conclude that the previous statements still hold for the other four cases of workload distribution in the ten-processor system.

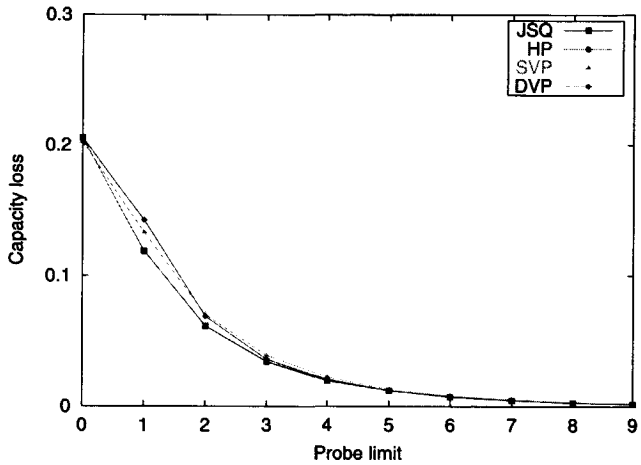


Figure 6.1: The capacity loss Δ^{T} for a case-1 workload as a function of the probe limit $L^{\mathcal{P}}$ $(P = 10)$.

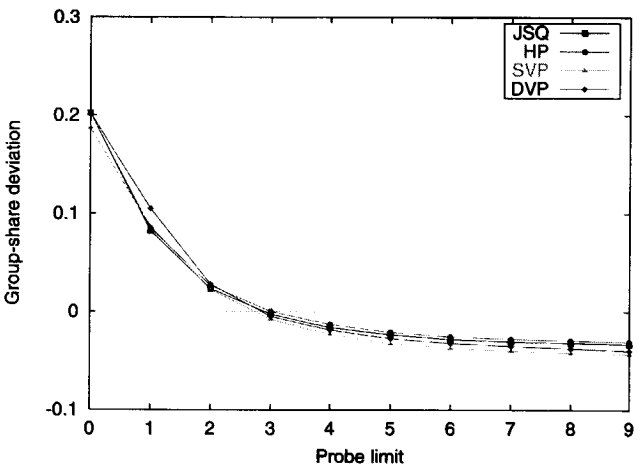


Figure 6.2: The group-share deviation Δ^{G} for a case-1 workload as a function of the probe limit $L^{\mathcal{P}}$ $(P = 10)$.

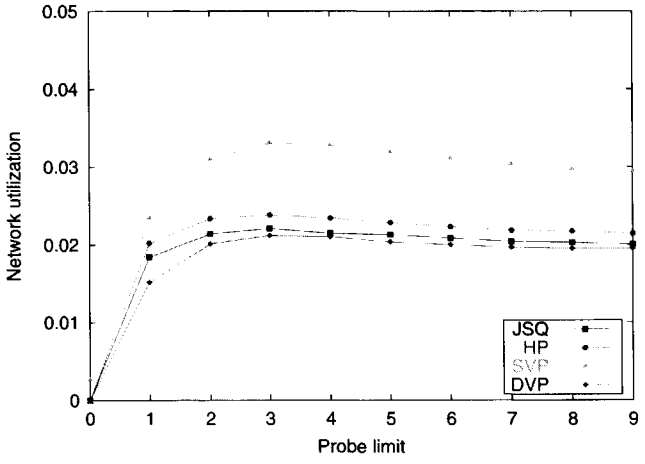


Figure 6.3: The network utilization ρ_{net} for a case-1 workload as a function of the probe limit $L^{\mathcal{P}}$ ($P = 10$).

For a case-1 workload distribution, we show the network utilization ρ_{net} as a function of the probe limit in Figure 6.3. Because $S^{\mathcal{M}} \gg S^{\mathcal{P}}$, the job migrations determine the network utilization. This figure confirms that for all policies, increasing the probe limit beyond $L^{\mathcal{P}} = 2$ only marginally improves the performance of the policies, because apparently the average number of job migrations hardly increases. With $L^{\mathcal{P}} = 2$, all policies almost obtain their maximal migration rate. However, the average number of migrations per unit of time is higher with SVP than with the other policies. This is mainly caused by the remigration of guest jobs, i.e., rule M4, because in many cases this results in the migration of a job running solely on a processor to an idle processor. So, SVP is somewhat more proactive with respect to job migration than the other policies.

Of course, we have only shown an isolated case out of many. As stated before, for the case with $P = 10$, we have conducted simulation experiments with all five cases of workload. In all cases, the performance of the policies with respect to Δ^{G} improves sharply between $L^{\mathcal{P}} = 0$ and $L^{\mathcal{P}} = 2$, and then stays rather flat between $L^{\mathcal{P}} = 2$ and $L^{\mathcal{P}} = 9$. This is an important result, and we will see in the next sections that it is valid under many conditions. The question as to how this is affected by changing the number of processors will be studied in Section 6.7.5. In the next sections, we will put $L^{\mathcal{P}} = 3$, unless noted otherwise. This ensures that both Δ^{G} and Δ^{T} will have reached their optimal values for practical purposes.

6.7.2 Effect of Periodic Probing

An important question is how periodic probing compares to on-demand probing. In Figure 6.4 we show Δ^{G} as a function of $\tau^{\mathcal{P}}$ in a ten-processor system with periodic probing ($L^{\mathcal{P}} = 3$) and a case-1 workload. As expected, Δ^{G} is an increasing function of $\tau^{\mathcal{P}}$. With $\tau^{\mathcal{P}} = 10$, the group-share deviation is almost as high as with the case $L^{\mathcal{P}} = 0$

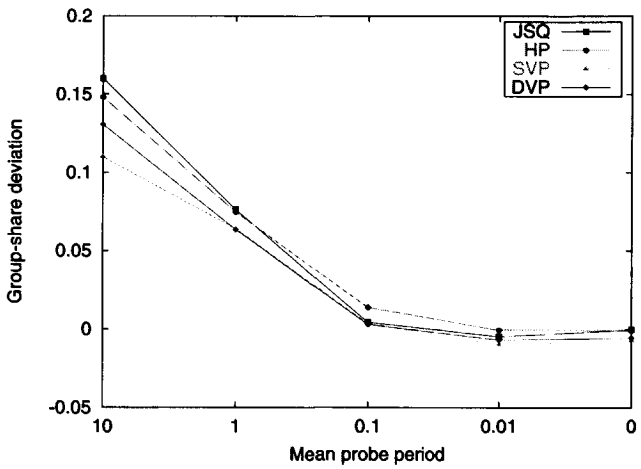


Figure 6.4: The group-share deviation Δ^{G} for a case-1 workload as a function of the mean probe period $\tau^{\mathcal{P}}$ ($P = 10$, $L^{\mathcal{P}} = 3$).

and on-demand probing, as can be seen from Figure 6.2. Clearly, there is no difference between on-demand probing and periodic probing when $L^{\mathcal{P}} = 0$. This implies that with periodic probing with $\tau^{\mathcal{P}} = 10$, the performance is comparable to that without probing at all, i.e., the static case.

It is interesting to compare the performances of on-demand probing and periodic probing under equal network loads and equal probe limits. From Figure 6.2, we find that when $L^{\mathcal{P}} = 3$, $\Delta^{\mathrm{G}} \approx 0$ with on-demand probing, irrespective of the global policy. Also, because $\rho = 0.8$, the mean job service time is unity, and $P = 10$, we have $\lambda = 8$, cf. (5.15) and (5.16). As long as the system is stable, this is also the average number of departures per unit of time. So the average number of probe *requests* per unit of time with on-demand probing is 16. With periodic probing, this number is simply $P/\tau^{\mathcal{P}}$, so the same number of probe requests is issued with periodic probing if $\tau^{\mathcal{P}} = P/16 = 0.625$. The interesting range for $\tau^{\mathcal{P}}$ is therefore between 0 and 1, which we show in greater detail in Figure 6.5.

From the figure it is clear that between $\tau^{\mathcal{P}} = 0.1$ and $\tau^{\mathcal{P}} = 1$, Δ^{G} increases substantially. With $\tau^{\mathcal{P}} = 0.6$, we have $\Delta^{\mathrm{G}} \approx 0.05$. So, with the same average number of probe requests per unit of time, on-demand probing ($\Delta^{\mathrm{G}} \approx 0$) clearly outperforms periodic probing ($\Delta^{\mathrm{G}} \approx 0.05$). It goes without saying that this depends on the probe service time $S^{\mathcal{P}}$. With its present value ($S^{\mathcal{P}} = 10^{-6}$), on-demand probing only causes small delays before jobs get scheduled, and therefore, these delays have a negligible effect on the group-share deviation. The effects of using stale information with periodic probing are far worse. Intuitively, this was to be expected: with on-demand probing the information policy only probes when necessary, while with periodic probing, many of the probe requests are unnecessary. In the remainder of this chapter we will restrict ourselves to on-demand probing.

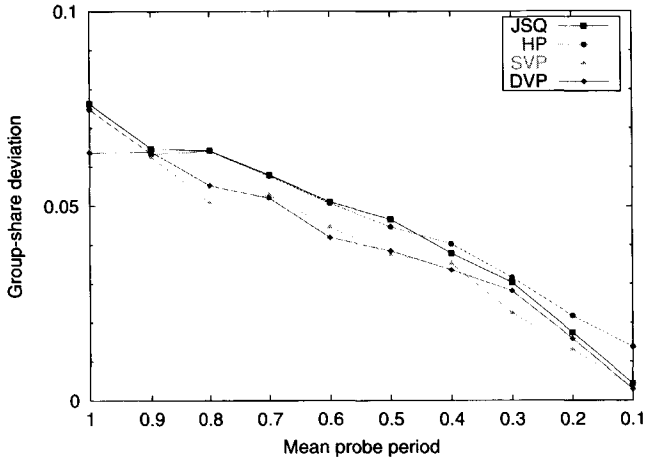


Figure 6.5: The group-share deviation Δ^{G} for a case-1 workload as a function of the mean probe period $\tau^{\mathcal{P}}$ ($P = 10$, $L^{\mathcal{P}} = 3$). This is a detail picture of Figure 6.4 on a linear scale.

6.7.3 Effect of the Probe Service Time

Although the probe service time $S^{\mathcal{P}}$ is very small, several orders of magnitude smaller than the mean migration service time $S^{\mathcal{M}}$, it is important to study its effects. This will provide insight into the margins we have should the probe delays increase (perhaps due to load other than probes and migrations on the network). In Figure 6.6, we show ρ_{net} as a function of $S^{\mathcal{P}}$ for a case-1 workload with $P = 10$. We have fixed the probe limit $L^{\mathcal{P}} = 3$, and we use on-demand probing.

The probes have hardly any effect on the network utilization for several orders of magnitude of $S^{\mathcal{P}}$, certainly not as long as $S^{\mathcal{P}} \leq 10^{-4}$. The network load is only determined by the migrations, not by the probes. Although we have not shown the figures, the performance measures Δ^{T} and Δ^{G} are unaffected by $S^{\mathcal{P}}$.

6.7.4 Effect of the Migration Service Time

Unlike the probe service time, the mean migration service time is expected to have a significant impact on the share-scheduling performance in a distributed system. In Figure 6.7 we show Δ^{G} for a case-1 workload as a function of $S^{\mathcal{M}}$ in a system with $P = 10$. We show ρ_{net} in Figure 6.8.

The group-share deviation Δ^{G} increases as $S^{\mathcal{M}}$ increases, albeit gradually. When $S^{\mathcal{M}} = 0.01$, the normal value, Δ^{G} is only slightly larger than with $S^{\mathcal{M}} = 0$. However, the effect of increasing $S^{\mathcal{M}}$ on the network utilization is significant. When $S^{\mathcal{M}} = 0.1$, the network load due to migrations is between 25% and 35%. We believe this should remain below 10% in practical situations, in order not to disturb the other network traffic. We conclude that the mean migration service time must be a critical factor in the design of a migration policy. In systems with high migration service times due to a slow network,

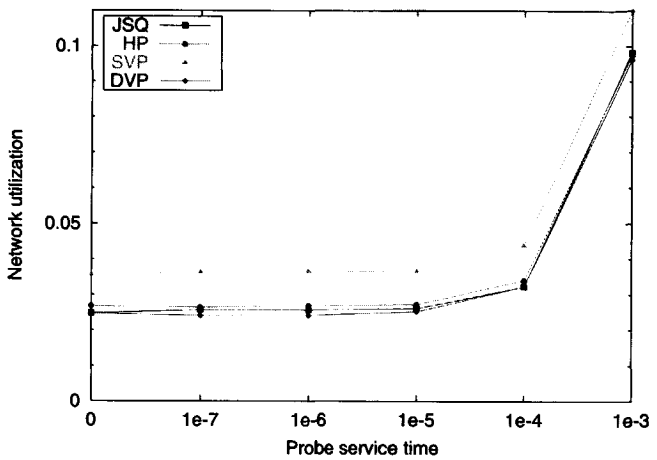


Figure 6.6: The network utilization ρ_{net} for a case-1 workload as a function of the probe service time $S^{\mathcal{P}}$ ($P = 10$, $L^{\mathcal{P}} = 3$).

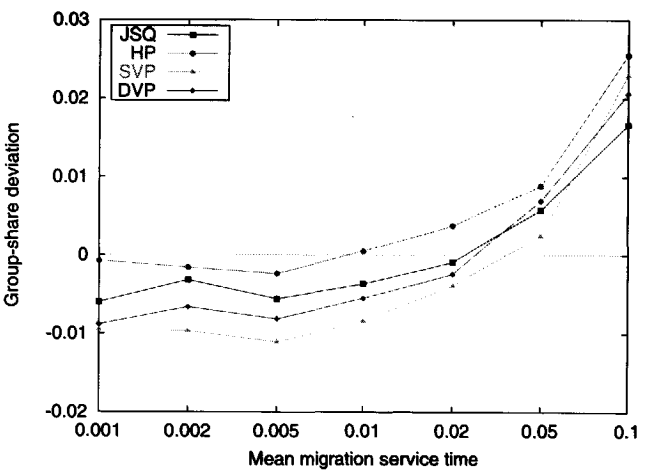


Figure 6.7: The group-share deviation Δ^{G} for a case-1 workload as a function of the mean migration service time $S^{\mathcal{M}}$ ($P = 10$, $L^{\mathcal{P}} = 3$).

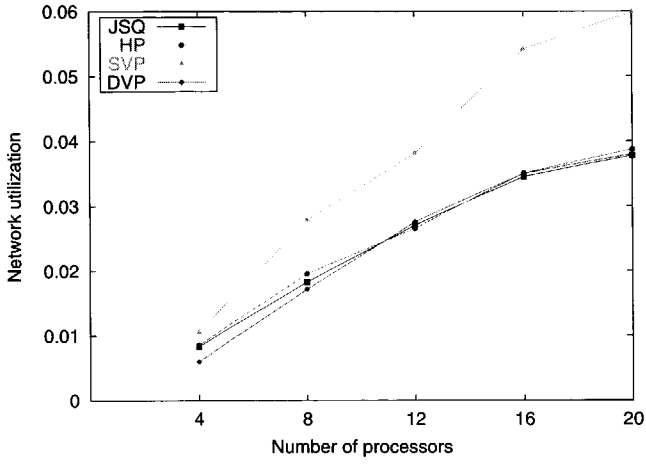


Figure 6.11: The network utilization ρ_{net} for a case-1 workload as a function of the number P of processors ($L^{\mathcal{P}} = 0.25 \times P$, see text for the connections between the points).

Obviously, $L^{\mathcal{P}} = 0.25 \times P$ is a better choice for the probe limit than $L^{\mathcal{P}} = 3$. We conclude that in order to keep up with an increasing number of processors, the information policy must increase the probe limit. Whether or not $L^{\mathcal{P}}$ should be linearly proportional to P is a subject for future study. At the present time, our simulation environment does not allow an extensive study of systems consisting of more than 20 processors with acceptable accuracy.

As to our second question: the network utilization increases when the number of processors increases, as is clear from Figure 6.11. This is a major point of concern for the scalability of the system. From our simulation results, we cannot decisively conclude whether or not the the network utilization flattens out as the number of processors increases.

6.7.6 Effect of the Service-Time Distribution

So far, we assumed that all job service times are independent, exponentially distributed random variables, all with mean 1.0. There are two major concerns regarding the validity of this model. First, the model used thus far does not include cases in which the mean job service times are unequal among the groups. Second, it is well-known that an exponential distribution is a poor approximation for the service-time distribution of "CPU hogs".

As for the first case, we consider a model in which the job service times are still exponentially distributed, but with different means among the groups. (So the job service time among all groups has a G-stage hyperexponential distribution.) Since $G = 2$, we define

$$\alpha \triangleq \frac{\mathbf{E}[S_1]}{\mathbf{E}[S_2]}, \tag{6.1}$$

where S_1, S_2 are the mean service times of jobs of groups 1 and 2, respectively. In Figure

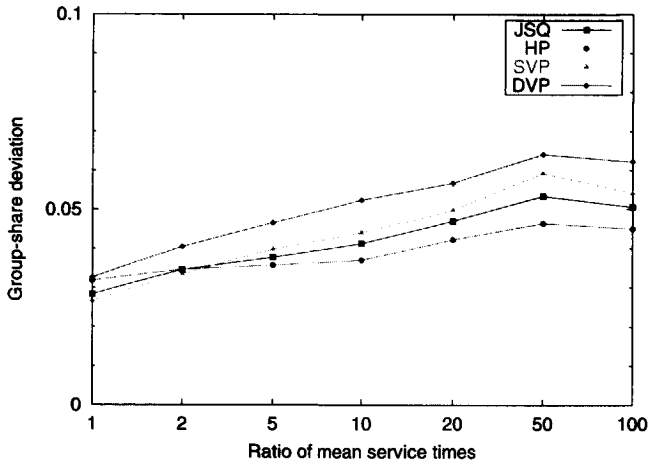


Figure 6.12: The group-share deviation Δ^G for a case-1 workload as a function of the ratio α of service time means ($P = 10$, $L^{\mathcal{P}} = 3$).

6.12, we show Δ^G as a function of α for a case-1 workload in a system with $P = 10$ and $L^{\mathcal{P}} = 3$. Note that the group loads ρ_{1*} and ρ_{2*} do not change as α changes, i.e., $\rho_{1*} = \rho_{2*} = 0.4$.

From the figure we conclude that the policies have sufficient robustness against large differences in service-time requirements between the groups. Clearly, the use of GPPS as the local scheduling discipline and the availability of job migration help to minimize the cases of unfairness that arise. Simulation experiments with the other four distributions of the workload among the groups confirm this statement. Noteworthy is that the group with the smallest mean job service time has the highest group-share deviation, in this case group 2. This is because the jobs of group 2 arrive in greater numbers per unit of time, and so the average feasible share of this group is higher than that of the other group. Also note that HP performs better than the other policies for this case of workload distribution and high values of α. This confirms the observation made in Chapter 5 that HP only performs well when there are large differences in workload characteristics among the groups. In other cases, it is inferior to JSQ.

For the second case, Leland and Ott [59] (see also Section 5.3) show from measurements that the distribution of the job service time has a heavier tail than the exponential distribution; they suggest a so-called Pareto distribution:

$$F(t) = 1 - a \times t^b, \; t \geq a^{-1/b}, \tag{6.2}$$

with $a > 0$ and $1.05 < b < 1.25$. For the suggested values of b, the distribution has infinite variance, but finite mean. There are many other cases in which heavy-tail distributions (such as Pareto and Weibull) better describe service-time or interarrival time processes. For an overview, see Feldman and Whitt [31] and the references cited therein. Borst et al. [6] point out the advantages of using the GPPS scheduling discipline as a means

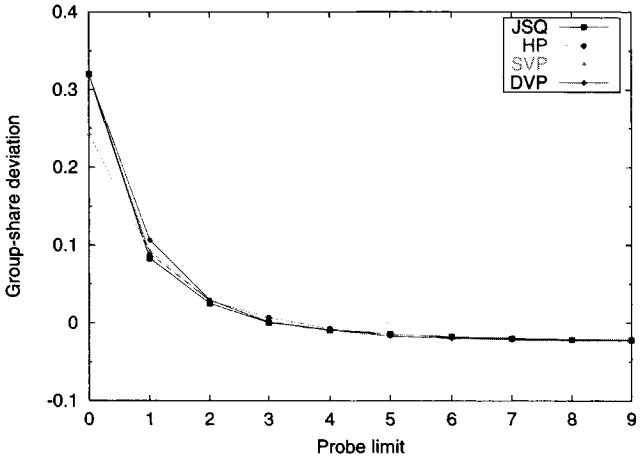


Figure 6.13: The group-share deviation Δ^{G} for a case-1 workload as a function of the probe limit $L^{\mathcal{P}}$ with Pareto-distributed job service times ($P = 10$, $\mathbf{E}[S] = 1.0$, $b = 1.25$).

of providing fairness among different groups in case service times have a heavy-tailed distribution.

In Figure 6.13, we show Δ^{G} as a function of $L^{\mathcal{P}}$ for a case-1 workload in a ten-processor system for Pareto-distributed service times ($b = 1.25$). If we compare the figure to Figure 6.2, we find that the performance degradation compared to the case with exponentially distributed service times is only significant for $L^{\mathcal{P}} = 0$, so the static case. The resulting improvement in group-share deviation between $L^{\mathcal{P}} = 0$ and $L^{\mathcal{P}} = 1$ is much more dramatic with the Pareto distribution. We believe the robustness against heavy-tailed service-time distributions is due to the use of job migration and of the GPPS scheduling discipline.

6.7.7 Effect of the Arrival and Probing Preferences

In this final section, we study the effects of the arrival and probing preferences. We study them simultaneously because they both affect the set of processors for which the global scheduler has state information. For all policies except SVP, the probing preference also determines the set of eligible processors for initial placement. In the figures, we will use the notation *arrival preference / probing preference / global policy*, for instance, HP/VP/JSQ.

We have conducted simulation experiments for the ten-processor system with all five cases of workload and all four combinations of arrival and probing preferences. Our main conclusion is that although the arrival and probing preferences can have a significant impact on Δ^{T} and Δ^{G}, none of the combinations clearly outperforms the others in all cases. We will only analyze JSQ, for which we show Δ^{G} as a function of $L^{\mathcal{P}}$ for a case-1 ($r_1^{\mathrm{G}} = r_2^{\mathrm{G}} = 0.5$, $\rho_{1*} = \rho_{2*} = 0.4$) and a case-2 workload ($r_1^{\mathrm{G}} = r_2^{\mathrm{G}} = 0.5$, $\rho_{1*} = 0.2$, $\rho_{2*} = 0.6$), in Figures 6.14 and Figure 6.15, respectively, for a ten-processor system. The effects described below are typical for the other policies as well.

From Figure 6.14, we find that although for the case-1 workload with $L^{\mathcal{P}} = 1$, the

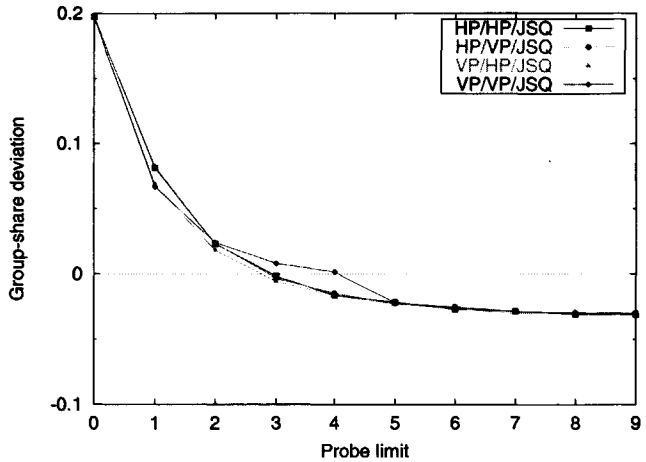


Figure 6.14: The group-share deviation Δ^G for JSQ under a case-1 workload as a function of the probe limit $L^{\mathcal{P}}$ for different arrival and probing preferences ($P = 10$).

combination of VP arrival and probing preferences yields a small reduction in Δ^G, it deteriorates the performance when $L^{\mathcal{P}} = 3$ or $L^{\mathcal{P}} = 4$. With VP arrival and probing preferences and $L^{\mathcal{P}} \leq 4$, jobs are always placed on a processor in the subset of the group the job belongs to, which leaves much of the system capacity unused. If either the arrival or the probing preference (or both) is HP, jobs are better spread among the processors, and because the group loads are proportional to the required group shares, the net result is a reduction in Δ^G. This effect is comparable to the better performance of JSQ compared to SVP under a case-1 workload, as was already pointed out in Chapter 5.

With a case-2 workload (Figure 6.15), the group loads do not correspond to the required group shares, and here we clearly see an improvement with VP arrival and probing preferences. Because $\rho_{2*} = 0.6$ and $r_2^G = 0.5$, the confinement of the jobs of group 2 to their subset causes the five processors in this subset to become unstable, and clearly $\Delta_2^G = 0$. When $L^{\mathcal{P}}$ increases from 1 to 4, the (internal) capacity loss in the subset of group 1 drops to zero. If $L^{\mathcal{P}} = 4$, the scheduling policy has complete knowledge about group 1's subset upon every arrival, and therefore $\Delta_1^G = 0$ as well. (Note that this already happens with $L^{\mathcal{P}} = 4$ instead of $L^{\mathcal{P}} = 5$, because the processor on which a job arrives is never probed. With VP arrival preference, jobs always arrive on processors in their own subsets.) This clearly shows the advantage of strictly separating the jobs of different groups (like SVP does) in cases where the group loads do not correspond to the required group shares. However, it also shows that this approach only makes sense when *both* the arrival and probing preferences are VP. Typically, with all global policies, the combined use of VP arrival and probing preferences has a negative effect under a case-1 workload and moderate probe limits (3 or 4), and a clear positive effect under a case-2 workload.

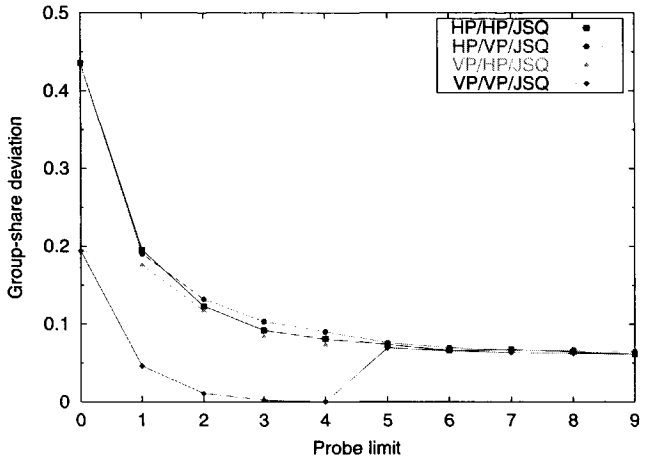


Figure 6.15: The group-share deviation Δ^G for JSQ under a case-2 workload as a function of the probe limit $L^{\mathcal{P}}$ for different arrival and probing preferences ($P = 10$).

6.8 Conclusions

This final chapter closes the gap between the static policies HPRS and VPRS of Chapter 4, and the dynamic policies JSQ, HP, SVP, and DVP of Chapter 5. The former policies use no system information at all, while in the system model used in Chapter 5, the latter policies have access to complete and up-to-date system information at all times. Clearly, none of these two cases is very realistic in practical situations. Therefore, in this chapter, we studied the effects of using incomplete and inaccurate state information by the four global policies of Chapter 5. However, unlike in Chapters 4 and 5, we were not concerned with the differences between the policies. We extended the system model to incorporate *probing*: a communication mechanism to obtain state information from the system. The amount of system information used is controlled by the probe limit $L^{\mathcal{P}}$ (which is the maximum number of processors probed for a scheduling decision). We considered both on-demand probing and periodic probing.

We found that in a system with realistic parameter values, on-demand probing is superior to periodic probing. With on-demand probing, probing only a fraction (20 to 30%) of the processors substantially improves the performance over not probing at all (which for most policies implies that each job is served until completion by the processor it arrives at). This is hardly affected by the probe service time $S^{\mathcal{P}}$, because this time is very small in realistic systems. Also, the policies are robust in the sense that the performance is not significantly affected by the service-time distributions of the jobs. However, the effect of the mean migration service time $S^{\mathcal{M}}$ is non-negligible. Therefore, with an increasing network load or an increasing number of processors, the policies should become more reluctant to job migrations. We also studied the effect of the number of processors and we found that as long as the number of processors is below 20, our results remain valid. The behavior for larger numbers of processors requires further study. Finally, we studied

the effects of arrival and probing preferences. We found that none of the combinations clearly outperforms the others.

Chapter 7

Conclusion

In this section, we present our conclusions and suggestions for future research. We stated the following research questions in Chapter 1:

- What are the objectives and performance measures for share scheduling in mathematical terms?
- What are suitable policies for share scheduling in uniprocessors, multiprocessors, and especially distributed systems?

7.1 Conclusions

We state the following conclusions:

1. *The objectives of share scheduling must take into account the system architecture and the workload.* This conclusion was drawn in Chapter 2. The amount of service a group can obtain from the system is limited by the number of jobs of that group present.
2. *Share scheduling presents little difficulties on uniprocessors and on multiprocessors.* On uniprocessors, one can use the Group-Priority Processor-Sharing policy introduced in Section 3.4.3. On multiprocessors, one can use the Multiprocessor Group-Priority Processor-Sharing policy introduced in Section 4.2.2. For both policies, the group-share deviation is zero at worst and the capacity loss is zero.
3. *On uniprocessors, non-discriminatory policies and non-preemptive policies are inadequate for share scheduling.* This is an important result from Chapter 3. In non-discriminatory policies such as First-Come First-Served and Processor Sharing, the delivery of the feasible group shares is too dependent on the distribution of the workload among the groups. In non-preemptive policies such as Head-of-the-Line and Up-Down, arriving jobs always have to wait until the job in service terminates. So, in general, such policies cannot guarantee the delivery of the feasible group shares.
4. *In distributed systems, even the simplest dynamic share-scheduling policies dramatically outperform random-splitting policies.* Convincing arguments for this statement

are given in Chapters 4 and 5. Even a very simple dynamic policy such as Join Shortest Queue outperforms Vertical Partitioning in Random Splitting, which we believe is close to the optimal in the class of random-splitting policies. From Chapter 6, we conclude that even in the presence of realistic network delays, and using only a fraction of the system-state information, the dynamic policies still outperform the random-splitting policies.

5. *In distributed systems, unless the number of processors is very high, both the global policy and the local policy have a significant effect on the share-scheduling performance.* This holds both for static policies, as concluded in Chapter 4, and for dynamic policies, as concluded in Chapter 5. Note though that there are some special cases in which the local policy has no effect, such as in the case of a fully partitionable system with Vertical Partitioning in Random Splitting.

6. *In distribution systems without job migration, the minimization of the capacity loss and the compliance with the feasible group shares are often conflicting objectives.* This is one of the conclusions of Chapter 5. For minimization of capacity loss, one should always put an arriving processor on an idle processor, if present. However, for compliance with the feasible group shares, it is better to reserve entire processors for groups so that these groups can be served at the proper rates when needed. In systems without job migration, one is forced to choose between minimization of capacity loss (by using JSQ or HP) and compliance with the feasible group shares (by using SVP or DVP).

7. *In distributed systems, unless the number of processors is very high, the availability of job migration substantially improves the performance of share-scheduling policies.* Evidence for this statement can be found in Chapter 5. The improvement is most striking for the Vertical-Partitioning policies SVP and DVP because these policies suffer from a high capacity loss when job migration is not available. For JSQ and HP, the benefits of job migration are smaller, but still substantial.

8. *In distributed systems, unless the number of processors is very high, policies that use a Vertical-Partitioning strategy outperform those with a Horizontal-Partitioning strategy.* This is a conclusion from both Chapters 4 and 5. The Vertical-Partitioning strategy is to reserve entire processors to groups and restrict the placement of a job to the processors assigned to the job's group. The Horizontal-Partitioning strategy is to spread out the jobs among the processors, and leave the delivery of feasible group shares to the local scheduling policy. If the number of processors is very high, the probability of queueing becomes negligible for most practical values of the total load, and the differences between the global policies vanish (as well as the differences between the local policies: jobs will seldom meet on one processor).

9. *It is feasible to run well-performing dynamic share-scheduling policies in a realistic distributed system.* This important conclusion follows from both Chapters 5 and 6.

7.2 Suggestions for Future Research

We state the following suggestions for future research:

1. *The extension of the objectives of share scheduling.* Even though we have gone through substantial effort in stating the objectives of share scheduling, some of the choices were made rather arbitrarily. We discussed some alternatives in Section 2.4.7. An example is the restriction of the performance measures for share scheduling to busy periods only, because users will not be interested in the periods during which they have no jobs present in the system. Another example is the inclusion of a history mechanism that changes the feasible group shares according to recent share deviations.

2. *The study of the queueing-theoretical problems of Chapters 3 and 4.* Some of the problem statements in Chapter 3 and in Section 4.3 describe interesting queueing-theoretical problems, even with Poisson arrivals and exponentially distributed service times, notably:

   - The derivation of closed-form expressions for the group-share deviation in a uniprocessor system under Head-of-the-Line, Priority Processors Sharing, or Group-Priority Processor Sharing.
   - The derivation of the optimal distribution of the total load ρ among the processors in order to minimize the capacity loss in a heterogeneous random-splitting system with more than two processors.

3. *The study of pattern-allocation policies for share scheduling.* In Section 4.3.2 we mentioned pattern-allocation policies: static global scheduling policies in which jobs are sent to processors according to a repetitive finite-length pattern. We believe such policies outperform random-splitting policies and are worth further study. Also, the combination of a dynamic global scheduling policy that employs probing according to a pattern (instead of probabilistically as in Chapter 6) seems very interesting for further study. This approach could decrease the required probe limit even further.

4. *The study of the Static Vertical Partitioning policy in systems without job migration.* The Static Vertical Partitioning policy does not perform well in systems without job migration. We believe there is still room for improvement of the policy. For instance, the present policy does not allow guest jobs when there is no job migration, but instead reserves processors to the groups that have them in their subsets. An alternative is to allow guest jobs even when there is no job migration, and use a checkpoint/restart mechanism to immediately release the processor when it is claimed by another group. In fact, this is much like the approach taken in the CONDOR system.

5. *The study of dynamic share-scheduling policies in heterogeneous distributed systems.* In Chapters 5 and 6 we have only considered homogeneous systems. A first approach to extending our work to heterogeneous systems is to redesign the policies such that they take heterogeneity into account.

6. *The study of share scheduling in very large clustered distributed systems.* We have not been able to study share scheduling in distributed systems with very large numbers of processors, in the order of hundreds or thousands. Some notable changes with the small to medium-sized systems we have been studying are that the large

systems are fully partitionable for all practical purposes, and that a single network connecting all processors is no longer a feasible solution for the information exchange. Instead, we believe the processors have to be clustered into subsystems in order to keep the network delays under control.

7. *The application of our results to share-scheduling problems in networking.* In this thesis, we have restricted ourselves to share scheduling in distributed systems under compute-intensive workloads. However, in another important application of share scheduling, that of data communications and computer networking, many of the assumptions made are no longer valid. Notable examples of such assumptions are the availability of job-preemption and job-migration mechanisms, and the exponential distribution of the job service time. Currently, share scheduling in networking receives much attention. It would be interesting to apply some of the principles presented in this thesis, such as Vertical Partitioning, to share-scheduling problems in networking, or to study the combination of share-scheduling objectives for the processors and the network (especially if jobs are both compute-intensive and communications-intensive).

8. *The study of share scheduling under workloads consisting of parallel programs.* We have restricted ourselves to workloads consisting of sequential jobs. However, the advent of parallel programming causes different types of workload to appear, consisting of jobs for which parts of the code can be run in parallel. Therefore, studying the performance of the share-scheduling policies under such workloads is a useful extension of the work in this thesis.

Bibliography

[1] A.O. Allen. *Probability, Statistics and Queueing Theory with Computer Science Applications.* Academic Press, second edition, 1990.

[2] F. Baccelli and Z. Liu. On the Execution of Parallel Programs on Multiprocessor Systems — A Queuing Theory Approach. *Journal of the ACM*, 37(2):373–414, 1990.

[3] F. Baskett, K.M. Chandy, R.R. Muntz, and F. Palacios. Open, Closed and Mixed Networks of Queues with Different Classes of Customers. *Journal of the ACM*, 22(2):248–260, 1975.

[4] A.J. Bernstein and J.C. Sharp. A Policy-Driven Scheduler for a Time-Sharing System. *Communications of the ACM*, 14(2):74–78, 1991.

[5] G.M. Birtwistle. *DEMOS — Discrete Event Modelling on Simula.* Macmillan, 1979.

[6] S. Borst, O. Boxma, and P. Jelenković. Asymptotic Behavior of Generalized Processor Sharing with Long-Tailed Traffic Sources. In *Proc. INFOCOM*, pages 912–921, 2000.

[7] S.C. Borst. Optimal Probabilistic Allocation of Customer Types to Servers. In *Proc. ACM Sigmetrics '95/Performance '95 (Special Issue of Performance Evaluation Review, Vol. 23, no. 1)*, pages 116–125, 1995.

[8] O.J. Boxma and J.W. Cohen. The M/G/1 Queue with Permanent Customers. *IEEE Journal of Selected Areas in Communications*, 9(2):179–184, 1991.

[9] J.P. Buzen and P.P.-S. Chen. Optimal Load Balancing in Memory Hierarchies. In J.L. Rosenfeld, editor, *Proc. IFIP 1974*, pages 271–275. North-Holland, 1974.

[10] T. Casavant and J. Kuhl. A Taxonomy of Scheduling in General Purpose Distributed Computing Systems. *IEEE Transactions on Software Engineering*, 14(2):141–154, 1988.

[11] L.M. Casey. Decentralised Scheduling. *Australian Computer Journal*, 13(2), 1981.

[12] Y.C. Chow and W.H. Kohler. Dynamic Load Balancing in Homogeneous Two-Processor Distributed Systems. In K.M. Chandy and M. Reiser, editors, *Computer Performance.* North-Holland, 1977.

[13] Y.C. Chow and W.H. Kohler. Models for Dynamic Load Balancing in a Heterogeneous Multiple Processor System. *IEEE Transactions on Computers*, 28(5):354–361, 1979.

[14] E.G. Coffman and L. Kleinrock. Feedback Queueing Models for Time-Shared Systems. *Journal of the ACM*, 15(4):549–576, 1968.

[15] E.G. Coffman, R.R. Muntz, and H. Trotter. Waiting Time Distributions for Processor-Sharing Systems. *Journal of the ACM*, 17(1):123–130, 1970.

[16] J.W. Cohen. *The Single Server Queue.* North-Holland, revised edition, 1982.

[17] J.W. Cohen. On the Analysis of the Symmetrical Shortest Queue. Technical Report BS-R9420, Centrum voor Wiskunde en Informatica, 1994.

[18] J.W. Cohen. Analysis of the Asymmetrical Shortest Two-Server Queueing Model. Technical Report BS-R9509, Centrum voor Wiskunde en Informatica, 1995.

[19] M.B. Combé and O.J. Boxma. Optimization of Static Traffic Allocation Policies. *Theoretical Computer Science*, (125):17–43, 1994.

[20] CONVEX Computer Corporation. *Revision Information for CONVEX Share Scheduler*, 1991. Report Number 710-004630-001.

[21] E. de Souza e Silva and M. Gerla. Load Balancing in Distributed Systems with Multiple Classes and Site Constraints. In E. Gelenbe, editor, *Performance '84*, pages 17–33. North-Holland, 1984.

[22] E. de Souza e Silva and M. Gerla. Queueing Network Models for Load Balancing in Distributed Systems. *Journal of Parallel and Distributed Computing*, 12(1):24–38, 1991.

[23] A. Demers, S. Keshav, and S. Shenker. Analysis and Simulation of a Fair Queueing Algorithm. In *Proc. of the ACM Sigcomm Symp.*, pages 1–12, 1989.

[24] D.L. Eager, E.D. Lazowska, and J. Zahorjan. A Comparison of Receiver-Initiated and Sender-Initiated Adaptive Load Sharing. *Performance Evaluation*, 6:53–68, 1986.

[25] D.L. Eager, E.D. Lazowska, and J. Zahorjan. Adaptive Load Sharing in Homogeneous Distributed Systems. *IEEE Transactions on Software Engineering*, 12(5):662–675, 1986.

[26] D.H.J. Epema. Processor-Management in Distributed Systems with a Compute-Intensive Workload. In *Proc. OpenForum Technical Conference*, pages 233–250, 1992.

[27] D.H.J. Epema. Decay-Usage Scheduling in Multiprocessors. *ACM Transactions on Computer Systems*, 16(4):367–415, 1998.

[28] D.H.J. Epema and J.F.C.M. de Jongh. Proportional Share-Scheduling in Single-Server and Multiple-Server Computing Systems. *Performance Evaluation Review*, 27(3):7–10, 1999.

[29] R.B. Essick. An Event Based Fair Share Scheduler. In *USENIX*, pages 147–161, 1990.

[30] G. Fayolle, I. Mitrani, and R. Iasnogorodski. Sharing a Processor Among Many Job Classes. *Journal of the ACM*, 27(3):519–532, 1980.

[31] A. Feldman and W. Whitt. Fitting Mixtures of Exponentials to Long-Tail Distributions to Analyze Network Performance Models. *Performance Evaluation*, 31(3–4):245–279, 1998.

[32] L.L. Fong and M.S. Squillante. Time-Function Scheduling: A General Approach to Controllable Resource Management. Technical Report RC 20155, IBM T.J. Watson Research Center, 1995.

[33] E. Gelenbe and G. Pujolle. *Introduction to Queueing Networks.* John Wiley and Sons, 1987.

[34] A. Goscinski. *Distributed Operating Systems, The Logical Design.* Addison-Wesley Publishing Company, 1991.

[35] A.G. Greenberg and N. Madras. How Fair is Fair Queuing? *Journal of the ACM*, 39(3):568–598, 1992.

[36] M. Harchol-Balter and A.B. Downey. Exploiting Process Lifetime Distributions for Dynamic Load Balancing. *ACM Transactions on Computer Systems*, 15(3):252–285, 1997.

[37] J.L. Hellerstein. Achieving Service Rate Objectives with Decay Usage Scheduling. *IEEE Transactions on Software Engineering*, 19(8):813–825, 1993.

[38] G.J. Henry. The Fair Share Scheduler. *AT&T Bell Laboratories Technical Journal*, 63(8):1845–1857, 1984.

[39] D.P. Heyman and M.J. Sobel. *Stochastic Models in Operations Research*, volume 1. McGraw Hill, 1982.

[40] D. Jackson. Maui Scheduler on Linux. In *USENIX*, 1999.

[41] D. Jackson, Q. Snell, and M. Clement. Core Algorithms of the Maui Scheduler. In *Proc. 7-th Workshop on Job Scheduling Strategies for Parallel Processing*, 2001.

[42] C. Jacqmot, E. Milgrom, W. Joossen, and Y. Berbers. Unix and Load-Balancing: A Survey. In *Proc. EUUG '89*, pages 1–15, 1989.

[43] N.K. Jaiswal. *Priority Queues.* Academic Press, 1968.

[44] E.P. Jansen. *Improving Production Schedules without Scheduling.* PhD thesis, Delft University of Technology, 1993.

[45] J.F.C.M. de Jongh. Central Dynamic Share Scheduling in Distributed Systems. Technical Report 93-48, Delft University of Technology, 1993.

[46] J.F.C.M. de Jongh. Central Dynamic Share Scheduling in Distributed Systems. In *Proc. Computing Science in the Netherlands*, pages 180–192, 1993.

[47] J.F.C.M. de Jongh. A Performance Comparison of Distributed Share-Scheduling Policies. In *Proc. 10-th UK Performance Engineering Workshop*, pages 89–102, 1994.

[48] J.F.C.M. de Jongh and D.H.J. Epema. Central Dynamic Share Scheduling in Distributed Systems. In *Proc. 2-nd Ann. Conf. of the Advanced School for Computing and Imaging*, pages 144–149, 1996.

[49] J. Kay and P. Lauder. A Fair Share Scheduler. *Communications of the ACM*, 31:44–55, 1988.

[50] D.G. Kendall. Some Problems in the Theory of Queues. *J. Roy. Statist. Soc. Ser. B*, (13):151–173, 1951.

[51] L. Kleinrock. Time-shared Systems, A Theoretical Treatment. *Journal of the ACM*, 14(2):242–261, 1967.

[52] L. Kleinrock. *Queueing Systems Volume II: Computer Applications.* John Wiley and Sons, 1976.

[53] D.E. Knuth. *Seminumerical Algorithms*, volume 2 of *The Art of Computer Programming.* Addison-Wesley Publishing Company, 2 edition, 1981.

[54] G.M. Koole. *Stochastic Scheduling and Dynamic Programming.* PhD thesis, Rijksuniversiteit Leiden, 1992.

[55] M. Kramer. A Processor Sharing Model with Several Types of Jobs and Preemptive Priorities. In E. Gelenbe, editor, *Performance '84*, pages 337–344. North-Holland, 1984.

[56] P. Krueger and M. Livny. The Diverse Objectives of Distributed Scheduling Policies. In *Proc. 7-th Int. Conf. on Distr. Comp. Syst.*, pages 242–249, 1987. Berlin, West Germany.

[57] K.G. Langendoen. Myrinet: The High-Speed Network for DAS. In *Proc. 13th Ann. Conf. of the Advanced School for Computing and Imaging*, page 259. Heijen, The Netherlands, 1997.

[58] E.L. Lawler, J.K. Lenstra, A.H.G. Rinnooy Kan, and D.B. Shmoys. *Sequencing and Scheduling: Algorithms and Complexity*, volume 4 of *Handbooks in Operations Research and Management Science.* North-Holland, 1992.

[59] W.E. Leland and T.J. Ott. Load-Balancing Heuristics and Process Behavior. In *Proc. Performance and ACM Sigmetrics*, pages 54–69, 1986.

[60] M.J. Litzkow, M. Livny, and M.W. Mutka. Condor - A Hunter of Idle Workstations. In *Proc. 8-th Int. Conf. on Distr. Comp. Syst.*, pages 104–111, 1988.

[61] M. Livny and M. Melman. Load Balancing in Homogeneous Broadcast Distributed Systems. *ACM Performance Evaluation*, 11(1):47–55, 1982.

[62] R.M. Metcalfe and D.R. Boggs. Ethernet: Distributed Packet Switching for Local Computer Networks. *Communications of the ACM*, 19(7):395–404, 1976.

[63] I. Mitrani. *Simulation Techniques for Discrete Event Systems.* Cambridge University Press, 1982.

[64] C.M. Moruzzi and G.G. Rose. Watson Share Scheduler. In *Proc. Lisa V*, pages 129–133, 1991.

[65] M.W. Mutka. Estimating Capacity for Sharing in a Privately Owned Workstation Environment. *IEEE Transactions on Software Engineering*, 18(4):319–328, 1992.

[66] M.W. Mutka and M. Livny. Profiling Workstations' Available Capacity for Remote Execution. In P.-J. Courtois and G. Latouche, editors, *Performance '87*, pages 529–544. North-Holland, 1987.

[67] M.W. Mutka and M. Livny. Scheduling Remote Processing Capacity in a Workstation-Processor Bank Network. In *Proc. 7-th Int. Conf. on Distr. Comp. Syst.*, pages 2–9, 1987.

[68] R.D. Nelson and Th.K. Philips. An Approximation for the Mean Response Time for Shortest Queue Routing with General Interarrival and Service Times. *Performance Evaluation*, 17:123–139, 1993.

[69] L.M. Ni and K. Hwang. Optimal Load Balancing in a Multiple Processor System with Many Job Classes. *IEEE Transactions on Software Engineering*, 11(5):491–496, 1985.

[70] T.J. Ott. The Sojourn-Time Distribution in the M/G/1 Queue with Processor Sharing. *Journal of Applied Probability*, 21(2):360–378, 1984.

[71] A.K. Parekh and R.G. Gallager. A Generalized Processor Sharing Approach to Flow Control in Integrated Services Networks—The Single Node Case. In *Proc. IEEE Infocom '92*, pages 915–924, 1992.

[72] K.M. Rege and B. Sengupta. Queue-Length Distribution for the Discriminatory Processor Sharing Queue. In *Proc. Conf. on Applied Probability*, 1993. Paris.

[73] C.G. Rommel. The Probability of Load Balancing Success in a Homogeneous Network. *IEEE Transactions on Software Engineering*, 17(9):922–933, 1991.

[74] K.W. Ross and D.D. Yao. Optimal Load Balancing and Scheduling in a Distributed Computer System. *Journal of the ACM*, 38(3):676–690, 1991.

[75] J. Sethuraman and M.S. Squillante. Optimal Stochastic Scheduling in Multiclass Parallel Queues. In *Proc. ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, pages 93–102, 1999.

[76] M. Shreedhar and G. Varghese. Efficient Fair Queuing using Deficit Round-Robin. *IEEE/ACM Transactions on Networking*, 4(3):375–385, 1996.

[77] W.E. Smith. Various Optimizers for Single-Stage Production. *Naval Research and Logistics Quarterly*, 3:59–66, 1954.

[78] W. Stallings. *Data and Computer Communications*. Macmillan, fourth edition, 1994.

[79] I. Stoica, H. Abdel-Wahab, and K. Jeffay. A Proportional Share Resource Allocation Algorithm For Real-Time, Time-Shared Systems. In *Proc. 17th Real-Time Systems Symposium*, pages 228–299, 1996.

[80] I. Stoica, H. Abdel-Wahab, and K. Jeffay. On the Duality between Resource Reservation and Proportional Share Resource Allocation. In *Proc. Multimedia Computing and Networking*, pages 207–214, 1997.

[81] A.N. Tantawi and D. Towsley. Optimal Static Load Balancing in Distributed Computer Systems. *Journal of the ACM*, 32(2):445–465, 1985.

[82] C.A. Waldspurger and E.W. Weihl. Lottery Scheduling: Flexible Proportional-Share Resource Management. In *Proc. First Symp. on Operating System Design and Implementation*, 1994.

[83] C.A. Waldspurger and E.W. Weihl. Stride Scheduling: Deterministic Proportional-Share Resource Management. Technical Memorandum MIT/LCS/TM-528, MIT Laboratory for Computer Science, 1995.

[84] Y.T. Wang and R.J.T. Morris. Load Sharing in Distributed Systems. *IEEE Transactions on Computers*, 34(3):204–217, 1985.

[85] W. Winston. Optimality of the Shortest Line Discipline. *Journal of Applied Probability*, 14:181–189, 1977.

[86] R.W. Wolff. Time Sharing with Priorities. *SIAM Journal of Applied Mathematics*, 19:566–574, 1970.

[87] R.W. Wolff. Work-Conserving Priorities. *Journal of Applied Probability*, 7:327–337, 1970.

[88] S.F. Yashkov. A Derivation of Response Time Distribution for a M/G/1 Processor Sharing Queue. *Problems of Control and Information Theory*, 12(2):133–148, 1983.

[89] E.R. Zayas. Attacking the Process Migration Bottleneck. In *Proc. 11-th ACM Symp. on Operating System Principles*, pages 13–24, 1987.

Appendix A

Simulation Techniques

In this thesis we have used discrete-event simulation as a means to evaluate the performance of scheduling policies. All simulations of models under stochastic workloads were performed with the DEMOS ('Discrete-Event Modeling on Simula') package [5]. We designed and implemented several extensions to DEMOS, such as processor-sharing queueing disciplines and time-series analysis [63], but the basic simulation engine was left undisturbed.

Of particular importance to us is the availability of a good random-number generator (RNG). In Section A.1, we explain how DEMOS generates random-number sequences. Subsequently, in Section A.2 we explain two well-known goodness-of-fit tests, the Chi-Square and Kolmogorov-Smirnov tests. We present the results of these and other empirical tests applied to the RNG of DEMOS in Section A.3. Finally, in Section A.4 we explain the methods used to obtain point and interval estimates from our simulation data.

A.1 Random-Number Generation in DEMOS

In DEMOS, a sequence of integers X_n, $n = 0, 1, \ldots$, is generated for each random variable used. Each draw from the random variable uses one or more successive numbers in the sequence. The value of X_{n+1} is calculated from X_n as

$$X_{n+1} = X_n \otimes 32 \otimes 32 \otimes 8. \tag{A.1}$$

The symbol $\otimes$ denotes multiplication modulo-M, with $M = 67099547$. In other words,

$$A \otimes B \stackrel{\triangle}{=} AB \bmod M. \tag{A.2}$$

The generator is of the *multiplicative* type, with period $M - 1$. We have, for $n > 1$,

$$1 \leq X_n \leq M - 1, \tag{A.3}$$

as long as $1 \leq X_0 \leq M - 1$.

The starting value of the sequence, X_0, is determined by a *seed generator*. The purpose of the seed generator is to generate well-spread initial values for different random variables, in order to minimize the (supposedly absent) dependency between them. The seed generator is a RNG in itself. (For obvious reasons, the seed-generator algorithm must

be different from the algorithm used for drawing from random variables.) In DEMOS, seeds are generated as a sequence of integers Y_i, with, for $i = 0, 1, 2, \ldots$,

$$Y_{i+1} = Y_i \otimes 7 \otimes 13 \otimes 15 \otimes 27. \tag{A.4}$$

The sequence $\{Y_i\}$ also has period $M - 1$. The initial value of this sequence is set by DEMOS to $Y_0 = 907$.

A.2 Goodness-of-Fit Tests

In this section we present the Chi-Square and Kolmogorov-Smirnov tests. These tests determine whether it is likely that a given finite sequence is drawn from a random variable with a specific distribution function.

A.2.1 The Chi-Square Test

A classical goodness-of-fit test is the Chi-Square (CS) test. The following description is taken from Knuth [53], pp 39–45. We consider a sequence of independent observations $X_1, X_2, \ldots, X_n$. The observations are classified into k mutually exclusive categories, numbered $1, \ldots, k$. Let Y_i denote the number of observations that fall into category i $(1 \leq i \leq k)$. We wish to determine whether the sequence $\{X_i\}$ is likely to be drawn from a random variable with a specific distribution function. To test this hypothesis, we assume it is true and we first calculate the probabilities p_i that an observation falls into category i, $1 \leq i \leq k$. Subsequently, we calculate the *chi-square statistic* V as follows

$$V \triangleq \sum_{i=1}^{k} \frac{(Y_i - np_i)^2}{np_i}. \tag{A.5}$$

In general, the higher V, the more unlikely our hypothesis.

For large n, V has a chi-square distribution with $k - 1$ degrees of freedom. As a rule of thumb, Knuth advises to choose $n \geq 5/p_i$, $i = 1, \ldots, k$. Using a chi-square percentile table, one can either reject or accept the hypothesis at a certain level of significance. The level of significance of a hypothesis test is the probability that the hypothesis, although true, is rejected.

A.2.2 The Kolmogorov-Smirnov Test

Another goodness-of-fit test is the Kolmogorov-Smirnov (KS) test. The KS test is typically used in cases where the outcomes of an experiment form a continuous range of possible values. Again following Knuth ([53], pp 45–56), suppose we have n observations, $X_1, \ldots, X_n$. We construct the *empirical distribution function* $F_n(x)$ of the sequence $\{X_i\}$, defined as

$$F_n(x) \triangleq \frac{\text{number of elements in } \{i | X_i \leq x\}}{n}. \tag{A.6}$$

The purpose of the KS test is to determine the likelihood that the sequence $\{X_i\}$ is indeed taken from a random variable with given *continuous* distribution function $F(\cdot)$. To that

end, some *measure* of the deviation between $F_n(\cdot)$ and $F(\cdot)$ is used, for which several alternatives exist [1]. We use the following two [53]:

$$K^+ = \sqrt{n} \max_x \{F_n(x) - F(x)\} \tag{A.7}$$

$$K^- = \sqrt{n} \max_x \{F(x) - F_n(x)\} \tag{A.8}$$

Again, for each of these statistics we have that the higher the statistic, the less likely it is that $\{X_i\}$ originates from a random variable with distribution function $F(\cdot)$. With a table holding the percentiles for the distributions of K^+ and K^- (these distributions are identical, and depend upon n), such as for instance given in [53], page 48, we can accept or reject our hypothesis at a certain level of significance.

A.3 Empirical Tests on the DEMOS Random-Number Generator

In this section, we consider a sample $U_1, \ldots, U_n$ taken from the DEMOS $U[0,1]$ (the uniform distribution on $[0,1]$) RNG, and apply statistical test to it.

A.3.1 Equidistribution Test

Our first question is whether the samples $\{U_i\}$ are really taken from a random variable with the uniform distribution function on $[0,1]$. Therefore, we apply both the Chi-Square and the Kolmogorov-Smirnov tests to the sequence. The CS test is performed 1000 times on consecutive sequences of 100 samples each. The interval $[0,1]$ is subdivided into 10 intervals of equal length, so $k = 10$. We thus obtain the chi-square statistics $V_1, \ldots, V_{1000}$, one for each sequence. Assuming our hypothesis is true, let π_p^{CS9} denote the p-th percentile of the chi-square distribution with 9 $(= k - 1)$ degrees of freedom, so

$$\mathbf{P}\left[V_i \leq \pi_p^{CS9}\right] = p. \tag{A.9}$$

From our observations, we subsequently calculate the frequencies S_p^{CS}, defined as

$$S_p^{CS} \triangleq \frac{\text{number of elements in } \{i | V_i \leq \pi_p^{CS9}\}}{1000} \tag{A.10}$$

Because π_p^{CS9} must be looked up in a table, we only calculate S_p^{CS} for $p = 0.01$, 0.05, 0.25, 0.5, 0.75, and 0.99. When our hypothesis is true, we must find that $S_p^{CS} \approx p$. In Table A.1, a table is shown holding p and S_p^{CS}. Clearly, the differences between p and S_p^{CS} are sufficiently small.

In a similar way, we apply the Kolmogorov-Smirnov test. The KS-test is performed to 1000 consecutive sequences of 100 samples each, resulting in 1000 observations of K^+ $(K_1^+, \ldots, K_{1000}^+)$ and K^- $(K_1^-, \ldots, K_{1000}^-)$. We define the percentile π_p^K as follows

$$\mathbf{P}\left[K_i^+ \leq \pi_p^K\right] = \mathbf{P}\left[K_i^- \leq \pi_p^K\right] = p. \tag{A.11}$$

p	S_p^{CS}
0.01	0.01
0.05	0.04
0.25	0.25
0.50	0.49
0.75	0.75
0.95	0.95
0.99	0.99

Table A.1: Observed frequencies S_p^{CS} for the chi-square test applied to the DEMOS RNG.

p	S_p^+	S_p^-
0.01	0.01	0.01
0.05	0.04	0.04
0.25	0.25	0.24
0.50	0.48	0.50
0.75	0.75	0.74
0.95	0.95	0.94
0.99	0.99	0.99

Table A.2: Observed frequencies S_p^+ and S_p^- for the Kolmogorov-Smirnov test applied to the DEMOS $U[0,1]$ generator.

For selected values of p, we again compare p with the observed frequencies S_p^+ and S_p^-, defined as

$$S_p^+ \triangleq \frac{\text{number of elements in } \{i|K_i^+ \leq \pi_p^K\}}{1000}, \tag{A.12}$$

$$S_p^- \triangleq \frac{\text{number of elements in } \{i|K_i^- \leq \pi_p^K\}}{1000}. \tag{A.13}$$

In Table A.2, a table is shown holding p, S_p^+, and S_p^-. Again we see that the observed frequencies closely match the expected.

A.3.2 Serial Test

The serial test is used to determine whether pairs of consecutive draws from a uniform random variable are uniformly distributed. In this case, we subdivide the interval $[0,1]$ in 10 disjoint intervals of equal length. Each possible pair of intervals to which a pair of consecutive samples belongs, now forms a category in a chi-square test. Every pair has equal probability of being hit. We perform 500 CS tests, each test applying to 2000 samples (and thus 1000 pairs). We define S_p^S as the frequency of observations for which the chi-square statistic is smaller than or equal to the p-th percentile. In Table A.3 we

p	S_p^S
0.01	0.00
0.05	0.05
0.25	0.22
0.50	0.49
0.75	0.75
0.95	0.95
0.99	0.99

Table A.3: Observed frequencies S_p^S for the serial test applied to DEMOS $U[0,1]$ generator.

show these frequencies for some values of p. Except for $S_{0.25}^S$, these values are close to their expected values.

A.3.3 Gap Test

The gap test examines the length between occurrences of U_i in a certain range. In our case, we determine the gaps between two occurrences that are in the interval $[0, 0.1]$. The p.m.f. for the gap length is clearly *not* uniformly distributed. In this case, the probability p_i of a gap of length i is

$$\begin{aligned} p_0 &= 0.1, \\ p_i &= 0.1(0.9)^{i-1},\ i > 0, \end{aligned} \tag{A.14}$$

In order to limit the number of categories for the CS test, we make a single category for all gaps with length 32 or more. The probability p_{32} of the occurrence of a gap of such length is

$$p_{32} = (0.9)^{32}. \tag{A.15}$$

We perform 100 CS tests, each test applying to 100 gaps. Let S_p^G denote the fraction of observed values of the chi square statistic that are smaller or equal than the p-th percentile. We show in Table A.4, the observed S_p^G for 7 values of p.

A.3.4 Coupon-Collector's Test

In our final test, the coupon-collector's test, the interval $[0, 1]$ is subdivided into 3 non-overlapping intervals of equal length. Again we are only interested in the interval to which a member of the sequence U_i belongs. We examine the length of the subsequences required to have all the intervals represented. In other words, such a subsequence contains for every interval a member that lies in it, and this condition is not met when the most-right member of the subsequence is taken away. The p.m.f. for the length of such a subsequence (clearly, in our case it cannot be smaller than three) can be found in [53], page 63. We perform 100 CS tests, each on 1000 subsequences. We use a single category for subsequences of length six and larger. Let S_p^{CC} denote the fraction of observed values of the chi-square statistic that are smaller than or equal to the p-th percentile. We show in Table A.5 the observed S_p^{CC} for the familiar 7 values of p.

p	S_p^G
0.01	0.00
0.05	0.04
0.25	0.25
0.50	0.51
0.75	0.72
0.95	0.95
0.99	0.98

Table A.4: Observed frequencies S_p^G for the gap test applied to the DEMOS $U[0,1]$ generator.

p	S_p^{CC}
0.01	0.03
0.05	0.09
0.25	0.27
0.50	0.54
0.75	0.76
0.95	0.98
0.99	0.99

Table A.5: Observed frequencies S_p^{CC} for the coupon-collector's test applied to the DEMOS $U[0,1]$ generator.

A.3.5 Conclusions

Clearly, for every random-number generator there is an associated test that will fail when applied to it. Nevertheless, we believe putting the random-number generator to the test can expose fundamental problems with it, such as a short period. The test results shown in this section have given us enough confidence that the RNG used by DEMOS is of sufficient quality.

A.4 Point and Interval Estimates with Batched Means

In order to obtain point and interval estimates from our simulation data, we used the method of *batched means*. In this method, the output data of a simulation are discarded for an estimated transient period T_t, after which data from the system are collected during B ($B \geq 2$) *subruns*, each of length T. If T is large enough, we may view the observations made in different intervals as stochastically independent. Let $\bar{\theta}_b$ be the observed average value of $\theta(t)$ during the b-th subrun, defined as

$$\bar{\theta}_b \triangleq \frac{1}{T}\int_{T_t+(b-1)T}^{T_t+bT} \theta(t)dt, \; b = 1, \ldots B. \tag{A.16}$$

An unbiased estimator $\bar{\theta}$ for θ is then given by the sample means of the $\bar{\theta}_b$'s:

$$\bar{\theta} \triangleq \frac{1}{B}\sum_{b=1}^{B} \bar{\theta}_b. \tag{A.17}$$

A confidence interval can be obtained by assuming that all the $\bar{\theta}_b$'s are independent, identically distributed random variables. However, generally this is not the case.

A more proper way of dealing with dependent samples is to estimate the covariance function (*time-series* analysis, see for instance [63]), to find a better estimator for the variance and, consequently, better confidence intervals. This method is applicable to covariance-stationary stochastic processes, for which the covariance function $\gamma(\cdot)$ is defined as

$$\gamma(b) \triangleq \mathbf{E}\left[(\theta_i - \theta)(\theta_{i+b} - \theta)\right], \; i, b = \cdots, -1, 0, +1, \cdots. \tag{A.18}$$

Note that by the definition of covariance stationarity, the right-hand side of Equation A.18 is independent of i. Also, $\gamma(-b) = \gamma(b)$ for all b. An estimator $\bar{\gamma}(0)$ for $\gamma(0)$ is provided by

$$\bar{\gamma}(0) \triangleq \frac{\sum_{b=1}^{B}(\bar{\theta}_b - \bar{\theta})^2}{B-1}, \tag{A.19}$$

and an estimator $\bar{\gamma}(b)$ for $\gamma(b)$, $b > 0$, by

$$\bar{\gamma}(b) \triangleq \frac{\sum_{i=1}^{B-b}(\bar{\theta}_i - \bar{\theta})(\bar{\theta}_{i+b} - \bar{\theta})}{B-1}. \tag{A.20}$$

Usually, we assume $\gamma(b) = 0$ for $b > m$ and some integer $m \leq B$. An estimator for the variance σ_θ^2 is

$$\bar{\sigma}_\theta^2 \triangleq \frac{\bar{\gamma}(0)}{B} + \frac{2\sum_{b=1}^{m}(B-b)\bar{\gamma}(b)}{B^2}. \tag{A.21}$$

Now $(\bar{\theta} - \theta)/(\bar{\sigma}_\theta/\sqrt{B})$ has approximately Student's t distribution with $B - 1$ degrees of freedom, so a confidence interval with coefficient $1 - \alpha$ is given by

$$< \bar{\theta} - \frac{z\bar{\sigma}_\theta}{\sqrt{B}}, \bar{\theta} + \frac{z\bar{\sigma}_\theta}{\sqrt{B}} >, \tag{A.22}$$

where z is defined by

$$\mathbf{P}\left[t_{B-1} \leq z\right] \leq 1 - (\alpha/2), \tag{A.23}$$

t_{B-1} being a random variable with Student's t distribution with $B-1$ degrees of freedom.

Appendix B

List of Acronyms

ASCI	Advanced School for Computing and Imaging
ATM	Asynchronous Transfer Mode
CSMA/CD	Carrier-Sense Multiple Access with Collision Detection
CS	Chi-Square
DEMOS	Discrete Event Modeling on Simula
DRR	Deficit Round-Robin
DVP	Dynamic Vertical Partitioning
EEVDF	Earliest Eligible Virtual Deadline First
FCFS	First-Come First-Served
FIFO	First-In First-Out
FQ	Fair Queueing
FQF	Fair Queueing based on Finishing times
FQS	Fair Queueing based on Starting times
Gbps	Gigabits per second
GPPS	Group-Priority Processor Sharing
GPS	Generalized Processor Sharing
HOL	Head-of-the-Line
HOL-PS	Head-of-the-Line Processor Sharing
HP	Horizontal Partitioning
HPRS	Horizontal Partitioning in Random Splitting
IDB	Idle Broadcast Algorithm
JPPS	Job-Priority Processor Sharing
JSQ	Join Shortest Queue
KS	Kolmogorov-Smirnov
LCFS	Last-Come First-Served
LST	Laplace-Stieltjes transform
MAP	Markov Arrival Process
Mbps	Megabits per second
MGPPS	Multiprocessor Group-Priority Processor Sharing
MJPPS	Multiprocessor Job-Priority Processor Sharing
MRTP	Minimum Response Time Policy
MSTP	Minimum System Time Policy

MTP	Maximum Throughput Policy
NP	Non-Polynomial
PB	Proportional Branching
PGPS	Packet-by-packet Generalized Processor Sharing
PID	Poll when Idle Algorithm
PLBS	Probability of Load-Balancing Success
pgf	probability-generating function
pmf	probability mass function
PPS	Priority Processor Sharing
PQ	Priority Queueing
PS	Processor Sharing
RNG	Random-Number Generator
RR	Round Robin
RSS	Random Selection for Service
STB	State Broadcast Algorithm
SVP	Static Vertical Partitioning
TFS	Time-Function Scheduling
UD	Up-Down
VLSI	Very Large-Scale Integration
VP	Vertical Partitioning
VPRS	Vertical Partitioning in Random Splitting
WFQ	Weighted Fair Queueing

Appendix C

List of Symbols

This list only contains symbols that have been used in more than one section. Notably, most symbols defined and used only in Section 4.2 and in Appendix A have not been included.

Symbol	Meaning	Definition
$\lvert \cdot \rvert$	Number of elements in a set	Section 2.2.1
$\lvert \cdot \rvert$	Sum of the elements of a vector or a matrix	Section 3.1
α	Ratio of service rates (PQ, $G = 2$)	Equation 3.38 (Section 3.3.1)
A_j	Arrival time of job j	Section 2.2.1
c	Total capacity	Equation 1.2 (Section 1.1)
c_p	Capacity of processor p	Section 1.1
Δ^{T}	(Expected) Capacity loss	Equation 2.43 (Section 2.5.2)
$\Delta^{\mathrm{T}}(t)$	Total share deviation (Capacity loss) at time t	Equation 2.42 (Section 2.5.2)
$\Delta^{\mathrm{T,MIN}}$	Minimum capacity loss	Equation 4.67 (Section 4.4.1)
Δ^{G}	(Maximum expected) Group-share deviation	Equation 2.40 (Section 2.4.6)
Δ_g^{G}	Expected group-share deviation of group g	Equation 2.39 (Section 2.4.6)
$\Delta_g^{\mathrm{G}}(t)$	Group-share deviation of group g at time t	Equation 2.38 (Section 2.4.6)
$\Delta^{\mathrm{G,MIN}}$	Minimum group-share deviation	Equation 4.130 (Section 4.5.1)
Δ^{J}	(Maximum expected) Job-share deviation	Equation 2.48 (Section 2.6.2)
Δ_g^{J}	Expected job-share deviation of group g	Equation 2.47 (Section 2.6.2)

$\Delta_g^{\mathrm{J}}(t)$	Job-share deviation of group g at time t	Equation 2.45 (Section 2.6.2)
D_j	Departure time of job j	Section 2.2.1
$F(\cdot)$	Pgf of joint number of jobs	Equation 3.8 (Section 3.1)
$F_g(\cdot)$	Pdf of service time of jobs of group g on a unit-capacity processor	Section 1.2
$f^{\mathrm{T}}(t)$	Total feasible share at time t	Equation 2.41 (Section 2.5.1)
$f_g^{\mathrm{G}}(t)$	Feasible group share of group g at time t	Equation 2.26 (Section 2.4.4)
$\bar{f}_g^{\mathrm{G}}(t)$	Maximum feasible group share of group g at time t	Equation 2.24 (Section 2.4.2)
$f_g^{\mathrm{J}}(t)$	Feasible job share of any job of group g at time t	Equation 2.44 (Section 2.6.1)
G	Number of groups	Section 1.2
$J(t)$	Set of jobs at time t	Equation 2.2 (Section 2.2.1)
$J_{g*}(t)$	Set of jobs of group g at time t	Section 2.2.1
$J_{*p}(t)$	Set of jobs on processor p at time t	Section 2.2.1
$J_{gp}(t)$	Set of jobs of group g on processor p at time t	Section 2.2.1
λ	Total arrival rate of non-permanent jobs	Equation 3.1 (Section 3.1)
λ_g	Arrival rate of non-permanent jobs of group g	Section 1.2.1
$L^{\mathcal{P}}$	Probe limit	Section 6.4
μ_g	Service rate of jobs of group g on a unit-capacity processor	Section 1.2
$m^{\mathrm{T}}(t)$	Maximum obtainable total share at time t	Equation 2.11 (Section 2.2.3)
$m_g^{\mathrm{G}}(t)$	Maximum obtainable group share of group g at time t	Equation 2.12 (Section 2.2.3)
N^{P}	Total number of permanent jobs	Equation 3.7 (Section 3.1)
$\boldsymbol{N}(t)$	Matrix holding the number of jobs (permanent and non-permanent) per group and per processor at time t	Equation 3.4 (Section 3.1)
$N(t)$	Total number of jobs (permanent and non-permanent) at time t	Equation 3.6 (Section 3.1)
$N_g(t)$	Number of jobs (permanent and non-permanent) of group g at time t	Section 3.1
$\boldsymbol{N}^{\mathbf{P}}$	Vector holding the number of permanent jobs per group	Equation 3.5 (Section 3.1)
N_g^{P}	Number of permanent jobs of group g	Section 3.1
$\boldsymbol{N}_{g*}(t)$	$G \times 1$-vector holding the	Section 4.3.1

	(total) number of jobs for each group at time t	
$\boldsymbol{N}_{*p}(t)$	$1 \times P$-vector holding the (total) number of jobs for each processor at time t	Section 4.3.1
$\boldsymbol{N}_{*,p_1 \cdots p_2}(t)$	$G \times (p_2 - p_1 + 1)$-matrix holding the number of jobs for each group on processors p_1 through p_2 inclusive at time t	Section 4.3.1
$\mathbf{P}_{\text{lbs}}(n, m)$	Probability of load-balancing success	Section 1.3.3
$\mathbf{P}_{\text{wwi}}$	Wait-while-idle probability	Section 4.3.2
π_{gp}	Probability that a job of group g arrives at processor p	Section 4.3.1
Ω_g	Subset of processors serving jobs of group g	Equation 4.138 (Section 4.5.3)
$o^{\text{T}}(t)$	Total obtained share at time t	Equation 2.7 (Section 2.2.2)
$o_g^{\text{G}}(t)$	Obtained share of group g at time t	Equation 2.6 (Section 2.2.2)
$o_{gp}^{\text{G}}(t)$	Obtained share of group g on processor p at time t	Equation 2.5 (Section 2.2.2)
$o_j^{\text{J}}(t)$	Obtained share of job j at time t	Equation 2.4 (Section 2.2.2)
P	Number of processors	Section 1.1
ρ	Total load (due to non-permanent jobs)	Equation 3.3 (Section 3.1)
ρ_g	Load due to (non-permanent) jobs of group g	Equation 3.2 (Section 3.1)
ρ_{g*}	Load due to (non-permanent) jobs of group g	Equation 4.34 (Section 4.3.1)
ρ_{*p}	Load on processor p (due to non-permanent jobs)	Equation 4.33 (Section 4.3.1)
ρ_{gp}	Load on processor p due to group g (due to non-permanent jobs)	Equation 4.32 (Section 4.3.1)
ρ_{net}	Network utilization	Section 6.2.2
R_j	Response time of job j	Equation 2.1 (Section 2.2.1)
r_g^{G}	Required group share of group g	Section 2.4.1
S_j	Service time of job j	Section 2.2.1
$\sigma_j^{\text{J}}(t)$	Service rate of job j at time t	Equation 2.3 (Section 2.2.2)
$S^{\mathcal{M}}$	Mean network service time for job migrations	Section 6.2.2
$S^{\mathcal{P}}$	Network service time for probe requests and replies, job placements, and job-migration requests	Section 6.2.2
$\tau^{\mathcal{P}}$	Mean probe period (periodic probing)	Section 6.4
$W_j^{\text{J}}(t)$	Remaining work of job j at time t	Section 2.2.2

w_g	Weight of group g (PPS/GPPS)	Section 3.4.2
w_{gp}	Weight of group g on processos p (MGPPS)	Section 4.2.2
w_{jp}	Weight of job j on processos p (MJPPS)	Section 2.3.2
$X = (X_{gp})$	Partitioning matrix (SVP)	Section 5.5.3
$x_g(t)$	Group priority of group g at time t (UD)	Equation 3.37 (Section 3.2.3)

Appendix D

Summary

Share Scheduling in Distributed Systems

Jan de Jongh

Ph.D. Thesis, February 2002

In this thesis, we study share scheduling in distributed systems. In share scheduling, one attempts to provide groups of jobs with a prespecified share of the system capacity, which is useful in multi-user distributed systems for high-performance computing applications, such as weather forecasting. In such systems, the delivery of shares to jobs or groups of jobs, in other words, have them make progress at certain rates, is more meaningful than traditional objectives, such as the minimization of the job response time.

We focus on two research questions:

1. What are the objectives and performance measures of share scheduling in mathematical terms? This is the subject of Chapter 2.

2. What are suitable policies for share scheduling in uniprocessors, multiprocessors, and distributed systems? These are the subjects of Chapters 3 through 6.

For the purposes of this thesis, the essential differences between a multiprocessor and a distributed system are the costs of communications and of the transfer of a job in progress between two processors. The latter mechanism is called *job migration*. We assume that in multiprocessor systems, communications and migrations are free of cost in terms of network delay and bandwidth. In distributed systems, communications suffer from delays, and job migration is costly, or even impossible.

In Chapter 2, we state the objectives of share scheduling. In our model, the workload of the system consists of jobs, each of which belongs to one of G groups. Essential in the definition of share-scheduling objectives is the notion of *shares*. The *obtained job share* is the fraction of the total system capacity used to serve a specific job. In an analogous way, we define the *obtained group share* of a specific group, and the *total obtained share*. For each of the obtained shares (i.e., job/group/total), we define a target share, the corresponding *feasible share*, which leads to the following three objectives, in decreasing order of importance:

- *Compliance with the feasible group shares:* to keep all obtained group shares close to the respective feasible group share at all times.

- *Minimization of capacity loss:* to keep the total obtained share close to the total feasible share at all times.

- *Internal fairness:* to keep the obtained job share of each job close to its feasible job share at all times.

The total feasible share is defined as the maximum share that can be obtained simultaneously by all jobs present in the system. In other words, the system should put to work as many of the fastest processors as possible. For the definition of the feasible group share, however, a different approach must be taken, because due to the heterogeneity of the system, the groups cannot always be provided with their maximum obtainable share simultaneously. We circumvent this problem by defining the feasible group share as the maximum obtainable share of a group on a homogeneous system with the same number of processors and the same total capacity as the original system. Also, the feasible group share of a group has a fixed upper bound called the *required group share*. The required group shares define how the share-scheduling policies should discriminate among the groups. Finally, we define the feasible job share as the obtained share of the group to which the job belongs, divided by the number of jobs of that group present. Hence, the objective of internal fairness simply implies that the jobs present of a group should be served at equal rates.

In Chapter 3, we study the suitability for share scheduling of some well-known scheduling policies on uniprocessors, such as First-Come First-Served (FCFS), Head-of-the-Line (HOL), Priority Queueing (PQ), and Processor Sharing (PS). We also introduce two new policies specifically designed for share scheduling: the Up-Down (UD) policy, which is non-preemptive, and Group-Priority Processor Sharing (GPPS), which is a processor-sharing policy. On uniprocessors, the share-scheduling objectives are almost impossible to achieve with non-discriminatory policies (such as FCFS and PS), i.e., policies that do not consider to which group a job belongs. In such policies, the delivery of the feasible group shares depends largely on the distribution of the workload (whether consising of permanent of non-permanent jobs, or both) among the groups. On the other hand, policies that provide unlimited preferential treatment to one group over another, such as Priority Queueing and Head-of-the-Line, do not perform very well either. This is because the service to lower-priority groups largely depends on the presence of groups of higher priority, so if the workload consists mainly of groups of higher priority, the low-priority groups will hardly be served. The UD policy is non-preemptive and discriminatory, but unlike HOL, UD discriminates among groups based on their accumulated deviation between the feasible and obtained group shares. Although UD performs better than HOL in some cases, in other cases UD turns into HOL for one or more of the groups, simply because the groups must wait too long before being served, which increases their priorities to unreachable levels for the other groups. In GPPS, all the jobs of a group present on a processor are served *jointly* at rate proportional to the required group share. The share-scheduling performance of GPPS is excellent.

In Chapters 4 through 6, we study share scheduling in systems with more than one processor. In such systems, one needs to decide which processor must serve which job, in

other words, one needs to design the *global scheduling policy*. The *local scheduling policy* decides when to serve the jobs on the processor, and at what rate. In Chapter 4, we study *static* global scheduling policies, i.e., policies in which the decisions taken do not depend on the state of the system, for instance in terms of the number of jobs present in the processors.

In Section 4.2, we consider a model of a multiprocessor system, in which we assume that job migration is available and free of any cost. We derive the necessary and sufficient conditions for the existence of a schedule of a fixed number of jobs on the processors such that each of the jobs is served (on the average) at a prespecified rate. As it turns out, share scheduling on multiprocessors is rather trivial, for instance with the Multiprocessor Group-Priority Processor-Sharing (MGPPS) policy, which provides groups with at least their feasible group shares and never suffers from capacity loss.

In Section 4.3, we consider so-called random-splitting policies, static scheduling policies in which arriving jobs are sent to processors according to fixed probabilities, independent of the state of the system. We consider both the minimization of the capacity loss and the compliance with the feasible group shares. In homogeneous systems, balancing the total load among the processors minimizes the capacity loss. For heterogeneous systems, we cannot derive a general expression for the loads on the processors that minimize the capacity loss, except for the two-processor case. For minimization of the group share deviation, we only consider homogeneous systems, and we introduce two basic policies: in Horizontal Partitioning in Random Splitting (HPRS) every processor receives the same traffic mix; in Vertical Partitioning in Random Splitting (VPRS) entire processors are dedicated to a group. We compare the two strategies and conclude that VPRS has the potential to outperform HPRS, even if the local scheduling policy is GPPS. This indicates the importance of a good global policy for share scheduling.

In Chapters 5 and 6, we study *dynamic* global scheduling policies. Such policies use information on the current state of the system in order to reach scheduling decisions. In Chapter 5, the global scheduling policies have complete and up-to-date knowledge of the system state, and job migration is free of cost. However, we limit the use of job migration to arrival and departure events only. We introduce four dynamic policies for share scheduling, Join Shortest Queue (JSQ), Horizontal Partitioning (HP), Static Vertical Partitioning (SVP), and Dynamic Vertical Partitioning (DVP). In JSQ and HP, the idea is to spread out the jobs among the processors. The difference between JSQ and HP is that JSQ does not consider to which group a job belongs, whereas HP attempts to spread out the jobs of each group. Both JSQ and HP are based upon HPRS. In SVP and DVP, the idea is to reserve entire processors to the groups according to the required group shares, i.e., to partition the set of processors. In SVP, partitioning is done beforehand, while in DVP, partitioning is done at runtime. The latter two policies are based upon VPRS.

From the performance comparison in Chapter 5, we conclude that dynamic policies perform much better than the random-splitting policies HPRS and VPRS. In general, the availability of job migration and the use of GPPS as the local scheduling policy substantially improve the performance. Finally, it turns out that SVP is the best choice for the global scheduling policy.

In Chapter 6, we study the information dependency of the four policies of Chapter 5 and we include the costs of communications and job migrations into the system model. In

distributed systems, certainly large ones, obtaining complete and accurate information on the state of the system at all times, as was the case in Chapter 5, is not very realistic. We therefore adapt the policies such that they only use information obtained by probing a random subset of processors of limited size. We introduce periodic probing and on-demand probing. The advantage of periodic probing is that information on the system state is always available, but always out-of-date to a certain degree. In on-demand probing, we always obtain fresh information for every scheduling event (which may induce more overhead than periodic probing), but arriving jobs (or idle processors) must wait for the probe reply before a scheduling decision can be taken. It turns out that all policies are very robust against the use of partial information. When only about 30% of the processors is probed, the performance is already close to what is achievable with full and up-to-date system information. Also, if using the same amount of network bandwith, on-demand probing outperforms periodic probing. Yet there are points of concern, such as the scalability of the system: as the number of processors increases, so does the load on the network due to job migrations. So, in large systems, the policies should be more reluctant towards the migration of jobs.

In Chapter 7 we state our overall conclusions. Our most important one is that share scheduling in distributed systems is much more complicated than on uniprocessors and on multiprocessors. In distributed systems, the availability of job migration and a good local scheduling policy, such as GPPS, is vital for good share-scheduling performance under all circumstances.

Appendix E

Samenvatting

Share Scheduling in Gedistribueerde Systemen

Jan de Jongh

Proefschrift, februari 2002

In dit proefschrift bestuderen we share scheduling in gedistribueerde systemen. Het doel van share scheduling is groepen jobs te voorzien van een voorafgespecificeerd deel (een "share") van de systeemcapaciteit, hetgeen zinvol is in multi-user gedistribueerde systemen voor rekenintensieve applicaties zoals weersvoorspellingen. In dergelijke systemen is het leveren van shares aan jobs of groepen jobs, met andere woorden, ze te bedienen met een bepaalde snelheid, zinvoller dan traditionele doelstellingen als het minimaliseren van job responstijden.

We richten ons op twee onderzoeksvragen:

1. Wat zijn de doelstellingen en prestatiegrootheden van share scheduling in wiskundige zin? Dit is het onderwerp van Hoofdstuk 2.

2. Wat zijn geschikte policies voor share scheduling op uniprocessoren, multiprocessoren en gedistribueerde systemen. Dit zijn de onderwerpen van Hoofdstukken 3 tot en met 6.

Voorzover dit proefschrift betreft, zijn de essentiële verschillen tussen multiprocessoren en gedistribueerde systemen de kosten van communicatie en van de verplaatsing van een job in uitvoering tussen twee processoren. Dit laatste mechanisme wordt *job migratie* genoemd. We nemen aan dat in een multiprocessor, communicatie en job migraties niets kosten in termen van netwerkvertragingen en bandbreedte. In een gedistribueerd systeem ondervindt communicatie vertragingen, en is job migratie kostbaar, of zelfs onmogelijk.

In Hoofdstuk 2 stellen we de doelstellingen vast van share scheduling. In ons model bestaat de werklast uit jobs die elk tot één van G groepen behoren. Essentieel in de definitie van share-scheduling doelstellingen is het begrip *share*. Het *behaalde job share* is het deel van de totale systeemcapaciteit wat aangewend wordt voor de bediening van een bepaalde job. Op een analoge manier definiëren we het *behaalde groep share* van een bepaalde groep, en het *totaal behaalde share*. Voor elk van de behaalde shares (dus

job/groep/totaal), definiëren we een doel share, het zogenaamde *haalbare share*, zodat we de volgende drie doelstellingen krijgen, in afnemende volgorde van belangrijkheid:

- *Het leveren van de haalbare groep shares:* het op elk moment klein houden van de afwijking tussen de behaalde groep shares en de respectievelijke haalbare group shares.
- *Minimalisatie van het capaciteitsverlies:* het op elk moment klein houden van de afwijking tussen het totale behaalde share en het totale haalbare share.
- *Interne eerlijkheid:* het op elk moment klein houden van de afwijking tussen het behaalde job share en het haalbare job share.

Het totale haalbare share wordt gedefiniëerd als het maximale share dat behaald kan worden door alle jobs in het systeem tezamen. Met andere woorden, het systeem dient zoveel mogelijk van de snelste processoren aan het werk te houden. Voor de definitie van het haalbare groep share moeten we echter een andere aanpak kiezen, omdat vanwege de heterogeniteit van het systeem de groepen niet altijd tegelijkertijd kunnen worden voorzien van het maximale haalbare groep share. We omzeilen dit probleem door het haalbare groep share gelijk te stellen aan het maximaal haalbare groep share in een homogeen systeem met hetzelfde aantal processoren en dezelfde totale capaciteit als het originele systeem. Ook heeft het haalbare groep share een bovengrens, het zogenaamde *benodigde groep share*. De benodigde groep shares definiëren hoe de share-scheduling policies onderscheid dienen te maken tussen de verschillende groepen. Tenslotte definiëren we het haalbare job share als het behaalde share waartoe de job behoort gedeeld door het aantal jobs van deze groep in het systeem. Aldus wordt de doelstelling van interne eerlijkheid simpelweg het gelijkelijk verdelen van het behaalde groep share onder de aanwezige jobs van de desbetreffende groep.

In Hoofdstuk 3 bestuderen we de geschiktheid voor share scheduling van een aantal bekende scheduling policies voor uniprocessoren, zoals First-Come First-Served (FCFS), Head-of-the-Line (HOL), Priority Queueing (PQ) en Processor Sharing (PS). We introduceren ook twee policies die speciaal ontworpen zijn voor schare scheduling: de Up-Down (UD) policy, die niet preemptief is, en Group-Priority Processor Sharing (GPPS), wat een processor-sharing policy is. Op uniprocessoren zijn de doelstellingen van share scheduling bijna onmogelijk te behalen met niet-discriminatoire policies (zoals FCFS en PS), dit zijn policies waarvoor het niet uitmaakt tot welke groep een job behoort. Met dergelijke policies hangt de levering van het haalbare groep share teveel af van de verdeling van de werklast (of dit nu permanente of niet-permanent jobs zijn, of beide) over de groepen. Anderzijds voldoen policies die ongelimiteerde voorrang geven aan één groep over de andere, zoals Priority Queueing en Head-of-the-Line, ook niet goed. Dit heeft als oorzaak dat de bediening van groepen met lage prioriteit teveel afhangt van de aanwezigheid van groepen met hogere prioriteit, dus als de werklast voornamelijk bestaat uit groepen met hogere prioriteit, zullen de groepen met lage prioriteit nauwelijks bediend worden. De UD policy is niet preemptief en discriminatoir, doch in tegenstelling tot HOL discrimineert UD tussen de groepen op basis van de geaccumuleerde afwijking tussen de haalbare en behaalde groep shares. Hoewel UD beter presteert dan HOL in sommige gevallen, komt de werking van UD in andere gevallen gewoon neer op HOL, omdat groepen soms te lang moeten wachten voordat ze in bediening worden genomen, wat hun prioriteit doet

toenemen tot voor de andere groepen onhaalbare hoogtes. In GPPS worden alle jobs die van een groep aanwezig zijn *gezamenlijk* bediend met een snelheid recht evenredig met het benodigde groep share. De prestaties van GPPS met betrekking tot share scheduling zijn uitmuntend.

In Hoofdstukken 4 tot en met 6 bestuderen we share scheduling in systemen met meer dan één processor. In dergelijke systemen moet men beslissen welke processor welke job bedient, m.a.w., men dient de *globale scheduling policy* te ontwerpen. De *lokale scheduling policy* beslist wanneer een job wordt bediend op de processor, en met welke snelheid. In Hoofdstuk 4 bestuderen we statische globale scheduling policies, dit zijn policies waarin de beslissingen niet afhangen van de toestand van het systeem, bijvoorbeeld in termen van de aantallen jobs op de processoren.

In Sectie 4.2 beschouwen we een model van een multiprocessor met gratis job migratie. We leiden de noodzakelijke en voldoende voorwaarden af voor het bestaan van een bedieningsschema zodat een vast aantal jobs wordt bediend (gemiddeld) op voorgespecificeerde snelheden. Het blijkt dat share scheduling op multiprocessoren relatief eenvoudig is, bijvoorbeeld met de Multiprocessor Group-Priority Processor-Sharing (MGPPS) policy, welke jobs voorziet van tenminste hun haalbare groep shares, en nooit last heeft van capaciteitsverlies.

In Sectie 4.3 beschouwen we zogenaamde random-splitting policies, dit zijn statische scheduling policies waarin aankomende jobs naar de processoren worden gestuurd volgens vaste waarschijnlijkheden, onafhankelijk van de toestand van het systeem. We beschouwen zowel minimalisatie van het capaciteitsverlies als de levering van de haalbare groep shares. In homogene systemen wordt het capaciteitsverlies geminimaliseerd indien de werklast wordt gebalanceerd over de processoren. Voor heterogene systemen kunnen we geen algemene uitdrukking afleiden voor de verdeling van de werklast zodat het capaciteitsverlies wordt geminimaliseerd, met uitzondering van het geval van twee processoren. Voor het leveren van de haalbare groep shares beschouwen we alleen homogene systemen, en introduceren we twee policies: in Horizontal Partitioning in Random Splitting (HPRS) ontvangt elke processor hetzelfde verkeersaanbod; in Vertical Partitioning in Random Splitting (VPRS) worden processoren zoveel mogelijk gereserveerd voor de bediening van jobs van een enkele groep. We vergelijken deze twee strategieën en concluderen dat VPRS in veel gevallen beter presteert dan HPRS, zelfs indien de lokale policy GPPS is. Dit onderstreept het belang van een goede globale policy.

In hoofdstukken 5 en 6 bestuderen we *dynamische* globale scheduling policies. Dergelijke policies gebruiken informatie omtrent de huidige toestand van het systeem in het beslissingsproces. In Hoofdstuk 5 hebben de globale scheduling policies complete en actuele informatie omtrent de systeemtoestand, en is job migratie gratis. Echter, we beperken het gebruik van job migratie tot aankomst- en vertrekmomenten van jobs. We introduceren vier dynamische policies voor share scheduling, Join Shortest Queue (JSQ), Horizontal Partitioning (HP), Static Vertical Partitioning (SVP), en Dynamic Vertical Partitioning (DVP). In JSQ en HP is het idee om de jobs te verspreiden over de processoren. Het verschil tussen JSQ en HP is dat JSQ geen rekening houdt met de groep waartoe een job behoort, terwijl HP probeert de jobs van elke groep te spreiden. Zowel JSQ als HP zijn gebaseerd op HPRS. In SVP en DVP is het idee om processoren te reserveren voor groepen in verhouding tot het benodigde group share, m.a.w. de processoren te partitioneren. In SVP gebeurt het partitioneren vooraf, in DVP gebeurt het

tijdens uitvoering. De twee laatste policies zijn gebaseerd op VPRS.

Uit de prestatie-vergelijking in Hoofdstuk 5 concluderen we dat dynamische policies veel beter presteren dan de random-splitting policies HPRS en VPRS. In het algemeen verbeteren de mogelijkheid van job migratie en het gebruik van GPPS als de locale scheduling policy de prestatie behoorlijk. Tenslotte blijkt SVP de beste keuze voor de globale scheduling policy.

In Hoofdstuk 6 bestuderen we de informatie-afhankelijkheid van de vier policies uit Hoofdstuk 5, en voegen we kosten van communicatie en job migraties toe aan het systeemmodel. In gedistribueerde systemen, zeker in grote, is het verkrijgen van complete en actuele informatie over de systeemtoestand, zoals het geval was in Hoofdstuk 5, niet erg realistisch. Daarom passen we de policies aan zodat deze nog slechts informatie gebruiken die wordt verkregen door het peilen van een willekeurige, in grootte beperkte deelverzameling van de processoren. We introduceren periodieke peiling en peiling op aanvraag. Het voordeel van periodieke peiling is dat informatie omtrent de systeemtoestand altijd beschikbaar is, maar ook altijd in min of meerdere mate gedateerd is. In peiling op aanvraag verkrijgen we altijd verse informatie voor elke beslissing (hetgeen meer kosten met zich mee kan brengen dan periodieke peiling), maar moeten aankomende jobs (of leeglopende processoren) wachten op het resultaat van de peiling voordat een schedulingbeslissing kan worden genomen. Het blijkt dat de policies zeer goed bestand zijn tegen het gebruik van onvolledige informatie. Indien slechts 30% van de processoren wordt gepeild, is de prestatie al dichtbij het maximaal haalbare met complete en actuele informatie. Verder presteert peiling op aanvraag beter dan periodieke peiling onder gelijke belasting van het netwerk. Toch zijn er een aantal punten van zorg, zoals de schaalbaarheid van het systeem: indien het aantal processoren toeneemt, neemt ook de netwerkbelasting ten gevolge van job migraties toe. In grote systemen zullen de policies dus terughoudender moeten zijn in het migreren van jobs.

In Hoofdstuk 7 staan de algemene conclusies. Eén van de belangrijkste is dat share scheduling in gedistribueerde systemen veel complexer is dan op uniprocessoren en multiprocessoren. In gedistribueerde systemen zijn de beschikbaarheid van job migratie en van een goede lokale policy, zoals GPPS, van vitaal belang voor goede share-scheduling prestaties onder alle omstandigheden.

Appendix F

Acknowledgments

Those who wish to be,
must put aside the alienation,
get on with the fascination,
the real relation,
the underlying theme.

Rush, *Limelight.*

Ik wil een groot aantal mensen bedanken voor hun bijdrage aan de totstandkoming van dit proefschrift. In de eerste plaats mijn promotor, prof.dr.ir. H.J. Sips. Henk, je nuchterheid, zakelijkheid en snelheid zullen een groot voorbeeld voor me blijven. Mijn toegevoegd promotor, dr.ir. D.H.J. Epema, was er vanaf het begin bij, en zal een aantal hoofdstukken nu wel uit zijn hoofd kennen. Dick, bedankt voor het door mij veel te lang op de proef gestelde geduld en vertrouwen. Daarnaast gaan mijn dank en gedachten uit naar prof.dr. I.S. Herschberg, bij wie ik aanvankelijk mijn onderzoek begon. Bob, je hebt het eind niet mogen meemaken. Geloof me, ik zal nog vaak het aantal manieren tellen waarop een zin gemisinterpreteerd kan worden. Waarschijnlijk zal ook ik er dan om kunnen lachen, mocht dit (wat?) afwijken van het (hopelijk) beoogde aantal van één. Ik dank de leden van de promotiecommissie voor het nauwgezet en kritisch doornemen van dit proefschrift. I express my thanks to prof. Miron Livny for reviewing the manuscript, and for giving birth to the CONDOR system, along with the many research opportunities it has provided.

Dr. René van Dantzig dank ik voor de samenwerking in de CONDOR projecten, en de geboden kijkjes in de wereld van de hoge-energiefysica. Verder bedank ik dr. Albert Schoute, die het vuur voor prestatie-analyse in mij aanwakkerde.

Een complete alinea van dank gaat uit naar dr. E. Roos Lindgreen R.E. Edo, de jaren dat je mijn collega was in Delft doen mij nog het meest denken aan het voorbijschieten van een raket op zeer korte afstand. We moeten het "even een biertje pakken" maar weer eens snel oppakken. Of een paar snaar- dan wel vingerbrekende licks uitwisselen.

De volgende (voormalige) studenten aan de TU Delft hebben een grote bijdrage geleverd aan het in dit proefschrift beschreven onderzoek: Merijn Broeren, Tom Dokoupil, Xander Evers, Péter Moskovits, Bas Muilwijk, Ralph Quapp, Peter van Sebille, Erik Slagter, Pieter Spronck, Jan Zeinstra. Ook bedank ik mijn voormalige collegae aan de TU Delft: Jan Heijnsdijk, Jaap Aalders, Joost Mebius, John Mugge, Paulo Anita,

Lieneke van de Kwaak, Trudie Stoute, Toos Brussee. Met weemoed denk ik nog wel eens terug aan de enorme verscheidenheid aan onderwerpen die de revue passeerden tijdens het leerstoeloverleg. Verder dank aan prof.dr. F.C. Schoute, Borut Stavrov, Alexander van Lomwel en Ronald in 't Velt voor de samenwerking in het BNET project.

Ik bedank het management en de (voormalig) medewerkers van TNO voor de hulp en de interesse die ik heb ontvangen. In het bijzonder (doch allesbehalve volledig) Jannik Daemen, René Struik en Thijs Veugen voor hun bijdrage aan hoofdstuk 4, Johan Pouwelse voor de enerverende discussies, Frank van Aken, Cor van Dam, Kurt "dim" van Dortmont, Jeroen Dezaire, Dick Fikkert, Daan Kloet, Henk Knol, Marcel van der Lee, Wouter Lotens, John Nieboer, Paula Smeele, Olaf Tettero, Frank van Vliet, Klaas de Waal, Jeroen van de Water en Danny Zwaard,

Van de Koninklijke Landmacht bedank ik Lkol. J. Meijer, Maj. H.J. Wendrich, KaptMarns H. Otter, en AAO T. Janssen van het buro SMP (Soldier Modernisation Programme).

Mijn familie en schoonfamilie bedank ik voor het begrip dat ik meerdere verjaardagen heb moeten missen: Cees, Dien, Theo, Truus, Elianne, Alex, Anne-Marie, Ronald, Frans, Miranda, Ino, Lucia, Richa, Eric, Maris, Pauline, Corné. Overigens, moeder Paap, wees gerust, ik heb geturfd. En nogmaals speciale dank voor de immens gave skivakantie.

Van mijn neef John moesten we veel te vroeg afscheid nemen. John, nogmaals bedankt voor alles wat je in je leven hebt gegeven.

Natuurlijk gaat dank uit naar mijn vrienden. Te vaak werden jullie goedbedoelde vragen naar de afronding beantwoord met tang-constructies met een opmerkelijk hoog "zicht-beperkt-tot-vijftig-meter" gehalte, waarvoor met terugwerkende kracht oprechte excuses. Op voornaam gesorteerd: Annet, Bert van Dijk, Bert de Vries, de leden van de Bujinkan Dojo Rotterdam, Evelyn, Fred, Geneviève (voor real-time SMS), Gerard, Harry "HEC" (voor de halfgeleiders "met"), Hermine (nu nog de slingers ophangen, buuv), Hester, Ko, Ludo, Lydi, Maddy, Mirjam, Peter Bessems, Peter "PGF" Feij, Remco, alsmede de overige leden van de Lighthouse Contest Group, in het bijzonder Mar (ik hoop dat ze daar ook contesten). Ook dank ik mijn fantastische buren aan de Vrijenbanselaan.

I'm very grateful to Anamarija Boban for telling me her remarkable and inspiring story. I envy your freedom, and I hope they'll never cage you.

My special friend Susan Young both demanded and deserves her own paragraph at the very least (they told me I was pushing it when I requested the separate appendix). Susan, thanks for the many things you taught me, and for providing me with the pepper when I needed it. Good luck with your business.

Ook dank aan de twee gitaren en het arsenaal aan DSP's die mij door de moeilijkste perioden hebben geholpen. Ik hoop dat de subtiliteit met de jaren komt, doch vrees het ergste.

Tenslotte (maar dat is dan ook een speciale plaats) ben ik oneindig veel dank verschuldigd aan Josien "pasta-tijger" Paap, Franka "nee, papa, dat zijn geen MIDI-voetpedalen, dat is een garage" de Jongh en Michel "uh uh" de Jongh. En ja, het embargo is eindelijk opgeheven; jullie mogen nu weer vragen "wanneer het af is," en "Nijntje" installeren.

domo arigato gozaimashita
Delft, december 2001
Jan de Jongh

Appendix G

About the Author

Jan (J.F.C.M.) de Jongh was born in Raamsdonksveer, The Netherlands, on October 7, 1965. He finished Athenaeum at the Katholieke Scholengemeenschap Etten-Leur e.o. in 1984, and obtained the M.Sc. degree in Electrical Engineering from Twenthe University of Technology in 1990. For his M.Sc. thesis, he designed and realized parts of a distributed operating systems for transputers. He fulfilled his military service from 1990 to 1991, and from 1991 to 1996 he was a Ph.D. candidate at Delft University of Technology, with the Faculty of Mathematics and Computer Science. In the meantime, as a freelancer he gave lectures on Computer Organization, Data Communications and Networks, and Network Management. Since 1996, he is with the Insitute of Appied Physics TNO at the Hague, The Netherlands, as a project manager and system architect. He worked on several multidisciplinary system-engineering projects for the Royal Netherlands Armed Forces.

His research interests are in performance analysis, scheduling, discrete-event simulation, distributed systems, and wireless ad-hoc networks. Other interests include software engineering, electromagnetic-wave propagation, signal theory, and astronomy. Jan is a radio amateur and stretches guitar strings beyond their specified operating range.

ISBN 90-901-5128-1