Decentralising Resource Management in Operating Systems

Rolf Neugebauer

Department of Computing Science
University of Glasgow

Submitted in partial satisfaction of the regulation for the award of the degree of Doctor of Philosophy

April 2003

Summary

Managing resources is one of the most important tasks an operating system has to perform. Managing resources involves policy decisions, such as how much of a resource should be allocated to competing resource consumers, and mechanisms, such as how to multiplex competing consumers. Traditionally, resource management policies are implemented by centralised entities, either individually per resource, by a scheduling algorithm or, at a higher level, using resource or Quality of Service (QoS) managers. Resources are "magically" allocated to consumers with little or no involvement of the consumers of the resources. Often, resources are virtualised to give competing consumers the impression of being the only consumer of that resource.

This dissertation explores operating system mechanisms to allow resource-aware applications to be involved in the process of managing resources under the premise that these applications (1) potentially have some (implicit) notion of their future resource demands and (2) can adapt their resource demands. The general idea is to provide feedback to resource-aware applications so that they can proactively participate in the management of resources. This approach has the benefit that resource management policies can be removed from central entities and the operating system has only to provide mechanism. Furthermore, in contrast to centralised approaches, application specific features can be more easily exploited.

To achieve this aim, I propose to deploy a microeconomic theory, namely congestion or shadow pricing, which has recently received attention for managing congestion in communication networks. Applications are charged based on the potential "damage" they cause to other consumers by using resources. Consumers interpret these congestion charges as feedback signals which they use to adjust their resource consumption. It can be shown theoretically that such a system with consumers merely acting in their own self-interest will converge to a social optimum.

This dissertation focuses on the operating system mechanisms required to decentralise resource management this way. In particular it identifies four mechanisms: pricing & charging, credit accounting, resource usage accounting, and multiplexing. While the latter two are mechanisms generally required for the accurate management of resources, pricing & charging and credit accounting present novel mechanisms. It is argued that congestion prices are the correct economic model in this context and provide appropriate feedback to applications. The credit accounting mechanism is necessary to ensure the overall stability of the system by assigning value to credits.

With two prototype implementations it is demonstrated that these mechanisms can be easily implemented and that the decentralised resource management yields the desired results, namely, achieving a socially optimal resource allocation.

Preface

Except where otherwise stated in the text, this dissertation is the author's own work and is not the outcome of work done in collaboration.
This dissertation is not substantially the same as any I have submitted for a degree or diploma or any other qualification at any other university.
No part of my dissertation has already been, or is currently being submitted for any other degree, diploma or other qualification.
Some of the work described in this dissertation has been published, and is described in references [Neu99, NM00, NM01, Neu01].

Acknowledgements

I would like to thank my supervisors Derek McAuley, Richard Black, and Peter Dickman (in chronological order) for their encouragement, valuable advice and discussions during the course of my research. Ray Welland, my professorial overseer, also deserves credit, not only for arranging additional funding for me, but also for temporarily stepping in when Mac left Glasgow.

I am indebted to Huw Evans, Peter Dickam, and Derek McAuley who have read earlier drafts of this dissertation. Peter and, in particular, Huw deserve special mentioning for their detailed comments on the final drafts of this dissertation.

Finally, I wish to thank all those who made my stay in Glasgow so pleasant and enjoyable. In particular: Ian and Huw, for providing, most persistently, the necessary distractions; Rory, Ian, and Lyndell for having been older office mates; Michael, Matt, Greg, Dani, Fabian, and many others. Thanks!

This work was partly funded by the European Union and the Department of Computing Science, University of Glasgow.

Contents

List of Figures

Chapter 1
Introduction

Managing resources is one of the main services provided by operating systems. An operating system has to allocate resources such as CPU, physical and virtual memory, disk and network bandwidth, between a number of competing tasks with different and variable resource demands of varying importance. Resource management can be divided into two main tasks: deciding how many resources should be allocated to competing consumers and the actual multiplexing of those resources. The former is referred to as *policy* and the latter as *mechanism*. Policies are typically centralised within a single entity, often for each resource individually, and aim to satisfy both the user's preferences and the application's resource requirements.

Resource management is either performed at a low-level within the operating system in the form of scheduling algorithms or at a higher level in the form of Quality of Service (QoS) managers. Scheduling algorithms only deal with individual resources, and, in general purpose operating systems, often implement mechanisms and policies. They often only provide crude and limited control to users and applications over these decisions. QoS managers, on the other hand, aim to meet user and application specified high-level goals, such as guaranteed, user-perceived levels of service, while optimising the overall resource utilisation. This typically requires both user and application requirements to be specified accurately and in advance — in itself a non-trivial task — and these requirements to be mapped to low-level resource allocations. Typically, neither of these approaches inform applications of changes in resource allocation — they manage resources "behind the scenes" — and only a fixed set of resource allocation policies is implemented.

In this dissertation I propose a radically different approach in order to support resource-aware, adaptive applications. For these applications, almost all resource management *policies* are eradicated from central entities, such as the operating system kernel or resource managers, and the operating system only provides some basic resource management *mechanisms* which allow these applications to actively manage resources themselves in a cooperative or competitive fashion. The mechanisms provided by each resource are based on a microeconomic theory, namely *shadow* or *congestion prices*. Tasks, or consumers of resources, are charged based on the congestion they cause to the system by consuming resources. For example, if a resource is only slightly loaded a task consuming some amount of that resource is not causing any "damage" to other consumers of the same resource. If, however, the load on that resource becomes too high, i.e., the resource becomes congested, a consumer, while receiving a sufficient amount of that resource for itself, may have a negative effect on other consumers. In economics, this is called an *externality* and shadow prices can be used to charge or tax consumers for this external effect — economists say they internalise the externality. Consumers interpret congestion charges as *feedback* signals, based on which they adjust their resource consumption. It can be shown, mathematically, that such a system with consumers merely acting in their own self-interest will converge to a social optimum (section 3.2 provides a detailed review).

The main contribution of this dissertation is the application of this theory to resources managed by operating systems. Furthermore, using two prototype implementations, it is demonstrated that the application of congestion prices to CPU resources is feasible and yields results as indicated by the theoretical treatment of the model.

In general, the idea of engaging consumers in the management of resources is not novel, albeit not widely deployed (see chapter 2 and in particular section 2.2), and researchers have proposed the use of economic ideas in this context before. Indeed this was quite popular in the context of large scale time-sharing systems (e.g., [Sut68, Nie70, Lar75]) with users making decisions on when to run jobs based on demand based prices. However, the approach presented in this dissertation follows a particular economic model more rigorously, applying the model at different time-scales, targeting different application domains, and encouraging the applications rather than the user to make resource allocation decisions.

In the remainder of this chapter a high-level synopsis of this dissertation is presented. The next section provides a basic overview of the key components of the resource management framework followed by a summary of the key operating system mechanisms provided by the architecture. Section 1.3 then motivates the architecture by providing some sample application areas in which it may be particular beneficial. Having set the context of this dissertation, in section 1.4 the thesis statement is formulated. This chapter is concluded with an overview of the remainder of this dissertation.

1.1 Overview

This dissertation proposes a new resource management model for operating systems in which applications play a key role in making resource management policy decisions. Figure 1.1 presents a high-level overview of this decentralised resource management architecture with the key components highlighted in blue. Each resource is extended by a pricing and charging module, which uses the resource usage accounting to determine prices and to charge applications for their resource usage. Each process has associated with it an account from which these charges are deducted. Users allocate credits to be placed in an application's account depending on how much they value the utility provided by that application. Each participating application has an adaption module which monitors the credit account and therefore the current charging rate and adjusts the resource consumption of that application. Users may aid this task by providing the adaption module with detailed, application-specific preferences for different modes of operation and the adaption module may be linked with the rest of the application code to adjust the mode of operation based on the user preferences and the observed charges.

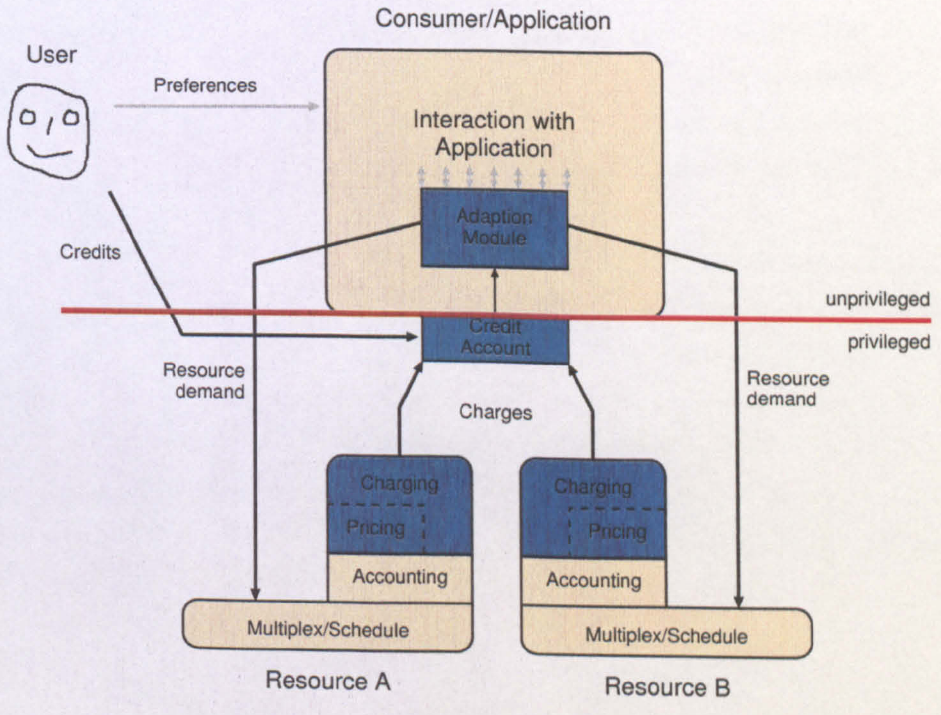


Figure 1.1: Overview of the decentralised resource management architecture

The diagram also indicates the separation between the privileged and unprivileged parts of the architecture[1]. The aim is to only provide mechanisms within the privileged parts of the architecture and to leave policy decisions to be made cooperatively amongst the consumers of resources as unprivileged operations. This allows the consumers to choose policies suitable for the needs of the applications and the users.

In this dissertation, the architecture and the mechanisms are described in economic terms. Prices and resulting charges are based on the theory of congestion prices and applications are presented as consumers of resources providing utility to users. Applications are charged for their resource consumption and attempt to maximise their utility minus the cost of providing this utility. A general economic framework was chosen to describe the architecture as it helps to leverage the substantial body of theoretical work in that area and because it has received much attention recently as an alternative scheme for flow control in computer networks.

However, the architecture could also be described in terms of control theory. Essentially, charges are feedback signals and the adaption module are controllers. The resource demand they impose on the resource is essentially an actuator. With these three components (feedback signal, adaption, and actuators) a traditional feedback control system can be constructed. However, the presence of multiple feedback loops, one for every application, interacting with each other at each resource, complicates matters significantly from a control theoretical point. Alternatively, the architecture could also be described in game theoretical terms, for example, as a variant of Axelrod's repeated Prisoners Dilemma [Axe90].

Stratford and Mortier [SM99a, SM99b] have independently proposed a very similar resource management architecture which is slightly more complex in that it incorporates the concepts of *contracts* for resource and credit allocations over variable time-scales; *contract trading*, allowing consumers to sell resources from their contracts back to resource managers; and *user agents*, implementing policy decisions on behalf of the user. Unfortunately, in [SM99a, SM99b] Stratford and Mortier do not provide any details on the implementation of their design. In particular, no details are provided on the pricing mechanisms, the required operating system mechanism, and, indeed, a detailed evaluation of the system.

[1]The terms privileged and unprivileged are used rather than kernel and user-space as the architecture is not tied to a particular operating system structure.

1.2 The mechanisms

This dissertation is mainly concerned with the operating system mechanisms necessary to decentralise resource management as outlined above. In summary, the mechanisms provided are as follows:

Pricing & Charging: The main mechanism of the proposed architecture is to send the correct feedback signals in the form of charges to consumers and to provide them with an incentive to adjust their resource consumption. I will argue that *shadow prices*, which capture the cost of congestion, and charges, based on the individual responsibility for generating congestion cost, provide the correct feedback signals to resource consumers.

Credit accounts: To provide consumers with an incentive to adjust their resource demands according to the charges they incur the availability of credits must be managed. A mechanism is required to attach a *value* to credits for applications and users if credits are not directly translated into real money[2].

Resource accounting: Resource management in general requires that resource usage can be accurately accounted to individual consumers. Unfortunately, many general purpose operating systems provide insufficient mechanisms for accounting resource usage. While resource accounting is not a mechanism especially provided by the decentralised resource management architecture, an accurate accounting mechanism is nonetheless required.

Multiplexing: With multiple clients consuming a resource, a mechanism is required to multiplex these clients. This task is usually performed by a scheduler, which typically also implements mid- to long-term resource allocation policies. For the decentralised resource management architecture, mechanisms from real-time scheduling are deployed to multiplex client requests in the short term, but the congestion pricing mechanisms cooperatively determine mid- to long-term resource allocations.

In chapter 4 these mechanisms are discussed in more detail and chapter 5 describes two implementations of these mechanisms in different environments.

[2]Following Ricardo's simple 1817 dictum in "Principles of Political Economy and Taxation": "It is not necessary that paper [money] should be payable in specie [i.e., gold] to secure its value, it is only necessary that its quantity should be regulated" (cited from "Crisp and even", *The Economist*, Dec. 22nd 2001 – Jan. 4th 2002, pp. 91–93.)

1.3 Application scenarios

The resource management architecture proposed in this dissertation advocates the active involvement of applications in the resource management process. There are two application domains where this would be particularly useful: multimedia applications and consolidated server systems.

Many multimedia applications have high and highly variable resource demands with strict timing requirements. This makes it difficult to manage resources for them efficiently. A significant part of the resource management research, in particular into QoS management, has focused on multimedia applications (see section 2.2 for an overview). Multimedia applications also have the interesting property of being highly adaptable. Typically, applications can operate in different modes of operation along different dimensions. A video display application, for example, can reduce the frame-rate, the quality of the decoded video, and the size of the decoded frame. These different modes can have a significant impact on the overall resource demands of the application.

In a traditional operating system environment, an adaptive multimedia application has to frequently monitor its performance and adjust its mode of operation. With the feedback provided by the architecture proposed in this dissertation, applications can take a much more proactive position, i.e., adjust their mode of operation *before* the performance in the current mode deteriorates. QoS managers, on the other hand, typically require the modes of operation and possibly resource demands for them, to be specified explicitly and in advance. Some approaches also allow users to express preferences for different modes of operation and the QoS manager then attempts to maximise the users' utility from the system. With the congestion pricing based approach much less specification is required from the applications. Instead, more intelligence is placed in the applications, with application developers and users choosing from a wide range of application specific adaption strategies. These strategies can range from very simplistic approaches to fairly sophisticated ones leveraging full application internal knowledge combined with additional user input. None of this information needs to be made explicit to a central entity, as required by many QoS managers.

A second application domain in which the decentralised resource management approach can be deployed beneficially is server systems, though adaption may be performed on different timescales. In this area, a trend to consolidate servers into fewer or even single servers can be observed. Rather than managing individual computers, each responsible for an individual server application, multiple server applications are executed on the same server with additional software providing performance isolation between them[3]. This is also of particular interest to Application Service Providers (ASPs), providing services to potentially competing customers, or, in fact, for the more radical proposals of Xenoservers [RPM+99, SM99a] which aims to allow networked applications to be executed on public servers closer to the data they might require.

In such consolidated server environments, a highly dynamic allocation of resources to competing consumers is desirable as the load on a system may change rapidly and, more significantly, bursty and very high loads can be generated. Furthermore, some server applications, such as web servers or e-commerce sites, towards which these server systems are targeted, can adjust their resource requirements under high load, by, for example, giving preferential treatment only to certain customers, dynamically adjusting the content they provide, or scheduling background maintenance tasks at off-peak times. For these applications, especially in a competitive environment, such as that at an ASP, providing feedback to the application enables them to make informed decisions about these options. Furthermore, a pricing-based approach, where charges can be related to real costs for customers, may be particular appealing to ASPs, and may provide the appropriate incentive to consumers to adjust their resource consumption. Moreover, it has been shown that, at least theoretically, congestion charges would generate the revenue necessary for capacity expansion [MV95a]. Interestingly, a recent paper [CAT+01] also proposes an economic-based approach for managing energy usage in Internet hosting centres.

In general, I view decentralised resource management as an enabling technology, which may be used in a variety of scenarios, some of which have been outlined above. However, this dissertation focuses mainly on the operating system mechanisms which are required to enable these scenarios. A detailed study of different application scenarios is beyond the scope of this work.

It needs to be stressed that active application involvement in the resource management process is not mandated by the decentralised architecture. Depending on the implementation of the framework, two options are conceivable: either default policies are provided for applications that are not capable of participating (see section 4.4), or, as in the FreeBSD prototype described in section 5.2, feedback is only provided to applications that are interested in actively participating in the management of their resources.

[3]Examples of such system software are Linux on IBM Mainframes (`http://www.ibm.com/s390/linux/`), IBM's LPAR technology [IBM01], VMware for servers (`http://www.vmware.com/products/server/`), Aurema's ARMTech (`http://www.aurema.com/`), or Sun's Solaris Resource Manager (`http://www.sun.com/software/solaris/ds/ds-srm/`).

1.4 Thesis statement

Resource management in operating systems has traditionally been performed by centralised entities implementing a fixed set of resource management policies with little or no involvement of the consumers of those resources. While it is accepted that a wide range of applications can adjust their resource demands, operating system resources are typically managed "magically" from an application's point of view, and adaptive applications are forced to observe performance degradation before being able to adapt to changing resource availability.

I assert that these applications should be more actively involved in the management of the resources they consume, in order to allow them to adapt to changing resource availability proactively. By providing feedback to applications and therefore expecting them to adapt, almost all higher-level resource management policies can be removed from privileged operating system entities, leaving the core OS to only provide simple, well understood, multiplexing mechanisms. Furthermore, I assert that congestion prices are applicable to all resources typically managed by an operating system and are suitable for providing the correct feedback to applications. This dissertation addresses the operating system mechanisms required to enable applications to participate in the management of resources.

1.5 Outline of the dissertation

Chapter 2 provides an overview of existing resource management approaches in operating systems. It focuses on scheduling algorithms and QoS management approaches. It also includes a general discussion on the impact of the overall operating system structure on its ability to manage resources accurately and the chapter describes a number of alternative approaches.

Chapter 3 serves two purposes: Firstly it provides an overview of general economic concepts and previous proposals for their application in managing computational resources. And, secondly, it introduces the general concept of congestion or shadow prices more formally, by reviewing the related work of its application to managing congestion in computer networks.

Chapter 4 is the core chapter of this dissertation. It presents the proposed architecture in detail and discusses its application in the context of operating systems. The chapter primarily uses CPU resource management as an example resource, but other resources are discussed as well.

Chapter 5 provides the background on, and a detailed description of, two implementations of the decentralised resource management architecture. A general simulation environment for scheduling algorithms and a prototype implementation under FreeBSD are presented.

Chapter 6 provides a detailed evaluation of these prototypes. The evaluation demonstrates that resource are shared in a weighted proportionally fair fashion despite each consumer attempting to maximise their own utility.

Chapter 7 summarises the main arguments of this dissertation, discusses future work, and presents the conclusions.

Chapter 2
Resource management in operating systems

Resource management in operating systems is performed at various levels. Traditionally, each resource is scheduled independently of other resources and scheduling algorithms are responsible for both multiplexing and resource allocation policies. Especially with the emergence of soft real-time multimedia applications this approach has been considered insufficient to provide predictable performance to these applications. As a solution, researchers have proposed high-level, integrated QoS managers, which aim at providing resource guarantees to complex multimedia applications.

In this chapter several of these approaches to resource management are reviewed to set the context for the novel decentralised resource management architecture. This review is structured in three parts. First, in section 2.1 an overview of CPU scheduling algorithms is given. Scheduling algorithms provide the low-level mechanisms and policies for allocating resources. In section 2.2 higher-level approaches, namely approaches to QoS management, are described. While these two sections cover the related work with respect to the actual management of resources, the structure of an operating system can also have a significant impact on how (especially on how accurately) resources can be managed. In section 2.3 these general operating systems issues are discussed.

2.1 CPU Scheduling algorithms

CPU scheduling performs the function of multiplexing the CPU resource among application programs. Applications, or more specifically, application mixes have largely varying demands on how the CPU should be multiplexed. Not surprisingly, there is a large body of research focusing on scheduling algorithms. These range from scheduling algorithms, traditionally used in general purpose operating systems[1], to hard real-time scheduling algorithms designed for specific narrowly focused (embedded) operating systems. This section first provides a general taxonomy of these scheduling algorithms (loosely based on that given in [RJS00]) and then discusses some examples in detail.

Scheduling algorithms for general purpose operating systems have to provide good interactive performance while ensuring high throughput for batch processing applications and some form of fairness between competing processes. When used for dedicated server applications, as is often the case, both fast response times for requests and high general system throughput are required. For dedicated server systems, overall fairness of resource allocations is not necessarily a requirement. However, for consolidated server systems, executing multiple server applications, fairness and load isolation are important issues.

For hard real-time scheduling algorithms it is paramount that all hard deadlines are met. This is often achieved by carefully engineering a system for a specific task and then statically analysing the resource requirements of the typically small set of processes executing on the system. As the a priori analysis of resource demands is sometimes difficult, if not impossible, resources are typically generously over-provisioned.

Some application domains, in particular multimedia applications, have timeliness requirements on resource allocations which are less stringent than those of hard real-time systems. While having deadlines, e.g., determined by a frame rate, deadline misses are undesirable but maybe acceptable to these applications. Applications with such a property are termed soft real-time applications. With the increased demand for multimedia applications on users desktops it is desirable to support them together with more conventional application mixes as provided by general purpose scheduling algorithms.

While scheduling algorithms can be categorised by their suitability for the above application scenarios, it is useful, when analysing a given scheduling algorithm, to distinguish between its steady state behaviour and its behaviour at mode changes. The steady state behaviour is given with a fixed task set while mode changes are defined by changes to the task set (tasks joining or leaving) and (user initiated) changes of scheduling parameters.

[1]The term "general purpose OS" is used loosely to describe operating systems for personal computers, workstations and servers. Most popular operating systems, such as the OS families of Microsoft Windows, Linux and other Unix style operating systems fall into this category.

In steady state behaviour, a scheduler should distinguish between importance and urgency [NL97]. Importance should determine the overall share of a resource a task is allocated while urgency should be used to make the actual scheduling decision, i.e., which task to schedule next. Importance can be indicated by the user, e.g., through the means of priorities, while urgency may be determined by the past resource consumption of a task, e.g., to give preference to unblocking interactive tasks, or the closeness of a task deadline.

Next, different categories of scheduling algorithms are reviewed, namely general purpose scheduling algorithms used in traditional operating system, classic hard real-time scheduling algorithms, scheduling algorithms which provide CPU reservations, and scheduling algorithms which emphasise proportional fair sharing of CPU resources. For each category the application scenarios supported and their behaviour in steady state and during mode changes are investigated. Then several specific scheduling algorithms are reviewed in detail.

2.1.1 "Traditional" general purpose schedulers

Most general purpose operating systems deploy scheduling algorithms aimed at application mixes of interactive and batch processing applications. These systems typically deploy priority scheduling algorithms where the priority of a process is dynamically adjusted based on its past usage of the CPU and/or current resource requirements (e.g., to speed up response times to a given event). The scheduler always selects a runnable process with the highest priority. If there are multiple processes with the same priority they are typically scheduled round-robin. Typically, scheduling decisions are made in fixed intervals or time slices. If at the end of a time slice a higher priority process is runnable, the current process is preempted. Preemption can also occur during a timeslice if the a higher process becomes runnable and, for example, the current process returns from a system call.

To the best of my knowledge, all general purpose operating systems in use today, deploy variants of this scheme. This includes the different derivatives of the UNIX operating system, starting with the original Unix [Tho78], and the more recent versions of the family of Windows desktop and server operating systems.

For example, 4.4BSD [MBKQ96], on which all modern BSD flavoured UNIX operating systems are based, maintains two different priorities per process: the current scheduling priority `p_priority`, on which all scheduling decisions are based, and the user mode priority `p_usrpri`. When a process executes in user mode these two priorities are identical. The kernel associates a sleep priority with every event a process can be blocked on when in kernel mode. When a process unblocks, its `p_priority` value is temporarily set to the sleep priority to allow blocking processes to complete blocking system calls promptly. When a process returns from kernel mode, its `p_priority` value is restored to the user mode priority `p_usrpri`.

The user mode priority depends on two factors: the recent CPU usage of the process (`p_estcpu`) and the `p_nice` value assigned to it by the user. Every four clock ticks the user mode priority of a process is recalculated using $p\_userpri = PUSER + (p\_estcpu/4) + 2 \times p\_nice$ where PUSER is the base priority for user mode processes[2]. The process' CPU usage `p_estcpu` is incremented each clock tick the process is executing and, in addition, is decayed every second based on the load average of the system using: $p\_estcpu = (2 \times load)/(2 \times load + 1) \times p\_estcpu + p\_nice$. Thus, for processes that have recently accumulated a large amount of CPU (large `p_estcpu` value) the user priority value will increase, resulting in a lower priority. Likewise, the user priority of heavily I/O bound processes that spend most of their time waiting for I/O operations will remain at a relative higher level.

The UNIX System V Release 4 (SVR4) has a completely redesigned scheduler framework compared to its predecessors [Vah96, Section 5.5]. It implements the concepts of scheduling classes which define different scheduling policies. The default scheduling class is the time-sharing class which also changes priorities dynamically and uses round-robin scheduling for processes with the same priority. However, unlike the BSD based systems, priorities are not recalculated in fixed intervals. Instead, priorities are changed based on events. Events include the completion of time slices, blocking, etc. Furthermore, processes may use different length time-slices depending on priority. A static dispatcher parameter table defines how different events affect priorities and defines the length of time-slices per priority. This approach promises greater flexibility and is more scalable as the periodic recalculation of priorities for all processes is replaced by a simple table lookup only involving one process. However, the subjective "feel" of the performance of the system heavily depends on the parameters in the table. Furthermore, the priority boosts and penalties related to various events required for a responsive system depend on the overall system load, thus manual re-tuning of these parameters may be necessary.

The recent versions of Microsoft Windows (NT/2000/XP) define several priority classes for threads [Mic01, Sol98, CJ98]. For threads of all priority classes except the real-time class, priorities are dynamically adjusted. Priorities are boosted for the threads with the current foreground window, for threads receiving user input, or when a thread wakes up after an I/O operation. After the priority of a thread has been raised, it is reduced for every completed time slice until it reaches the thread's base priority.

[2]Note that in BSD systems numerically lower values correspond to higher priorities.

Apart from providing traditional time-sharing functionality most modern general purpose operating systems also provide some rudimentary scheduler support for real-time applications as defined by the POSIX standard [ISO96]. This functionality is typically provided by a real-time scheduling class often mapped to a privileged range of priorities. Unlike priorities in the time sharing class, priorities in the real-time scheduling class are not adjusted dynamically and real-time processes have a higher priority than any process in the default time-sharing scheduling class. The POSIX standard defines two types of real-time scheduling policies: FIFO and Round-Robin. Processes using the round robin policy are preempted after a configurable time quantum has expired while processes using the FIFO policy are only preempted when higher priority processes become runnable. This form of real-time scheduling is provided by SVR4, 4.4BSD derived systems (e.g., FreeBSD), Linux, and Windows NT/2000/XP.

A major problem with this mix of real-time scheduling and time-sharing scheduling is that it is extremely difficult to configure a system to support a mix of applications. Using a fixed set of three different applications — interactive typing, batch processing, and video display — [NHNW93] tried to find an adequate combination of scheduling class and priorities under SVR4. This also included settings for the X-Server process which was used by the interactive typing and the video display applications. The authors of [NHNW93] found that the combinations were either ineffective or resulted in complete system lockups and concluded that while *"SVR4 UNIX provides many controls for changing scheduler performance, they are virtually impossible to use successfully"*. Furthermore, if it almost impossible to find an adequate combination of parameters for a static set of applications, it is infeasible to do the same for a dynamic mix of applications.

2.1.2 Real-time scheduling

Dedicated real-time systems form a specialised area in computing science research and there exists a huge body of research on just hard real-time scheduling algorithms. An exhaustive review is beyond the scope of this dissertation and only a brief overview of related scheduling algorithms is provided.

The real-time literature distinguishes between *periodic*, *aperiodic*, and *sporadic* tasks. Periodic tasks enter their execution state at regular intervals and typically have hard deadlines. [PDP93] defines aperiodic tasks as tasks whose execution states cannot be determined in advance as they usual depend on the occurrence of external or internal events and whose deadlines are soft, i.e., deadlines whose adherence is desirable but not critical to the functioning of the system. The authors of [PDP93] define sporadic tasks as aperiodic tasks whose deadlines are hard deadlines[3].

Furthermore, tasks can be categorised into preemptable and non-preemptable tasks. A preemptable task's execution can be interrupted by tasks with a higher priority while non-preemptable must be executed until they complete.

Real-time scheduling algorithms can be categorised into static and dynamic algorithms. A static algorithm is one in which a feasible schedule, i.e., a schedule in which all deadlines are met, is computed offline and not changed during the execution of the schedule. While the run-time overhead of these algorithms is very low, they are too inflexible to be deployed in the context of a dynamic operating system environment. With dynamic algorithms, the exact schedule is not known in advance; it is computed dynamically at run-time. This allows for new tasks to enter the schedule at any time, provided that the schedule remains feasible. This obviously induces a higher run-time overhead than static algorithms but provides for more flexibility.

As this dissertation is mainly concerned with resource management in general purpose operating systems the primary interest is on dynamic scheduling algorithms that deal with preemptable tasks. For systems supporting multimedia applications, scheduling algorithms for periodic tasks are of particular interest as those applications often have inherent periodic behaviour, e.g., fixed intervals between frames of a video or virtual reality game applications.

The two classic algorithms for scheduling preemptable, periodic real-time tasks are Rate Monotonic (RM) and Earliest Deadline First (EDF) [LL73]. In their seminal paper Liu and Layland study these two algorithm in an environment which has to fulfil the following requirements (directly taken from [LL73]):

A1: The requests for all tasks for which hard deadlines exist are periodic, with a constant interval between requests.

A2: Deadlines consist of run-ability constraints only — i.e., each task must be completed before the next request for it occurs.

A3: The tasks are independent in that requests for a certain task do not depend on the initiation or completion of requests for other tasks.

[3]Note, that alternative definitions exist. Hyden [Hyd94], e.g., defines aperiodic tasks as tasks with stochastic arrival rates and multiple instances of the same task way arrive in very short periods of time; and sporadic tasks are classified as aperiodic tasks with a minimum limit between the arrival times of multiple instances of the same task. However, the precise definition of these terms is not relevant in the context of this dissertation and are only given for completeness.

A4: The run-time for each task is constant for that task and does not vary with time. Run-time here refers to time which is taken by a processor to execute the task without interruption.

A5: Any nonperiodic tasks in the system are special; they are initialisation or failure-recovery routines; they displace periodic tasks while they themselves are being run, and do not themselves have hard, critical deadlines.

RM is a fixed priority based scheduler, i.e., a priority is computed for a task once and is maintained unchanged throughout its lifetime. For RM the priorities are assigned relative to the tasks' periods: the shorter the period the higher the priority. In [LL73] a proof is given that this approach is optimal among fixed priority algorithms, i.e., given a set of tasks, RM always produces a feasible schedule if any other fixed priority algorithm does.

The processor utilisation u of a system with m tasks can be defined as: $u = \sum_{i=1}^{m} C_i/T_i$ with C_i denoting the run time and T_i the period of task i. In [LL73] it is shown that the achievable processor utilisation U for RM, and in fact for every fixed priority scheduling algorithm adhering to the above assumptions, is $U = \sum_{i=1}^{m} C_i/T_i \leq m(2^{1/m} - 1)$, approaching $ln2$ for large task sets. This means that if $U < ln2$ a feasible schedule can be constructed. This represents a lower bound on the achievable utilisation. For average task sets a feasible schedule can be found with an utilisation up to 88% [LSD89]. An optimal resource utilisation (U $\leq$ 1) can be achieved if all periods are harmonic, i.e., each period is an integer multiple of every period of smaller duration. A simple admission control policy aiming for a feasible schedule is to admit tasks up until the pessimistic lower bound of $U = ln2$ is reached. Since this potentially leaves the resource underutilised, one could use spare resource time for unreserved background tasks or use the more optimistic lower bound of 88%. Furthermore, an exact analysis of the task set, if feasible, might be used to yield a more optimal resource allocation.

While RM uses fixed priority values for each task, EDF computes dynamic priorities at run-time based on tasks' deadlines. The deadline of a periodic task is the end of the period at which it started. The EDF algorithm chooses at any given point in time to run the task with the smallest deadline value. Thus, the closer the deadline of a task, the higher its priority. Under the assumptions outlined above, all tasks meet their deadline if the processor utilisation is: $U = \sum_{i=1}^{m} C_i/T_i \leq 1$ [LL73]. Admission control for EDF is fairly straightforward: new tasks are admitted as long as the utilisation of the resource is less then 1. EDF is very attractive because it can achieve high utilisation and it is optimal in the sense that if there exists any algorithm that can schedule a set of tasks without missing a deadline, then EDF can also schedule the tasks without missing any deadlines.

Both algorithms behave optimally when the processor utilisation conforms to the given scheduleability test, however they behave differently under overload. Under RM the tasks with the longest period will be the first to miss its deadline since it has the lowest priority. While this behaviour is predictable, it is undesirable since the period of a task and thus its RM priority is not related to its importance. The way EDF degrades under overload is not that easily predictable. The general consensus is, that EDF performs poorly under overload and can even result in a state where the resource is constantly busy, but no deadlines are met. Furthermore, it is undetermined which tasks will miss their deadlines under overload. There have been proposals, e.g., [SLR86, Mil90], to overcome these problems, namely to assign some measure of importance to the tasks. However, these proposals make this very simple and well understood algorithm more complicated and the general consensus seems to be to simply not use EDF in environments in which overload can occur.

A number of proposals have been made to incorporate support for aperiodic or sporadic tasks in real-time systems. These are not very widely used in the domain of general purpose operating systems, therefore, only a brief overview is given. In [PDP93] five different approaches dealing with aperiodic tasks are described. The simplest approach is to execute aperiodic tasks as background tasks, i.e., they are only executed if no periodic tasks are active. A second approach, known as *polling*, uses a periodic task with fixed priority to serve aperiodic service requests. This approach has the obvious problem of the incompatibility of periodic and aperiodic tasks. The Priority Exchange (PE) and Deferrable Server (DS) approaches both use a high priority periodic server to maximise responsiveness of aperiodic tasks. In the DS approach, the server maintains its priority throughout its period, thus it can service aperiodic tasks with its high priority. With PE approach, on the other hand, the server exchanges its priority with the priority of the highest priority task pending if no aperiodic task requests occur at the beginning of the server's period. The final approach, known as Sporadic Server (SS), also uses a periodic server. Its response time performance is comparable to PE and DS, its implementation is, according to [PDP93], less complex.

2.1.3 CPU reservations

Scheduling algorithms that support CPU reservations are closely related to real-time scheduling algorithms, in fact systems that offer CPU reservations typically deploy real-time scheduling algorithms to support multimedia applications. CPU reservations provide applications with load isolation and predictable resource allocations, often in a periodic fashion. For example, an application may reserve $15ms$ of CPU time every $100ms$. The scheduling algorithm would then guarantee that this application would not receive less than the reserved amount every $100ms$.

In [MST94] a mechanism called Processor Capacity Reserves is presented. It is one of the first papers to highlight the relationship between resource reservations, presented as processor capacity reserves, and real-time scheduling algorithms in the context of a multi-media operating system. Previously, these algorithms had been mainly discussed in the context of classic hard real-time systems.

The principal abstraction described is the *processor capacity reserve*, which allow applications to request a reservation of the processor's capacity. Once the request has been granted, i.e., after performing an admission control check, the system guarantees that the reservation is met. Applications can increase their reservation if permitted by the admission control system and can always decrease it.

The processor capacity reserve abstraction is translated into a kernel abstraction known as *reserve*. Reserves are used to track reservations and measure processor usage for each program. The measurement is used to enforce reservations. Reserves can be passed from one thread to another, for example, during an Inter-Process Communication (IPC) call, to account for resource consumption independently of threads.

Capacity specification is based on periodic tasks; programs can request a percentage of the CPU to be reserved to them during an arbitrary period. Percentage and period define the rate of progress. The authors of [MST94] suggest that programs with known periods, but unknown or varying resource requirements during the period, should request conservative worst-case estimates. Programs with no natural computing rate, i.e., non-periodic tasks or batch processing applications, get assigned a rate determining the completion time of the application (details are not provided).

For the scheduler framework supporting processor capacity reserves both RM and EDF based algorithms were considered and admission control for both algorithms is discussed in detail (see section 2.1.2). Although EDF is theoretically preferable since it allows reservations up to 100% of the processor, the authors argue that, considering accounting inaccuracy, the effect of critical sections and synchronisation problems, 100% reservation cannot be achieved and both RM and EDF are suitable.

A similar approach to processor capacity reserves is presented in [Hyd94] where "processor bandwidth" is used analogously to processor capacity reserves. Hyden proposes a scheduling algorithm based on EDF. This work was carried out in the context of an operating system called Nemo. Nemo and its scheduling algorithm were the predecessors of the Nemesis operating system [LMB+96] (described in more detail in section 2.3.1) and its scheduling algorithm, known as Atropos [Ros95, Bar98]. Atropos is an implementation of the classic EDF algorithm extended by two features: support for latency sensitive tasks and a mechanism to deal with any "slack" CPU time. In addition to the standard EDF parameters of period p and slice s tasks have an extra latency hint l and boolean flag x associated. If a task has been blocked for a time longer than a period and unblocks at time t its next deadline is calculated using the latency hint rather than its period. Thus by choosing $l < p$, an unblocking task may be given preferential treatment. Essentially, its deadline is moved forward at the risk of potential deadline misses by other tasks during the first period after unblocking. This approach may still result in long dispatch latencies for tasks which only block briefly, i.e., unblock before their current deadline expires. These tasks are given preferential treatment if the CPU should become idle. Atropos also distributes "slack" CPU time in a round-robin fashion amongst tasks for which the x flag is set.

In Rialto [JRR97], applications can make CPU reservations by requesting Y units of time every X units of time. This is similar to the other reservation based systems described in this section, but the Rialto scheduler provides stronger guarantees in that the reservation is guaranteed continously, i.e., at *every* time T a task will receive at least Y units of time in the interval $[T, T + X]$. These guarantees are given for activities which do not block.

In addition to CPU reservations, Rialto also provides time constraints allowing threads to dynamically request a piece of code associated with a time constraint to be executed between a start time and a deadline. Deadlines of constraints may be tighter and execution times may be higher than the CPU reservation of the associated activity. A feasibility analysis, or admission control, is performed for any requested time constraint and, when accepted, time constraints are guaranteed to be met.

Unlike the other CPU reservation based systems, the Rialto scheduler is not based on one of the classic dynamic real-time scheduling algorithms. Instead, Rialto uses a static precomputed scheduling graph that allocates specific future time intervals to activities ensuring that CPU reservations can be met. The scheduling graph presents a repeating schedule of all accepted CPU reservations and forms the basis for the feasibility analysis of time constraints. The graph is recomputed whenever CPU reservations change or activities enter or leave the schedule. Thus, the runtime overhead for a static task set is small (complexity O(1)), while the assumed infrequent recalculation of an optimal scheduling graph is NP hard (for the implementation the authors of [JRR97] use a simpler algorithm which provides an approximation of the optimal graph). Depending on the assumed dynamics of the active task set, it is not clear if this tradeoff is better than using a dynamic real-time scheduling algorithm with simple admission control, such as EDF.

There are two common criticisms about reservation based systems. First, reservation based systems need to deploy a run-time admission control systems in order to make guarantees about resource reservations. A simplistic approach would simply be to accept new reservations until the capacity of the resource is exhausted. This potentially leads to inflexibility and unfairness since tasks arriving later than others may be denied access to a resource even if they are more important [NL97]. Furthermore, even for conventional tasks with no special timeliness requirements, reservations have to be made to prevent their potential starvation.

The second criticism is that consumers of a resource have to specify their resource requirements in advance in order to make a reservation. This is non-trivial for most applications. Multi-media applications, for example, may have highly bursty resource requirements (e.g., see figure 5.1 for decoding times of an MPEG video stream). If the reservation is made for the peak resource requirement (which itself might be unknown) the resource is likely to be underutilised. If, however, a reservation is made based on the average resource requirement, some application specific deadlines might be missed due to insufficient resources. Moreover, the admission control system would have to provide incentives to applications to make *correct* reservations.

These issues are typically not addressed with reservation based systems. Instead, they are deferred to a higher level entity such as a QoS manager (see section 2.2).

2.1.4 Proportional fair schedulers

A recent trend has been the introduction of proportional share schedulers into general purpose operating systems (e.g., [WW94, WW95, GGV96, SAWJ+96a, NL97, JSMA98, DC99, BPM99]). In proportional share allocation all tasks in the system are guaranteed to make progress at a well defined uniform rate. This progress is frequently expressed in terms of virtual time [Zha91], which advances relative to real time. These scheduling algorithms are closely related to the fair queueing models that have been proposed for packet scheduling in computer networks (e.g., [Nag87, DKS90, Zha91, PG93, BZ96]).

Proportional fair schedulers aim at providing a perfect sharing of resources. However, resources cannot be shared in infinitesimally small units (e.g., due to fixed packet sizes, time slicing, prevention of context thrashing). Therefore, the share a consumer should receive in an ideal system (often referred to as the *fluid-flow model*) and the share of the resource a consumer has received at any given point in time may be different. In [SAWJ+96a] this difference is called service time *lag*. The lag is a measure for the allocation accuracy of a given proportional share scheduler and, ideally, should be bounded to a constant.

2.1.4.1 Virtual time based scheduling

Most proportional share algorithms are formulated in terms of virtual time. Virtual time flows at a rate proportional to the weights of the active tasks and thus abstracts over changes in the task set. The relationship between real time and virtual time can be defined as (taken from [SAWJ$^+$96a]):

$$V(t) = \int_0^t \frac{1}{\sum_{j \in \mathcal{A}(\tau)} w_j} d\tau \qquad (2.1)$$

with $\mathcal{A}(\tau)$ denoting the set of all active processes at time τ. The flow of virtual time changes to "accommodate" all active clients in one virtual time unit. This notion of virtual time can be used to express the service time a task should receive in the fluid-flow model. In this model, a task with the weight w_i should receive a service time proportional to the sum of the weights of all active tasks:

$$S_i(t_0, t_2) = w_i \int_{t_0}^{t_2} \frac{1}{\sum_{j \in \mathcal{A}(\tau)} w_j} d\tau \qquad (2.2)$$

From equation 2.1 and 2.2 it follows:

$$S_i(t_1, t_2) = (V(t_2) - V(t_1))w_i \qquad (2.3)$$

Thus, the lag, or deviation of the service time in a real system from the fluid-flow model, can be viewed as a problem of how well the virtual time is estimated in the real system.

The general model of virtual time is used by a number of CPU and fair-queueing packet scheduling algorithms, for preemptive and non-preemptive schedulers, and for soft real-time and non real-time systems, demonstrating the expressive power of virtual time. The initial work was done in the area of link sharing in datagram networks. Therefore, the general modus operandi of virtual time scheduling algorithms is first described in the terminology of packet scheduling. Later in this section its application to CPU scheduling is discussed.

Bit-Round Fair Queueing (BRFQ) and Weighted Fair Queueing (WFQ) [DKS90], also known as Packet Generalised Processor Sharing (PGPS) [PG93], aim at emulating the fluid-flow model in which packets would be transmitted bit by bit in a (weighted) round-robin fashion among different flows or clients. Each packet is assigned a timestamp representing its virtual finish time (VFT) in the fluid model. If v_0 is the virtual time the kth packet arrives for flow i and l_i^k its length, then the VFT_i^k can be calculated as $VFT_i^k = v_0 + \frac{l_i^k}{w_i}$. The VFT can be interpreted as the amount of service, normalised with respect to the weight, the flow i has received immediately after the kth packet is served. Thus, WFQ transmits packets from different flows in the order of their VFT.

There are two well known problems with WFQ. First, in WFQ, flows could be serviced long *before* they would complete in the fluid-flow model. Therefore, some scheduling algorithms (e.g., [SAWJ$^+$96a, BZ96, BZ97]) include the notion of *eligibility*. A task is considered eligible if it would receive service in the fluid-flow model. Thus, the resulting system conforms more closely to the idealised fluid-flow model. Systems taking eligibility into account have a provably smaller bound for the lag. For example, in Earliest Eligible Virtual Deadline First (EEVDF) [SAWJ$^+$96a, SAWJ$^+$96b] and WF^{2}Q [BZ96], the lag is bounded by the maximum time quantum or maximum packet size respectively.

The second problem with WFQ is maintaining the virtual time necessary to compute the VFT. For an active flow, i.e., a flow which has outstanding packets queued, v_0 is simply the VFT of the previous packet. However, for idling flows, the current virtual time at the time a new packet arrives needs to be maintained. The original WFQ proposal simulates the fluid flow model and essentially maintains the exact virtual time. However, the simulator is complicated and computationally intensive (essentially, it implements equation 2.1). More recent algorithm [SAWJ$^+$96a, GVC96, BZ96, BZ97, DC99] instead estimate the virtual time in the system by setting the virtual start time v_0 of a packet to the maximum of either the VFT of the previous packet of that flow or to the smallest virtual start time of any packet from the other active flows. This estimate is either accurate or slightly conservative.

The virtual time based scheduling algorithms have been discussed so far in the context of packet scheduling in datagram based networks. They can be applied in different ways to CPU scheduling. Packet scheduling is usually non-preemptive — once the transfer of a packet has started it has to be finished. However, the length of a packet is known at arrival and networks typically define a Maximum Transfer Unit (MTU), limiting the maximum length of a packet. EEVDF uses the simplest equivalent for CPU scheduling and uses a fixed scheduling quantum, as is the case in most general purpose operating systems. However, EEVDF *assumes* that all active tasks at the start of a quantum will use their entire quantum. This assumption is necessary to calculate the VFT. However, if a task becomes inactive before its quantum expires, the task's lag can be zero, positive or negative. If the lag is zero, than the task can simply leave, as it has received its fair share of the resource. If the lag is negative or positive the task has received more or less service than it should have in the ideal fluid model. In this case the authors of

[SAWJ$^+$96a] propose to adjust the virtual time *as if* the task would have executed until its lag is zero and distribute the task's lag to the other tasks proportional to their weights.

BERT [BPM99], a non-preemptive CPU scheduler based on WF^{2}Q+ [BZ97], requires tasks to specify the length of a request and uses this to calculate the VFT of the task. To an extent this choice reflects the environment, namely the path-oriented[4] operating system SCOUT [MP96, MT97], for which BERT was developed. BERT also implements a mechanism called Fair Queueing (FQ) stealing which allows more important tasks to steal cycles from less important tasks. For this purpose the user can divide tasks into important and unimportant ones using a simple user interface. The amount of cycles a task can steal is bounded and BERT preserves the relationship between real time and virtual time during stealing as weights are adjusted in correspondence to the number of cycles stolen.

Start-time Fair Queueing (SFQ) [GGV96, GVC96] takes a different approach. Instead of executing tasks in the order of their VFT SFQ uses a task's virtual start time v_0. The authors demonstrate that, using this approach, the lag is bounded, but is dependent on the number of clients. However, SFQ has the advantage that it naturally supports non-uniform time quanta, no a priori knowledge of the computational requirements is necessary, and it can cope with variable capacity resources (WFQ and related algorithms implicitly assume a fixed capacity in the calculation of the VFT). The assumption of variability of capacity is required by the desire to deploy SFQ in hierarchical schedulers.

Yet another approach is used by Borrowed Virtual Time (BVT) [DC99]. BVT uses a relatively small time quantum, e.g., $100\mu s$, and essentially uses this to directly model the fluid-flow model, i.e., it ignores the resulting quantisation error. In order to prevent constant CPU context switches, also known as thrashing, a context switch allowance is introduced, which allows a task to execute for a minimum amount of time. The more interesting feature of BVT, however, is the introduction of virtual time *warping*. Warping is used to support latency sensitive applications by dispatching unblocked tasks earlier rather than later. To achieve this aim, latency sensitive tasks have an associated *warp factor*, a constant which is subtracted from their virtual timestamps. Thus, the scheduler will run them earlier than in the corresponding fluid-flow model. The time a task can run warped and the time between warps are controlled by two additional task specific parameters. The base BVT algorithm is very simple. However, configuring the scheduler for a variety of applications and application mixes appears more complicated: some per-task parame-

[4]The term "path-oriented" refers to the primary abstraction in SCOUT, a "path". Paths are responsible for moving and manipulating data form one I/O device to another.

ters are expressed in virtual time, others in real time. Not surprisingly the majority of the paper is dedicated to the configuration of the scheduler.

Virtual time based scheduling algorithms have also been proposed to support real-time tasks (e.g., [SAWJ$^+$96a, GGV96, BPM99, DC99, BP00]). The general idea in these system is to control the weights assigned to different tasks. By policing the weights assigned, tasks have guaranteed execution rates. Then, if the lag is bounded for a given scheduling algorithm, guarantees can be given about the completion times of a task's particular work unit. The proponents of this approach argue that these type of guarantees are sufficient for soft real-time tasks. A generalised treatment of the use of virtual time based schedulers for real-time scheduling can be found in [FP97, BP00].

2.1.4.2 SMART: A Scheduler for Multimedia And Real-Time

A particular feature-rich and influential virtual time based scheduling algorithm is SMART [NL97]. SMART aims at supporting a mix of real-time multimedia and conventional applications. It allows real-time applications to execute blocks of code under time constraints and provides notifications to applications if the time constraints cannot be met. SMART uses both priorities and shares: priorities allow users to specify their preference for one process over another; and shares may be used to influence the amount of CPU a process receives compared to other processes of the same priority if there is CPU contention. The base scheduler is a standard virtual-time based scheduler to which two main features are added: a bias of the VFT and a reordering of real-time tasks based on EDF. For conventional processes, a bounded bias is used to defer long running batch processing applications during transient overloads by adding it to the VFT of each task. The bias is calculated similarly to the dynamic priority recalculations used in the SVR4 time-sharing scheduler. Scheduling decisions are based on a value tuple of priority and Biased Virtual Finishing Time (BVFT) of a process. A process with a higher priority is said to have a higher value tuple, for processes with the same priority, one with a lower BVFT has a higher value tuple.

In general, tasks with the higher value tuple are scheduled. However, the second special feature of SMART, the EDF based reordering of real-time tasks, is used to trade off urgency of lower priority real-time tasks against the importance of higher valued real-time tasks. More precisely, if the process with the highest value tuple is a conventional process, that process is run. If the process with the highest value tuple is a real-time task, a candidate set is created, to which all real-time processes with a higher value tuple than the highest valued conventional process are added. From the candidate set of real-time tasks, processes are inserted into an EDF schedule starting with the process with the highest value tuple until either the candidate set is empty or the EDF schedule becomes infeasible. The real-time processes that cannot be inserted into the schedule are then notified by the OS that their resource requests cannot be granted. The resulting EDF schedule may result in lower valued processes being run before higher valued processes. In general, this approach trades off instantaneous fairness for better real-time and interactive response times.

2.1.4.3 Lottery Scheduling

A number of other proportional fair share algorithms have been proposed; the most prominent being lottery scheduling [WW94]. Lottery scheduling is a probabilistic, proportional share scheduler — scheduling decisions are based on a lottery with lottery tickets representing resource rights. Resource allocations are determined by holding a lottery, the resource is granted to the process holding the winning ticket. Thus, lottery tickets represent relative resource rights, since the fraction of a resource represented by one ticket varies dynamically in proportion to both the total number of tickets in the system and the contention for the resource. In the worst case, a process receives a share of the resource proportional to the number of tickets it holds. Furthermore, lottery scheduling is probabilistically fair. Since a lottery has a random result, the actual proportion of a resource a process receives is not guaranteed over shorter periods. However, the accuracy of the allocation improves with the number of lotteries held. Waldspurger and Weihl argue that with a scheduling quantum of 10 milliseconds a reasonably fair distribution can be achieved over sub-second intervals.

A number of further abstractions are supported to form a "modular resource management" framework. *Ticket transfers* allow a process to yield its resource rights to another task, for example, when a process needs to block on a reply from a server. This provides a mechanism to avoid priority inversion. Processes may transfer parts of their tickets to multiple clients. *Currencies* and *ticket inflation* are mechanisms which allow a more flexible management of resource rights within groups of mutually trusted processes, e.g., processes belonging to a particular user. A unique currency can be used to express resource rights within a group of processes. Each local currency has to be backed by tickets in a more primitive currency, forming an arbitrary acyclic currency graph. This allows, for example, a user, to locally inflate tickets by creating more of them. Exchange rates between currencies are implicitly changed by ticket inflation. *Compensation tickets* may be awarded to processes which only use a fraction of their allocated resource quantum, essentially inflating the value of that process' allocation until the start of the next quantum. This mechanism ensures that processes yielding the resource before their quantum expires may still receive their fair share of the resource

A number of researchers have revisited lottery scheduling since its introduction and proposed improvements. In particular, Sullivan et al. [SS00, SHS99] have extended the resource management abstractions to include multiple resources. They introduce resource specific tickets to provide insulation between different resources and allow applications, via *ticket exchanges*, to barter with each other over resource specific tickets.

2.1.5 Hybrid and hierarchical schedulers

Since designing one scheduler to accommodate both conventional and real-time tasks is difficult, a number of hybrid schemes have been proposed. Typically, a base scheduler is used to schedule both tasks and other schedulers, forming a hierarchy of schedulers. Different policies have been proposed for the base level scheduler, as discussed below.

Hybrid, priority based schemes, such as the Windows NT scheduler [CJ98] or the Unix System V scheduler [Vah96, Chapter 5] assign higher, typically fixed, priorities to real-time tasks than are assigned to tasks scheduled by a conventional scheduler. This results in all real-time tasks being scheduled before any conventional tasks, irrespective of their importance. It has been demonstrated [NHNW93] that this might result in pathological behaviour in which run-away real-time tasks prevent users from regaining control of the system.

Proportional-share based hybrid schemes (e.g., [BJ95, WW95, GGV96, SAWJ97, DC99]) deploy a proportional-share base-level scheduler. This prevents real-time tasks from monopolising the resource, but potentially prevents them meeting all their deadlines in the name of fairness. Often, the base-level proportional share scheduler is combined with an admission control system to guarantee a share of the resource to the set of real-time tasks.

In reservation based schemes (e.g., [LMB$^+$96]) the base scheduler is a classic real-time scheduler allowing resource guarantees to be given to tasks. Conventional tasks are scheduled when the resource is not busy, i.e., after all real-time tasks have been serviced. As in priority based schemes, this might lead to resource starvation of conventional tasks. Alternatively, conventional tasks can be grouped together as one real-time "task", with a conventional scheduler scheduling processes within that group. While this prevents resource starvation of conventional tasks it might put an additional strain on the base level scheduler by introducing artificial real-time tasks.

CPU inheritance scheduling [FS96] presents a special case of hybrid schedulers. In this scheme arbitrary threads can act as scheduler for other threads, allowing for a wide range of policies to be implemented and spawning arbitrary domains such as processes, jobs, users, and user groups. Threads can donate their share of the CPU to selected threads while waiting on events. Donating threads can be notified if the thread it donated its CPU to does not require it anymore and running threads can be preempted if the donor wakes up and the CPU is given back to the donor thread. The authors of [FS96] anticipate that special scheduler threads exist in the system that donate most of their CPU time to other threads. Furthermore, each physical CPU has one root scheduler thread associated with it, forming the root of the scheduling hierarchy. However, it is unclear what overheads including multiple context switches this approach incurs or what influence the multiple levels of scheduling threads have on wake-up latencies?[5].

2.1.6 Feedback-Driven schedulers

Recently, researchers have proposed applying concepts from control theory to the management of resources in operating systems [LMB$^+$96, BN98, SGG$^+$99, LSTS99]. At the core, these systems deploy a real-time scheduler and then, using a feedback controller, adjust the resource allocations of processes.

Steere et al. [SGG$^+$99] argue that scheduling should be based on the notion of progress. Their feedback-driven resource allocator, implemented under Linux, monitors the progress of applications via *symbiotic interfaces*, which map application specific notions of progress to a uniform progress metric, and adjust the "reservation" of the applications for an RM based real-time scheduler so that the progress is distributed uniformly.

The controller is the central component in their architecture. It receives feedback of the progress of applications via the symbiotic interfaces and dynamically assigns proportion and periods. The controller distinguishes between four different types of application: real-time, aperiodic real-time, real-rate, and miscellaneous. Real-time tasks supply both proportion and period while aperiodic real-time tasks only specify the required proportion. Real-rate applications do not specify either of these but instead provide a metric of progress. Applications which do not provide any information at all fall into the miscellaneous category. For real-time applications the controller does not adjust proportion and periods but treats the supplied information as a reservation. If the system is under significant overload it initiates a renegotiation of the reservations. For aperiodic real-time tasks the controller has to assign a period determining the deadlines for the task. If the application does not supply a progress metric a systemwide default

[5]The paper only describes a prototype implementation as a user-level thread package on top of FreeBSD, thus definitive answers to these concerns cannot be given.

value is used. Real-rate tasks are the main subject for the controller since they do not specify their resource requirements but have requirements on the achieved throughput, as expressed by their progress metric (ranging between $-1/2$ and $+1/2$). The controller, executing in regular intervals, attempts to level the progress metric at 0 and uses a Proportional-Integral-Derivative (PID) function, commonly used in control theory, to adjust the share of real-rate application. The system can be overloaded, either by new real-time tasks arriving or by the adjustments of shares made by the controller. In this case, newly arriving real-time processes are rejected by an admission control system, and for other processes their allocation is *squished* in a manner not described in detail in [SGG+99].

The second feedback based system is FC-EDF [LSTS99, LSA+00]. Unlike Steere et al.'s resource allocator, aimed at general purpose operating systems, FC-EDF is targeted at dynamic, adaptive real-time environments and is based on an EDF scheduler. It is motivated by the observation that traditional real-time scheduling algorithms are mainly targeted at static environments with well-known workload characteristics. However, the authors concede that in a dynamic, complex system it will be impossible to meet all deadlines, without significant over-provisioning of resources. They therefore argue that for such systems the aim should be to meet as many deadlines as possible and propose to use a control theoretical framework to meet this. The aim of which is to minimise the deadline-miss ratio, i.e., the percentage of tasks that miss their deadlines[6]. In soft real-time environments a small percentage of missed deadlines is acceptable, especially if it yields significantly higher resource utilisation. In FC-EDF, a controller periodically monitors the deadline-miss ratio (*controlled* variable) and compares it with a predefined acceptable value (set or *reference* point). The controller then uses a control function, again PID, to determine by how much to change the overall requested CPU utilisation $\Delta CPU(t)$ (*manipulated* variable). This is done to minimise the error between controlled variable and set point.

FC-EDF uses two different mechanisms to adjust the requested CPU utilisation (*actuators*). First it attempts to accommodate the change $\Delta CPU(t)$ within the existing task set by changing the modes of operation of the existing tasks — the service level controller. This is facilitated by the assumed task model borrowed from the imprecise computation model [LLS+91] which describes a task as a tuple of vectors referring to the different modes of operation, their resource requirements, and their value to the user — essentially defining a discrete utility function. If the service level controller cannot accommodate the entire requested change, $\Delta CPU(t)$, by

[6]The task model used assumes that tasks only have a single deadline – processes with periodic deadlines are modelled as periodically arriving tasks. This definition of a task is common in the real-time literature.

changing the modes of operation of existing tasks, the admission control system, as the second actuator, is instructed to take the remaining change into account when performing admission control for newly arriving tasks.

2.1.7 Discussion

In this section a wide range of differing scheduling algorithms have been presented. For a general purpose resource management framework, as proposed in this dissertation, all these approaches have their shortcomings.

Priority based schemes are problematic for soft real-time applications, such as multi-media applications, as they specify *what* to schedule and not *when* and *how much* [Pra97]. Assigning a high priority to an application which requires only a small amount of a resource, albeit in a timely fashion, requires these applications to be trusted as high priority applications may monopolise the resource [NHNW93]. Furthermore, in order to provide predictable resource allocations, global knowledge about other application's priorities and behaviour is required.

The first problem of high priority processes is addressed with proportional fair schedulers — they ensure isolation between processes by making sure that they make progress at a defined rate (proportional to their weight). However, in order to provide predictable resource allocations some global knowledge, namely the sum of weights of all processes, needs to be known and controlled. Furthermore, with most proportional fair scheduling algorithms it is difficult to achieve the timely allocations of resources, that are required by some applications, such as digital audio applications [JR00] or soft modems [JS01]. These types of applications typically require real-time scheduling algorithms. Proportional fair share schedulers only provide upper bounds on the timeliness of allocations, typically dependent on the number of active processes and/or the sum of weights.

Both priority based and proportional fair scheduling algorithms manage resource "magically"; they do not provide any feedback to applications when resource allocations are changed (a notable exception is SMART, which provides a notification mechanism informing applications when deadlines cannot be met — this, however, is a fairly coarse grained feedback). The consequence of this "behind the scenes" approach is that those applications which can adjust their resource demand, can only do so *after* they have experienced unexpected performance degradation due to insufficient resources, rather than being able to avoid undesirable side-effects by *proactively* adjusting their behaviour.

In contrast, reservation based scheduling algorithms, borrowing heavily from hard real-time scheduling algorithms, provide firm guarantees about resource allocations. While this offers predictable performance to applications it requires an admission control system to ensure the guarantees. Processes requiring more resources or processes arriving late may be rejected. Furthermore, simplistic implementations of an admission control system provide no incentive to (1) not reserve a large amount of the resource, e.g., peak-rate, and (2) adjust an existing reservation if other processes may benefit from it.

Feedback driven schedulers have been proposed as a solution to the problems related to reservation based systems. A feedback controller periodically observes the system, and dynamically changes the resource allocations of active processes. However, one might argue that processes should be notified when their resource allocations are changed, i.e., feedback based systems again manage resource magically behind the scenes. Furthermore, for the controller to be able to adjust the overall systems applications have to provide either a metric for their progress or a detailed model of their modes of operation. It is unclear, especially for the latter, if this is feasible is a general purpose operating system environment.

2.2 QoS Architectures

With the increased interest in multi-media applications and their significant demand for more than one resource, many researchers have proposed QoS architectures. These are often concerned with providing QoS *end-to-end*, i.e., including *all* resources required by an application. QoS architectures are typically structured in layers (e.g., network layers, OS layer, application layer). At each of these layers *QoS specifications* describe the resource demands of the application and the QoS architecture provides a *QoS mapping* mechanism to translate higher level specifications to lower level ones with the aim of freeing the user from representing applications' resource requirements in lower level, system terms. In terms of system resources, QoS architectures typically operate with resource reservations in order to provide resource guarantees or contracts to the higher layers. Resource reservations are combined with admission control and, often, with a mechanism for contract renegotiation.

The following sections introduce a number of different QoS architectures that aim to provide end-to-end QoS management. A good survey and a generalised framework is presented in [ACH98].

2.2.1 The QoS-A model

The Lancaster University *Quality of Service Architecture (QoS-A)* [CCG+93, CCH94, BCC+94] is a layered architecture which uses a modified version of the Chorus microkernel [CBRS93] to provide QoS support in the end-systems and an experimental ATM testbed to provide network connections. The key abstractions in QoS-A are *flows* which characterise the production, transmission, and consumption of multimedia data streams; *service contracts* which are agreements about a resource allocation between users and providers; and *flow management* which encapsulates monitoring and maintenance of service contracts.

The main contributions of QoS-A are the definition of a comprehensive architecture for QoS management and the definition of interfaces between the components of the architecture [CCH94]. This interface, for example, allows the specification of a flow's QoS requirements to the lower level of the system in terms of bandwidth requirements, delay, loss and jitter. In addition to these flow characteristics, a request for the establishment of a flow also indicates the level of commitment required from the provider (e.g., deterministic, probabilistic, or best-effort). This information is used for an admission control test [BCC+94]. Also encapsulated in the flow specification is information about different adaption options for the flow. This information is used for contract renegotiation in the event of changes in the system load. The authors envisage that users or application programmers can specify flow QoS requirements in terms of high level, commonly used channel types, such as "Standard Video", and a QoS mapper service would convert these to specifications for the next level down. Then for each further level, QoS specifications are mapped to the next level down. Unfortunately, no details are provided on how this mapping is implemented and how the overall system performs.

2.2.2 QualMan and OMEGA

OMEGA [NS96, NS95] and its successor QualMan [NhCN98] aim at providing end-to-end QoS. OMEGA presumes that each individual resource in a distributed system, e.g., network and OS resources, is able to provide service guarantees or has real-time facilities and OMEGA aims at integrating these into a networked multimedia system. OMEGA's key component is the QoS Broker [NS95] which runs on each of the end-systems. QoS brokers either operate as buyers, if they wish to establish a connection with another end-system, or as sellers, if they are contacted by another end-system. The communication between brokers is performed using a QoS Broker protocol, which is modelled in layers, in a similar way to network protocols, compromising an application layer (or subsystem) and a transport subsystem. Both of these make use of the OS resources in the end-systems and the QoS broker additionally translates the higher level resource requirements at each layer to the OS resources.

OMEGA translates QoS specifications between four different QoS views: User (e.g., TV quality video), application (e.g., 20 frames per second (fps)), system (e.g., $10ms$ of CPU cycles every $50ms$) and network (e.g., 10Mbits/s of bandwidth). The mapping from user (or perceptual) QoS to application level QoS specifications are performed by a tuning service, allowing users to specify QoS using their senses. The user can adjust application QoS parameters, such as frame rate or picture size of a sample audio/video clip using a simple set of sliders. The slider values encode application QoS parameters, such as sample size, sample rate, period between samples, and end-to-end delay, for periodic uncompressed data streams and Constant Bit Rate (CBR) data (on which the prototype is focused). The authors realise that this mapping from perceptual QoS to application QoS is non-trivial and still an open research issue. Application QoS parameters are then translated by the QoS broker into network (e.g., packet size, packet rate) and system (e.g., task priorities and periods, and buffer space requirements) QoS parameters using fixed bidirectional mappings. Admission control is performed using these translated QoS specifications using a number of admission tests, first at the end-systems for the application subsystem (both seller and buyer) and then for the transport subsystem. The admission test also accounts for time dependencies of the tasks and streams. Admission control is actually performed as a form of negotiation with three possible results: accept, reject, and modify. The latter case caters for the situation where a subsystem may be able to suggest different QoS parameters to achieve the same higher level QoS specification, which is why bidirectional translation between different specifications is used. QoS parameters are renegotiated periodically, where only one parameter can be changed per negotiation.

QualMan is the successor of OMEGA and addresses some of the lessons the authors learned from OMEGA. In particular, the QoS Broker is split up to move some of its functionality closer to the individual resources. In QualMan each resource has a broker associated with it, which accepts system level QoS requests (e.g., CPU reservations in the form of period and utilisation) and performs its own admission control. However, there is still a central QoS Broker which performs QoS translation and negotiation as in OMEGA[7]. In QualMan, the QoS translation can be aided by a probe-based system for estimating system QoS parameters [NHK96] which, during the negotiation phase, determines the statistical resource requirements for media streams.

[7]The paper also seems to suggest that applications may request system level QoS directly from the resource brokers. However, the authors also argue this may result in a deadlock situation when applications request and are directly granted resources in a different order.

[KN97] describes a mechanism for how these estimates can be used for admission control and to determine CPU scheduling parameters for Variable Bit Rate (VBR) MPEG video streams.

In general, both OMEGA and QualMan do not seem to address the issue of resource revocation, thus exhibiting similar problems to reservation based systems (see section 2.1.3). In both architectures, admission control seems to be performed on a first-come first-served policy, placing a disadvantage on clients arriving later, and neither architecture seems to include mechanisms to provide clients with an incentive to reduce their resource demands or to even choose the appropriate QoS parameters, given the overall system utilisation.

2.2.3 Q-RAM

The Q-RAM model [LLS+99, LLRS99, RLLS98, RLLS97], developed in the context of resource kernels [RJMO98, OR98], strives for an optimal allocation of multiple resources to concurrent adaptive applications, which can operate at different levels of quality. In order to achieve this aim, the authors introduce a general model, which, unlike many other systems, encapsulates the notion of the end-user's utility.

The model assumes a set of applications T_n and a set of shared system resources R_m. Each application i has a number of application specific quality dimensions (e.g., picture format, colour depth, frame rate, etc) Q_{id_i}. These quality dimensions define a d_i-dimensional space Q_i of quality points. Each application has associated with it a task profile which is partially populated by the application developer and partially by the user. The application profile contains: the quality space, Q_i; a quality index, which is an ordering of quality points, where the order is defined by a "better than" relation; a utility function, which could be defined as the weighted sum of per quality dimension utility functions; and a resource profile, that defines a relation between potential allocation of resources and quality points. A user can also specify minimum QoS requirements in terms of minimum quality values along each dimension.

The aim of this model is to maximise the system utility, defined as the (weighted) sum of application utilities under the constraints of limited resources and the resource profiles. It has been proven that even the optimisation problem considering a single resource and a single quality dimension is an NP-hard problem, and therefore, so are more complex allocation problems involving multiple resource and/or multiple quality dimensions [Lee99]. However, in [RLLS98] a practical solution to this problem is presented under the assumption of a concave utility function, while [LLRS99] relaxes this assumption and presents an optimal and two near-optimal algorithms (one with a bounded distance from the solution and one with a user-specified distance from the solution). The results suggest that the first of the near optimal solutions is close to the optimal solution and has feasible run-times for use on-line.

This theoretical solution has some practical considerations. First, it is infeasible to expect the user to specify utility values for all possible quality points along all quality dimensions (an example application of a video phone can have tens of thousands of quality points). However, by presenting a number of sample single dimensional utility functions, the user only has to specify a few points out of this vast space. [Lee99] presents an example user interface for the above mentioned video phone application.

The second main issue is that the optimisation problem requires knowledge of the relation between resource allocation and quality points. Unfortunately, the authors do not provide details on how this relation can be constructed. This task is especially difficult as the resource requirements for a given quality point may vary dramatically over time and are also machine dependent.

The third issue is that of interfaces. Since the authors assume a centralised entity to perform the optimisation problem, the user, application, and resource profiles need to be communicated to the entity performing the optimisation. Again, these are not specified in more detail.

2.2.4 AQUA

AQUA is an architecture which focuses on providing QoS in the end-system, aiming in particular at multi-media applications [LY96, LYF97]. Each resource in AQUA is associated with a *resource manager* which includes policies for scheduling, admission control, and resource reservations. Applications use a *QoS specification library* to specify their resource requirements for each resource. Applications may omit some resources if the demand is not known in advance. Each application also has an application level *QoS manager* which interacts with the resource managers and adjusts the application's resource requirement. And finally, a *QoS adaption library* is provided to facilitate the cooperation between resource managers and QoS manager.

AQUA uses a uniform QoS specification for all resources. This specification includes an application class (CBR, VBR, or, Available Bit Rate (ABR)); a rate range over which resources should be allocated; a delay range, specifying the maximum delay of resource allocation acceptable; and an optional payload, specifying the quantity of the resource required. The QoS manager takes the specification and makes resource requests to the resource managers. The resource managers measure the resource consumption over an averaging interval and feed these values back to the QoS managers. The QoS manager then compares the received QoS with the desired QoS and adjusts the resource demand based on application specific policies. Furthermore, the resource managers can also request, again via the QoS manager, that an application changes its resource demand when the overall resource demand changes. Admission control and adaptation requests are based on averaged resource utilisation and the authors have evaluated the system for one resource, namely CPU, using a trace driven simulation.

2.2.5 Discussion

In this section a number of QoS architectures have been introduced, most of which use some form of layering to free users and application programmers from having to directly specify resource demands for each individual resource. The three main challenges in this area are then (1) the mapping of QoS specifications between the different layers; (2) dealing with admission control and resource renegotiation under changing resource availability; and (3) the definition of appropriate APIs which allow for both the specification of application-domain specific QoS requirements and the interaction between the different layers.

The mapping of QoS specifications is non-trivial both for multimedia applications and server applications, although most of the work on QoS management has focused on the former. The resource demand for individual work units, such as decoding a frame of an MPEG encoded video stream, are highly variable, depending, for example, on the encoding and the content of the stream. The authors of [BMP98] report a factor of three difference in decoding times between different frames of the same MPEG encoded video. Should the mapping take the peak or the average decoding times into account when determining the resource demand? Similar problems arise in server applications, where, depending on the type of request, it is difficult to predict how much of which resource is required to serve an incoming request. Thus, reliably deriving concrete resource allocations from high-level descriptions such as desired frame-rate or requests per second is still an open research issue. I argue that, for these types of application, some form of feedback based approach is more promising than attempting to establish static mappings between application QoS specifications and resource demands. From the described systems, only AQUA appears to advocate providing feedback to applications.

The second problematic area with the described QoS architectures is related to admission control. Most of the architectures require an admission test for a task's translated QoS requirements in order to provide QoS contracts to applications and suggest resource renegotiations should the general resource availability change. However, most architectures seem to deploy a simple, but unfair, first-come first-served policy for admission control and do not seem to provide applications with an incentive to adjust their resource demands when resource availability changes. Unless the QoS mapping is performed by a central entity, i.e., the resource request can be trusted, and applications are required to provide alternative modes of operation, the applications or users have very little incentive to not simply request the best individual QoS achievable. In this respect, Q-RAM is certainly the most advanced approach. By taking the user and application utility into account, Q-RAM can find an optimal allocation of resources and make the correct trade-offs should the resource availability change. However, it is unclear what incentive users have to truthfully reveal their utility functions. Furthermore, Q-RAM also requires a full specification of resource demands for different modes of operation which raises the same problems as those discussed with QoS mapping. For Rialto, a similar user-centric approach has been proposed [JLDB95], however, their resource planner is not described in detail and is considered as future work.

The construction of appropriate APIs is a third problematic area. Depending on the QoS architecture, the APIs need to both support the declaration of application-specific QoS specifications and enable the communication between the different layers of the architecture. While this is mainly an engineering effort, it is non-trivial given the complexity of the QoS architectures described above. Part of the complexity of the APIs can be found in the centralised approach advocated by most QoS architectures, as they require the explicit specification and communication of QoS requirements to the central QoS manager. This communication to a central entity may also prevent frequent renegotiation of QoS contracts.

2.3 General operating system issues

The previous two sections covered approaches on how to manage resources in an operating system, both at the lower level through scheduling algorithms and at a higher level through QoS management architectures. In addition, the general structure of an operating systems may also have a significant impact on the way resources are managed. In this section some of the issues related to this impact are discussed by describing a number of example systems.

The discussion starts with an overview of the Nemesis operating system which probably presents the most radical approach (section 2.3.1). This discussion of Nemesis is of particular significance in the context of this dissertation as one of the main evaluation systems for the decentralised architecture is based on Nemesis' key abstractions. Sections 2.3.2 to 2.3.4 then discuss three other approaches for retrofitting more advanced resource management concepts into existing systems.

2.3.1 Nemesis

The Nemesis operating system [LMB$^+$96] was designed and built from scratch (mainly at the Computer Lab, Cambridge University) to offer genuine support for multimedia data stream types by providing QoS guarantees for all shared resources. In more traditional operating systems a significant amount of resources are consumed anonymously, i.e., unaccounted for, because a significant proportion of code is executed in the kernel, or in shared servers, on behalf of processes. In a multimedia operating system this may lead to an undesired effect, termed *QoS-Crosstalk*, where one process could influence the performance of other processes by causing contention for shared resources. In Nemesis, this problem has been addressed by multiplexing shared physical resources only once and at the lowest possible level, facilitating accurate accounting of resource consumption. The resulting operating system is vertically structured [Bar96], with most of the functionality usually provided by traditional operating systems instead of being executed by the applications themselves, implemented as user-level shared libraries (see figure 2.1)[8].

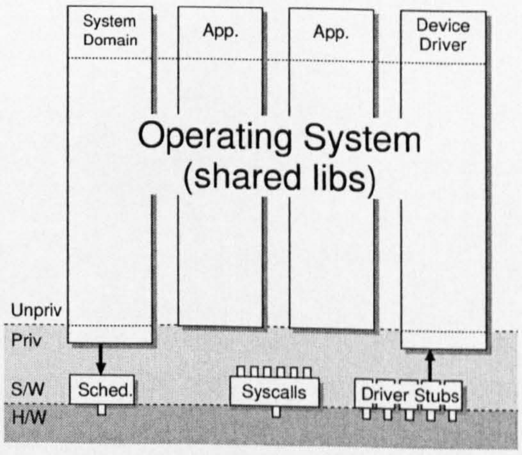


Figure 2.1: Nemesis: A vertically structured operating system

The task model in Nemesis distinguishes between three entities: Scheduling Domains (SDOMs), Activation Domains (ADOMs), and Protection Domains (PDOMs). SDOMs are entities the scheduler allocates the CPU resource to. Each SDOM contains one or more ADOMs which the scheduler explicitly activates via scheduler activations [ABLL92]. Usually there is a one-to-one

[8]This structure is comparable to Exokernel systems [KEG$^+$97], though the motivation behind the design is different. The principal motivation for the Exokernel design was to allow applications to optimise the implementation of various system components using application-specific knowledge.

mapping between SDOMs and ADOMs, forming the closest equivalent to a "process" in more traditional operating systems. Hierarchical scheduling schemes can be formed by associating more than one ADOM with an SDOM, with the SDOM scheduling its ADOMs. Furthermore, it is also conceivable, though not implemented, that one ADOM is associated with more than one SDOM allowing for flexible combinations of resource allocations and executions. PDOMs are the entities on which memory protection is enforced. They form the closest equivalent to an address space in a multiple address space operating system[9].

Nemesis provides QoS guarantees for the following resources: CPU [LMB+96], memory [Han99b], I/O devices such as the network interface [BBDS97] and disk drives [Bar97], the audio device[Ree98], and framebuffer devices [Bar96]. Processes can make reservations for the CPU in the form of a slice of s nanoseconds per period p nanoseconds which are then scheduled using the Atropos scheduler. For memory, individual processes can request ranges of virtual memory which are guaranteed to be backed by a specified number of physical pages. Processes are then responsible for their own virtual memory management. Device drivers for I/O devices are implemented as privileged, user-level processes which register interrupt handlers with the system. The interrupt handler typically only clears the interrupt condition, and sends an event to the device driver process, effectively decoupling interrupt notification from interrupt servicing. The device driver process only implements infrequent out-of-band management functions and performs a single de-multiplexing function for the hardware device (e.g., packet filtering for network devices)[10]. I/O requests are scheduled using the Atropos scheduler which is also used for CPU scheduling. All higher level functionality, such as network stack processing, is performed at the user level, utilising (shared) libraries. Similarly, processes *own* individual pixels or regions of pixels of the framebuffer device and all higher level drawing primitives are performed by the processes themselves. Again, protection and access control is managed by the device driver.

As a result of this OS architecture, most activities typically performed by an operating system kernel are instead performed by the applications. Thus, virtually all resources consumed can be accounted to individual processes and processes can request guaranteed, absolute shares for each resource. In a recent paper [NM01] we have illustrated how this accurate accounting can be

[9]It is worth pointing out, that the current version of Nemesis only supports the one-to-one mapping of ADOM, SDOM, and PDOM. However, Neil Stratford once extended this implementation to include almost arbitrary mappings. Unfortunately, this was never included in the mainstream Nemesis source tree [Str98].

[10]With appropriate hardware support, as provided, for example, by some network cards, de-multiplexing can be mainly performed in hardware, thus reducing the resources needed by the device driver. A software mechanism, known as *call-privs* [BBDS97, Bar96], also allows some of the de-multiplexing costs to be accounted to the clients.

extended to account for battery energy consumption by individual processes as a necessary step towards energy management in mobile devices.

The Nemesis base system includes a complete windowing system, network stack functionality, and simple file system access, implemented mainly as user-level shared libraries. Using this base system we have developed a set of shared libraries providing a Unix-like API, allowing a range of unaltered Unix applications to be used with Nemesis [NB00, NDH$^+$99, ND99, NB98]. This Unix personality also includes support for the X-Window client API `Xlib` [NH99].

2.3.2 Resource Containers

Resource containers [BDM99], developed at Rice University, attempt to provide advanced resource management mechanisms in general purpose operating systems but are specifically targeted at large-scale server systems. The key observation is that in traditional operating systems processes are both protection domains and resource principals, i.e., entities to which resource are allocated and accounted. This creates a mismatch between the requirements of large-scale server applications, where individual activities, such as servicing a client's requests, may require the participation of multiple processes and individual processes may service requests from multiple clients. The resource container work addresses this mismatch by separating the concept of protection domains from that of resource principals, to enable fine grained and robust resource management in server operating systems. Resource containers are the entities to which resources are allocated (comparable to Nemesis' SDOMs). Actual work is performed by *activities*, which may spawn multiple threads in different protection domains, and which are dynamically bound to a resource container. Resources consumed by an activity are accounted to its current resource container.

The resource container abstraction also addresses the problem with monolithic kernel designs that a significant amount of resources can be consumed by the operating system kernel and often are not accounted to the activity causing the resource consumption. A well publicised example is the network subsystem, especially for receiving packets from the network. In traditional operating systems, such as most UNIX flavours, incoming network traffic processing is interrupt driven: the arrival of a packet is signalled by the card through a hardware interrupt. The interrupt handler receives the packet from the card and, after some minimal hardware specific processing, puts it into a protocol specific input queue before scheduling a software interrupt. In the context of the software interrupt handler all the protocol processing is performed before the processed data is placed into a per socket queue from where user programs can retrieve the data. There are two main problems with this approach. First and foremost, as the interrupt handlers are executing at a higher priority than any user level program, a system can *livelock* under heavy load with all CPU resources being spent on processing incoming network traffic — a phenomenom termed Receive Livelock [MR97]. Secondly, CPU resources spent on processing incoming network traffic is largely accounted to whichever process is executing at the time of the interrupt handler execution rather than to the recipient of the network packet. To tackle the receiver livelock problem, Mogul & Ramakrishnan in [MR97] proposed switching between interrupt driven mode and polling under heavy network loads. The resource container prototype uses an alternative technique, called Lazy Receiver Processing (LRP) [DB96], which tackles both problems by de-multiplexing incoming network traffic into separate queues as early as possible (either in hardware or in the hardware interrupt handler) and then performs protocol processing in the context of and with the priority of the receiving process or resource container. Essentially, LRP can be seen as an implementation of Nemesis' general device driver architecture [Bar96] in the context of a monolithic operating system kernel.

The authors of [BDM99] have implemented resource containers in the context of FreeBSD and Digital UNIX 4.0D for CPU scheduling and network interfaces. However, they stress that resource containers are a generic abstraction or mechanism which also allows for the accounting of other resources. For example, *cluster reserves* [ADZ00] extend the notion of resource containers to clusters of commodity PCs used as web server farms. Resource containers on individual cluster nodes are associated with cluster reserves in order to achieve performance isolation between different services hosted by the cluster. [CAT+01] describes a similar system, however, with the primary aim of managing energy consumption in server farms. Resource containers are used to account energy consumption to individual customers of a server farm. [ZFE+01] uses the resource container abstraction to account for battery energy consumption in mobile computing devices. Energy consumed by an activity is accounted to its associated resource container.

2.3.3 Resource Kernels

The resource kernel work [RJMO98, OR98] is built on and significantly extends the earlier work on processor capacity reserves [MST94] which only addressed CPU resources. Instead, resource kernels present an overall resource-centric approach in order to support real-time and multi-media systems where multiple applications with different timing constraints are executed concurrently.

Resource kernels provide a unified abstraction for time based resource reservations. A resource request consists of explicit and implicit parameters. The explicit parameters specify the nett usage of the resource in terms of computation time, C, every T time units, timeliness requirements in terms of a deadline D and starting time S, and the lifetime L of a resource allocation. The implicit parameter B (blocking factor) is used to place an upper bound on the priority inheritance protocol used to prevent priority inversion. The resource reservation abstractions also distinguish between hard, firm, and soft reservations. Processes with hard reservations are only allowed to use their resource reservations of C within T time units and will not receive any additional resources. Processes with firm reservations may use extra resources only if all other reservations have been satisfied. Processes with soft reservations may receive additional resources even if not all reservations have yet been satisfied.

This unified resource model is used in a prototype for both CPU and disk bandwidth. For CPU scheduling a fixed priority scheme based on either deadline or periods (for either an RM or EDF style scheduler) is used with a custom admission control test which takes the exact schedule into account. For disk scheduling an EDF based scheme, called just-in-time scheduling, is used. For higher level resource management policies, the authors refer to the related work on Q-RAM (see section 2.2.3).

In comparison with the two systems described previously, it is unclear how resource kernels deal with accounting of resource consumption to activities, in particular, considering its implementation in an micro-kernel environment, where considerable amounts of resource may be consumed in server processes on behalf of client processes. However, considering the resource kernel work is based on Processor Capacity Reserves [MST94] which uses the *reserve* kernel abstraction to track reservations independently of threads, it may be assumed that a similar abstraction is deployed in resource kernels.

2.3.4 Eclipse

Eclipse aims to provide support for soft real-time, multimedia, and server applications. The most recent variant of Eclipse, referred to as Eclipse/BSD, is based on FreeBSD [BBG+99c, BBG+99a, BGSS99], an earlier version was based on Plan 9 [BGÖS98]. Eclipse/BSD provides QoS guarantees to selected applications for CPU, memory, disk, and network resources through *resource reservations*. A resource reservation determines a fraction of a resource exclusively set aside for use by one or more processes. An admission control check ensures that all resource reservations can be met. Applications can sub-divide their reservation hierarchically, e.g., assign sub reservations to individual tasks or clients.

Applications access the reservation subsystem through a special filesystem mounted as `/reserv`. The top-level directory of this filesystem contains a directory for each of the physical resources. Subdirectories of these directories represent resource reservations. Each reservation directory r contains a file containing two values: a minimum absolute value m_r of that resource and a weight ϕ_r, which is r's share of its parents resource reservation. An admission control test ensures that the sum of minimum resources of child reservations is less than or equal to the minimum resource of the parent reservation. Resource reservations which can not have children are called *queues* and a resource reservation may contain multiple queues. Queues are the entities on which scheduling decisions are made and to which consumed resources are accounted. Each process is associated with a *reservation domain* which is a list of a process' *root reservations*, one for each resource. Each root reservation contains a default queue and system calls are provided to change the queues. This allows processes to execute different work items for different "activities". Reservation domains are accessible through FreeBSD's `/proc` filesystem. By default, child processes inherit the reservation domain of their parent process. However, the user or parent process can create new reservation domains for child processes by assigning new root reservations to the child's reservation domain. These root reservations have to be descending from the parent's root reservations.

The `/reserv` interface provides a flexible abstraction for making and managing resource reservations. Internally, resources are scheduled using proportional share scheduling algorithms. CPU is scheduled using Move-To-Rear List Scheduling (MTRLS) [BGÖS97], a virtual-time based scheduler with special support for processes blocking on I/O operations. Disk bandwidth is scheduled according to YFQ [BBG$^+$99b], another virtual-time based scheduling algorithm. Outgoing network traffic is scheduled using WF^{2}Q [BZ96] while incoming network traffic is managed similarly to the LRP approach deployed in the resource containers implementation. Unfortunately, the mechanism for memory reservations is not described in the Eclipse publications.

2.3.5 Discussion

Recent work on general operating system resource management issues has been motivated by the desire to support either soft real-time, multimedia applications, or server applications. Both application scenarios require performance isolation between, and QoS guarantees for, concurrent "activities". In the most general case, activities are units of computation for which an application wishes to perform separate resource allocation and accounting. Thus, activities can be equivalent to traditional processes or groups of processes, or may be independent of processes, e.g., related to client requests in a server application.

Two different approaches for achieving isolation have been presented. Isolation can either be achieved through the general operating systems structure, as in the case of Nemesis, or through the introduction of new abstractions in existing operating systems, such as resource containers or reservation domains in Eclipse. While the former approach is certainly more radical, the latter provides better support for legacy applications, possibly at the expense of achievable accuracy of resource allocations.

QoS guarantees are typically provided in the form of resource reservations supported by real-time scheduling algorithms. This typically allows better control over the timeliness requirements of multi-media applications. However, proportional-fair scheduling algorithms may also be deployed, especially if the resource shares are policed by an admission control test. In general, the systems discussed only provide fine grained control over resource allocations for some resources (Nemesis being the notable exception). While this is usually sufficient for most applications, it raises the concern that contention for an "unmanaged" resource, e.g., virtual memory, may significantly impact the resource guarantees provided for other resources.

The main difference between the systems discussed is the granularity at which activities are defined. Resource kernels seem to simply map activities to processes, while the other three systems (and others, e.g., activity objects in Rialto [JRR97]), provide mechanisms for similar flexibility for defining activities independently of other OS abstractions, such as protection domains. This is very important for the more general problem of resource management.

In general, it has to be stressed that accurate accounting of resource consumption to entities specific to a particular application domain is the key to the ability to manage and control resources for that application domain.

2.4 Summary

This chapter has provided background on resource management in operating systems. The general task of managing resources can be divided into two parts: deciding how resources should be allocated to competing consumers and the actual multiplexing of resources, e.g., deciding which task to allocate a resource to next for time based resources. The former is referred to as *policy* and the latter as *mechanism*[11].

Some scheduling algorithms, e.g., traditional scheduling algorithms used in general purpose operating systems and, to a lesser degree, proportional fair share algorithms, implement both mechanism and some policy. As a result, it is more difficult to control timely resource allocations. Real-time schedulers, typically used in reservation based systems, provide much better control over the multiplexing of resources. However, they require a separate entity, e.g., an admission control system or a QoS manager, to implement resource allocation policies.

Policies implemented in conjunction with reservation based systems should provide consumers with an *incentive* to make reservations over the appropriate amount of resource. What is appropriate depends both on the user preferences and the general resource availability. Since both may change over time, consumers also need to be given an incentive to renegotiate their resource reservations. Unfortunately, very few systems provide these incentives and they implement very simplistic policies.

The second issue is that most QoS architectures are centralised in order to manage multiple resources, which are typically scheduled independently of each other by the operating system. The centralised approach raises the problem of designing suitable APIs, and also requires applications to provide explicit, often high-level, specifications of their resource demand. Furthermore, it is a non-trivial task to map these QoS specifications to low-level scheduling parameters.

Finally, most approaches to resource management discussed in this chapter, both QoS architectures and scheduling algorithms, fail to involve the application in the management of resources. Scheduling algorithms may "magically" shift resource allocations without notifying the consumers. QoS managers, once a resource contract has been established, typically do not provide feedback to applications, although these applications may be able to choose different resource allocations to satisfy the same level of user perceived QoS, e.g., by trading resources against each other or by correcting an initially overly pessimistic resource estimate.

These issues will be revisited in section 4.6 which discusses how they are addressed by the decentralised resource management architecture.

[11]Note, that multiplexing itself also contains some element of policy. However, this "policy" is typically well defined and enforces low-level resource access. The main policy decisions in resource management are concerned with the amount of resources consumers receive and to deal with changing resource availability.

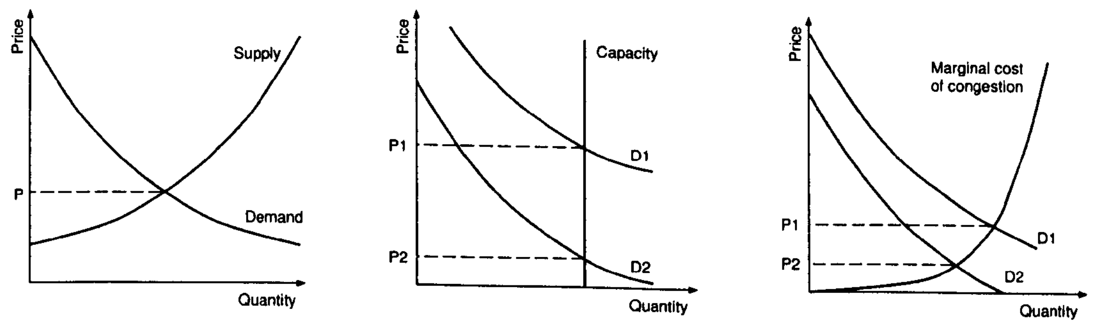
Chapter 3
Pricing computational resources

This chapter provides background information on general economic models and their application to the management of computational resources. It also describes related work, especially related work on congestion or shadow pricing as has been proposed for use in communication networks. Initially, some very basic economic models are introduced. These models are then used to describe, in more detail, specific mechanisms for establishing prices. Finally, how these mechanisms can be applied to establish prices for computational resources is discussed.

In economics the area which is primarily concerned with the process of allocating scarce resource and the establishment of prices for commodities is called *microeconomics*. The basic model used in microeconomics is that of a *market* where *demand* and *supply* determine the price of commodities. Demand denotes the quantity of a commodity that buyers, given a certain price, are willing to purchase; correspondingly, supply is the quantity of a commodity sellers are prepared to sell at each possible price. In general, the higher the price, the fewer buyers are willing to purchase a commodity, while more sellers are inclined to sell the commodity, and vice versa. This relationship is typically expressed in terms of supply and demand curves such as depicted in figure 3.1(a).

From the figure it is clear that, at high prices there is an excess of supply while at low prices there is an excess of demand. However, at some intermediate price the quantity demanded equals the quantity supplied and the market reaches an *equilibrium* at a price known as the equilibrium price or *market clearing* price (marked P in the figure). In economic theory, an ideal market will converge to this equilibrium point and stabilise although buyers and sellers are acting independently of each other and pursue their own interests. One often cited aim for such markets is to achieve *Pareto optimality*: an allocation of resources where no-one can be made better off without someone else being made worse off.

While some researchers have proposed deploying variations of this basic model for allocating computational resources in distributed systems (e.g., [CT93, WHH$^+$92, Wel96]), the model is unsuitable for deployment as a system for resource management in an operating system. The model assumes a large number of buyers and sellers with none being powerful enough to individually have an impact on the market. In an operating system there is typically only one seller per commodity and a small number of buyers (processes). Furthermore, the supply of commodity is limited — processor capacity, network bandwidth, physical memory, etc. are typically fairly scarce resources with individual consumers easily being capable of consuming the entire capacity. Thus, the conditions of an ideal market and perfect competition are not given.



(a) Supply and Demand (b) Fixed Capacity (c) Congestion Prices

Figure 3.1: Supply and Demand — The Basics

However, since the operating system is in control of the supply side of commodities, a resource controller could still model the behaviour of sellers in an idealised market. This leads to the model described in [MV95b, MV96] in the context of pricing schemes for the Internet. As argued above, computational resources are of fixed capacity, thus the supply of the commodity is fixed. If the price for this commodity is high, users would consume less of it and, conversely, if the price for the commodity is low, users are assumed to consume more of it. Thus, the price should be determined by the demand for the commodity. This, again, can be depicted as supply and demand curves (see figure 3.1(b)). The graph shows two different demand curves (D_1 and D_2) for different levels of demand. The supply curve is fixed as the capacity of the resource is fixed. As in the previous model, the optimal, market clearing price is where the demand curve crosses the supply "curve", resulting in the two prices P_1 and P_2 for the two levels of demand.

In [MV95b] a third pricing model is presented. Assuming that a shared resource of fixed capacity is only lightly loaded then one user increasing its share slightly does not create any additional cost. If, however, the resource is utilised close to capacity, that marginal increase may impose an unreasonably high cost on the *other* users in the form of delay or exclusion. In economic terms this *congestion cost* is an "externality", a phenomenon closely related to the "tragedy of the commons" [Har68] and typically *shadow prices* are used to cover these external costs. In other words, shadow prices make users aware of the external cost they impose on others. Suppose it is possible to determine the relationship between the utilisation and the congestion cost, then the marginal congestion cost of an increase of resource utilisation by one user can be determined. Then, as depicted in figure 3.1(c), the optimal price is when the demand curve intersects the curve representing the marginal cost.

Independent of the actual mechanism of establishing resource prices one can distinguish between prices which are based on reservations and prices which are based on actual resource usage. With *reservation based* prices, users make requests for resources and prices are established before actual resource consumption. While this allows the suppliers of resources to give guarantees that users will actually receive the resources they purchased, it requires users to estimate their resource requirements as accurately as possible. Conversely, with *usage based* prices, users are charged only for the resources they consume. In both schemes, prices can be based on demand; in reservation based schemes, the demand for a resource is expressed through the estimates while in usage based schemes the demand is based on actual usage. Note, that usage and reservation based pricing schemes can be combined into a single scheme. See section 3.2.2.2 for an example.

In the next sections different pricing schemes for computational resources are introduced. These are organised into two categories: systems using market based mechanisms (section 3.1) and pricing schemes based on congestion prices (section 3.2).

3.1 Market based mechanisms

As described in the introduction, prices can be formed based on supply and demand. A common approach to determine market clearing prices is through auctions. A number of researchers have proposed deploying auction based schemes for resource allocations in computer systems, e.g., [MD88b, WHH+92, Bog94, MKH+96, Wel96, YWI96]. In this section, a general overview of auctions is given, followed by descriptions of representative systems.

There are a variety of different auction schemes which differ in the way clients submit bids and how the final price is determined. In general, one can distinguish between four major auction types ([Ago96] provides an excellent overview):

- English auction, also known as first-price, open-bid auction: Bidders submit bids with higher and higher prices and the highest bidder wins, paying the final price he bid. Every bidder knows the bids other make.

- Dutch auction: The seller offers lower and lower prices until a buyer claims the item at the last offered price. Every bidder knows the bids other make.

- First-price sealed-bid auction: Fixed bids are submitted and the highest is accepted at the price of that bid. Bidders do not know the bids of other bidders.

- Vickrey Auction or second-price sealed bid auction: Fixed bids are submitted and the highest bid wins at a price equal to the second highest bid. Bidders do not know the bids of other bidders.

A fifth category, though not normally identified as one of the four classic auction forms, is the Double auction. Buyers and seller simultaneously submit bids and offers, generating supply and demand profiles. In a particular variant of the Double auction, known as Continuous Double Auction (CDA), sellers and buyers simultaneously submit offers and bids and at any time a seller is free to accept a bid of a buyer and a buyer is free to accept an offer of a seller. CDA is widely used in financial markets.

While the four classic categories represent quite different auction mechanisms it might seem surprising that, under certain assumptions, they result, at least theoretically, in the same prices. This is the case when the bidders have different private evaluations of the goods on auction, and behave in a risk neutral and symmetric way. If bidders have common values attached to the goods on auction then the English action yields the highest prices followed by the Vickrey auction; Dutch auction and first-price sealed-bid auctions yield similar prices. For Dutch and first-price sealed-bid auctions it is irrelevant if bidders have private or shared common evaluations of the goods because bidders behave the same as they have the same information available. For second price and English auctions it depends on whether the bidders are certain about their private evaluation or are uncertain about the common value of the goods. If bidders have independent private values, both auction schemes yield the same price. Furthermore, the risk characteristics of the bidders plays a role as well. For risk averse bidders first-price and Dutch auction result in higher prices.

For establishing prices for computational resources, sealed-bid auctions have the advantage that they do not require any iteration and do not require clients to be active while bidding. All clients submit their bids and the highest bid wins. Both English and Dutch auctions require some form of iterations and thus may incur a higher runtime overhead. Clients of computational resources have different, independent evaluations of the resource; a certain amount of CPU may be more valuable to a batch processing application than to a simple interactive application. Thus, to a resource manager aiming to maximise its profit, the choice of auction mechanism is largely irrelevant. However, Vickrey auctions have the advantage that they offer a clearer strategy to the bidders: the literature suggests that the dominant strategy for bidders in a Vickrey auction is to submit a bid equal to his true evaluation of the resource (see [MV95b, Ago96] and the literature cited therein).

The simple auction models described above are able to allocate a single good to competing users or multiple items of a good to multiple users only interested in single items. However, if users want more than one item of a good the simple auction models may result in an inefficient allocation, i.e., they may not find the equilibrium price (see [WWWM98, Section 6] for examples). In [VM94, WWWM98] the Generalised Vickrey Auction (GVA) model is presented, an extension of the standard Vickrey auction, which allows for the efficient allocation of multiple goods. The basic idea of GVA is that a winning bidder gets charged the amount which could have been generated by the goods he won had he not participated in the auction. The rationale behind this scheme is that bidders are best off if they truthfully reveal their preferences and evaluation for the goods on auction. More formally, consider a set of $a = 1, \ldots, A$ users, and user a chooses a set of x_a goods; x denotes the choices of all users $x = (x_1, \ldots, x_A)$ and x_{-a} denotes the set of all choices except x_a. Each user is assumed to have a quasi-linear utility function $u_a(x) + m_a$ where m_a is the amount of money held by user a. If the utility functions are strictly concave[1], there will be an allocation x^* which is Pareto optimal. Suppose each user reports a utility function $r_a(\cdot)$, which does not necessarily need to be a truthful revelation of its real utility function $u_a(\cdot)$, then an auctioneer would compute: $x^* = max \sum_a r_a(x)$ and, under GVA, each user has to pay: $V_a = G_a(r_{-a}) - W_{-a}(x^*)$ with $W_{-a}(x^*) = \sum_{b \neq a} r_b(x^*)$ and $G_a(r_{-a}) = max \sum_{b \neq a} r_b(x_{-a})$.

The $W_{-a}(x^*)$ component of the cost represents the total value of the allocation x^* without the user a. The function G_a could be any function, however the function given presents the total value the auction would achieve if user a would not have participated in the auction. Thus, it represents the second-prize analogue to the simple Vickrey auction. In [VM94] and [WWWM98] it is argued that, if users reveal their true preference to the auction, i.e., $r_a(\cdot) = u_a(\cdot)$, then GVA will compute an optimal allocation. However, as noted in [WWWM98], the GVA mechanism for auctioning multiple goods is an NP-complete computation.

[1]This is a common assumption indicating a diminishing marginal utility as x increases.

In the following sections, a number of systems deploying an explicit auction mechanism are described. Not surprisingly, most systems deploy a variant of Vickrey auction, as it does not require any iterations of the auction process and encourages users to reveal their real preferences.

3.1.1 Spawn

The Spawn distributed computation economy [WHH+92] represents one of the first implementations of a market based resource allocation. It allows for the coarse grained, decentralised allocation of idle computational resources in a network of heterogenous workstations. On each workstation an auction process controls the sale of idle computing resources to potential buyers. Idle computation resources are divided into slices and are auctioned one at a time. During a slice, a process has exclusive access to the resource. The auction process continously accepts bids for the next available slice. Potential buyers place bids consisting of a length of time and a quantity of funds. For the basic auction model a Vickrey auction was chosen because it provides the incentives to clients to place bids corresponding to their real evaluation. The basic auction process can be parameterised to accommodate bids with varying lengths. The authors suggest using a linear function relating cost to the length of time of the bids, allowing for the amortisation of startup costs for the winning bidder while avoiding inconvenience to a user returning to his "idle" workstation. However, it is unclear how this parameterisation is factored into the auction process.

Applications consist of worker modules and managers. Worker tasks implemented by the worker modules perform the actual computation of the application. The simplest case constitutes a single worker task corresponding to a monolithic application executing in a single process. However, decomposable applications which are aware of the underlying distributed nature of the system may have a number of tasks and subtasks forming a tree structure of tasks. Managers are associated with worker tasks and are responsible for the execution of worker tasks. Thus, a manager and a worker task form a subtask of the application. Managers of decomposable applications may choose to spawn additional subtasks on other nodes to complete the application's work. Before a worker task can be created at a new node a new manager for that worker task needs to be spawned on that node. The manager then takes part in the auction, and if successful, creates the new local worker task.

The basic concept of funding deployed in Spawn is that of sponsored computation. A manager serves as a funding sponsor for all its children tasks, allowing it to dynamically control the relative fraction of funding allocated to its children and thereby forming a sponsorship hierarchy. A top-level manager controls the total amount of funding available to the application, while users control the amount of funding available to their applications, and, ultimately, system administrators control the amount of funding available to users. Neither managers nor users can create nor destroy funds, however, the prototype implementation does not enforce this. Managers can deploy a number of strategies to allocate funds to their subtasks. In a simple strategy a manager would allocate funds equally to all its subtasks while a more sophisticated manager may allocate more funds to more efficient subtasks, e.g., tasks located on faster workstations.

3.1.2 Agoric systems

Miller and Drexler have authored several papers advocating the use of marked-based mechanisms for resource management in computer systems [MD88a, MD88c, MD88b, MKH+96, MTHH97]. They refer to systems which apply economic techniques to the management of computational resources as *agoric systems*[2].

In [MD88b] an auction algorithm for scheduling CPU resources is presented. The algorithm is called the escalator-bid auction and is a variation of the Vickrey auction. The variation was introduced to tackle the problem of process starvation; if the market price for a slice of the processor stays above a process' bid, the process will never run, and, thus, won't be able to raise its bid. In the escalator-bid auction, processes with an initial bid are placed on an "escalator" which linearly increase the bid over time depending on the speed of the escalator. Different escalators have different "speeds". The process with the highest bid is selected, and after executing a cycle of the escalator, i.e., increasing the bids of other processes, it is charged the highest bid (i.e., the second-highest bid before it was removed). The initial bid and the escalation rate together form the priority of a process. If a selected process does not have enough funding in its expense account it will not be run, but placed on a stationary escalator with a bid equalling its expense account. The system has two additional parameters, the maximum initial bid and the maximum escalation rate. Since bids for all processes on non-stationary escalators will eventually grow large enough, starvation is not an issue. The authors argue that a process' initial bidding strategy should be to place it on the fastest escalator with an initial bid of zero. Under this condition the escalator-bid auction guarantees non-starvation to all processes.

Based on the more general work described in [MD88a, MD88c, MD88b], the authors developed an auction system for allocating bandwidth in an Asynchronous Transfer Mode (ATM) network [MKH+96], with the related patent [MTHH97] containing some additional informa-

[2]From the Greek word "Agora" for a square serving as a marketplace.

tion. The auction method is a variant of the Vickrey auction. Users submit their bids in bid slates consisting of any number of individual bids for a combination of resources[3]. When bid slates are submitted to the auction process, first, all individual bids are eliminated from a slate for which an individual resource request exceeds the capacity or a maximum allocation for that resource. The auction process then selects the user with the highest bid as the winner. Unfortunately, due to the inherently incomprehensible patent language, it is somewhat unclear how the prices for the winning bids are calculated. However, at least from the textual description, it appears that a variant of the GVA auction is deployed.

The use of bid slates allows users to bid for different allocations of multiple resources and the GVA style auction ensures that an optimal allocation is found for competing users. Furthermore, GVA also encourages the users to reveal their real evaluations of the resource allocations. In this context it is worth noting that the GVA auction is not aiming at maximising monetary return to resource managers but to allocate resources to maximum declared user values. However, given the computational complexity of GVA it is unclear at which timescales resources are reallocated. Furthermore, the process requires a centralised auction process with a global model of the network which accepts bid slates from users scattered around on the ATM network.

3.1.3 WALRAS

Michael Wellman's WALRAS system [Wel96] provides an infrastructure for implementing market based control systems. It is built around the concept of a market-oriented programming environment and paradigm offering an object-oriented implementation of general equilibrium theory in which two basic classes of agent, consumers and producers, interact solely via a bidding protocol. Consumers submit bids which maximise their utility and producers aim at maximising their profits. Based on the two basic classes of agents, subtypes of these can be created, which either implement more specific but still general features of an agent, or provide application specific roles for agents. Producers and consumers submit supply and demand bids in the form of a partial function of quantity versus price. The WALRAS system then computes the equilibrium price for that auction. In essence, the WALRAS system implements the perfect competition model depicted in figure 3.1(a).

[3]The definition of bid slates differs slightly between [MKH+96] and [MTHH97]. However, since the latter provides a more detailed description of the bidding process its definition is used in the main text. In [MKH+96] a bid slate is formed from non-overlapping, monotonically increasing bid segments describing a straight line segment on a graph of price as a function of quantity of one resource.

In [CW98] a detailed accord of the algorithm deployed in WALRAS to achieve equilibrium is given. The WALRAS algorithm is based on the classic Walrasian tatonnement process in which agents respond to price signals for individual goods sent out by an auctioneer. More specifically, an auctioneer is responsible for each separate good. Each auctioneer starts with sending out a randomly chosen price for its good to agents. Agents then compute their demand functions for the individual goods they are interested in (consumers are interested in goods which are part of their utility functions and producers are interested in goods they produce) and send them back to the corresponding auctioneer — the agents submit their bids for individual goods. The auctioneer then computes the market clearing price from the bids and notifies the bidders of the new price. These in turn re-evaluate their demand functions. This iteration is continued until an equilibrium is reached. The important feature of this auction mechanism is that it is asynchronous, i.e., agents can submit bids at any time. The main difference between the WALRAS mechanism and the classic tatonnement process is that in WALRAS, agents respond with demand functions rather than single points on their demand curve. The authors show that the WALRAS algorithm converges to the price equilibrium under the assumption of convex preferences (demand and supply functions) and gross substitutability[4]. However, as noted in [Wel96], the WALRAS model does not allow for the modelling of monopolies or small groups of agents, where the actions of one agent may have a significant impact on the price or the other agents.

The WALRAS system has been used to model a number of non-computing related resource allocations problems, e.g., [Wel96] describes its application to a distributed transportation resource allocation and information services within networks.

In [YWI96] the application of the WALRAS system to the problem of network bandwidth allocation in a virtual networked meeting environment, called FreeWalk, is described. FreeWalk provides a three-dimensional space in which users, represented by their corresponding audio and video feeds, can informally meet. Each user, represented by a FreeWalk client, can move around freely and perceives the other participants relative to their distance, i.e., users further away are displayed in smaller windows mapped into the distance in a three dimensional space, while participants closer by are displayed in larger windows. Participants outside a specified distance are not displayed. As each user has a different, constantly changing view of the three-

[4]"Two goods are gross substitutes if an increase in the price of one of the goods causes an increase in the demand for the other" [Var92, p. 395]. N.B. according to Cheng and Wellman [CW98] gross substitutability is only essential for the *formal* proof of convergence to an equilibrium, while in experiments, convergence can be observed without this assumption.

dimensional meeting space, the corresponding audio and video streams are sent directly between the users, i.e., are not multicast. A so called community server is responsible for tracking users' positions in the meeting space and to allocate bandwidth to clients.

The market model for bandwidth allocation in FreeWalk uses four different types of goods: two network goods, raw bandwidth and QoS, and two different time periods, current and future. The distinction between raw bandwidth and QoS goods is motivated by the idea that QoS (different levels of service) is "produced" from raw bandwidth by the agents in the market, and that QoS is what clients primarily value. Thus, client 1 may receive QoS level $q_{1,2}$ for the communication with client 2 at price $P_{1,2}$. The distinction between current and future time periods is introduced to provide incentives to inactive clients to transfer network resources allocated to these clients to more active participants.

Each client receives an initial endowment of goods which it is free to trade at the current market price. In the FreeWalk system, endowments only consist of current and future bandwidth, and do not contain QoS goods, as they must be produced from bandwidth. Each consumer has a known utility function which is dependent on its endowment both in current and future bandwidth and its current and future QoS. The authors assume a known mapping between the bandwidth and QoS which is dependent on the distance of the participants in the spatial meeting space. The notion of future QoS is then given by the same mapping for the future bandwidth and the distance factor set to unity. Consumers are the individual Freewalk clients which naturally seek to maximise their utility given the current market prices and their endowment. Producers are modelled in the community server and attempt to maximise their profits based on the current price and a set of technically feasible bandwidth/QoS pairs (technologies). Prices are determined by the WALRAS algorithm described above.

The motivation for the two time periods, current and future, allows clients to trade off current bandwidth with future bandwidth in return for future QoS. The WALRAS algorithm is executed in fixed slices and deploys a rolling time horizon of T timeslices. Then the current period refers to the current time slice and the future period refers to the remaining $T-1$ slices. At each increment of the time slice, each client is allocated a new endowment in current bandwidth and new future bandwidth based on its current share of future bandwidth. Unfortunately, the authors do not discuss how the initial endowment with bandwidth is determined nor how clients entering the system later are initially endowed.

In [Bog94] a number of market mechanisms for allocating computing times based on the general WALRAS approach are discussed. The author considers various models including auctioning off individual time blocks of CPU time and creating a futures market for time blocks of CPU time. These are deemed impractical as they require processes to know in advance their computing time demands and cannot account for processes whose computing time demands depend on other processes to complete. Instead, the author of [Bog94] proposes a processor rental scheme. In this scheme, tasks bid for the processor and the task with the highest bid is awarded the processes until it either runs out of credits, completes its computation, or is outbid by another task. The winning task is charged a price between the second highest bid and its own bid, thus modelling a Vickrey auction[5].

3.1.4 Discussion

In this section a number of market based resource allocation systems for computational resources were introduced. The resource allocation problem is typically expressed in classic market economic terms with buyers, sellers, and a market or auctioneer to establish market clearing prices. This model may be applicable in the context for which it has been proposed, i.e., distributed computing systems, where multiple market agents supply and demand resource. However, for local resources managed by an operating system there is typically only one supplier offering a fixed capacity resource and only a small number of consumers. Furthermore, there is no per unit cost associated with producing an extra unit of a resource, e.g., there is a one-off cost in providing the CPU, but then offering cycles is essentially free. Furthermore, operating system resources can become congested (see next section), causing additional costs typically not considered in market based approaches. However, since the operating system controls the resources it could still model the supply side of a market, but I argue that markets are simply the *wrong* economic model for managing operating system resources.

Furthermore, some of the approaches discussed share the same problems as some of the QoS architectures discussed in the previous section: complex APIs for bid vectors and complex, NP complete algorithms for establishing prices at the equilibrium. They can however provide incentives to consumers of resources to truthfully reveal their resource preferences. A further critique of the market based approach is that it may require all clients to be synchronised, i.e., submit bids at the same time. None of the reviewed systems addresses this issue in detail.

[5]Note that this work appears to precede the WALRAS algorithm described above.

3.2 Congestion Prices for Communication Networks

Recently, the application of pricing mechanisms to communication networks, in particular the Internet, has received much attention in the research community. It is generally understood that a network offering different classes of services to users requires some pricing mechanisms to encourage users to choose the appropriate service class for their traffic (e.g., [She95b, MV95c]). This is required due to the limited supply of network resources; if network capacity was available in abundance, there would be no need to prioritise traffic of individual users or applications. However, if demand for, say bandwidth, exceeds the supply, mechanisms for allocating resources become more important.

In this context, it is important to distinguish between two types of cost: fixed costs for providing the network infrastructure and variable costs dependent on the users' network usage. While the fixed cost is easily recoverable through fixed subscription fees, variable usage based costs are more difficult to determine. Ideally, usage based prices should accurately reflect costs so that users can compare the benefits of their actions to the cost of their actions [MV95c]. Consider a lightly loaded network. A user sending an additional packet does not cause any additional costs, while in a heavily loaded network an additional packet could cause congestion and therefore cause delay or packet loss to other users. Thus, if prices reflected these additional *social* costs, a user would decide whether his marginal benefit of sending an additional packet is greater then the marginal social cost caused by the additional packet.

In economics, congestion is called an "externality". In general, an externality exists when the action of one agent directly affects the environment of another agent [Var92, p. 432ff]. Classic examples for externalities are pollution, or road congestion. In general, agents seek their own benefits without considering the cost they impose on others. Consider a firm A which produces x units of good at a cost $c(x)$. Then, if the price for a unit of the good is p then firm A seeks to optimise $max\ px - c(x)$ with an equilibrium solution of $p = c'(x)$. Now assume, that the production of good x imposes a negative, external cost of $e(x)$ on firm B, then a merged firm would seek to optimise $max\ px - c(x) - e(x)$ with a solution of $p = c'(x) + e'(x)$. Economists say that $c(x)$ denotes the private cost and $e(x)$ the social cost of good x. Prices which capture the social cost are known as *shadow prices*.

There are a number of ways to deal with externalities. In small communities social norms may be established to prevent congestion externalities. However, in larger, non-cooperative environments pricing mechanisms are more appropriate. There are a number of ways to determine shadow prices, and researchers have proposed a number of mechanisms in the context of communication networks. In the following sections some examples are presented.

3.2.1 Smart-Markets — MacKie-Mason and Varian

MacKie-Mason and Varian are two economists who have been investigated pricing mechanisms for the Internet. In [MV95b] the authors argue that users of the Internet should face prices which reflect the real resource costs and consider five different types of cost:

1. The fixed cost of providing the network infrastructure.
2. The incremental cost of connecting to the network.
3. The incremental cost of sending another packet.
4. The social costs of delaying other users' packets when the network is congested.
5. The cost to expand the network capacity.

The authors argue that costs of type 1 should be covered by a flat access fee and costs of type 2 should be paid for by the user setting up the connection, as a one-time connection fee. Essentially, costs of the first two types form the traditional pricing model for the majority of the Internet today. The other three types of cost are related to the actual usage and the degree of congestion in the network. In summary, the authors argue that prices should be based on three principles: (1) a positive packet charge close to zero when the network is not congested; (2) a larger positive packet charge when the network is congested and; (3) a fixed connection charge.

In the appendix of [MV95b] and, in greater detail in [MV95a], a more formal treatment of this model is presented. Let x_i denote a user's use of the network resource and $Y = X/K$ denote the total network utilisation, with K denoting the network capacity and $X = \sum_{j=1}^{n} x_i$ the total use of the resource. External costs, such as delay and exclusion depend on Y. Users' utility functions are of the form $u_i(x_i, Y) + m_i$, with m_i denoting the number of credits that a user has to spend on other things. In general, it is assumed that u_i is a differentiable and concave function of x_i and a decreasing concave function of Y. Further, assume that $c(K)$ is the cost of providing the capacity K.

Given a fixed capacity K an efficient allocation maximises the welfare $W(K)$, i.e., the sum of the benefits minus the costs:

$$W(K) = \max_{x_j} \sum_{j=1}^{n} u_j(x_j, Y) - c(K) \tag{3.1}$$

with the solution satisfying the first order condition:

$$\frac{\partial u_i(x_i, Y)}{\partial x_i} = -\frac{1}{K}\sum_{j=1}^{n}\frac{\partial u_j(x_j, Y)}{\partial Y} \tag{3.2}$$

This equation gives the social optimum for the resource allocation, where the marginal benefits from the usage for user i equals the marginal costs he imposes on the other users. Equation 3.2 denotes the shadow price p_e of congestion, which measures the total marginal congestion cost that an increase of x_i imposes on the users:

$$p_e = -\frac{1}{K}\sum_{j=1}^{n}\frac{\partial u_j(x_j, Y)}{\partial Y} \tag{3.3}$$

If the price for resource usage is p_e then, for user i, the maximisation problem becomes:

$$\max_{x_i}\ u_i(x_i, Y) - p_e x_i \tag{3.4}$$

with the solution:

$$\frac{\partial u_i(x_i, Y)}{\partial x_i} + \frac{1}{K}\frac{\partial u_i(x_i, Y)}{\partial Y} = p_e \tag{3.5}$$

Considering the definition of p_e in (3.3), for large n the second term on the left hand side will be negligible relative to p_e and the solution becomes equivalent to the solution of the social optimum.

Using this basic model, MacKie-Mason and Varian consider a number of scenarios. For example, it can be shown that the shadow price, apart from providing a measure of social cost, also provides the incentive to expand the capacity under certain conditions (namely: $W'(K) = p_e\frac{X}{K} - c'(K)$). Further, it is shown, that in a competitive market with a number of suppliers of network resources, suppliers are forced (due to competition) to charge the socially optimal prices given in equation 3.2. Further analysis investigates the schemes without usage fees and in a monopolistic environment (see [MV95a] for details).

In [MV95b] the authors propose implementing the usage based charges in this model as a "Smart Market" where prices fluctuate constantly reflecting the current degree of congestion. This could be implemented using a variant of a second-price sealed-bid auction whereby each packet contains a field indicating how much its sender is willing to pay. The network admits all packets with bid prices exceeding the current cutoff amount determined by the marginal congestion cost imposed by the next additional packet.

3.2.2 Frank Kelly et. al.

Frank Kelly and his collaborators have developed a number of pricing schemes for communication networks. While these are conceptually similar to the scheme proposed by MacKie-Mason and Varian (see previous section), their treatment is mathematically more rigorous. Much of this work has been carried out in the context of the CA$hMAN project[6]. This work had a great influence on research on congestion pricing in networks.

Most of Kelly et al.'s work has been carried out in the context of multi-service ATM networks, but the results have also been applied to packet switched networks, such as an enhanced Internet, offering differentiated or integrated services. In general, it distinguishes between elastic traffic and traffic requiring guaranteed services. Elastic traffic corresponds to the ABR traffic class in ATM networks, which provides a best effort service with an explicit congestion control mechanism; and guaranteed services correspond to the CBR, VBR, and ABR with specified Minimum Cell Rate (MCR) traffic classes, which provide varying degrees of guarantees to users (see [ATM99] or a computer networks textbook, e.g., [PD00, chapter 6.5], for a detailed description of these service categories). The use of an economic framework is partially motivated by the desire to provide incentives to users of a network to choose an appropriate service class. For example, a customer willing to pay twice as much for bandwidth in the best effort class should receive twice as much bandwidth.

3.2.2.1 Elastic traffic

A congestion pricing model for elastic traffic is presented in [Kel97b] which considers ABR traffic with zero MCR. In [KMT98] the authors consider, in greater detail, fairness and convergence criteria of this model especially in the context of large scale networks. In [GK99b] the authors discuss how this congestion pricing model can be used to implement a decentralised transmission rate control in the end-nodes of a packet switched network such as the Internet. End-nodes are provided with sufficient information and the correct incentives to use the network efficiently. Peter Key [Key01] provides an excellent introduction and summary of this work.

The basic congestion pricing model is similar to that presented by MacKie-Mason and Varian and argues that users should be charged the marginal increment in cost that a marginal increment in load causes. More formally, consider a network as a set of J resources with K_j denoting the finite capacity of each of the resources $j \in J$. Let $r \in R$ denote a route, i.e., a

[6]Charging and Accounting Schemes in Multi-service ATM Networks, ACTS Project AC-039, September 1995 to August 1998.

non-empty subset of J, and associate a route with a user. Let $A_{jr} = 1$ if $j \in r$ and $A_{jr} = 0$ otherwise. Further, define the 0–1 matrix $A = (A_{jr, j \in J, r \in R})$ and the vector $K = (K_j, j \in J)$. If a user receives a rate x_r along a route r then his utility is denoted by $U_r(x_r)$. $U_r(x_r)$ is assumed to be an increasing, strictly concave and continously differentiable function of x_r for $x_r \geq 0$, i.e., the traffic is elastic. Let $x = (x_r, r \in R)$ and assume that utilities are additive, then an optimal allocation of rates would solve:

$$\begin{aligned} maximise &\quad \mathcal{U}(x) = \textstyle\sum_{r \in R} U_r(x_r) \\ subject\ to &\quad Ax \leq K \\ over &\quad x \geq 0 \end{aligned} \tag{3.6}$$

However, as argued above, if a resource is heavily loaded the network incurs some cost $C_j(y_j)$ such as delay or loss depending on the load y_j of the resource. Then the optimisation problem becomes:

$$\begin{aligned} maximise &\quad \mathcal{U}(x) = \textstyle\sum_{r \in R} U_r(x_r) - \sum_{j \in J} C_j(\sum_{r \in R} A_{jr} x_r) \\ subject\ to &\quad Ax \leq K \\ over &\quad x \geq 0 \end{aligned} \tag{3.7}$$

If $C_j(y_j) = C_j(\sum_{r \in R} A_{jr} x_r)$ is differentiable with $\frac{d}{dy} C_j(y_j) = p_j(y_j)$ and $p_j(y_j)$ is a non-negative, continously increasing function for $y \geq 0$ then $\mathcal{U}(x)$ has a unique maximum [KMT98]. For $x_r > 0$ at the optimum $U_r'(x_r) = \sum_{j \in r} p_j(y_j)$ is satisfied. The function $p(y_j)$ represents the marginal increase in cost at the resource for a marginal increment in load, while $U_r'(x_r)$ represents the marginal increase in utility. Equilibrium is reached when the marginal increase in utility matches the marginal increase in cost.

While the above optimisation problem is mathematically tractable, it requires knowledge of users' utility functions, which are unlikely to be known to the network. Instead, Kelly proposes to separate the problem into an optimisation problem for the users and one for the network. Suppose a user is charged a price λ_r per unit transmitted and is allowed to freely vary x_r. Then optimisation can be decomposed:

$$\begin{aligned} User &\quad \max_{x_r} \ U_r(x_r) - \lambda_r x_r \\ Network &\quad \max_{\lambda_r} \ \textstyle\sum_{r \in R} \lambda_r x_r \end{aligned} \tag{3.8}$$

In [Kel97b] it is shown that there does exist a price vector $\lambda = (\lambda_r \in R)$ which solves both the user and the network optimisation problem as well as the utility maximisation of equation 3.6.

For an alternative formulation, assume that a user is willing to pay an amount w_r per unit of time and receives in return a flow of rate x_r proportional to w_r, i.e., $x_r = w_r / \lambda_r$. Then the user optimisation problem becomes max $U_r(w_r/\lambda_r) - w_r$. If the users' utility function is of the form[7] $U_r(x_r) = w_r log(x_r)$, which fulfils the requirements on the users' utility functions stated above, then at the optimum $w_r = x_r\lambda_r = x_r \sum_{j\in r} p_j(y_j)$. The resulting allocation x is proportionally fair to the unit charge.

Kelly et. al. suggest that each resource should continously generate feedback signals at rate $y_j p_j(y_j)$ and each user receives a proportion x_r/y_r of these feedback signals, i.e., each user is charged proportionally to his share of the resource. Users are then encouraged to adapt their rate accordingly. More specifically, Kelly et. al. suggest that elastic users execute a Willingness-To-Pay (WTP) algorithm:

$$\frac{d}{dt}x_r(t) = \kappa\Big(w_r(t) - x_r(t)\sum_{j\in r} p_j(t)\Big) \tag{3.9}$$

This algorithm provides a steady increase of the rate proportional to $w_r(t)$ and a decrease of the rate proportional to the rate of feedback signals (charges). The constant κ influences the rate of convergence. In [KMT98] it is shown that, for a constant $w_r(t) = w_r$ this decentralised system converges to the stable point with a proportional fair allocation as described above. In essence, the system forms a classic Walrasian tatonnement process, described in section 3.1.3, where users respond to price signals from the resource and is in contrast to the "smart market" model described in section 3.2.1. In [KMT98] the effects of stochastic perturbation on the measurement of the load y_j and time lags in the delivery of the feedback signals are also considered.

While the presentation so far has considered optimisation problems and convergence, only qualitative statements about the shadow prices $p_j(y_j)$ have been made, namely, that each increment in load should be charged the cost increment that it causes. In [GK99b] the authors consider two network models (slotted-time and a more realistic queueing model) which show that the shadow prices are straightforward to identify. In the slotted-time model a resource has the capacity per slot to transfer N equally sized packets with any excess lost. Thus, lost packets can be assumed to be the cost of congestion. The authors demonstrate that for loads which are generated by independent Poisson random variables as "users", a mechanism which ***marks*** each packet arriving in a time slot in which the total capacity is exceeded, produces a charge (in marks) equalling precisely the shadow price at the resource, as derived from the Poisson characteristic of the users' traffic. Furthermore, the charge per unit time for each user r is precisely the fair charge $x_r p(y)$ described above. For more general statistics other than Poisson the authors of [GK99b] demonstrate that for *small* increments of the load the above identities hold.

[7]Note, that [KMBL99] shows that this is not a necessary requirement.

For the more realistic queueing model, a finite length queue is modelled which can serve a single packet per unit time. If more than one packet arrives each time unit, the queue fills up and if its capacity is exceeded, packets are lost. In this model a *busy period* is defined between the time when the queue starts filling up and the time only one packet is left in the queue. Again, the cost can be identified as the number of lost packets, and ideally every packet which passes through the queue from the start of a busy period until a packet loss occurs should be marked/charged as it contributed to the packet loss. However, while in the slotted-time model it is quite easy to mark packets contributing to packet loss, this is impracticable in the queueing model as it is difficult to determine in advance if a packet passing through the queue will in fact contribute to a packet loss for a packet arriving later. Instead, the Gibbens and Kelly propose to maintain a count of packets leaving the queue from the start of a busy period and at the time of packet loss mark all packets currently in the queue and a sufficient number of packets afterwards. While this doesn't mark all packets contributing to the packet loss it will mark the right number of packets. In an alternative scheme all packets leaving the queue after a packet loss until the end of a busy period could be marked.

For both models the authors of [GK99b] present the results of some experiments with different control mechanisms in the end-notes designed to achieve different user objectives. Elastic users implement a variation of the WTP algorithm presented above; intermittent users behave as elastic users for a random period of time before being inactive for a random period of time; a file transfer user attempts to transmit a file as quickly as possible for a fixed amount. The experiments demonstrate that elastic users with different w_r achieve different, proportional service rates and that under fluctuating demand, introduced by intermittent and file transfer users, the users manage to share the resource between them keeping the total throughput approximately constant.

Other people have investigated how this pricing scheme for elastic traffic can be implemented in more realistic network environments. Courcoubetis et. al. [CSS96] describes an integration into the flow-control mechanism for ABR traffic in ATM networks. Essentially, the Resource Management (RM) cells are used to convey the congestion information from the switches to the end-systems. Key et. al [KM99, KMBL99] describe an implementation and simulation results for TCP/IP networks using the proposed Explicit Congestion Notification (ECN) mechanism [Flo94] to convey the marks/charges to the end-systems.

One obvious drawback of Kelly's idealised charging scheme is that it may send out feedback signals too late, i.e., after packets were lost. Furthermore, depending on the timescales, e.g., Round-Trip Time (RTT), feedback signals may arrive after any transient congestion has occurred and clients may have adjusted to a congestion situation that no longer exists. In order to send feedback signals early, i.e., before packet loss occurs, Gibbens and Key suggest using a Virtual Queue (VQ) shorter that the real queue and base per packet charges on packet loss in the virtual queue [GKT00, GK01] (Kunniyur and Srikant discuss a similar approach [KS01b, KS01a]). Another implementation related issue recently discussed is the debate about using single bits versus multiple bits to convey the charges [Str00]. Using a single ECN bit the end-system has to average over a significant number of packets in order to determine the charging rate, thus potentially exaggerating the delayed feedback caused by the RTT delay. Instead, Stratford and Barham [Str00] have experimented with multiple bits per packet header and provide positive as well as negative feedback to the end-systems, depending on whether a link is idle or congested (Ganesh et al. [GLS00] present a similar approach using non-binary prices).

3.2.2.2 Guaranteed services

For charging guaranteed services Kelly et. al. [Kel97a, CKW97, CS98, CKSW98] propose a Time-Volume scheme $a_0 \times T + a_1 \times V = T(a_0 + a_1 \times M)$ which accounts for both resource reservations through the time component T and actual usage through the volume component V. The cost of a connection is dependent on static parameters, such as the parameters for the token bucket used for policing the traffic contract, and dynamic parameters which can be easily measured, such as volume. Time-Volume pricing is a specialised form of a family of per unit time charges which are expressed as linear functions of the form:

$$a_0 + a_1 g_1(X) + a_2 g_2(X) + \cdots + a_n g_n(X) = a_0 + a^{\top} g(X) \tag{3.10}$$

with $g_1(X) \ldots g_n(X)$ being measurements or functions of measurements from observations $X = (X_1, \ldots, X_T)$. The coefficients $a_0, \ldots, a_n$ are solely dependent on static parameters, such as policing parameters. Charges are thus linear in the chosen measurements. This linear dependency makes it easier to implement the charging at run-time as well as making charges more transparent to users.

To determine the coefficients $a_0, \ldots, a_n$ the authors use the notion of effective bandwidth [Kel96]. Conceptually, effective bandwidth refers to the bandwidth a link needs to reserve for a connection in order to satisfy its particular statistical characteristics and requirements. The general form of the effective bandwidth of a source is:

$$\alpha(s,t) = \frac{1}{st} log E\left[e^{sX[0,t]}\right] \quad 0 < s, t < \infty \tag{3.11}$$

with $X[0,t]$ being the total load produced in the epochs $1,\ldots,t$ and s and t being system parameters which depend on the characteristics of the multiplexed traffic and link parameters such as capacity and buffer size[8]. Since the notion of effective bandwidth provides a way to assess the resource usage, charges can be based on the effective bandwidth of a source. However, this requires the effective bandwidth of a source to be estimated. In [Kel97a] Kelly considers a number of schemes, such as using a priori information like the characteristics of past sources of the same type, and a posteriori measurements. However, he argues that both these approaches have severe drawbacks. The a priori scheme is analogous to an all-you-can-eat restaurant where prices are based on the average amount users have consumed before and therefore encourages users to consume more resources than they need and penalises users which do not consume up to the average. Effective bandwidth estimates based on a posteriori measurements do not take into account a priori expectations of the users. Consider a user requesting, in good faith, a connection with a high peak rate but then only transmitting very little traffic. Using a posteriori measurements, the user will only be charged very little, even though the a priori estimates might have been much higher. Essentially, the network is carrying too much risk.

Instead, Kelly proposes a scheme which is based on equation 3.10 and defines an upper bound $\bar{\alpha}(m,h)$ on the greatest effective bandwidth possible with a traffic contract h and measured mean rate m. $\bar{\alpha}(m,h)$ is concave in m and h may be interpreted as a policed peak rate. Through Lagrangian methods[9] equation 3.10 can be rewritten as:

$$a_0 = \bar{\alpha}(m,h) - \lambda_m^{\top} m, (a_1[m,h],\ldots,a_n[m,h]) = \lambda_m^{\top} = \left(\frac{\partial\bar{\alpha}(m,h)}{\partial m_1},\ldots,\frac{\partial\bar{\alpha}(m,h)}{\partial m_n}\right) \tag{3.12}$$

and the simplified version of the per unit time charges form of the Time-Volume charge becomes:

$$\bar{\alpha}(m,h) - \frac{\partial\bar{\alpha}(m,h)}{\partial m_1}m + \frac{\partial\bar{\alpha}(m,h)}{\partial m_1}M \tag{3.13}$$

where M is the measured mean rate and m_1 is a value for the mean rate announced by the user before connection admission[10]. In essence this equation defines a set of tangents on $\alpha(m,h)$ with the user choosing one by specifying the mean rate m_1.

The coefficients of the pricing scheme are dependent on a conservative approximation of the effective bandwidth $\bar{\alpha}(m,h)$. To be efficient and easy to implement, this approximation should be simple and in [CS98, CKSW98] the authors propose a number of different schemes: The

[8]For example, the effective bandwidth of an on/off source of peak rate h and mean rate m is $\alpha(s,t) = \frac{1}{s}log\left[1+\frac{m}{h}(e^{sh}-1)\right]$. See [Kel96] for more examples.

[9]Lagrangian methods are a standard mathematical method to find optima under constraints.

[10]See [Kel97a] for a similar description for users with unknown peak and mean rate.

first, called "on-off" or "peak/mean", is based on an on/off traffic source providing an upper bound to $\alpha(m,h)$ which solely depends on the peak rate which is assumed to be policed. If, in addition, a token bucket policing mechanisms is used at the source, a tighter upper bound can be defined. This bound is called "simple bound". A third approximation is based on the observation that the worst case output of a token bucket policed source consists of blocks of inverted T patterns. By using a statistical model of this pattern the "T approximation" can provide an upper bound to $\alpha(m,h)$. Experiments with trace data derived from real traffic shows that "simple bound" and "T approximation" yield considerably better results than the "on-off" approximation, as their approximation of $\alpha(m,h)$ is more accurate.

As a concluding remark, it has to be noted that the Time-Volume scheme described in this section can also be used in conjunction with ABR source which have MCR requirements. Traffic up to the specified MCR is charged according to the Time-Volume scheme and traffic exceeding the MCR is charged according to the congestion prices described in the previous section, thus combining the two pricing schemes.

3.3 End-user perspective of network pricing

Elaborate schemes proposed for network pricing, especially in the context of Internet pricing, pose an interesting question on the effect and impact on end-users. In [Odl01] Andrew Odlyzko provides an interesting insight into these issues by providing a historical perspective of pricing for several communication services. Odlyzko focuses mainly on the business to consumer market and argues that customers are predominantly attracted to simple pricing models, in particular flat-rate pricing. This argument is supported by a number of studies which demonstrate that the end-users prefer flat-rate charges over usage based charges even if the flat-rate charges result in higher cost to the user. Possible explanation for this are risk avoidance and overestimation of usage by the users. Furthermore, flat-rate charges can make economic sense to service providers, not least because they ensure a steady stream of revenues and lower the per transaction cost by removing the need for usage-based charging and billing, but also because of the obvious preference of their customers. This observation is supported by the fact that historically for communication services there has been little relation between the cost of providing a service and the price charged to the end-user. Odlyzko concludes that pricing for the Internet should be based on simple flat-rate based schemes even at the cost of the efficiency of resource usage.

Odlyzko's argument seems to appear in stark contradiction to the aims of the congestion pricing proposals reviewed above. For instance, he writes: "people are also averse to varying prices, and are more willing to accept variations in quality than in price". From an end-user perspective congestion pricing, if applied explicitly, would appear even more complex than e.g., the original calling tariffs for mobile phone with time-of-day and recipient dependent call charges, because a user would also have to take into account the *current* state of congestion within the network.

However, most researchers do not primarily view network congestion prices as prices in economic terms to which end-users would be exposed. Instead, congestion pricing is used to model fair sharing of congestible network resources. Moreover, prototype implementations of congestion pricing for networks typically operate at the level of individual flows and are used as a replacement for TCP-like flow control algorithms, not as a charging scheme end-users will ever encounter.

There is, however, an economic argument for price differentiation if different service qualities are available: why would any user choose a lower quality if a higher quality is available at no extra cost? Whether ISPs will offer per-flow service differentiation to end-users is a separate question. Key et al. [KMBL99] discuss a number of options on how congestion charges could be used to determine charges in real money terms, starting with pricing for aggregate flows between ISPs, and allude to a scheme where guaranteed service could be offered by means of brokers, managing the risk of fluctuating prices on behalf of the users. However, irrespective of this argument, congestion pricing is still a valid proposition as a *flow control* mechanism capable of achieving fair sharing of network resources.

3.4 Summary

This chapter has provided an introduction in the use of economic models for managing resources in operating systems, networks, and distributed systems. I distinguished between market based approaches and congestion prices. In a market based approach prices are established by supply and demand, e.g., through an auction mechanism. While some researchers have proposed the use of market based approaches for distributed system, I have argued that they do not present the correct economic model for resources in an operating system context. The main arguments are that the supply side would need to be modelled by the operating system, that bidding would have to be synchronised, and that the required APIs would be complex

An alternative scheme, namely congestion pricing, has recently received much attention in the context of computer networks, in particular the Internet. This is based on standard economic practice which suggest that prices should be matched by cost [MV97]. That is, the fixed costs for providing network capacity and operational costs should be covered by fixed subscription fees. Then the direct usage costs are negligible. However, when the network becomes congested the social cost of congestion is incurred. The general idea of congestion pricing mechanisms is to make this social cost explicit to the end users in order to provide them with an incentive to adjust their behaviour. This chapter discussed a number of approaches, both theoretical and practical, showing how this concept can be applied to computer networks. The theoretical results show that congestion or shadow prices, based on the marginal cost of congestion, can achieve a socially optimal resource allocation. Under certain conditions, it can be demonstrated that such a system with independent users acting in their own self-interest converges to a fixed point at which the allocation is proportionally fair. Furthermore, revenues from congestion prices also provide the incentive to network operators to expand capacity. These theoretical results can also be observed in the more practical work, which is mainly concerned with implementation issues of how to convey congestion charges to the end-systems.

The same argument for congestion pricing in networks also applies for resources managed by an operating system. Essentially, if a resource, e.g. the CPU, is not congested its usage should be free. In the next chapter it will be demonstrated that operating system resources can become congested and that the consumption of a resource by one consumer can have an external negative impact on other consumers. I therefore argue that congestion prices are the correct economic model for operating system resources.

Chapter 4
Congestion pricing in operating systems

This chapter describes the three main components of the decentralised resource management architecture: the two OS mechanisms for pricing and charging and example application strategies. However, first the underlying theoretical model of congestion prices applied to operating systems is introduced. The model, described in section 4.1, is based both on general micro-economic theory and the prior work of its application to congestion avoidance in communication networks described in chapter 3. In section 4.2, how this model can be applied to resources managed by an operating system is discussed. A particular focus is placed on the management of CPU. It is argued that the CPU resource can become congested and that congestion externalities can be observed. It is discussed in detail how congestion can be detected, how shadow prices can be identified, and how congestion charges can be applied. Congestion detection and shadow prices for other resources are considered in less detail.

The second operating system mechanism is concerned with the management of credits used to pay for congestion charges. Allowing users to assign different amounts of credits to consumers[1] is necessary for service differentiation — if consumers of resources could spend as many credits as they like there would not be any incentive for them to adjust their behaviour. In section 4.3 a detailed description of these account mechanisms is provided.

While this research is mainly concerned with the mechanisms necessary to eliminate centralised resource management policies, this approach puts an additional onus on the application developer or user agents. In section 4.4 some of the issues concerned with resource consumer-adaption are discussed.

[1]In this chapter the generic term "consumer" is used to refer to the entities to which an operating system accounts resources to. These may be processes, threads, resource containers, or other operating system abstractions as discussed in section 2.3.

In this chapter the focus is on consumers of a single resource. In section 4.5 issues related to consumers of multiple resources are discussed.

The chapter is rounded off by comparing the decentralised resource management architecture to previous approaches to resource management (section 4.6) and a discussion on how this new form of resource management can be deployed in different contexts, such as personal workstations or server environments (section 4.7).

4.1 The model

Before discussing the use of congestion prices for decentralising resource management in operating systems, a formalised notion of the general model is provided. This model follows the one presented in, e.g., [KMT98] or [KM99], and is a simplified version of the model discussed in section 3.2.2.1. Consider a user i of a resource whose preference for consuming the resource at a rate of x_i is expressed through the utility function $u_i(x_i)$. Assume further that $u_i(x_i)$ is a concave and non-decreasing function of x_i — an assumption which characterises an elastic user [She95a]. For a user, it is natural to seek to maximise the utility $u_i(x_i)$ over $x_i \geq 0$. However, resources, such as CPU, disk and network bandwidth, memory, etc., are finite. Thus, if every user would seek to maximise their utility, resources would become overloaded and congested. Assume that the resource is incurring a (congestion) cost at rate $C(y)$ with $y = \sum x_i$ being the load of the resource.

In such an environment, a social planner, seeking to achieve a *socially optimal* resource allocation, would attempt to maximise the *net utility* of the systems:

$$\max \sum_i u_i(x_i) - C(y) \quad \text{with} \quad y = \sum_i x_i \tag{4.1}$$

For convex cost functions $C(y)$, at the optimum

$$u_i'(x_i) = p(y) \quad \text{with} \quad p(y) = C'(y) \tag{4.2}$$

is satisfied where $p(y)$ can be interpreted as the shadow price of the load. In other words, the optimum is achieved when the marginal benefit for user i equals the marginal cost to all users.

This solution is mathematically tractable, however, it relies on the explicit knowledge of users' utility functions. This is problematic as utility functions are typically not known explicitly. Instead, Kelly et al [KMT98] suggest decomposing the optimisation problem into an optimisation problem for each user and an optimisation problem, not involving the user's utility functions, for the resource.

Assume that a user i is charged at a rate t_i proportional to the amount of the resources x_i consumed. Then, it is natural for the user to maximise the net utility $U_i(x_i)$:

$$\max U_i(x_i) = u_i(x_i) - t_i x_i \tag{4.3}$$

Under the assumption of a monotonically increasing, concave, and continously differentiable utility function, the unique solution is:

$$u_i'(x_i) = t_i \tag{4.4}$$

If the system is aiming for socially optimal resource allocations according to equation 4.1 then it will set the charge t_i to the shadow price $p(y)$. Thus, if the price is right, the individual users' optimisations drive the system towards the social optimum.

Furthermore, it can be shown mathematically [KMT98, KMBL99] that if users adjust their rate of resource consumption x_i according to:

$$\Delta x_i(t) = \kappa_i \Big(x_i(t) u_i'(x_i(t)) - x_i(t) p(y(t)) \Big) \tag{4.5}$$

then, again under the assumptions of concavity of u_i and convexity of $C(y)$, each user will converge to a unique fixed point at the social optimum. The factor κ_i controls the rate of convergence.

Finally, if one assumes utility functions being of the general form $w_i log(x_i)$, satisfying the conditions of an elastic user, it can be demonstrated that, if all users adapt according to equation 4.5, the systems yields a weighted proportional fair allocation of resources with $x_i = w_i/p(y)$. The factor w_i can be interpreted as the willingness-to-pay, i.e., it denotes the rate at which a user is willing to pay for the resources consumed.

This is depicted in figure 4.1. The four panels show an elastic user with a utility function of the form $w_i log(x_i)$ with two different valuations (top panels with $w_i = 1$, bottom panels with $w_i = 0.5$) at two different prices (left panels $p = 2$, right panels $p = 4$). This graphical representation confirms the proportional fair allocation with independently acting users. The optimal allocation for a user willing to pay twice as much as another (top vs. bottom panels) is twice as large. Similarly, the proportionality of charges and resources consumed results in proportional optimal allocations for different prices (left vs. right panels).

As a socially optimal resource allocation is desirable in the context of an operating system, one of the main issues is to determine the right price, i.e., $p(y) = C'(y)$ where $C(y)$ is the external cost of the load y. In the following section a number of approaches for identifying congestion prices for a number of operating systems resources are discussed. It is worth pointing out

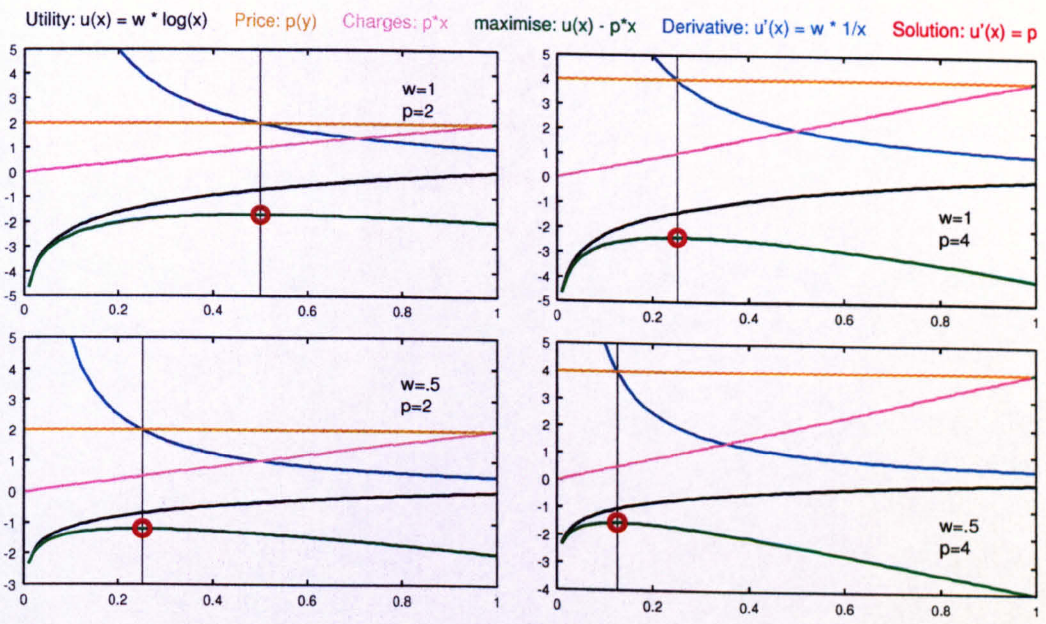


Figure 4.1: Elastic user optimisation — graphical solution for varying $p(y)$ and w_i

that the assumptions made for the mathematical proofs can be significantly relaxed in practise. In particular, proportional fairness can be achieved with other adaption strategies.

4.2 Identifying Shadow Prices

Identifying the right prices is vital to achieve a socially optimal allocation of resources. The prices should reflect the external cost of resource contention or congestion. In communication networks there is a clear indication of congestion, namely dropped packets or cells, and therefore a way of assessing the external cost of congestion. Essentially, all packets which have passed through a node prior to a dropped packet (since the node was last idle) have contributed to the dropping of the packet by congesting the node.

For operating system resources a similar measure needs to be identified for every resource managed. In this section, it is discussed how shadow prices can be identified in an OS context. The main focus is on CPU resources. Subsequently, other resources such as physical memory, disk bandwidth, network I/O, and battery energy are investigated.

In general, the approach is to require consumers of a resource to reveal their preferences for the resource. Then resource congestion can be identified if not all preferences can be satisfied. Note, that such an approach does not require admission control, as no strict guarantees are given that all preferences will be satisfied. Instead, one could think of these preferences as "soft reservations". Through the charging mechanism consumers are encouraged to truthfully reveal their preferences.

4.2.1 CPU congestion pricing

Following the general approach outlined above, assume a soft real-time environment with a real-time scheduler, e.g., EDF [LL73]. Such an environment has the advantage that it allows consumers of CPU resources to state their preferences for a resource in advance. For an EDF based scheduler, resource preferences can be defined by a tuple of period and slice. The scheduler then takes care of the short term scheduling decisions while the resource requests correspond to medium term resource allocations (e.g., on the scale of seconds). This approach can also be described as separating *urgency* from *importance* [NL97]. The scheduling algorithm determines which task to run next (the one with the earliest deadline) and the resource requests (combined with the decentralised resource management) determine the importance of a task, i.e., the share of the resource ($slice/period$) a task receives.

In a preemptive real-time environment an admission control system ensures that, for an EDF based scheduler, the sum of requests does not exceed 100%, which guarantees that all resource requests can be satisfied. In a soft real-time environment *occasional* deadline misses are acceptable and can be interpreted as an indication of resource congestion, i.e., missed deadlines are the external cost of CPU resource congestion. Thus, shadow prices can be based on it. Missed deadlines can be interpreted as the analog to dropped packets in communication networks.

If a task misses a deadline, all tasks which contributed to that miss should be charged proportionally to their responsibility. These are essentially all tasks which consumed CPU time between the last deadline met by that task and its missed deadline. This is illustrated in figure 4.2. The top of the figure shows the time line of a schedule for five tasks $T_{1..5}$ which, for simplicity, all have the same periods and deadlines labelled $D_{1..5}$. All tasks meet their first deadline. However, task T_5 misses deadlines D_2 and D_4 as some processes vary their share.

Ideally, the tasks responsible for the deadline miss should be charged. In the example schedule, these are all the tasks which executed between the end of the first period and the first missed deadline. The same applies for the second missed deadline and the ideal charging periods are marked in the upper time-line in the figure. During these periods, tasks should get charged proportionally to their resource consumption.

There are several practical problems with using the ideal scheme. Ideally, one would like to charge tasks for their consumption as they consume the resource. This, however, is impossible

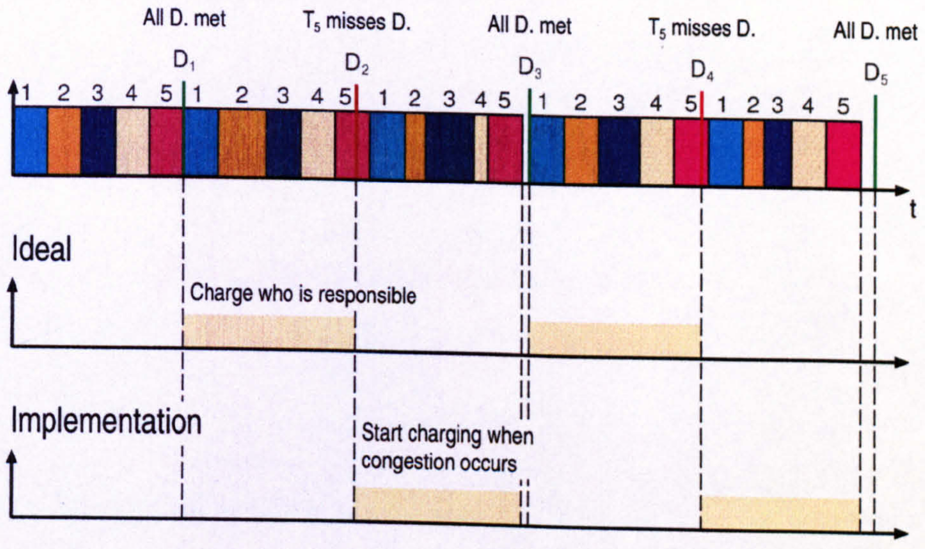


Figure 4.2: Different charging schemes

to implement as it would require knowledge of *future* events, i.e., knowing if a deadline will be missed. In the example, the system would need to know at the end of the first period that T_5 will miss its next deadline. This knowledge can only be derived if the exact resource requirements are known in advanced. Unfortunately, exact resource requirements are typically not known in real systems. If they were, managing resources would be much easier.

A second approach to implementing an ideal scheme could gather usage information as time passes and charge tasks after a deadline is missed based on their previous resource consumption. In this example, all tasks would be charged, at time D_2, an amount proportional to their resource consumption between D_1 and D_2. Such an approach may seem feasible if all tasks have exactly the same deadlines, as in the example. However, consider a more realistic example in which tasks have different deadlines and different periods. If any of the tasks miss a deadline the system needs to know how much CPU each of the tasks have consumed since this task last met a deadline in order to calculate the charges. This information would have to be maintained for each task individually. This is clearly impractical as it imposes a high overhead for maintaining the usage history. Furthermore, as charges are interpreted as feedback signals by tasks, it would seem desirable for charges to be applied continously as tasks consume resources.

As an implementation of the ideal model seems infeasible, an alternative charging scheme is required. A similar problem exists in communication networks, Gibbens and Kelly [GK99b] and Key et al [KMBL99] have suggested charging an appropriate number of packets which pass through a congested queue *after* the congestion occurred. This ensures that the correct amount of feedback is provided to resource consumers at the risk that some consumers may be charged although they have not been responsible for the resource congestion. However, it is argued that on a statistical basis the correct consumers are charged.

The same idea can be applied to CPU shadow prices. Instead of starting to charge at the start of the previously met deadline, one simply starts charging once a deadline has been missed and continues until the next deadline for that task is met. For the example schedule in figure 4.2 this is illustrated in the bottom time line. Tasks are charged for their resource usage during the period starting with T_5 missing deadline D_2 and continues until T_5 meets its next deadline D_3. From the example, it is clear that using this scheme, tasks T_2 and T_4 will get charged less than they would under the ideal scheme while T_3 would get charged more. However, this effect is amplified in this contrived example and in practise (see chapter 6 for examples) does not appear to be a problem.

In order to provide a continuous stream of charges proportional to the resource consumed, it is useful to define a *minimum time unit* which, under congestion, corresponds to a charge of one credit. A possible implementation of this approach may deploy the periodic timer interrupt most operating systems use to maintain the system software clock and/or to perform scheduling related functions such as updating usage statistics or priority re-computation. If the scheduler detects a missed deadline (i.e., an overload) the interrupt service routine would charge the currently running task one credit on every execution until the scheduler detects the next deadline is met. On standard PC style hardware this periodic timer can be set to periods as small as $122\mu s$ offering a high resolution feedback signal. However, general purpose operating systems use the periodic timer interrupt for scheduling related activities, such as recalculating priorities and schedulable timers. In these systems it is not advisable to increase the timer interrupt frequency to such high levels[2].

An alternative approach is to modify the code responsible for de-scheduling the current task. If the system is currently in congestion the de-scheduling code would simply charge the current task an amount proportional to the number of minimum time units consumed. Such an approach would require marginally more changes. However, an operating system typically performs some resource accounting in these routines which can easily be extended to also per-

[2]In Nemesis, however, the periodic timer interrupt is only used for maintaining wall clock time and, on alpha architecture, the interrupt service routine is implemented entirely in PAL code. Changing the timer period to $122\mu s$ and adding code for implementing the charging scheme does not impact the overall performance of the system.

form the charging. In practise, this approach provides charges at a fine enough granularity (see chapter 5).

4.2.1.1 Avoiding congestion

One problem with the pricing and charging model presented above is that in order for charges to be generated resource congestion must occur first. However, it is desirable to avoid congestion altogether as systems in overload tend to behave less predictably. Instead of basing charges on the external cost of congestion, charges could be based on the *probability of congestion*. This would generate some charges and therefore feedback signals to resource consumers prior to congestion and would provide them with an incentive to back-off their resource consumption, thereby potentially avoiding the system being overloaded.

This approach has its equivalent in communication networks. For example, Random Early Detection (RED) [FJ93] was proposed to avoid congestion in packet-switched networks by "randomly" dropping packets if the average queue size exceeded a threshold. In the context of congestion pricing in communication networks, RED in combination with ECN [Flo94] has been proposed to convey charges. Gibbens and Key [GKT00, GK01] proposed a similar scheme based on a Virtual Queue.

For CPU shadow prices a scheme akin to RED can be easily deployed[3]. In general, the probability of resource congestion increases with higher resource utilisation. Therefore, shadow prices can be based on the utilisation of the resource. More specifically, a charging probability is assigned for each minimum time unit consumed, with the probability dependent on the resource utilisation.

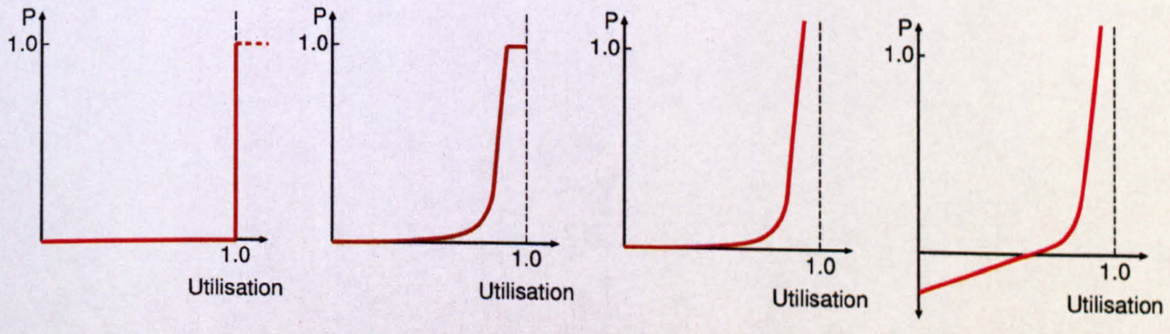


Figure 4.3: Charging probabilities

[3]Virtual queue base schemes are not applicable as it is difficult to identify queues in this context.

Figure 4.3 illustrates a number of examples. The left most panel illustrates the scheme introduced in the previous section. If the system utilisation is below 1.0 no congestion occurs and consumers are not charged for their resource consumption. If the resource is over-utilised the probability that a consumer gets charged for a unit of CPU time consumed becomes 1.0. The second panel shows an exponential function of the utilisation (i.e., $e^{a(y-b)}$) — the higher the CPU utilisation the more likely it is for a consumer to be charged for a unit of CPU time consumed. This scheme also accommodates a configurable offset from full utilisation to allow a target utilisation to be set. The third panel shows an alternative scheme which allows charges to be above one credit per minimum time unit. The final panel shows an example which provides "negative charges" when the resource is only lightly loaded and positive charges as in the previous example. The rationale behind this scheme is that the resource may also provide an indication to consumers that resources are underutilised, so that they can adjust their resource demand more quickly.

These example demonstrate that introducing the notion of a charging probability for minimum time units is flexible enough to accommodate a range of different pricing schemes. In general, setting the charging probability based on utilisation should reduce the risk of congestion and, in the event of congestion, the consumers responsible for the congestion are more likely to be charged.

4.2.1.2 Request-Usage charges

The discussion so far assumed elastic applications which were capable of consuming all requested resources. Therefore, the pricing schemes discussed were only concerned with resource utilisation and resource usage. A significant amount of real life applications have either bursty or constant resource requirements. These applications may not use the entire requested amount. For reasons of overall system stability[4] it is desirable that these applications make the "right" resource requests. For example, bursty applications could make requests based on their mean or peak resource demand depending on system load. Within a microeconomic based resource management system it is natural to provide this incentive by charging for resource requests as well as for resource usage.

A pragmatic approach is to split the charges for each consumer based on some systemwide ratio β as shown in equation 4.6, where x_i is the resource usage and r_i the resource request of consumer i and $p(\cdot)$ is a price function. The price function could be the same as the ones

[4]Some scheduling algorithms, such as EDF, do not perform well when in constant overload.

discussed in the previous section.

$$\text{charge}_i = \beta\left(x_i p(\sum x_i)\right) + (1-\beta)\left(r_i p(\sum r_i)\right) \tag{4.6}$$

Using the parameter β, the system is configurable to shift emphasis from purely usage based charging to purely request based charging. Due to the request component of a charge, bursty applications are charged even if they only consume very little resources. This should provide an incentive to them to request an affordable amount of the resource.

This Request-Usage scheme is similar to Kelly's Time-Volume scheme [Kel97a] (see section 3.2.2.2) which is based on the notion of effective bandwidth. The Request-Usage scheme is mathematically less sophisticated but appears to work well in practice (see chapter 6).

4.2.2 Other resources

The previous section was mainly concerned with identifying shadow prices for CPU resources. In this section, other resources are discussed. For every resource, the initial challenge is to identify congestion and the cost of congestion. For some resources there is an obvious indication of congestion (i.e., dropped packets in networks) for others this is less obvious. However, once the congestion has been identified the same or similar techniques to the once discussed in the previous section, for example for congestion avoidance, can be directly applied to different resources.

4.2.2.1 Memory

Most operating systems abstract over the limited physical memory available in a system with a virtual memory management subsystem typically using the local hard-disk as backing store — "inactive" pages of physical memory are stored or paged to disk in order to make the memory available for more active data. Paging activity can have a severe impact on the overall system performance, as disk I/O is typically several orders of magnitude slower than main memory accesses. In standard operating systems, some heuristic, such as Least Recently Used (LRU), is deployed to decide when, and which, pages are paged-out to disk. Thus, in the case of virtual memory management, resource congestion can be easily identified — if paging of other consumers' pages is required to serve a consumer' physical memory demands then there is resource congestion and all consumers using physical memory are responsible for this congestion, in proportion to their usage of physical memory[5].

[5]Note, that this description is deliberately oversimplified. In practise, operating systems make tradeoffs between using pages as buffer caches and user-level programs; they use shared libraries and layered VM systems which

Analogous to the discussion on CPU, congestion pricing consumers can be charged based on the number of physical pages of memory they use if the OS has to page-out. Alternatively this can be based on the probability of paging being required. Again, charges can be seen as feedback signals and consumers could free up used memory. Interestingly, some modern derivates of Unix, e.g., IBM's AIX, already implement a very rudimentary feedback mechanism using signals if a high-water mark is exceeded. However, full blown congestion pricing for physical memory would be more interesting in the context of self-paging systems which provide QoS within the virtual memory system [Han99a, Han99b]. In such a system, complex application-specific paging strategies [McD01] can be implemented and could benefit from the feedback provided by congestion charges.

4.2.2.2 Disk bandwidth

In most modern variants of the Unix operating system, disk I/O requests are queued in a per device queue and are serviced by the bottom-half of the block device driver (see, e.g., [Vah96, Chapter 16] or [MBKQ96, Chapter 6] for a detailed description). Completion of an I/O request is indicated through an interrupt. Typically, I/O requests are reordered using a specific disk scheduling algorithm to improve device throughput by minimising disk-head seek times (see, e.g., [SCO90, SV98] for a discussion of different scheduling algorithms).

Conceptually, as the block device I/O infrastructure uses queues to manage concurrent I/O requests, similar techniques to the ones developed for network routers and switches could be deployed (see section 3.2). Essentially, one could define a maximum acceptable queue length for outstanding I/O requests and identify congestion if this queue length is exceeded. Also, techniques like VQ or RED may be applicable to help prevent this congestion. As an alternative to monitoring queue lengths, the system could also monitor I/O request completion times. If the servicing time for an I/O request exceeds a device dependent threshold, congestion can be identified. This approach has the advantage that it monitors requests directly and is independent of any I/O request reordering. Akin to the exponential marking scheme, discussed above, a marking probability for I/O requests could be set as a function of either queue length or service times.

It is worth pointing out that the two approaches discussed in the previous paragraph are orthogonal to more advanced disk scheduling algorithms, such as Anticipatory scheduling [ID01], or filesystem prefetching, e.g., [SSS99].

complicate the accounting of physical pages to consumers. These and many other implementation specifics would need to be addressed by an actual implementation of congestion prices for memory.

Alternatively, some disk schedulers, e.g., the Nemesis disk scheduler [Bar97], deploy a deadline based scheduler. In fact, the Nemesis disk scheduler is based on Atropos, the CPU scheduler used in Nemesis. If an OS uses such a scheduler, similar techniques to those discussed in the previous section on identifying CPU congestion and shadow prices could be applied for disk access.

As with identifying congestion in the VM subsystem, the discussion in this section deliberately ignores many implementation details but rather focuses on conceptual approaches. For example, when using the queue length to identify congestion, without further studies it is unclear on which time scales, and with what degree of multiplexing of requests from different consumers, I/O request queues are operated. Furthermore, the service time for I/O requests varies significantly depending on the current head position — I/O requests may have different, unknown "sizes". This may significantly affect the applicability of results from network congestion control. Furthermore, unlike CPU or network packet scheduling the ordering of disk requests has a significant impact on the overall performance (throughput as well as latency). Therefore, the interactions between congestion pricing and disk scheduling and/or request tagging, as supported by modern disk drives, needs careful studying. Also, the discussion above focuses on individual disk devices. For systems with multiple disks, possibly configured as software RAID, this conceptual model would have to be expanded.

4.2.2.3 Network I/O

Operating systems treat outgoing network traffic in a similar way to disk I/O requests. Outgoing network packets are typically queued until they are transmitted on the wire. Concurrent requests are then serviced by a variety of different queueing disciplines (e.g., FCFS or some proportional fair scheduler like WF^2Q). Outgoing network traffic can also be scheduled using deadline based schedulers, e.g., in Nemesis a variant of Atropos is used [BBDS97]. So, the general techniques for congestion prices for disk I/O request also apply for outgoing network traffic.

Incoming network traffic is largely controlled by CPU capacity for protocol processing, thus mechanisms like LRP [DB96] (see section 2.3.2) in combination with congestion prices for CPU resources should suffice to control this resource.

4.2.2.4 Energy

Recently, there has been increased research activity in energy management in operating systems. The main motivation is the increased use of mobile devices such as laptop computers or PDAs, but environmental issues, such as overall power consumption and the noise generated by active cooling, also play a role.

In a recent paper [NM01] we investigated how congestion pricing can be applied to managing energy, focusing mainly on battery energy management in mobile devices. The first obstacle to applying the general model outlined in section 4.1 is that energy is consumed by all devices in a computer system. Thus, accounting energy consumption to individual consumers, as necessary to apply the correct charges, is more difficult — a problem exaggerated by the general lack of fine grained resource accounting in general purpose operating systems. In [NM01] we propose to exploit the mechanisms deployed in the Nemesis OS, for accounting for the usage of traditional resources (e.g., CPU, network, disks, and displays), to account energy consumptions to individual consumers. A similar approach is presented in [ZFE$^+$01], which extends the notion of resource containers [BDM99] to account for energy consumption.

Accounting for a consumer's energy consumption forms only the basis on which advanced energy management can be performed. In the research community, it is now widely accepted [Ell99, FS99, VLE00] that energy management should be performed at a higher level and may involve applications themselves. In [FS99] it is impressively demonstrated how a variety of applications, executing in different modes of operation, can have a significant impact on a system's energy consumption. Flinn and Satyanarayanan therefore argue that applications should form a collaborative relationship with the operating system.

Unlike the other resources discussed so far in this section, for energy it is initially not apparent how resource contention and the cost of resource contention can be identified. There are no obvious queues or deadlines to be considered. Instead, for battery energy, we adopt the *goal directed* approach, proposed in [FS99]: to a user the primary, most meaningful, energy-related performance metric is the lifetime expectancy of the current battery charge. In other words, the user should be able to specify how long the current battery charge should last, and the energy management system should strive to achieve this goal while maximising the utility provided to the user. Given the current battery capacity the system can calculate the maximum average discharge rate of the battery acceptable to achieve the user's goal. If the current discharge rate exceeds this average discharge rate, the system runs the risk of not being able to meet the user's expectation. This can be interpreted as energy contention and the consumers responsible for the excess energy consumption should be charged proportionally to their current energy consumption[6].

More specifically, in intervals Δt the reduction of battery capacity ΔE can be measured (courtesy of the Advanced Configuration and Power Interface (ACPI)[Int99]). If ΔE exceeds the maximum amount of energy E_{max} the system is allowed to use in that interval, we charge every consumer, i, proportional to the energy ΔE_i it consumed during that interval, thus the term $t_i x_i$ from the decomposed model equals $\Delta E_i / \Delta E$. As the battery capacity decreases non-linearly [ZFE+01], even under constant load, E_{max} needs to be recalculated periodically (e.g., the Odyssey prototype [FS99], implementing a similar mechanism, uses adaption intervals Δt of half a second). This approach for identifying shadow prices for energy consumption is similar to the simple "slotted time" model discussed in [GK99b] for network congestion prices.

As with the other resources discussed in this section, this proposed model for managing energy will require additional considerations for implementation. Most importantly, it is not known how accurately energy consumption can be accounted to individual resources and applications. This is clearly dependent on the accuracy provided by the specific ACPI implementation.

4.2.3 Comparison to congestion pricing in networks

The congestion pricing mechanism described in this section was inspired, and is based on, similar proposals in the area of communication networks. However, there are some notable differences, which make its application in an OS context easier.

First, in networks congestion charges are only delivered with RTT delays. Thus, the feedback they provide may be delivered too late to the end-systems which then react to a past transient congestion. In an operating system, congestion charges can be applied immediately and applications can potentially react more quickly.

Recently, there has been a discussion amongst researchers about using single bit vs. multiple bits for charges [Str00]. This argument is motivated by the desire to provide more fine-grained feedback than the single ECN bit available in the standard IP header. With a single bit, the end-system has to sample many packets to determine the current charging rate, thus exaggerating the delayed feedback caused by the RTT delay. For congestion prices for CPU this discussion does not apply. If a higher resolution of feedback signals is required then the minimum work unit

[6]For desktop computers, the goal could instead be for the system to not need active cooling. An external cost of contention can then be identified, if active cooling is required. For server systems, a system administrator could set a target energy consumption and resource contention can be identified if this target is exceeded.

can simply be made smaller, thus providing finer-grained feedback. For other resources, e.g., virtual memory or disk I/O, where the rates of resource consumption is discrete and relatively low, using an approach akin to multiple bits may prove useful.

Furthermore, in computer networks, packets may traverse multiple congested routers. If a single bit is used to indicate congestion charges the sum of marked packets an end-system observes may not accurately reflect the actual congestion price unless the charging/marking probability is very small. This problem does not apply when applying congestion charges in the context of operating systems as charges from different congested resources can be observed separately.

And finally, operating system interfaces are easier to change than network protocols. Changing network protocols, such as adding an additional field to a header in order to provide more detailed feedback, requires the agreement of all parties involved and may require routers and other network equipment to be updated. In an operating system only local changes are needed.

4.2.4 Summary

In this section different ways of identifying congestion and the related shadow prices have been discussed for a variety of resources that are typically managed by an operating system. In summary, there a number of different approaches to identify congestion.

Request based: Consumers of a resource reveal their resource preference through resource requests, which are treated as soft reservations. If the resource cannot satisfy all resource requests, resource congestion can be identified. This approach can be applied to all resources for which a given operating system provides a mechanism for resource reservations.

Goal directed: The goal directed approach is related to the reservation based approach. The user or a system administrator sets a target utilisation of a resource and if this target utilisation is exceeded resource congestion can be identified.

Queue based: For resources which use a queue to multiplex concurrent resource requests, or which can be modelled as queues, similar mechanisms to the ones discussed for congestion prices in communication networks can be applied. Congestion externality occurs if either the queue is full or a certain queue length threshold is exceeded.

Resource specific: Some resources, as for example, discussed for virtual memory, provide a natural form of congestion control, e.g., paging activity. In these cases, it is straightforward to identify shadow prices.

Which of these techniques is appropriate depends not only on the type of the resource but also on the way a given operating system supports this resource. For example, as discussed above, disk drivers could either use a request queue or a deadline based disk scheduler. For the latter, clearly a reservation based scheme is more natural to use, while for the former, a queue based approach is more appropriate.

In general, it is essential that the operating system is able to account resource consumption to consumers of resources as consumers should be charged in proportion to their resource consumption in the case of congestion.

4.3 Managing credits

Identifying congestion and the associated shadow prices is the most important operating system mechanism in the context of the described decentralised resource management system. The second important OS mechanism required is the management of credits and accounts, against which resource consumers are charged. For the overall stability of the system it is required that consumers cannot create credits themselves or accumulate arbitrary amounts of credits. In other words, if credits do not carry any value, there is no incentive for consumers to adjust their resource consumption and there would be no means to differentiate between services. Furthermore, if it was possible for consumers to either create or accumulate arbitrary amounts of credits, individual consumers could easily, either maliciously or through software bugs, price other consumers out of the market and monopolise a resource. Therefore, credits need to be policed and managed by the system. Essentially, limiting the amounts of credit available to consumers provides the incentive for them to adapt their resource demands[7].

The general idea is that each consumer has associated with it an account from which it gets charged. If an account is empty and a resource is congested, i.e., the consumer would get charged for its resource consumption, then the consumer is prevented from using that resource. If, however, a resource is not congested, i.e., no charges are applied, even a consumer with an empty account can use the resource.

A user allocates credits to each of his consumers of OS resources. This could either be done as a one off payment — the completion of a task is worth a certain amount of credits to the user — or the user specifies a rate at which credits should be allocated to a consumer (in $credits/s$). As charges are rate-based, i.e., are observed in $credits/s$, it is natural to base the credit allocation

[7]Note, that if credits are associated with real money, managing credits is not necessarily required as, at least theoretically, congestion charges would generate the revenue necessary for capacity expansion [MV95a].

on rates as well. Thus, a user specifies at which rate a given consumer is allowed to spent credits on congestion charges. This approach is similar to assigning priorities in traditional operating systems. However, it is potentially easier to understand as it has a direct real-life equivalent. If a user allocates twice as many credits to one consumer than to another, the user values the work performed by the first consumer as being twice as important as the work performed by the second. Naturally, if both consumers are non-blocking, the first consumer should receive twice as many resources as the second[8].

After the user has specified how many credits per second are allocated for each of the consumers, an operating system mechanism will place the appropriate amount of credits into a consumers account in fixed intervals. It is important to prevent consumers from accumulating an arbitrary amount of credits over time, as could happen if no resource congestion occurs for a while or the consumer is idle. Therefore, the absolute amount of credits in an account should be policed. A simple token bucket (e.g., [Par94, page 260ff]) as shown in figure 4.4 fulfils this requirement. Tokens, or credits, are placed in a fixed sized bucket, or account, of size β at a rate of $w\ credits/s$. If the bucket is full, newly generated credits are discarded. Credits are removed from the bucket when the consumer consumes resources and resource congestion is identified, i.e., if the shadow price is non-zero. A consumer is allowed to consume resources if (1) it has credits available to pay for the congestion charges, or (2) if a resource is not congested.

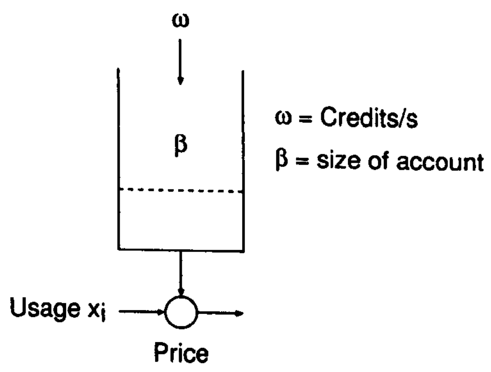


Figure 4.4: A token bucket scheme for managing credits

The token bucket scheme for credits permits credit spending rates of up to $\beta + \tau \times w$ in an interval τ with a long term average of w. To prevent excessive accumulation of credits which would allow consumers to sustain longer periods of heavy congestion without adjusting

[8]The experimental results, presented in chapter 6, confirm this intuition and, under certain conditions, this can also be proven theoretically[KMT98, KMBL99].

their resource demands the size β of the bucket should be sized close to the per second credit allocation rate w. In the implementations (see chapter 5) β is typically set to $2 \times w$, allowing for consumers with small credit allocations some slack should there be a sudden rapid increase in congestion charges. Note, that in general, limiting the bucket size to close to the per second rate of credit allocations is not a problem for well behaved applications. The upper limit on the number of credits which a consumer can accumulate merely provides a means of dealing with malicious, uncooperative, or buggy consumers.

In practise, this simple token bucket scheme for managing credits has been proven to be sufficient. However, should further policing of peak credit spending rates be required, a Token Bucket with Leaky Bucket rate control [Par94, pp. 262-263] could be implemented to provide a policed upper bound on the burst credit spending rate. In this context, it is also worth pointing out that a simple Leaky Bucket scheme for policing credit spending rates is not sufficient as neither resource demands nor congestion charges are likely to be isochronous, i.e., both are expected to be bursty.

On the more practical side, the operating system should provide an interface which allows consumers to query their current account, the rate at which the user allocates credits to them, and, most importantly, a mechanism which allows them to easily establish the current charging rate. This would be difficult by simply observing the current level of the token bucket, as the system is periodically placing new credits into that bucket. This would require consumers to sample the account frequently, resulting in an unnecessary overhead. In the implementations, accounts are only maintained internally and consumers can simply query the total amount of charges incurred or the charges incurred since they last queried the charges. Thus, by keeping timestamps on the samples, consumers can easily establish the current charging rate at a granularity best suited to them.

It is worth pointing out that the token bucket for credits only provides a minimum mechanism for managing credits. Higher level mechanisms, such as the "top half" of lottery scheduling, i.e., tickets and currencies (section 2.1.4.3 and [WW94]) and windowed ticket boost [PMG99], or user agents varying $w_i(t)$ on behalf of the user[9] can be implemented to compliment this basic mechanism. In a multi-user environment a mechanism is also required to allow an administrative entity, e.g., system administrator, to allot credit rates to individual users of a system.

[9]As a simple example, in a workstation environment the windowing system's input focus may be used to increase the rate of credit allocation for the current "foreground" application [SM99b].

4.4 Consumer Strategies

In this chapter, the operating system mechanisms for pricing and managing accounts have been described. In this section, different adaption strategies for consumers are discussed which can be used to react to feedback signals provided by the charges.

Following the decomposed model introduced in section 4.1, consumers of resources observe charges of the form $t_i x_i$, i.e., proportional to their resource consumption x_i. A rational consumer aims to maximise their utility $U_i(x_i)$ minus these charges:

$$\max U_i(x_i) = u_i(x_i) - t_i x_i \tag{4.7}$$

A general assumption is that a consumer's utility is elastic with a diminishing marginal increase of utility as the resource allocation increases. In other words utility functions, at least qualitatively, are assumed to be monotonically increasing and concave. With this in mind, a consumer's optimisation problem has the unique solution $u_i'(x_i) = t_i$.

In a dynamic system, overall resource consumption and shadow prices, and therefore, the charges, change constantly. Consumers will have to adjust x_i dynamically to deal with these changes. Consumers of a resource thus observe the charging rate $t_i x_i$ and adjust their resource consumption or reservation by Δx_i according to some application specific strategy. Applications may also change their mode of operation in response to the feedback signals. e.g., reduce or increase video decoding quality or display frame rate.

Given the account model outlined in the previous section, it is natural for elastic consumers to attempt to match the rate of charges to the rate at which they are allotted credits by the user. Essentially, each elastic consumer will run a rate adaption algorithm so that its willingness to pay w_i, as assigned to by the user, equals the rate $t_i x_i$ at which charges are observed.

A wide range of different rate adaption strategies are conceivable. These range from a simplified TCP-like algorithm implementing an additive increase/exponential decrease strategy, to more sophisticated, control theory based algorithms.

One simple sample strategy can be derived from the general adaption strategy described in equation 4.5. Assuming a general utility function of the form $w_i log(x_i)$ this equation can be transformed into Gibbens and Kelly's WTP algorithm [GK99b, Key01]:

$$\Delta x_i(t) = \kappa_i(w_i(t) - x_i(t) \times p(y(t))) \tag{4.8}$$

With this algorithm, a consumer observes charges proportional to the load they impose on a resource and increase or decrease their consumption according to whether the charge is higher or lower than the amount they are willing or allowed to pay. The parameter κ_i influences the rate of convergence.

An alternative approach could use a classic controller from control theory, namely a PID controller, which has also been used in other feedback based resource management systems for operating systems [SGG+99, LSTS99]. A PID controller adjusts the resource consumption rate based on the combination of a proportional, integral, and derivative component of an observed error:

$$\Delta x_i(t) = C_{iP} e_i(t) + C_{iI} \int e_i(t) + C_{iD} \frac{de_i(t)}{dt} \tag{4.9}$$

where C_{iP}, C_{iI}, C_{iD} are constants and $e_i(t)$ represents the error between a configurable willingness to pay rate and the rate at which congestion charges are incurred, i.e., $e_i(t) = w_i(t) - p(y(t))$.

In general, consumers are free to choose which adaption strategy to use. It is conceivable that the operating system provides a variety of default adaption strategies in the form of user-level shared libraries for the application developer or even the user to choose from.

The system can also accommodate non-adaptive legacy applications. Consumers are guaranteed a share of the resource proportional to $w_i / \sum w_i$ since $\sum w_i$ is policed through the accounts mechanism and the maximum charging rate is fixed by the mechanism for identifying shadow prices. Consumers which do not perform any Δx_i adaption are able to sustain charges up w_i. Therefore the simplest way to support non-adaptive legacy applications is to make resource requests for them which are proportional to their weight and the maximum charging rate. The FreeBSD prototype, described in detail in section 5.2.4, uses this mechanism to support non-adaptive legacy applications.

More complex strategies other than the simple charging rate control algorithms discussed in this section, may be deployed. For example, a user may specify that a consumer should perform a certain task before a given deadline at a minimum or fixed cost. This freedom enables application developers and users to choose from a wide range of different strategies borrowing from game theory, control theory, economics and mathematics. Some sample strategies for different types of application are discussed in [GK99a, KM99] in the context of communication networks. Furthermore, application developers are encouraged to use the feedback provided to pro-actively adapt the behaviour of their applications. A detailed discussion of these issues is beyond the scope of this dissertation, which is primarily focused on the enabling mechanisms.

4.4.1 Non-convex utility curves

The theoretic model of congestion prices, more precisely, the analysis of optimality of resource allocations, assumes strictly continuous, convex utility curves. While it can be expected that some applications fulfil this criterion, many real applications might not. Often, realistic applications require a minimum amount of a resource before being of use to a user. Furthermore, real utility curves are likely to be non-continuous as real applications are likely to only offer distinct modes of operation with different resource demands.

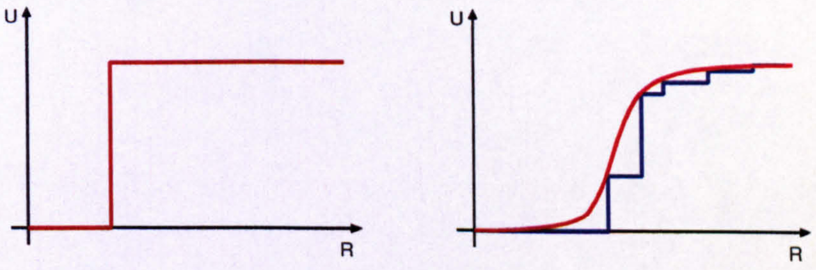


Figure 4.5: Examples of non-convex utility curves

For realistic applications one can distinguish two common cases for non-convex utility curves. These are illustrated in figure 4.5. The left curves represents an application which requires a fixed amount of a resource in order to work. However, once it receives this amount of the resource any further allocations of resource do not allow it to offer any further utility to the user. The system can accommodate these types of application in a similar way as described above for legacy applications. Since an application is guaranteed a share of a resource proportional to its credit allocation and the maximum charging rate is known, a user could assign an appropriate amount of credits to this type of application and the application does not need to perform any Δx_i adaption. Either the user considers them valuable enough and provides them with sufficient funding to sustain their service rate even under resource congestion, or they simply do not execute.

A sample utility curve for a different type of application is depicted on the right hand side of figure 4.5. This type of application also requires a minimum amount of a resource to perform any useful work and then has a sharp increase in the utility provided which then tails off in a convex form with more resources added. Typical applications with this type of utility curve are multimedia applications which may have a quite high initial resource demand for decoding video frames and then are able to offer increased quality video playback with a higher resource allocation. Again, it is up to the user to provide sufficient funding for these types of applications to allow them to operate in the region of the utility curve where they can offer any utility to the user. However, once the application is able to operate in the convex part of the utility curve it can utilise the adaption techniques described above.

The right hand side utility curve also illustrates the non-continuous nature of real applications (in blue), contrasting them with the idealised form (in red). An application may offer distinct modes of operation offering different amounts of utility to end-users. A switch between these modes may be distracting to an end-user (e.g., a change in video playback resolution) so that application programmers would typically implement a hysteresis for mode changes. Again, if an application does not receive enough funding to continuously operate in a mode desired by the user, the user could simply increase the credit allocation for that application.

In general, it can be assumed that applications with non-convex utility curves fall into the two categories described. Furthermore, typically, applications with a relatively high resource demand belong to the second category, while applications with a low overall resource demand fall into the first. The reason for this is clear: even if an application with a small resource demand is adaptive and may consume more or less resource based on its mode of operation, this adjustment of resource demand is still relatively insignificant when compared to the total amount of resource available. In other words, applications with a small resource demand might as well request resources at a peak rate and be treated as fixed rate applications.

Applications with non-convex utility functions do place a certain onus on the end-user, namely the end-user has to provide sufficient funding to these types of applications. However, this task can be supported by appropriate user interfaces which allow an end-user to both monitor the system and to adjust credit allocations. Two prototypes of such interfaces are presented in section 5.2.6.

4.4.2 Dealing with ill-behaved consumers

If consumers get involved with resource management and can choose any application strategy then ill-behaved or even malicious consumers may try to sabotage or play the system in order to gain an unfair advantage over other consumers or even deliberately cause harm to other consumers. The credit policing account mechanisms described in section 4.3 may prevent the worst ill-effects by simply reducing the number of credits available to consumers.

Even with such protection in place it is conceivable for consumers to "play" the system. While this, to an extent, is encouraged — may the best strategy win — there may be "strategies" which could cause unnecessary fluctuation or oscillation in service rates received by other consumers. For example, a malicious consumer may deliberately attempt to drive a resource into heavy congestion and, as soon as congestion charges are applied, retract to a minimum resource consumption or be inactive for a period of time. Also, ill-configured adaption strategies may cause similar effects. For example, the PID strategy introduced above requires up to four parameters to be specified. Choosing an unfortunate combination for these factors may result in erratic behaviour.

In the evaluation chapter the interaction of these and other ill-behaved consumers and "good citizens" is investigated in detail. However, there are a few mechanisms which may provide additional protection. For example, using the congestion charges based on congestion probability (section 4.2.1.1) increases the likelihood of malicious consumers, who are trying to drive the resource into congestion, being charged for their behaviour. Another protection mechanism may be to prevent a consumer from making rapid and significant changes to their resource requests, thus encouraging more measured resource adaption.

It is worth pointing out that apart from these technical measures, social measures may be more effective in dealing with ill-behaved consumers. In a mainly single-user workstation environment, a user has little incentive to use applications or adaption strategies which are ill-behaved and have a negative impact on the user's other consumers. In a shared multi-user environment, users with ill-behaving consumers may be subject to peer-pressure from other users or administrators can ultimately ban users with ill-behaving consumers from using the system.

4.5 Multiple resources

The discussion so far has focused on the management of individual resources. However, applications typically make use of multiple resources. From a user's point of view, it is desirable to have just one currency for all resources used by a consumer, i.e., the user allocates a credit rate for its consumers and consumers then have to decide on which resources to spent these limited funds.

However, consider a user allocating a larger amount of credits to a consumer for CPU, disk I/O, and memory. If memory momentarily is not congested, and the consumer performs less disk I/O than anticipated, the consumer could potentially use the "surplus" credits to monopolise the CPU resource. Thus, from a systems view, it is desirable to use different currencies for different resources. This approach is also favoured by Sullivan and Seltzer [SS00, SHS99] which present a system using the lottery scheduling resource management framework (see section 2.1.4.3) to manage multiple resources. They argue that different currencies are necessary in order to provide insulation between different resource and consumers.

Unfortunately, in such a system an undesirable burden is placed on the user who has to allocate credits in different currencies to each of the consumers rather than simply stating how important a particular consumer is. One possible solution is to assume that for a given consumer all resources are equally important. The user would assign a credit rate w_i to a consumer and the system would assume equal allocations for all n resources, i.e., the credit allocation rate for each resource would be policed to $w_i/n\ credits/s$ for each resource. In such a system it is conceivable to allow consumers to exchange or trade credits for different resource with each other. For this purpose, Sullivan and Seltzer [SS00] proposed Ticket Exchanges, so that, for example a CPU intensive consumer could exchange its disk I/O tickets (or credits) for more CPU tickets with an I/O intensive application. An alternative would be to allow consumers to change credits of different currencies, but tax each exchange as a deterrent against frequent changes.

A single currency for all resources might still be feasible in an economic framework if consumers are given an economic incentive to spent their credits appropriately. In the single resource pricing scheme discussed in section 4.2.1.1 minimum work units were charged *one* credit with a probability determined by the level of resource congestion. Thus each resource had a maximum charging rate at probability 1.0 depending on the size of the minimum work unit. By deploying a scheme which inflates prices beyond this maximum charging rate of one credit per minimum work unit consumers *might* be given an incentive to spent credits on different resources (if they can) or, are at least discourage from spending most of their credits on one resource, as it becomes exceedingly undesirable to spent credits on just one resource.

Without further investigation and practical experiences with congestion pricing for multiple resources it is unclear which of these alternatives to use. The single currency model is certainly more appealing from a user's point of view but might introduce instability and potential exploits for malicious consumers to the system. These issues will need to be addressed with future work.

4.6 Related work revisited

In chapter 2 existing approaches to resource management in operating systems have been reviewed and several problems with them have been identified. Specifically, these are issues related to (1) QoS mapping, (2) admission control, and (3) complex APIs. Now, with the decentralised resource management architecture introduced in detail, these issues are revisited and it is shown how the new architecture addresses them.

The decentralised resource management architecture separates mechanisms from policies [LCC$^+$75] as is generally considered a good practice which all higher level resource management architectures adhere to. Low level mechanisms merely multiplex resources, and implement pricing, charging, and accounting. However, unlike previous approaches to resource management, higher level policies are not centralised within a single entity, such as a QoS manager: decisions about how much of a resource a consumer requires are made by the consumers — policies are decentralised.

Centralised QoS managers typically export complex APIs to allow a range of different types of applications to specify their QoS demand in a application specific manner. In contrast, the API presented by the decentralised architecture is very simple: resource consumers can request resources via a single interface per resource and can enquire about their current charging rates — there is no need for more information being exchanged between applications and resources. The per resource interfaces should be very simple, merely reflecting the way the resource is multiplexed. Thus, whether a particular type of application is supported depends on the capability of the multiplexing mechanisms rather than on support for it by a higher level API. For most resources, especially the CPU, the capabilities of multiplexing mechanisms and their interfaces are well understood, as indicated by the review of scheduling algorithms in section 2.1. Likewise, the interface for managing credits, both for the user and the consumers is simple: users can specify credit allocations in very much the same way they would assign priorities to applications and consumers can, through a simple call, find out the number of credits they have been charged. This simplicity is in stark contrast to the interface definitions of, e.g., QoS-A (section 2.2.1), OMEGA (section 2.2.2) or Q-RAM (section 2.2.3) which define interfaces at a significantly higher level, i.e., at the application level.

In most existing QoS architectures QoS managers typically attempt to map these high level QoS definitions to lower level resource allocations. This mapping is complex, requires dynamic adjustments and, most importantly, knowledge about the performance of the application with respect to the stated high-level QoS requirements. Most of the reviewed QoS architecture (OMEGA, QualMan, Q-RAM) accomplish this task by assuming a static mapping from application level QoS requirements to resource allocations (determined either off-line or during a calibration phase at application startup). However, it has been argued that this approach is infeasible due to the high variability of resource demands of applications and its dependency on the application's input (e.g., differently encoded video streams). The decentralised architecture addresses this problem by not performing this mapping outside the application. Instead, an application *may* perform the mapping from an application level QoS specification to low-level resource requests itself; potentially leveraging full application specific knowledge. This mapping does not need to be explicit, i.e., an application may simply try to maintain a certain QoS level; it can be performed continuously, adjusting to changing resource demand and availability; and is assisted and encouraged by the decentralised architecture through the feedback provided by the pricing information.

This type of flexibility would be difficult to achieve in a centralised architecture as more and more information would have to be passed to a centralised resource manager, thus making its API more complex. For example, a video decoding application would require an interface to inform the manager about its current frame-rate and decoding quality level. A much simpler approach would be that proposed by Steere et al. with their feedback-driven scheduler (discussed in section 2.1.6). In their system applications only have to provide a simple progress metric to a centralised controller. While this solves the problem of complex APIs and mapping, it is unclear how the performance of complex applications can be reduced to a single progress metric. The decentralised architecture does not have such a constraint and instead enables applications to choose progress metrics specific to the applications and to adjust their resource demand accordingly.

The third problem with existing resource management approaches has been identified as admission control and resource renegotiation, in particular under changing resource availability. QoS managers have to implement an admission control system to provide resource guarantees to applications. Simplistic admission policies, e.g., first-come-first-served, are unfair and suboptimal as later arriving, more important tasks may be denied access. Furthermore, under conditions of changing resource availability QoS managers typically propose to use one of two different schemes: either to change resource allocations based on detailed application knowledge or to engage in resource renegotiation with the consumers. It has been argued that the former exhibits the same problems as QoS specification mapping, in that the resource manager would need to know how many resources a consumer would require in different modes of operations. The latter approach is more promising. However, it requires consumers to be provided with an incentive to reduce their resource demand. The decentralised architecture addresses this issue by charging consumers for their resource consumption. By basing the charges on the external cost of congestion, consumers are made aware of the impact they are having on the overall system performance and by limiting the availability of credits used for paying for these charges consumers are given an incentive to adjust their resource demand with changing resource availability. This feedback in the form of charges is provided to resource consumers continuously, allowing them, to some degree, to adjust (or "renegotiate") their resource demand at timescales suitable to them, rather then at times chosen by a resource manager. Furthermore, by controlling the availability of credits and through an appropriate pricing mechanism the need for an admission control policy at the resource is alleviated.

From the reviewed QoS architectures, AQUA is conceptionally similar to the decentralised approach. In both systems, resources provide feedback to the applications which adjust their resource demand based on the feedback. However, AQUA does not seem to provide any mechanisms, other than prescribing a QoS manager, to force applications to adapt their resource demand. Furthermore, by basing the decentralised approach on a sound micro-economic theory (see chapter 3), theoretical results from other areas of resource management can be leveraged. These results state that under certain assumptions, a decentralised approach can yield optimal allocations as well, maximising the overall system utility. Techniques proposed for QoS architectures are still applicable in the decentralised resource management architecture. For example, techniques for mapping QoS requirements may be deployed by the resource consumers directly in order to make more accurate resource demands. Techniques for mapping user and application utility to an application's mode of operation, proposed in the context of Q-RAM, may be valuable for applications to react to feedback signals. Classifications of applications, as used by some systems, may be used to develop default adaption strategies for applications. However, an important difference is that the decentralised approach *does not* force the use of any of these techniques.

4.7 Use in different contexts

The description of the decentralised resource management architecture in this chapter has so far focused on the low-level operating system mechanisms. However, there are a number of additional considerations depending on the context in which such a system would be deployed. Three different contexts are considered: a user's personal workstation, a server internal to an organisation, and an external server. The latter could be part of a server farm at an application service provider, a node in a compute cluster, or even part of a public computing infrastructure (such as the proposed by the XenoServer project [RPM+99, HHKP03] or built by the PlanetLab project[10]). The distinguishing feature between internal and external servers is that external servers have potentially competing users from different administrative domains.

The approach to congestion pricing in operating systems as described in this chapter can be applied directly in the context of personal workstations. Typical target applications in this

[10]`http://www.planet-lab.net`

context are adaptive multimedia applications or applications which perform compute-intensive yet non-time critical background tasks, e.g., document indexers. A user can assign different credit allocations to different applications in a similar way as he would assign priorities. Applications can directly enquire their credit allocation and charging rates using simple system calls. If funds for an application are insufficient a user could directly increase funding for this application. Furthermore, the task of assigning and changing credit allocations to applications could be facilitated through user interfaces or simple user agent softwares. In section 5.2.6 some initial prototypes for this purpose are described.

Congestion pricing "could" also be applied to manage resources in local servers — servers which are deployed within an administrative domain such as workgroups or small companies to provide services to users within that group. However, in general congestion pricing as a resource management mechanism is only useful for resources that (can) become congested and there is a strong argument that congestion within local servers should be avoided — resources for such services should be over-provisioned or at least sized to provide an adequate level of service.

Applying congestion pricing in the context of an external server is significantly more interesting and challenging. As described above external servers are servers providing services on behalf of non-cooperating remote users. A typical application scenario for external servers are hosting centres using consolidated servers, i.e., hosting multiple competing applications on a single server or small group of servers, maybe deploying virtual machine techniques such as VMWare ESX. Even more challenges are posed by emerging public computing infrastructures allowing arbitrary users to run potentially short-lived services on a computer system "somewhere" in the network, e.g., as proposed by the XenoServer and PlanetLab projects.

Such environments suggest that real money will be used to provide services to users or customers. Then the question becomes whether congestion pricing should be used to charge customers real money and, if so, how should users be made aware of the fluctuating prices inherent to congestion prices. While the answer to the first question is difficult to provide — it would make sense from an economic point of view, however, issues such as customer preferences and general market conditions may be prevailing — the second question is directed more at whether it would be feasible to implement such as pricing scheme.

A simple way of implementing congestion pricing in external servers involving real money would be to enable customers to specify a fixed maximum amount of money they are willing to pay per unit time for the service. This knowledge would only be made available to their application which would take this information into account when changing resource requests. A more elaborate scheme could involve customers making use of a partial autonomous agent at the server making limited decision about spending for them. Comparable schemes have been proposed in the context of MUSE [CAT+01], a system to manage energy in hosting centres, and the modified version of REXEC used for managing jobs in the cluster computers [CC00]. In both systems customers specify a policy on how much money should be spent on their behalf. This policy could be as simple as stating a maximum spending rate. It is anticipated that each external server also exports an interface via which customers can query pricing information, e.g., current price or the price history. This interface could be simply web-based or use, e.g., a multicast group to disseminate pricing information to all customers. It is important, however, that using an agent of some sort frees customer from having to react to congestion pricing information at the short time scales at which they change — a customer can occasionally check the performance of his service (via an application specific interface) and then take pricing information into account in order to decide whether to adjust the policy implemented by the agent.

The external server example obviously requires further infrastructure for it to be feasible. For example, mechanisms for authentication and payment are required. One possible approach is presented by the XenoServer project which models part of these mechanisms similar to the way credit card companies operate [HHKP03]. A detailed discussion of these issues is beyond the scope of this dissertation and the remainder of this dissertation focuses on the application of congestion pricing in personal workstation environments.

4.8 Summary

In this chapter the application of congestion pricing mechanisms to managing resources in operating systems has been discussed. Particular emphasis was put on the mechanisms which an operating system has to provide in order to decentralise resource management. These mechanisms are the identification of congestion and congestion prices, and an appropriate mechanism for managing credit accounts. An accurate accounting mechanism is assumed to be provided by the operating system.

I have identified 4 different ways of identifying congestion — request based, goal directed, queue based, and resource dependent — and have demonstrated how these can be applied to identify congestion in a variety of resources managed by an operating system. In this discussion, particular emphasis was placed on the management of CPU resources. For CPU resource management a soft real-time approach was proposed as it allows consumers of a resource to express their preferences in the form of soft reservations and real-time schedulers can be used to provide resource allocations at a fine granularity while medium to long term allocations are provided by the decentralised approach. In the context of CPU resource management, the importance of actually avoiding congestion entirely by basing congestion charges on the probability of congestion rather than on congestion itself has been discussed. In general, it has been demonstrated that operating system resources are indeed congestable and exhibit congestion externalities.

The second important mechanism has been identified as that of managing credit accounts, used by consumers, to pay congestion related charges. It has been argued that a rate allocation scheme for providing credits to consumers offers a useful abstraction: the user decides at which rate credits are allocated to consumers. A simple token bucket scheme is deployed to prevent consumers accumulating arbitrary amounts of credits. This is mainly a safeguard against ill-behaving consumers and should not impact well-behaved consumers.

Finally, this chapter provided a discussion on possible adaption strategies for consumers. Consumers are encouraged, through the congestion charges and the limited availability of credits, to adjust their resource demands. A natural behaviour is to adjust the resource demand so that the charging rate matches the rate at which the user has allocated credits. Two different algorithms (WTP and PID) for achieving this goal have been described. However, it is worth stressing that consumers are free to choose any strategy they like. Measures beyond the credit controlling mechanism, to protect the system from ill-behaved consumers, have been discussed.

Congestion pricing mechanisms for CPU resource management have been implemented in two different environments. Background to these implementations is provided in the next chapter and chapter 6 provides an evaluation of these implementations.

Chapter 5
Implementations

To evaluate the model of congestion pricing for operating system resources that has been outlined in the previous chapter, I have implemented two prototypes. In this chapter these implementations are described in detail — an evaluation of the congestion pricing mechanisms is given in the next chapter.

For initial experiments and comparative studies I have implemented a generic simulation environment for scheduling algorithms. The simulator offers an environment in which different CPU scheduling algorithms and resource management mechanisms and policies can be implemented quickly and evaluated under similar workloads. Thus, it allows for the detailed qualitative and quantitative comparisons of different scheduling and resource management approaches. The simulator is described in the next section.

Although simulations are a very practical tool to quickly explore different approaches, they often model only certain aspects of real systems and, therefore, the results should be carefully interpreted. More specifically, the simulator only provides a highly idealised simulation environment for CPU scheduling. To validate the results gathered from the simulator, I have also implemented a prototype under FreeBSD. The FreeBSD implementation has the advantage of a mature, industrial strength workstation and server OS. The FreeBSD prototype is described in 5.2.

5.1 The simulator

The simulator is a discrete-event simulation environment for scheduling algorithms and workloads. It provides abstractions for processes, schedulers, and inter-process communication and synchronisation. These abstractions are derived from their equivalents in Nemesis, as it already provides a rich and flexible notion for these fundamental abstractions. In practise, the abstractions are shown to be powerful enough to implement a variety of scheduling algorithms, including hierarchical schedulers (see section 5.1.1 and 5.1.4). The task model is flexible enough to model both artificial and realistic workloads (see section 5.1.5).

The simulator is implemented[1] in Python [vRD00], taking advantage of both its object-oriented and functional programming features. The basic abstractions are implemented as base classes and can be easily extended or partially replaced to suit a variety of different scheduling algorithms. For example, the basic IPC mechanism can be extended with scheduler specific techniques to avoid priority inversion. Python's interpreted runtime environment also facilitates rapid development and prototyping.

In the remainder of this section, the simulator is described in more detail. First, the general abstractions are introduced, followed by an description of how different scheduling algorithms can be added and how a variety of workloads are modelled. This section is rounded off with a discussion of how the decentralised resource management mechanisms are implemented in this environment.

5.1.1 The task model

The simulator offers a general task model mirroring the conceptual task model used in Nemesis (see section 2.3.1). Tasks are composed from SDOMs and ADOMs (Nemesis' PDOMs are not modelled as the simulation environment does not need to provide memory protection). The simulator provides an n-to-m mapping from SDOMs and ADOMs allowing for very flexible combinations of resource allocations and executions. However, as in Nemesis, the default is a one-to-one mapping, forming the closest equivalent to a "process" in more traditional operating systems.

An SDOM can be in one of 4 states. Different schedulers may only make use of a subset of these states. If an SDOM has any associated ADOMs which are willing to receive CPU time it is either in the `run`, `wait`, or `unblocked` state. The `run` state indicates that the SDOM is eligible to receive the CPU, while the `wait` indicates that the SDOM has already received its quantum of CPU and is currently waiting to become eligible for the next quantum. The `unblocked` state may be used by schedulers to indicate that the SDOM only became runnable recently, and a given scheduler may pay it special attention in order to reduce unblocking latencies. The `block`

[1]The initial implementation was derived from a small test harness for the Nemesis Atropos scheduler (originally written by Paul Barham at the Computer Laboratory, University of Cambridge, U.K.). However, it was subsequently completely redesigned and significantly extended for use in this dissertation.

state indicates that the SDOM currently does not have any associated active ADOMs and is therefore blocked awaiting an external event.

In the simulator, SDOMs and ADOMs are implemented as classes. The base `SDOM` class implements the simple case of a one-to-one mapping between SDOMs and ADOMs. When the scheduler selects an SDOM to be run, its associated ADOM is activated. To implement hierarchical scheduling schemes, the `HSDOM` class provides an interface via which the scheduler can inform the SDOM of changes in its ADOMs' states (i.e., if they block or become unblocked). If an SDOM is awarded the CPU by the scheduler, it can nominate one of its active ADOMs to be run. However, to keep the main scheduler implementations clean and simple, the base `SDOM` class is used as the default. The base `SDOM` class can be extended to accommodate scheduler specific attributes and methods, e.g., to contain priorities, resource reservations, and deadlines.

In contrast to SDOMs, ADOMs are generic and independent of any specific scheduling algorithm and implement various types of workloads. As with SDOMs the `ADOM` base class implements the basic functionality of all ADOMs. The main method of the `ADOM` class is `activate()` which is called whenever the ADOM gets allocated the CPU by an SDOM. The simplest implementation consumes the entire allocation (see section 5.1.3 on how CPU time is consumed). Subtypes of the `ADOM` class provide different implementations of `activate()` to implement different types of workload (see section 5.1.5 for details). The `ADOM` class also provides methods to block the ADOM or to yield the current allocation. Furthermore, the `ADOM` class implements the user-level part of the basic IPC mechanism, described in section 5.1.2. The `ADOM` base class is derived from Nemesis' Virtual Processor (VP) interface which, in Nemesis, is also used to multiplex user-level threads and to implement thread synchronisation.

5.1.2 The event mechanism

The simulator provides a very basic IPC mechanism which is a direct implementation of the *event channels* abstraction[2] used in Nemesis (see e.g., [LMB+96, section III.a]). Communication between two ADOMs is accomplished via communication Endpoints (EPs) which are connected via event channels. Channels are unidirectional, allowing a *single* numerical value to be conveyed from a transmit (TX) EP to a receive (RX) EP via the asynchronous `send_event()` function, that is provided by the simulator core. Currently, each ADOM has a fixed length array of EPs. The simulator core provides functionality to connect two EPs via a channel (`bind_ep()`). The

[2]Strictly speaking, event channels are a synchronisation mechanism which can be used to implement a variety of higher-level IPC mechanisms.

ADOM class provides methods to allocate and free EPs, to send an event of a particular value via a TX EP, and to read the value of an RX EP.

The event mechanism also interacts with the scheduler, allowing it to unblock blocked ADOMs and SDOMs, and to give preferential treatment to these tasks to reduce dispatch latency. If an ADOM sends a value over an event channel via `send_event()`, a reference to the receiving ADOM is placed into a FIFO managed by the simulator core, indicating to the scheduling algorithm that the ADOM has received an event. The `send_event()` code also places a reference to the RX EP into a ADOM specific FIFO. This allows ADOMs to easily check for any pending events. As an obvious optimisation, an EP is not placed on the FIFO if it has already been placed into it due to a previous event.

It is worth noting that the event mechanism does not impose *any* restrictions on the interpretation of the event values. Furthermore, a receiver is not able to reconstruct any intermediate values an EP might have had in between subsequent reads. Any semantics carried by events are defined at a higher level between ADOMs.

One such mechanism, used in Nemesis, is an Event Count (EC) [RK79]. In the simulator, ECs are provide by a separate class, `Events`, and ensure monotonically increasing event values. The `Events` class provides methods to read the value of an EC, to advance an EC by an increment, and to block the ADOM until the EC has reached a specific value (or, optionally, a timeout has expired). To provide an IPC mechanism, ECs can be attached to connected EPs — advancing an event count then results in an event carrying the updated value to be sent to the peer EP in the peer ADOM.

In Nemesis, ECs are used for both inter and intra ADOM synchronisation. For example, [Bla95] demonstrates how a variety of user-level thread synchronisation primitives can be built using ECs and *sequencers*. For inter-domain communication, both a local RPC mechanism [Ros95] as well as a bulk data transfer mechanism, known as *RBufs* [Bla95], are built using ECs for synchronisation.

For the simulator, only the basic EC mechanism is provided for inter ADOM communication, because, so far, it has been sufficient to model simple interactions between ADOMs. For example, a synchronous Local RPC call can be modelled using two ECs per ADOM connected via a pair of EPs: The client's TX EC is connected to the server's RX EC, and vice versa. To "call" the server, the client advances its TX EC by one and than awaits an advance on its RX EC. The server ADOM does the reverse, starting, however, with awaiting an advance on its RX EC. More complex interactions, such as a server serving multiple clients, can be built in a similar fashion.

5.1.3 The simulator core

The core of the simulator provides the infrastructure in which simulations are executed. More specifically, it provides the equivalent of the functionality of the Nemesis Trusted Supervisor Code (NTSC) as well as the event machinery driving a simulation. The most important facility provided by the core is the notion of simulation time. Simulation time is maintained in the global variable `Now`. Time is typically interpreted in units of nanoseconds, although the simulator itself does not have a notion of time units — time is simply a monotonically increasing value.

Simulation time is advanced through the `consume_time()` method, which, for example, is used by ADOMs to consume CPU time in their `activate()` method. Different parts of the simulator can schedule events to be executed at specific times. There are two types of scheduled events: those which are part of the main simulation, such as preemption times, and those a user can schedule, e.g., for changing scheduling parameters or adding new tasks at specific times. Only the former are scheduled at the exact time. The latter are scheduled some time after the specified time, whenever the simulator core was entered for other reasons. A caller to `consume_time()` specifies by how much the simulation time should be advanced, i.e., how much time should be consumed on its behalf. The scheduler core then checks if this would advance the simulation time beyond scheduled events of the first type. In this case, simulation time is only advanced to the time of the scheduled event. The simulator core returns from a `consume_time()` call with the time to which it actually advanced the simulation time.

This mechanism allows for the implementation of both preemptive and non-preemptive scheduling algorithms. For non-preemptive scheduling algorithms, an ADOM is allowed to consume as much time as it wants until it either blocks or yields — `consume_time()` always returns the time passed in and advances scheduling time by that value. For preemptive algorithms, an ADOM is only allowed to consume time until the scheduling time reaches the time of the next scheduling decision, as set by a preemptive scheduler. These times can be either periodic, to model periodic timer interrupts used in some operating systems, or non-periodic, modelling a programmable timer.

The core simulator essentially executes in a loop which invokes the selected scheduler and activates the ADOM selected by the scheduler. While an ADOM is active, simulation time is advanced as described above. Within the loop, user scheduled events are also executed at the scheduled time. Optionally, a very small perturbation of simulation time can be introduced on every run through the loop. This may be used to introduce — albeit, in a very crude way — variations seen in real systems, so some results look less artificial.

The simulator core also provides some of the facilities implemented by the NTSC and other system services in Nemesis. Namely, it provides functions to block and yield an ADOM, to bind two communication EPs together, to send events, and to create and delete ADOMs. While most of this functionality could be implemented elsewhere, it keeps the structure of the simulator closer to that of Nemesis.

Finally, the core also provides a generic tracing facility through which all scheduling related events are timestamped and logged to a file. Tools are provided to extract interesting information from these log files. The tools are described in more detail in section 5.1.7.

5.1.4 Implementing schedulers

In the environment provided by the task model and the simulator core, the implementation of different scheduling algorithms is straightforward. All schedulers extend the base `Scheduler` class and typically extend the `SDOM` class to include scheduler specific attributes. The `Scheduler` class provides methods for adding and removing ADOMs and to change their scheduling parameters, e.g., priority. The main method of the `Scheduler` class is `reschedule()`. This method is invoked by the simulation core whenever a scheduling decision has to be made and typically performs the following steps:

1. De-schedule the currently active ADOM/SDOM combination. This may include updating some usage statistics for the SDOM and, optionally, blocking the ADOM.

2. Check if blocked ADOMs became runnable, i.e., check for expired time-outs and delivered events. The latter is facilitated by the kernel event FIFO in which `send_event()` places ADOMs which have received events.

3. Run the scheduling algorithm and select the next SDOM and an associated ADOM to allocate the CPU to. Also, for preemptive scheduling algorithms, a time for the next schedule needs to be calculated.

For hierarchical schedulers, these steps are slightly more complicated. If an ADOM wishes to block, all its associated SDOMs need to be notified and they may also need to be blocked if they have no other associated runnable ADOMs. During the unblocking phase, any unblocking ADOM needs to be added to all its associated SDOMs, unblocking them if necessary. In the final step, the winning SDOM needs to nominate one of its associated ADOMs as described above.

While most scheduling algorithms only extend the SDOM class with scheduler specific fields, the object-oriented design allows for the straightforward extension of other basic simulator primitives. For example, the implementation of a Lottery scheduler [WW94] naturally extends the event mechanism to implement ticket transfers from a blocking task to the task it is blocked on.

A wide variety of scheduling algorithms have been implemented in the simulation environment. These include real-time scheduling algorithms, such as RM, EDF [LL73], and Atropos [Ros95, Bar98], virtual-time based, proportional fair algorithms, such as Stride scheduling [WW95] and BVT [DC99], as well as classic priority based and round robin algorithms. In the experience of the author, an implementation of a new algorithm only requires a few hours (obviously, depending on its complexity) and requires about 140 lines of Python code to be written, a significant portion of which is boilerplate code, dealing for example, with adding and removing tasks or (un)blocking them.

5.1.5 Modelling workloads

Workloads are implemented by providing subtypes of the basic ADOM class. In particular, workloads are specialisation of the ADOM class' `activate()` method. With the simulator, a number of individual workloads are provided which can be easily combined to create more complex workload mixes. The workloads provided can be categorised into "toy" workloads modelling very simplistic applications, which can be used to highlight certain aspects of a scheduling algorithm, and realistic workloads, which are derived from real applications. First a number of "toy" workloads are described.

The simplest workload models a batch application, always consuming its entire CPU allocation. Thus, the `Batch` class simply calls the `consume_time()` function with a very large time value every time it gets activated. The `Block` and `Yield` workloads both consume a configurable amount of CPU time before blocking or yielding respectively. For the `Block` workload, the time it blocks for is also configurable.

The `PingPong` class is a simple workload making use of the event mechanism. Two instances of this class play ping-pong with event counts. After consuming a specified amount of CPU time, the first ADOM increments a TX event count by one and blocks until its RX event count is incremented by the other ADOM. This workload can be used to highlight the unblocking characteristics of a given scheduling algorithm.

An extension to this simple workload can be used to model a server applications. The server ADOM has an RX/TX endpoint pair for each of it clients. Clients send "requests" by sending an event on their TX endpoint and then block on their RX endpoint waiting for the server's response. The server task consumes some CPU time in order to "satisfy" the client's request and then sends its response by sending an event on the appropriate TX endpoint. A server task can implement different internal scheduling algorithms to decide which client to serve if different clients send requests concurrently.

A variety of bursty workloads can be modelled using Markov processes. Markov processes describe stochastic processes with discrete states. Probabilities are assigned to the transition from one state to another. The decision of which state to change to only depends on the current state and not on past states. Thus, Markov models can be described by a transition probability matrix. In the simulator, Markov workloads can be generated using the `Markov` class. Objects of this class are instantiated with a list of states and a probability matrix defining the probabilities for transitions between states. In each state the ADOM either consumes a specified amount of CPU time or blocks for a specified time. This general interface allows the definition of simple, two-state ON-OFF Markov workloads as well as more complex workloads.

5.1.5.1 Multimedia workloads

Multimedia applications are generally regarded as requiring special treatment from a scheduler — a number of scheduling algorithms have been designed specifically to support them (see section 2.1). Multimedia workloads, in particular interactive ones, place special demands on a scheduler as they have strict timeliness requirements, often combined with overall high, *and* highly variable, resource demands.

The highly variable resource demands can be attributed to the encoding and compression used for video streams. The MPEG format [LG91], for example, uses three different types of frame encodings, which require different amounts of CPU to decode. The three types are `I` (intra picture) frames, `P` (predicted picture) frames, and `B` (bidirectional predicted picture) frames. In general, `I` frames require more decoding time than `P` and `B` frames as they contain the entire image. In contrast, `P` and `B` frames only encode differences from reference frames, thus less data needs to be decoded. Reference frames can be `I` and `P` frames. Further, frames are grouped into Groups Of Pictures (GOPs) which contain sequences of frames from one `I` frame to the next.

As the different types of frames require different amounts of CPU to decode, the GOP sequence has an impact on the overall resource demands and the variability of the resource demands. Furthermore, the content of a video stream can have a significant impact on the resource demands as well. Scenes with highly dynamic content require more information to be encoded in P and B frames, thus require more CPU resources to decode. In general, it is rather difficult to predict a priori the CPU resource requirements for decoding any given stream. For example, Bavier et al. [BMP98] achieve an accuracy of 25% of predicted decoding times over actual decoding time with predictions based on observations of frame type and size.

Resource demands can be reduced by lowering the quality of the resulting image prior to the main decoding stage. MPEG, and also JPEG [Wal91], compress an original stream by converting 8×8 pixel samples into the frequency domain, using Forward Discrete Cosine Transform (FDCT), quantising them to values from 1 to 255, and then Huffman encoding them. For decoding, this process is reversed with the final transformation from the frequency domain to the spatial domain being performed by an Inverse Discrete Cosine Transform (IDCT). The IDCT is the most resource intensive operation in the decoding chain, and by reducing the information passed into it, the resource demands can be lowered at the cost of a lower quality. Information can be reduced after the Huffman decoding, by setting coefficients which are below a certain threshold to zero. The visual effect of this information reduction is that frames look more "blocky" as fewer levels of details are displayed.

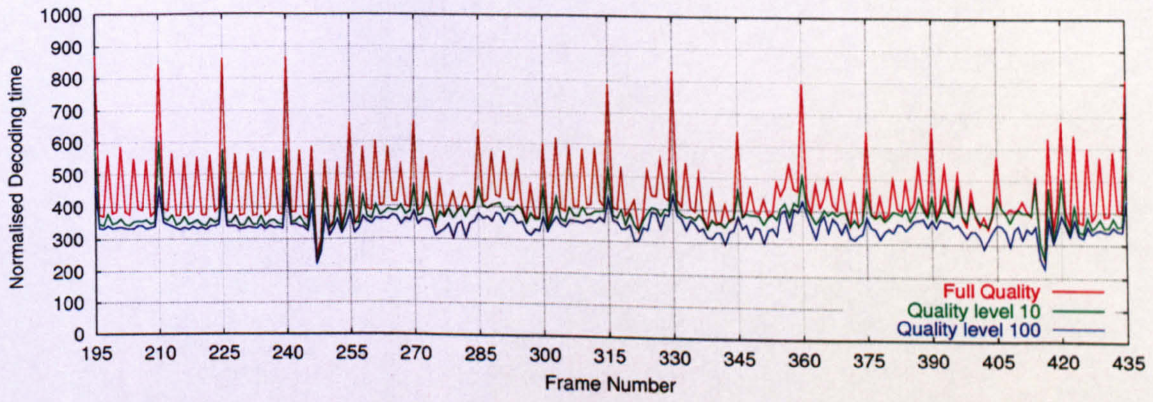


Figure 5.1: Decoding times for an encoded MPEG video at different quality levels

In order to model a variety of workloads based on video decoding, an MPEG decoder, `mpeg2play`, and a Motion JPEG decoder[3] have been instrumented using the per-process cycle counter, avaliable on Digital Alpha based systems[Mos97]. The instrumentation measures the per-frame decoding times for a given stream at a specified quality level. The resulting decoding times are normalised to values from 0 to 1000 and stored in a table with a frame per row and each column representing the decoding time at a different quality level. For an MPEG stream the frame type is also stored in the table.

[3]Originally written by Neil Stratford at the Computer Laboratory, University of Cambridge, U.K.

Figure 5.1 illustrates a sample taken from video stream, using an IBBPBBPBBPBBPBB grouping of frame types, with three quality levels[4]. In the graph every I frame falls directly on a tick on the x-axes. Clearly visible are the different decoding times for different types of frames, scene changes (e.g., after frame 240), and the reduction of resource requirements for reduced quality decoding.

Within the simulator, ADOMs, implementing video decoding workloads, are initialised with a file containing the table of normalised frame decoding times, the desired frame rate, and a factor for translating the normalised decoding times into simulator time scale. The simple video workload ADOM consumes the time for decoding a frame and then blocks until the next frame is due. When unblocked, the ADOM writes an entry to the log file that a frame has been "displayed".

More complex, adaptive workloads can be implemented by an ADOM comparing the currently achieved frame rate with the requested frame rate. For example, if a frame's deadline was missed by a certain threshold, an adaptive player may consider skipping the next frame or reducing the quality of the decoded image, as described above. Furthermore, in a real decoder application, the decoding step can be decoupled from the displaying step by executing them in separate threads. By introducing a buffer between the two threads, the decoder can, potentially, make more efficient use of allocated time slices, as it need not to block after each frame. Similar behaviour can be modelled within an ADOM by introducing a FIFO of decoded frames and periodically checking if the head of the FIFO needs to be "displayed". In [Neu99], I demonstrated, using a sample Motion JPEG video application, that some these techniques can be used to make the resource demand of adaptive multimedia applications more elastic.

As the video workload ADOMs write log entries for the frame display times, their performance can be evaluated. In general, the simulator environment provides tools to extract information, such as how many frames have been displayed on time, or how much jitter was introduced. This allows the comparison of different scheduling algorithms for multimedia workloads and different adaption strategies under different workload conditions[5].

[4]The values were obtained on a Alpha PC164LX 533MHz and, for this stream, the normalised value of 1000 is equivalent to 16846394 cycles or around 31 milliseconds.

[5]I have also instrumented an MP3 decoder, measuring audio frame decoding times. However, its resource demand is intrinsicly linked with the buffering performed by the audio device driver. For example, under FreeBSD the device driver essentially polices an audio application like a leaky bucket — if the buffer is full the application is blocked and if the buffer is empty, the audio breaks up. In order to evaluate the performance of audio applications, this or other audio device driver models would have to be implemented within the simulator.

5.1.5.2 Interactive workloads

Interactive applications are another type of application which may require special treatment by a CPU scheduler. Typically, interactive applications have fairly low resource requirements – although applications like a word processor may perform a significant amount of work while active. However, the response time of the system to user events, such as displaying characters after they have been typed, should be fairly low. The author of [Shn92] recommends system response latencies to be less than 50-150 ms.

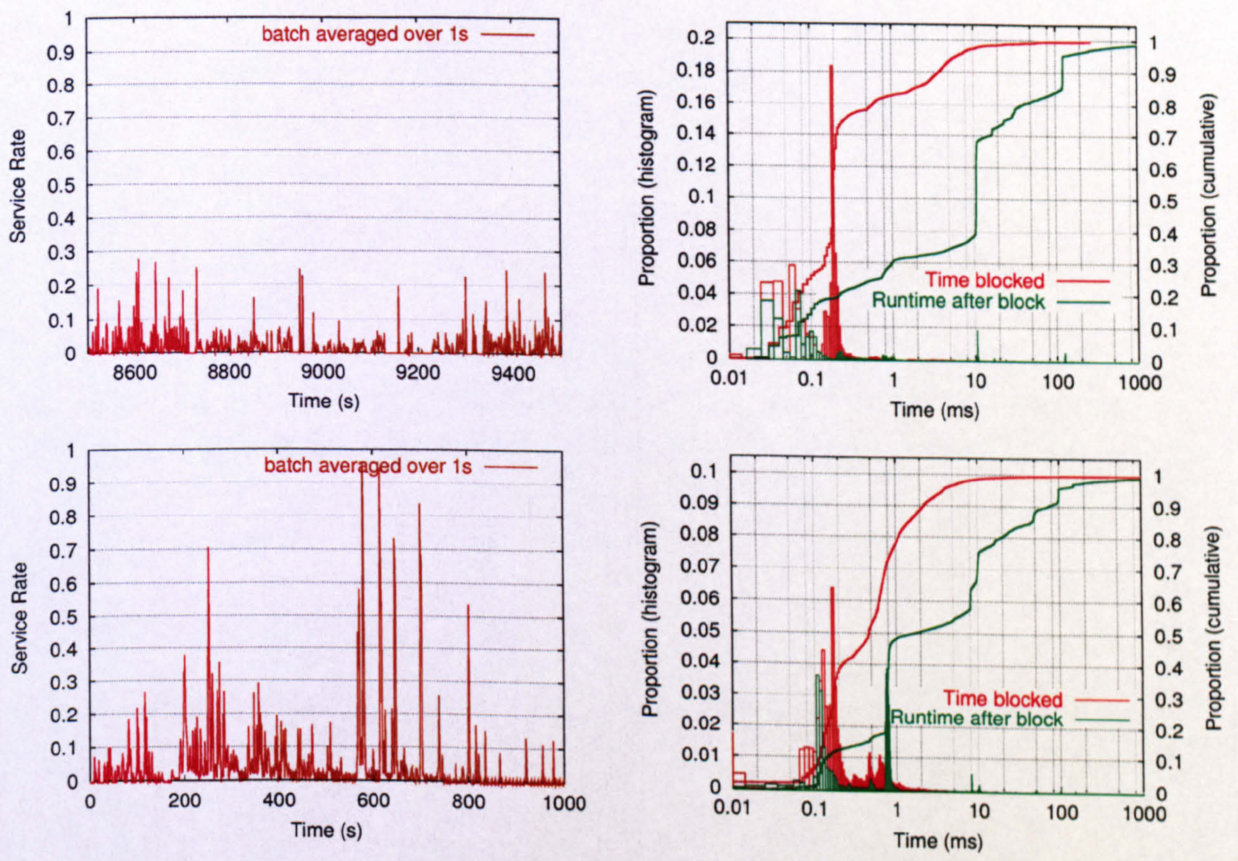


Figure 5.2: `emacs` (top) and `netscape` (bottom) characteristics

To illustrates the resource requirements of interactive applications, a scheduling trace from an instrumented FreeBSD kernel is used (the instrumentation is described in section 5.2.2). The trace was collected during a typical working day from the author's workstation (a 533MHz Alpha PC164LX with 128MB of RAM). Figure 5.2 shows sample data extracted from the trace for two interactive applications: `emacs` and `netscape`. The two panels to the left show the very bursty nature of CPU consumption during a period of heavy use of the respective applications.

The two panels to the right summarise the dynamic nature of the applications by showing the distributions of times for which the applications have been in the blocked state and the amount of CPU they consumed between two blocked states over the entire day. The graphs contain both histograms, relative to the left y-axis, and cumulative distributions, relative to the right y-axis (note the different scales, and the logarithmic x-axis). For both applications, the graphs show heavy tail distributions of both blocked times and run times which reflect the bursty nature of the workloads.

To model interactive workloads in the simulator, one could use traces such as the ones summarised above. However, using such trace data may give the wrong impression of accurately modelling the behaviour of such applications — the trace data depends on: the system on which it was captured; the overall system load; and the specific scheduler used. These factors also influence a user's actions. Instead, interactive workloads are better modelled using a stochastic process, e.g., Markov processes as described above, to just capture the bursty nature of interactive applications. While this is clearly not ideal, it prevents the misconception of an accurate model for interactive applications.

The suitability of a given scheduling algorithm with respect to interactive applications can be evaluated using a set of standard tools provided with the simulator (described in section 5.1.7). The main performance metric for interactive applications would be dispatch latency, i.e., the time between the time an ADOM became runnable after being blocked and the time it subsequently is awarded the CPU. As noted above, this time should be below 50-150 ms.

5.1.6 An example simulation

To illustrate how the various parts of the simulator are combined to execute a simulation, figure 5.3 shows a sample simulation. In essence, simulations are Python programs which make use of modules and classes provided by the scheduling simulator.

In the sample simulation, first the simulator core is initialised and a trace file is created (lines 3-5). The simulator core will introduce random perturbations to the scheduling time with a maximum of $1\mu s$ as indicated by its parameter. Next, in lines 7-8, a scheduler is created and registered with the simulator core, in this example an instance of Nemesis' Atropos scheduler is created (see section 2.1.3). The parameter is the time quantum for best effort tasks, in this case $500\mu s$. In the next step, a number of ADOMs modelling different types of workload are created (lines 11-16). Note, that this simply creates `ADOM` objects but does not actually add them to the scheduler. The next block of code (lines 19-21) illustrates how a pair of event channels is set up

```
01: # [imports omitted]
02: # set up the simulation
03: sim   = SchedSim(us(0.1))                          # a new simulation with small time perturbations
04: trace = open('sample.log', 'w')                    # open trace file
05: sim.create_trace(trace)                            # create trace file of all events
06:
07: sched = Atropos(us(500))                           # create an Atropos scheduler
08: sim.set_sched(sched)                               # tell the Simulator core about it
09:
10: # create domains
11: dom0 = PingPong('Ping', us(200))                   # consume 200us then ping
12: dom1 = PingPong('Pong', ms(2))                     # consume 2ms then pong
13: dom2 = Block('Block', ms(5), ms(1))                # consume 5ms then block for 1ms
14: dom3 = Markov('Markov1', ms( 2),.75, ms(5),.5)     # ON/OFF Markov
15: dom4 = Markov('Markov2', ms(.2),.80, ms(2),.8)     # ON/OFF Markov
16: dom5 = Batch('Batch1')                             # consume all you can get
17:
18: # set up event channels
19: ep0 = dom0.get_ep_pair()
20: ep1 = dom1.get_ep_pair()
21: kernel.bind_ep_pairs(dom0, ep0, dom1, ep1)
22:
23: # Add ADOMs to the scheduler
24: sched.add_domain(dom0, ms(10), ms(1), ms(10), 0)
25: sched.add_domain(dom1, ms(10), ms(2), ms(10), 0)
26: sched.add_domain(dom2, ms(10), ms(2), ms(10), 0)
27: sched.add_domain(dom3, ms(10), ms(2), ms(10), 0)
28: sched.add_domain(dom4, ms(10), ms(1), ms(10), 0)
29: sched.add_domain(dom5, ms(10), ms(2), ms(10), 0)
30:
31: # create events
32: sim.at(sec( 30), lambda:sched.set_param(dom5, ms(10), ms(1), ms(10), 0))
33: sim.at(sec( 30), lambda:sched.set_param(dom0, ms(10), ms(2), ms(10), 0))
34: sim.at(sec(100), lambda:sched.set_param(dom5, ms(10), ms(2), ms(10), 0))
35: sim.at(sec(100), lambda:sched.set_param(dom1, ms(10), ms(1), ms(10), 0))
36: sim.at(sec(200), lambda:sim.stop())
37: sim.run()
```

Figure 5.3: Sample simulation

between the two PingPong ADOMs. After creating the communication channel all ADOMs are added to the scheduler with the specified parameters for period, slice, latency hint, and, extra time flag (lines 24-29). Finally, a number of scheduled events are created using Python's lambda function (lines 32-26). The scheduled events change the scheduling parameters for two ADOMs after 30s and 100s. The simulation is stopped after 200s.

When run, this simulation creates a log file called sample.log which can be subsequently analysed by a set of tools described in the next section. To investigate the same task set with a different scheduler, only a different scheduler needs to be instantiated and the scheduling parameters need to be changed accordingly. Further, as a simulation is essentially a Python program, the full functionality of the programming language can be used to set up simulations. This, for example, allows for the systematic exploration of choices for options.

5.1.7 The tools

As part of the simulation environment a number of tools are provided to parse log files, to extract numerical information, and to perform a variety of data analysis tasks on the data[6]. As the trace files only contain log entries for scheduling events, the parser essentially creates a state machine for each ADOM, changing its states whenever an event occurs. At state changes, the requested information may be extracted and written to an appropriate data file.

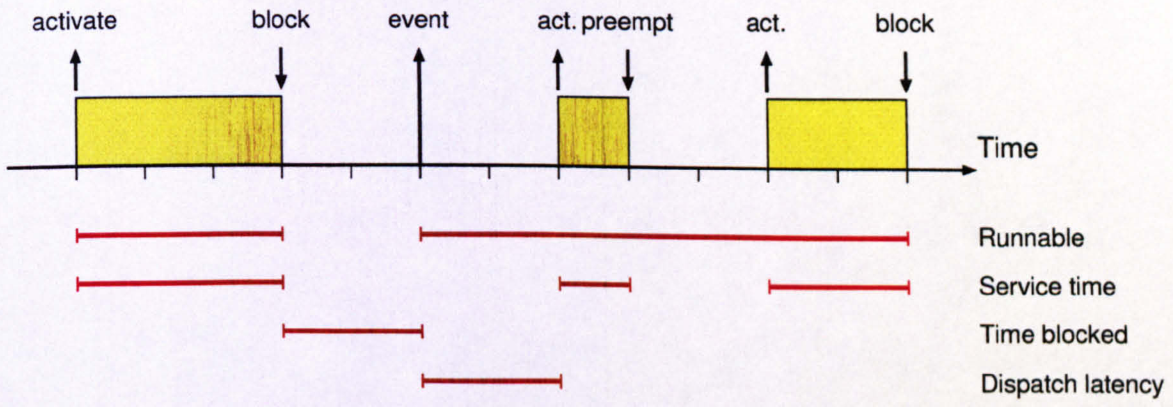


Figure 5.4: Extracting information from trace file events

To illustrate this, consider the time line of events for an ADOM displayed in figure 5.4. Logged scheduling events are marked with arrows. Below the time line a number of interesting metrics are drawn. For example, service time can be extracted by monitoring activation and "de-activation" events, such as block or preemption events. In general, the tools first extract this raw data and then presents it in a more compact form.

To give an example, service times can be extracted either as raw data, as shown in the figure, or as cumulative service time over the lifetime of an ADOM. To evaluate criteria such as fairness of allocations, service rates can be calculated from the raw data by averaging over a configurable window size. Currently, both batch and moving averages can be calculated. Others, such as exponentially-weighted moving averages or *Flip-Flop* [KN01], can be easily added. For real-time schedulers with purely periodic task sets, averaging windows can also be based on multiples of the period rather than specifying a fixed window size.

To evaluate the suitability of a scheduler, especially for interactive tasks, dispatch latencies are an important measure. Dispatch latency denotes the time between an ADOM receiving an

[6]The same tools are used to analyse trace data gathered from an instrumented FreeBSD kernel, described in section 5.2.

event or a timeout expiring and the ADOM being activated. As with service times, the tools are able to provide a range of presentations of dispatch latencies. Along with the raw data, dispatch latency distributions and cumulative distributions can be extracted. These distributions may be used to demonstrate that a particular scheduling algorithm provides an upper bound on dispatch latencies. Similar data can be extracted for blocking times.

The tools can also be used to correlate different data sets. For example, scatter plots of time blocked vs. dispatch latencies may be used to analyse if a scheduling algorithm provides preferential treatment for long or briefly blocking tasks. Likewise, plots of time blocked vs. service time after unblocking may be used to characterise workload behaviour.

Sometimes it is important to investigate the exact schedule a particular task set/scheduler combination produces. For this purpose, a tool is provided which generates an annotated scheduling time line, not unlike the one presented in figure 5.4. In these time line graphs, events and block timeouts are annotated with arrows. For real-time schedulers, deadlines and missed deadlines are also marked.

Finally, the tool set also provides a number of tools to extract scheduler and workload specific trace date, for example, to calculate deadline misses for real-time schedulers or to analyse the performance of video decoding workloads. In practise, the tracing mechanisms have been flexible enough to extract all data necessary to characterise and to analyse different schedulers and workloads.

5.1.8 Considering overheads

The simulator currently presents an idealised system which does not account for overheads introduced by, e.g., context switches or system calls. It only introduces some optional "jitter" to simulation time. To take system overheads into account the simulator core would need to be extended to advance simulation time, using `consume_time()`, on a context switch or whenever system calls are executed.

In order to get an idea of the scale of these overheads, the cost for basic operations needs to be measured on a real system. As the simulator models the Nemesis operating system, it is natural to measure the overhead for a number of basic operations under Nemesis. These measurements can then be used to decide whether it would be necessary to also model these overheads in the simulator. To measure the overheads in Nemesis, a number of micro-benchmarks[7] were

[7]These benchmarks were mainly written by Paul Barham at the University of Cambridge Computer Lab. But, other people, including the author of this dissertation, also contributed individual benchmarks.

performed on two different platforms: a 533MHz Alpha PC164LX system and an 200MHz Intel Pentium. The results are summarised in figure 5.5.

Alpha			Intel		
Benchmark	Time(ns)	Cycles	Benchmark	Time(ns)	Cycles
ntsc_yield()	3092	1648	ntsc_yield()	10061	2006
Events$Advance	145	77	Events$Advance	407	81
ntsc_send()	45	23	ntsc_send()	729	145
Event Channel Ping Pong	13674	7288	Event Channel Ping Pong	103112	20564
IO Ping Pong	16875	8995	IO Ping Pong	136347	27193
NULL RPC	23777	12674	NULL RPC	166230	33153

Figure 5.5: Micro-benchmarks for AXP PC164LX 533 MHz and Intel P200

For each operation implemented by the simulator, its equivalent on Nemesis is executed in a tight loop 10000 times after an initial warmup loop of 1000 iterations. The time is recorded at the start and the end of the main loop. The numbers given in the table are the average time and the number of cycles for a single operation, i.e., the recorded time divided by the number of iterations. All benchmarks were run on an otherwise idle system.

The first benchmark, labelled `ntsc_yield()`, continously yields the CPU. This causes a full de-schedule of the process, a run through the scheduler, and an activation through the user-level thread scheduler back to the thread which called `ntsc_yield()`, as it is the only active thread in the process. The slightly slower execution of this benchmark in terms of cycles on the P200 can largely be attributed to the higher cost of crossing the user-kernel boundary required for the system call.

The second benchmark, `Events$Advance`, measures the cost of advancing a *local* event count, essentially an atomic operation as described in section 5.1.2. Performance in terms of cycles spent is comparable between the two architectures. The third benchmark, `ntsc_send()`, measures the cost of sending an event on a "bogus" event channel[8]. On the Intel platform the overhead of the system call is significant.

The fourth benchmark measures the time of a round-trip ping pong of events using event channels between two processes. Thus, two runs through the scheduler and two `ntsc_send()` calls on a valid event channel are required during each iteration. The cost for this benchmark

[8]Sending an event on a "bogus" channel performs most of the work a `send_event()` call would perform on a connected event channel. The RX count, however, is not updated and the RX endpoint is not placed into the peer's event FIFO, as described in section 5.1.2. These are comparatively cheap operations.

is significantly higher on the P200 — it requires almost three times as many cycles than on Alpha system. This again, can partly be attributed to the system call overhead. Also, the cost of a switch between different protection domains is considerably higher on Intel architectures compared to the Alpha architecture (see [Han99a] for a detailed discussion). The final two benchmarks measure round-trip times for different IPC mechanisms, built on top of the event channel mechanism, one using the bulk data transfer mechanism and the other using the local Remote Procedure Call (RPC) mechanism. The increased time required, compared with the simple channel ping pong, can be attributed to the additional overheads introduced by these mechanisms. However, the significantly higher overhead for the NULL RPC call on the Alpha system, compared to the "event channel ping-pong" benchmark, may be caused by less optimised stub code generation.

The results from these micro-benchmarks could be incorporated into the simulator through a configuration file specifying the cost of certain operations for different architectures. However, for most applications the overhead is sufficiently small to be ignored. Furthermore, the performance of a real system is likely to be more influenced by other factors, such as caching effects, which are more difficult to account for in the environment of the simulator. Therefore, it makes more sense to keep the simulator simple and simply take these limitations into account when interpreting simulation results.

An alternative way of assessing scheduling overheads is to examine the accuracy of CPU allocations over a wide range of scheduling parameters. For this purpose, a small benchmark was constructed. When active, the benchmark program consumes all CPU allocated to it by means of an endless loop. In a control thread, the CPU reservations are systematically changed and the received share of the CPU is recorded. More specifically, the control thread systematically alters the period and the percentage of the CPU allocation for the benchmark process. After a period of time equivalent to 1000 periods, the accounting information maintained by the scheduler and the number of iterations through the endless loop are recorded.

The results from this benchmark on the two test systems are shown in figure 5.6. The two left-hand graphs show the percentage of time received by the benchmark process over a wide range of different periods at different percentages of CPU allocation. All values are averaged over five runs with error-bars showing the minimum and maximum values (variations between runs are insignificant so the error-bars are not visible in the figures). The left-hand graphs illustrate that the allocation of CPU time to the benchmark process is very accurate on both platforms. Only for very short periods of less than 200 μs can one see a divergence between the allocation the process should receive and the allocation it actually receives. This trend is

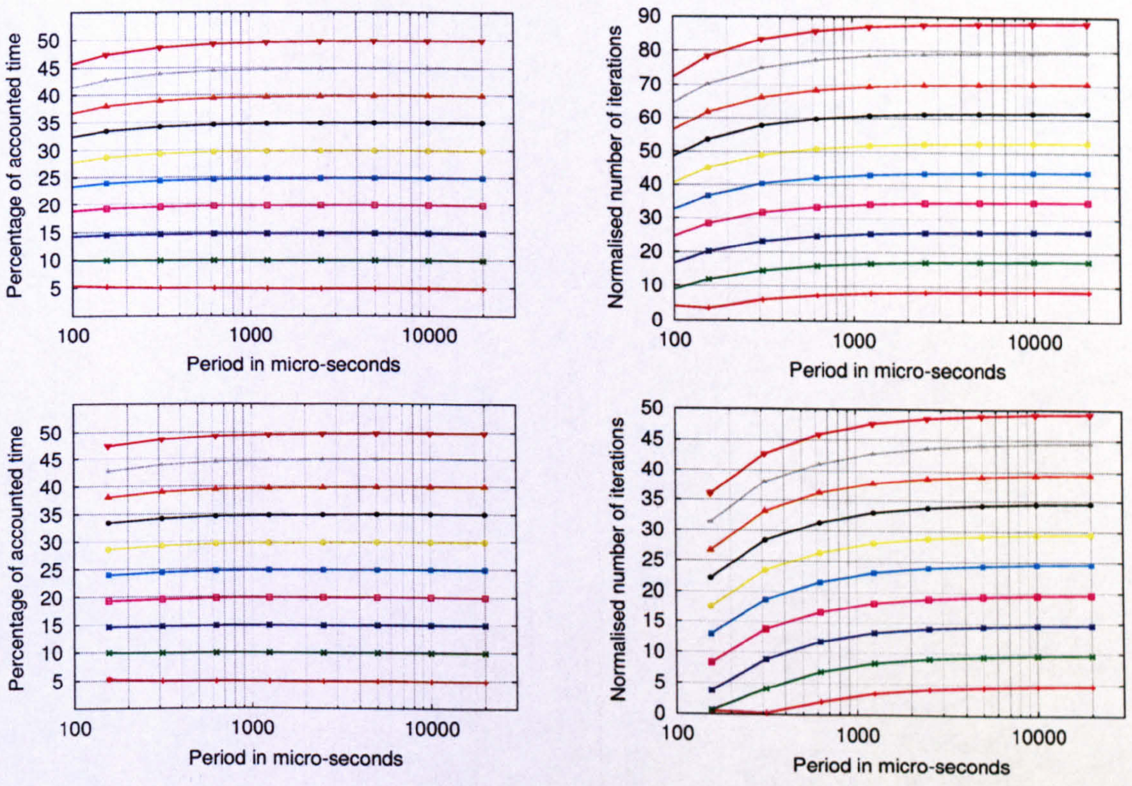


Figure 5.6: Scheduling accuracy for AXP PC164LX (top) and Intel P200 (bottom)

stronger for higher allocation percentages. It is worth pointing out that the benchmark could not be run on the Intel P200 for periods shorter than around 150 μs as the machine would "lock up", probably spending most of its time processing the programmable timer interrupt without making any real progress.

While the graphs on the left-hand side show the percentage of the CPU received by the benchmark application, the right-hand graphs show how much work, i.e., iterations of the endless loop, has been performed by the benchmark application. Again, these are averaged over 5 runs with minimum and maximum values shown as error-bars. One can clearly see that the amount of work performed by the benchmark is proportional to its CPU allocation. However, in particular on the P200, the amount of work performed declines significantly for shorter periods. This can be attributed to the scheduling overhead, which as indicated by the micro-benchmarks, is higher on the Intel Pentium based system.

In summary, Nemesis is capable of allocating CPU resources accurately over a wide range of timescales (from hundreds of microseconds upwards) and is capable of accounting for the CPU usage accurately (as indicated by the left-hand graphs in figure 5.6). However, the performance perceived by the applications themselves may be reduced for shorter periods due to scheduling overheads. For the simulator, this means that as long as care is taken to ensure that periods are not below a certain threshold, for example, $500\mu s$, no additional measure must be taken to model scheduling overheads.

5.1.9 Implementing congestion pricing

In the next chapter, the simulator is used to explore a variety of congestion pricing mechanisms and application adaption strategies. An extended version of the Atropos scheduler is used as the base scheduler. To cater for the variety of pricing mechanisms a subclass of Atropos, called `AtroposECO`, is used. `AtroposECO` is the base class for classes implementing different pricing mechanisms and maintains information such as CPU utilisation and sum of requests. CPU utilisation is calculated in configurable intervals using an Exponential Weighted Moving Average (EWMA) filter[9] [Tha98, KN01]. Different pricing mechanisms are implemented as subclasses of `AtroposECO` and may use this information or other information, such as missed deadlines, to calculate a current price.

The de-schedule code has been modified to charge SDOMs based on charging probabilities as described in sections 4.2.1 to 4.2.1.2. The pricing mechanism sets a probability with which the current SDOM is charged one credit for each time unit its active ADOM consumes. The length of the time unit is configurable. The base class also contains a mechanism to periodically charge SDOMs for their resource requests based on a price set by a pricing mechanism.

The `SDOM` class has been extended to contain a token bucket for credits as described in section 4.3. Credits are placed into the bucket at regular intervals in accordance to the set willingness to pay value set for that SDOM. SDOMs with an empty token bucket are not run if the usage price is non-zero and are placed on the wait queue if the request price is non-zero. The token bucket is only maintained internally. For SDOMs a separate attribute is maintained containing the total charges incurred and resource consumed. An ADOM associated with an SDOM, can access this attribute to calculate the rate at which its SDOM incurs charges. Based on this observation they may adjust their resource consumption.

Different adaption strategies are implemented as subclasses of the `ADOM` class. Simple adaptive workloads are created by merging `ADOM` subclasses implementing workloads, as described in section 5.1.5, and subclasses implementing adaption strategies using Python's multiple inheritance features. For example, a batch processing workload using the WTP strategy, is created by

[9]The reactivity of the filter to newer values can be controlled by a single parameter α, called the *gain*.

sub-classing from both the `Batch` class and the `WTP` class. More complex adaptive workloads may require tighter integration of the two subclasses.

For subsequent analysis, various events, such as changes to the charging probability and the observed marking rate, are written to the log files. As part of the tool set, tools are provided to extract this information for further analysis.

5.1.10 Summary

In this section a simulation environment for scheduling algorithms has been introduced. This environment is used in the next chapter to explore a number of different design choices for congestion pricing for CPU resource management. The simulation environment builds on some key abstractions introduced by the Nemesis operating system, which, in the experience gained with the simulator, have proven to be more than sufficient to support a wide variety of scheduling algorithms and workload models. Currently, the simulator and tools consist of around 6000 lines of Python code[10] and implement more than 10 different scheduling algorithms and a variety of workloads.

The workloads provided are mainly "toy" workloads which can be used to highlight certain characteristics of a given scheduling algorithm. More realistic workloads can be modelled to the extent that they capture the *essential* characteristics and properties of real workloads. In general, I believe, that it is not necessarily desirable, if at all possible, to model workloads *accurately*, as they often exhibit complex interactions with the operating system itself. So far my experience has shown that simplified workloads are sufficient to highlight certain features or problem areas of a given scheduling algorithm.

The simulator provides a good environment in which to experiment with congestion pricing mechanisms and adaption strategies. Different pricing mechanisms can be easily explored and directly compared with each other. Similarly, the simplified environment the simulator provides, allows for a direct comparison of different adaption strategies, avoiding the interactions with other parts of a real operating systems.

However, it must be stressed that simulation results should only form the basis for exploring congestion pricing mechanisms in real operating systems, as certain aspects of the simulator are oversimplified. For example, the simulator does not take into account: any overheads (e.g., for context switches or systems calls); limited timer accuracies, typically found in mature operating systems; or, indeed, interactions with other subsystems of an operating system, such as the virtual

[10]According to 'SLOCCount' by David A. Wheeler `http://www.dwheeler.com/sloc-count.`

memory system, I/O subsystem, and the network interface. Some of these issues are addressed in the next section, which provides background on a prototype implementation of congestion pricing in the FreeBSD kernel.

5.2 The FreeBSD prototype

FreeBSD[11] is an advanced BSD UNIX derived from the Berkeley 4.4 BSD distribution [MBKQ96] and widely used in server and workstation systems. It currently runs on both Intel ix86 and DEC Alpha platforms (other ports, including Intel's IA64, PowerPC, and Sparc64 are planned). In this section, a prototype implementation of the congestion pricing framework for CPU resource management under FreeBSD 4.3-STABLE is described.

For CPU scheduling, FreeBSD uses a typical decay usage priority scheduler which employs multi-level feedback queues [MBKQ96], similar to the schedulers found in other UNIX derivates. Processes are assigned priorities and the process with the highest priority is selected to run for a time quantum. Priorities are dynamically adjusted reflecting the resource usage of the processes. In addition to this relatively standard Unix scheduler, FreeBSD also defines two other scheduling classes (similar to the POSIX real-time scheduling classes): the real-time class and the idle class. Processes of the real-time class are scheduled with a higher priority than processes of the standard scheduling class, while processes of the idle class are scheduled with a lower priority. Priorities in both classes are not adjusted dynamically. The real-time class has two subclasses, FIFO and round robin. Processes of the round robin class are preempted after a configurable time quantum, if processes of the same priority are runnable, while processes of the FIFO class are only preempted by higher priority processes. Processes are put in the real-time or idle classes by using the privileged system call `rtprio(2)` or the system utilities `rtprio(1)` and `idprio(1)` respectively. The real-time scheduling classes defined by POSIX are mapped onto the same mechanism.

Nieh et al. [NHNW93] have demonstrated that this type of real-time support is inappropriate for multimedia workloads; indeed, using these extensions may result in locking up the entire system. Furthermore, the congestion pricing framework, introduced in the previous chapter, requires its clients to be able to make requests for absolute resource allocations, rather than the relative allocations provided by the priority based scheme of the standard FreeBSD scheduler. To provide CPU resource reservations, I have modified the FreeBSD kernel to provide a new scheduling class capable of providing CPU reservations to processes. This implementation is described in detail next.

[11]`http://www.freebsd.org/`

5.2.1 Providing CPU reservations

CPU reservations are offered to individual processes by providing a new scheduling class. Processes of this class have a higher priority than processes of the default scheduling class, but a lower priority than processes of the real-time class. Processes can reserve a share of the CPU using a new system call, `eco_sched()`[12]. For simplicity, CPU reservations are expressed as the number of microseconds per one second. However, for scheduling purposes a shorter epoch (or period) may be used with reservations scaled accordingly. The scheduling epoch can be configured dynamically at run-time using FreeBSD's `sysctl(8)` interface[13]. The default epoch is $1s$ but for some experiments, requiring more fine-grain sharing of the CPU, it is reduced to $100ms$. If a process requests a change of its CPU reservation, its share during the current epoch is adjusted proportionally to the share of the CPU it has already received during that epoch. The relationship of the CPU reservation scheduling class to the other scheduling classes is depicted in figure 5.7.

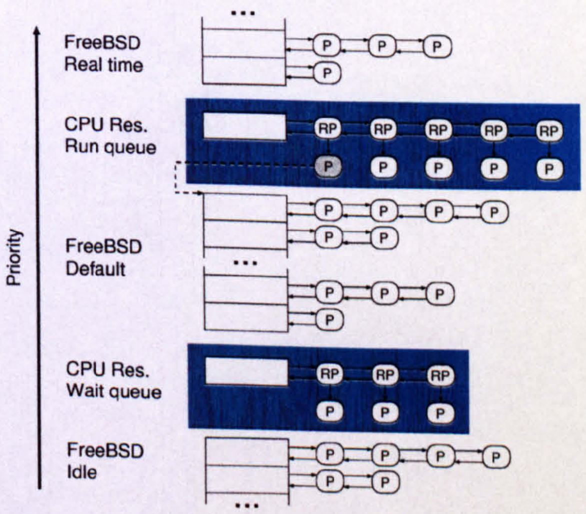


Figure 5.7: FreeBSD CPU reservation scheduling class

A runnable process of the reservation scheduling class resides on one of two queues: the run queue if it has not consumed its entire reservation yet, or the wait queue, if it has already received its share of the CPU but is still runnable. Blocked processes are kept on the standard FreeBSD kernel data structures for blocked processes. Every time a process relinquishes the CPU, either

[12]To limit the number of new system calls, all new system calls necessary are implemented via sub functions to this system call, akin to `fcntl(2)`.

[13]The `sysctl(8)` interface allows user programs to read and write kernel specific state variables.

voluntarily or through preemption, its runtime is subtracted from its current reservation. If a process becomes runnable again it is put at the end of the run queue provided it has not yet consumed its entire reservation; otherwise, it is put at the end of the wait queue. A periodically scheduled function renews the reservations every scheduling epoch and moves processes from the wait queue to the run queue, if appropriate.

When the scheduler has to pick a new process to run, it first checks if there are any real-time processes runnable, and if so a process is selected based on its real-time priority. If no real-time process is runnable, and there are processes on the run queue associated with the reservation scheduling class, the scheduler selects the head of that run queue. If the run queue is empty, a process from the standard scheduling class is selected, again based on priority. If there are no runnable processes in the standard scheduling class, processes on the CPU reservation wait queue are scheduled round-robin. And, finally, if the wait queue is empty, processes from the idle class are selected.

If a process of the reservation scheduling class is selected, it is given the CPU either until it blocks or yields or until it is preempted. Processes are preempted either by real-time processes becoming runnable, or because they have consumed their entire reservation, or a configurable preemption interval expires (default $10ms$ which can be changed at runtime using the `sysctl(8)` interface). The preemption interval prevents a process with a large allocation from monopolising the CPU for an excessive interval. Preemption within the scheduling class, i.e., consumption of the reservation or the preemption interval, is implemented using the kernel's `timeout(9)` function [VL87], which is scheduled by the periodic timer interrupt. To increase the accuracy of the preemption, the periodic timer interrupt frequency has been raised from 100Hz to 1000Hz. For non-blocking processes this may lead to accounting inaccuracies of at most $1ms$[14]. This effect is not cumulative, as the measured run time of the previous epoch is taken into account when the reservation is renewed every second.

The CPU reservation scheme as described so far works fine for individual processes placed into the new scheduling class by the user. However, processes that are part of the default scheduling class may be starved of CPU time just as they can be starved by processes of the real-time class [NHNW93]. To prevent this starvation, the default scheduling class is also run as a "dummy" process within the new class, i.e., a CPU reservation can be requested which all processes of the default class share. If the dummy process is selected by the scheduler, a process from the default

[14]Higher timer resolutions could be achieved using Soft Timers [AD00]. Soft Timers provide potentially higher timer resolutions with less overhead by also checking for scheduled events at trigger points, such as system call exits or hardware interrupts.

scheduling class is selected as usual. CPU time is then accounted to the dummy process. Like other processes, if it uses its entire reservation, the dummy process is put on the wait queue. However, if the run queue is empty, processes of the default class are selected as normal. Resource consumed by these processes are not accounted to the dummy process. If the system is idle, i.e., there are no runnable processes in the real-time or default scheduling class and the run queue of the reservation scheduling class is empty, processes are selected from the wait queue in a round-robin fashion. This mechanism gives priority to processes with CPU reservations, but favours processes of the default scheduling class, if all reservations have been satisfied.

The new scheduling class has been implemented with only minor modifications to the standard kernel. Most of its functionality is implemented in a separate source file. Most modifications were made in the `chooseproc()` function, mainly to allow the default scheduling class to be part of the new scheduling class. Minor modifications were necessary to `setrunqueue()` and `remrunqueue()` to deal with processes of the new scheduling class. The functions `tsleep(9)`, `await(9)`, and `yield()` had to be modified slightly to update the CPU usage account in the case where a process blocks or yields. And, obviously, `fork()` and `exit()` required modification to deal with process creation and termination. Currently, newly forked processes are placed in the default scheduling class, even if the parent process is part of the reservation scheduling class.

5.2.1.1 Discussion

The current implementation of CPU reservations is fairly simplistic — it merely provides a means of allowing individual processes to make CPU reservations. In this section a number of possible extensions and improvements are discussed. However, it is worth pointing out that the simple implementation has been sufficient for the prototype of the resource management framework, and the possible extensions and improvements would not fundamentally change the way the resource management is implemented.

An obvious limitation is the round-robin scheduling of processes in the new scheduling class. For larger active sets of processes in this class this may lead to unacceptable dispatch latencies (i.e., for n processes, a worst case delay of n times the preemption interval). A possible solution would be to insert unblocking processes at the *front* of the run queue in order of their wakeup priority — the `tsleep(9)` kernel function, used for event-based process blocking, allows the specification of a priority with which a newly woken process is made runnable. So far, I have not observed a dispatch latency, even for a non-buffering MP3 decoder or interactive applications, which would require such a change.

Another possible extension is to introduce a two level hierarchy of schedulers, where, by default, all child processes of a process in the new scheduling class are scheduled as one entity. The new scheduling class only selects such a process group and a secondary scheduling algorithm selects a process from this group to be scheduled. Such an approach would be beneficial especially in server systems where typically a service forks off a number of child processes to serve incoming requests. A server administrator could then manage CPU reservations for the service rather than individual server processes.

Ideally, the prototype should also contain more flexible abstractions for resource management, e.g., resource containers [BDM99] or reservation domains [BBG$^+$99c] (described in section 2.3) to provide a more accurate accounting and a flexible framework for different types of applications. The code available for these prototypes is only applicable to earlier versions of FreeBSD, changes the base system substantially, and would require further changes to accommodate the requirements of the decentralised resource management framework. To keep the experimental environment simple and manageable, I decided against using these prototypes. However, for a more generally applicable implementation, the use of either of these approaches would be beneficial.

5.2.2 Tracing facility

In addition to CPU reservations, the kernel was modified to provide detailed traces of all scheduling related information. The implementation of this tracing facility is similar to the Linux Trace Toolkit [YD00]. Essentially, all places in the kernel where scheduling decisions are made, such as selecting a new process, or blocking and unblocking of processes, have been instrumented. The tracing facility also traces creation and destruction of processes. When the tracing facility is active, these events are timestamped and written to a kernel-internal circular buffer. The superuser can activate tracing by starting a daemon process which periodically reads this data from a pseudo device and writes it to disk. When started, the daemon also takes a snapshot of the currently running processes.

For detailed analysis, the binary file, produced by the tracing daemon, can be converted offline into the format used by the scheduling simulator. This allows the simulation analysis tools, described in section 5.1.7, to be used for the analysis of traces from a FreeBSD system. However, typical traces obtained over a longer period of time, say an entire day, may contain trace information for a lot of "uninteresting" processes. Also, one is often interested in the resource consumption of an entire group of processes, for example, of a larger compile job where a significant number of short lived processes are created to perform the task. For this purpose, tools are provided to extract the process hierarchy from the trace file (similar to the `pstree` utility) and to extract only trace information about "interesting" processes, summarising all the other processes into a single background process, or to "collapse" the data of an entire sub-hierarchy into one process.

It is worth pointing out that the tracing facility is implemented independently of the CPU reservation system. The tracing facility can be conditionally compiled into an otherwise standard FreeBSD kernel by adding the appropriate option in the standard kernel configuration file. The tracing facility can also be used in conjunction with the CPU reservation system and the dynamic resource management prototype described below.

5.2.3 Evaluation of CPU reservations

In this section a brief evaluation of the CPU reservation subsystem is given. First, the ability to reserve CPU allocations and isolate processes from each other is demonstrated and contrasted with the default scheduler. The main focus of this evaluation is on compute-bound processes due to the limitations of the current prototype (see section 5.2.1.1). The tracing facility is used to provide detailed information on CPU resource allocations. Secondly, the overhead, introduced by the various alterations made to the FreeBSD kernel, is evaluated.

In figure 5.8, the effect of CPU reservations is illustrated by showing the service rates received by a number of processes. For this experiment, a 200 MHz Pentium was used and the service rates were averaged over one second. The use of a slow processor has the advantage of highlighting the effect of resource allocations by providing a more resource constrained environment. For the experiment, initially, a set of processes was started using the default scheduling class with different `nice(1)` levels. Then, after about 375 seconds some of these processes are entered into the new scheduling class with different CPU reservations. The processes include an MP3 decoder (labelled `mpg123`) and a number of processes executing an endless loop (labelled `loop`). To provide some background load, a large parallel compilation job compiling the FreeBSD kernel is executed. For the figure, the CPU resource consumed by all processes of this compilation are summed together and shown as one process, labelled `make`.

Initially, all processes are started in the default scheduling class with different nice levels. In the figure, the nice levels are indicated in the key by the first number in brackets. Four `loop` processes are started with nice levels of -10, -5, 5, and 10 respectively. Lower nice levels represent higher priorities. A fifth `loop` process is started with the default nice level of 0 as is the

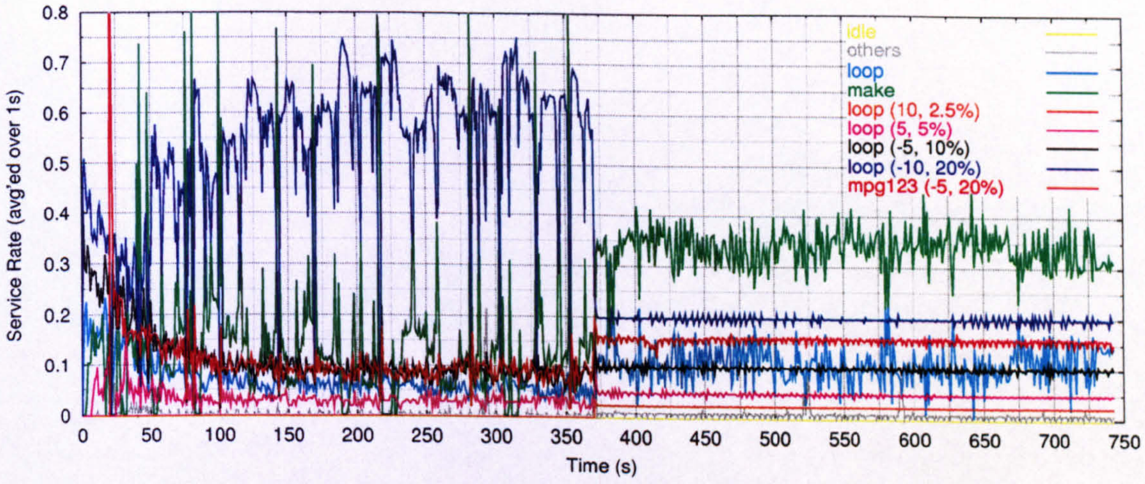


Figure 5.8: Nice levels vs. CPU reservations

compilation process. The MP3 decoder is started with a nice level of -5. From the graph it is clearly visible that, although the different nice levels for the different `loop` processes correspond to different service levels, there is no clear relationship between the different nice levels and service rates. Furthermore, the allocation is fairly bursty, although the service rates are averaged over a relatively long period of one second. It is also worth noting that, although the MP3 decoder has the second highest priority, the sound continously breaks up and plays back at an essentially unacceptable quality.

For the second part of the experiment some of the processes are placed into the new scheduling class providing CPU reservations. The four `loop` processes with non-default nice levels are given reservations of 2.5%, 5%, 10%, and 20% as indicated in the graph by the second number in the key. The MP3 decoder is given an allocation of 20%, while the fifth `loop` process and the compilation processes are left in the default scheduling class. In stark contrast to the first part of the experiment, processes with CPU reservations receive exactly the share of the CPU they have reserved, with very little variation in allocations over time. Note, that for this part of the experiment an averaging interval of one second is the smallest meaningful interval, as it represents the granularity at which CPU reservations are made. The `make` and the best-effort `loop` process share the remaining CPU resources and still show the bursty nature of allocations. The MP3 player uses less than its reservation of 20% and consumes an almost constant 15% of the CPU – there are no audible drop-outs in the decoded audio stream. As described in section 5.2.1, the surplus CPU resources are first allocated to processes of the default scheduling class. Due to the non-blocking `loop` process in this class, all surplus cycles are consumed by this class and no spare cycles are allocated to processes of the CPU reservation class which have used up their allocation.

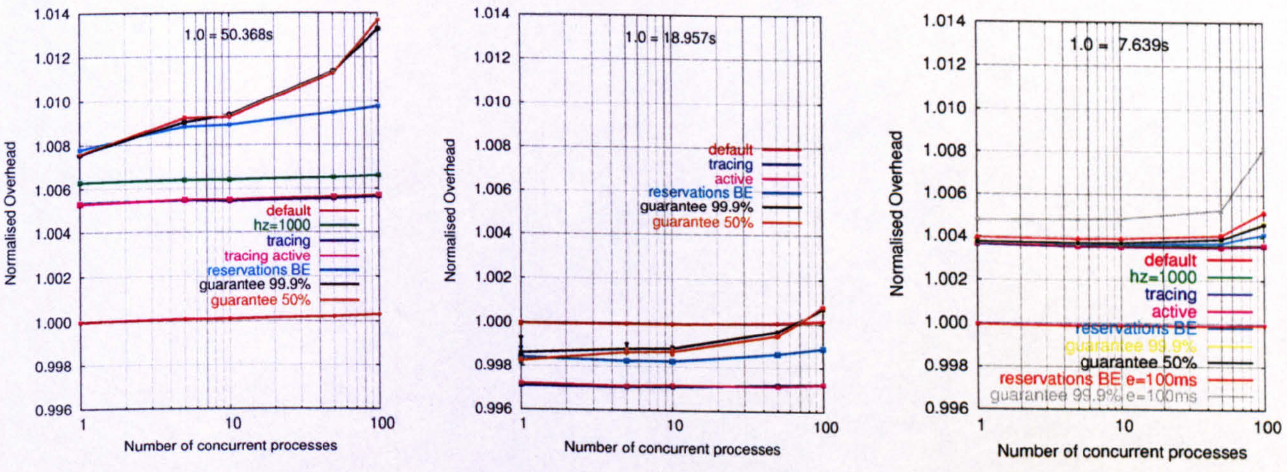
This experiment clearly demonstrates that the CPU reservation prototype is capable of providing absolute shares of the CPU to processes in the CPU reservation scheduling class.

Next, the overhead introduced by the various modifications made to the FreeBSD scheduling subsystem are evaluated. The overhead is measured using long running, entirely compute bound processes. For this purpose a modified version of a benchmark by Larry McVoy called `mhz`, part of `lmbench` [MS96], is used[15]. The original `mhz` benchmark was modified so that it is guaranteed to run longer than one scheduling epoch. To determine the overhead, initially a single instance of the benchmark is executed with the default kernel configuration and the time to execute it is measured. This forms the baseline performance and all further experiments are normalised to this value. Subsequently, concurrent instances of the benchmark process are started. By measuring the time for all processes to complete and normalising it by the number of concurrent processes, one can evaluate how well any given configuration scales with the number of active concurrent processes. This procedure is repeated for a number of different kernel configurations, which subsequently enable more and more features of the modified kernel. The results of this experiment are shown in figure 5.9 for three different computers: The Pentium 200MHz and Alpha PC164LX, already introduced, and a 1.3GHz AMD Athlon. The figure uses the average of five runs for each data point, with error bars indicating the minimum and maximum of the five runs. There are only insignificant variations between runs, therefore the error bars are not visible in the graphs.

The different lines in the graphs correspond to the different kernel configurations: `default` is the default kernel configuration with all modifications disabled at kernel compile time. The line labelled `hz=1000` represents the default configuration with an increased periodic timer interrupt frequency (1000Hz instead of the default 100Hz). The line labelled `tracing` shows the normalised overhead for a kernel configuration with the tracing code compiled in and the line labelled `tracing active` uses the same kernel but with the tracing daemon actually collecting trace data. The remaining lines are different measurements from the kernel configuration with the CPU reservation scheduling class compiled into the kernel. For the first line, labelled `reservations` BE[16], none of the active benchmark processes are using CPU reservations, thus this line represents the overhead introduced to normal processes. For the second and third run,

[15]The modifications are courtesy of the "Plug in Scheduler Policies for Linux" project:
`http://resourcemanagement.unixsolutions.hp.com/WaRM/schedpolicy.html`.

[16]Short for Best Effort.



(a) Intel Pentium 200MHz (b) Alpha PC164LX 533 MHz (c) AMD Athlon 1.3GHz

Figure 5.9: Scheduling overhead for different number of concurrent processes

all benchmark processes are placed into the CPU reservation scheduling class with total guarantees amounting to 50% and 99.9% respectively. The graph for the Athlon processor included two additional lines which repeat the best effort and the 99.9% runs with a different scheduling epoch and preemption interval (100ms epoch with 1ms preemption interval) and measure the additional overhead for finer-grain multiplexing of the CPU.

From figure 5.9 two main observations can be made: First, the overall overhead introduced by the modifications is not significant (<1%). Second, the single biggest contributer to the overhead is the increase in periodic timer interrupt frequency, necessary to achieve higher timer resolutions. This overhead does not occur on the AXP PC164LX as FreeBSD on Alpha platforms already uses a periodic timer interrupt of 1024Hz by default. In more detail, a number of other observations can be made. First, one can also conclude that the overhead introduced by the tracing facility is insignificant. In fact, for the P200, the kernel configurations with the tracing facility enabled perform better than the kernel configuration with only the increased HZ value. A similar phenomenom can be observed on the Alpha platform[17]. Secondly, when using the CPU reservation scheduling class, an overhead, albeit small, is introduced for processes not using the new scheduling class. Thirdly, for processes in the CPU reservation scheduling class there appears to be an almost linear increase in the overhead with the number of processes. This overhead can probably be attributed to the linear traversal of the list of all processes in this

[17]This may be attributed to caching issues resulting from slightly different code layouts. On the Alpha platform this could be investigated further using Compaq's Iprobe tools, which provide a simple means of using the performance counters available on that processor (`http://www.support.compaq.com/alpha-tools/software/`).

class every second, in order to update their CPU reservations and move them from the wait queue to the run queue when appropriate. One possible optimisation would be to only update the allocations of processes already on the wait queue and update the information for other processes, i.e., those residing on the run queue and those being blocked, only when needed. However, it is not clear whether this only distributes the costs over a longer period or whether it would significantly reduce the overhead despite the significantly more complex implementation. In either case, I would argue that the linearity of the overhead is negligible for the number of processes expected to be in the CPU reservation class (< 100); especially when considering the small scale of the overhead on faster machines. Furthermore, there is no significant difference in overhead between the two experiments where the benchmark processes are given a reservation of 50% and 99.9%. This is not entirely unexpected as there are no active processes in the default scheduling class. This means that almost the entire remaining CPU cycles, which would preferentially be given to processes of the default scheduling class, are instead allocated to the processes on the wait queue, i.e., the processes which have exhausted their CPU reservations[18]. There does not seem to be a significant overhead involved. The final observation to be made is that although the higher context switch rate for the last two experiments on the Athlon does introduce an additional overhead, this overhead is relatively small compared to the overhead introduced with the higher timer frequency and is negligible for a smaller number of processes.

To summarise this section, the CPU reservation scheduling class introduced in the previous section is indeed capable of providing CPU guarantees to individual processes without allowing them to starve processes in the default scheduling class. Processes within the CPU reservation class are isolated from each other as well as from processes in the default scheduling class. This is achieved with little extra overhead. Indeed, most of the overhead can be attributed to the higher timer interrupt frequency necessary for better timer resolutions. The approximately linearly increasing overhead per processes in the CPU reservation class is negligible for the number of processes expected to be in that class.

5.2.4 Implementing congestion pricing

The CPU reservation scheduling class allows individual processes to request a share of CPU. This forms the basis for the implementation of decentralising resource management. The de-

[18]If there were active processes in the default scheduling class, the result for the experiment with 50% allocation would naturally be much worse. That, however, would defeat the purpose of the experiment of measuring overheads.

centralised resource management extends the CPU reservation scheduling class in a number of ways.

Similar to the scheduler simulator, the reservation scheduling class maintains utilisation and total resource request information. The utilisation information is updated every one second epoch, again using an EWMA filter. The gain parameter α of the filter can be controlled via the `sysctl(8)` interface. Prices, and therefore charges, are combinations of utilisation and requests, as discussed in section 4.2.1.2. The emphasis for request vs. usage charges, as controlled by the parameter β, can also be configured using the `sysctl(8)` interface. Usage charges are applied as part of the de-scheduling code of the reservation scheduling class, which also maintains per process usage statistics. Charges for resource requests are applied every epoch in the routine which renews the reservations for each process. An exponential function of the form $e^{a*(x-1)}$ is used as the pricing function, with $a = 10$ working well in practise. As with all arithmetic calculations required, values are scaled to a large integer range, as floating point arithmetic is not permitted in kernel space.

The structure representing a process in the reservation scheduling class is extended to implement the token bucket for credits as described in section 4.3. Using the sub-functions of the `eco_sched()` system call, a process can query the current number of credits in the account and, more importantly, the number of credits charged since the last call. This provides a process with a simple facility to determine the current charging rate. The credit allocation for a process can be set by the user using a new utility called `ecoprio(1)`, modelled after the standard FreeBSD utilities, `rtprio(1)` and `idprio(1)`, used to place processes in the real-time and idle scheduling class respectively. A routine executed at the end of each scheduling epoch places new credits in a process' credit token bucket.

As the default scheduling class is also a member of the reservation scheduling class it also has a credit account and is charged for its usage and resource requests. The system administrator can assign a credit allocation to the default scheduling class, thus guaranteeing it a share of the CPU for the processes it manages. Optionally, and also controlled by `sysctl(8)`, an adaption based on the WTP strategy is performed for the default scheduling class every epoch.

Figure 5.10 summarises the interface for processes actively participating in the CPU resource management. These are implemented as sub-functions to the `eco_sched()` system call. In

Function	Description
ECO_SET_SHARE	Request a CPU reservation
ECO_GET_SHARE	Query current CPU reservation
ECO_GET_ACCOUNT	Query current account
ECO_GET_DCHARGES	Query Δcharges since last call
ECO_GET_TOTAL_CRED	Query total amount of credits in circulation
ECO_GET_TOTAL_SHARE	Query total CPU reservations

Figure 5.10: `eco_sched()`: Resource management interface

addition, processes can query their current credit allocation rate using the `rtprio(2)` system call[19].

Figure 5.11 shows pseudo code an application may use for a periodic adaption. The code example implements an endless loop, which could, for example, be executed in a separate thread within a process[20]. The loop is executed every second and determines the current requested share of the CPU (line 20), queries the charges incurred since the last call (line 23), and calculates the time since the last call (lines 26-30). It then computes the Δx of the desired change using the WTP strategy (line 33). It finally requests an updated share of the CPU and goes to sleep for a second (lines 36-42). Naturally, an adaptive application can extend this skeleton code to perform application specific adaption. A more realistic example is given in section 5.2.5.1.

Non-adaptive applications can coexist with adaptive applications. When a user places an application into the reservation scheduling class and assigns a credit allocation to it, its initial share is set to a proportion equal to $(credit\ allocation)/(maximum\ charging\ rate)$. A non-adaptive application is guaranteed this share of the CPU as it can pay for the charges even at the highest price. Furthermore, if a user changes the credit allocation of a non-adaptive consumer, its share is also updated. The system distinguishes between adaptive and non-adaptive consumers by setting a flag in a process' state when it calls ECO_GET_DCHARGES. It is assumed that only adaptive consumers have an interest in their charging rate, thus processes with the flag cleared

[19]On FreeBSD this system call is used for lookup and change of priorities of processes which are not in the default scheduling class. As the credit allocation is the closest equivalent to priorities it was decided to use this system call. `rtprio(2)` already is a misnomer as the system call is also used by the idle scheduling class.

[20]Threads in FreeBSD are implemented in user-level and are multiplexed over a standard process. This is unlike, for example, the Linux implementation where threads are almost treated as independent processes. The process and thread model in FreeBSD is currently being changed radically to model a scheduler activation [ABLL92] based approach. [Eva00] provides a good overview of the current thread implementation, its limitations and the new design.

```
adapt() {
        int share, share_delta;
        int charges;
        struct timeval  q_time, new_q_time;
        struct rtprio   rtp;
        struct timespec rqtp;
        u_int64_t       t_delta;

        rqtp.tv_sec  = 1; /* 1s */
        rqtp.tv_nsec = 0;

        gettimeofday(&q_time, NULL);
        goto sleep;

        for (;;) {
                /* Check current credit allocation */
                res = rtprio(RTP_LOOKUP, 0, &rtp);
                if (rtp.type == RTP_PRIO_ECO) {
                        /* Get current share */
                        res = eco_sched(ECO_GET_SHARE,  &share);

                        /* Get charges since last call */
                        res = eco_sched(ECO_GET_DCHARGES,  &charges);

                        /* Get current time */
                        gettimeofday(&new_q_time, NULL);
                        t_delta = (u_int64_t)(new_q_time.tv_usec -
                                              q_time.tv_usec) +
                                (new_q_time.tv_sec - q_time.tv_sec) *
                                (int64_t)1000000;

                        /* Calculate change in share using WTP with kappa=0.1 */
                        share_delta = WTP(.1, rtp.prio, charges, t_delta)

                        /* Request new share */
                        share += share_delta;
                        res = eco_sched(ECO_SET_SHARE, &share);
                        q_time = new_q_time;
                }
        sleep:
                /* sleep for one second */
                nanosleep(&rqtp, NULL);
        }
}
```

Figure 5.11: Pseudo code for adaption

are assumed to be non-adaptive. This mechanism provides simple support for legacy applications which require a fixed share of a resource.

5.2.4.1 Discussion

The implementation of congestion prices under FreeBSD is only a proof of concept implementation. Apart from the relatively coarse grained control for resource requests and the difficulties of accounting for resource consumption, as discussed in section 5.2.1.1, the congestion pricing mechanism currently has no concept of different users and requires super-user privileges to assign credits to resource consumers. Furthermore, the implementation does not address process relationships well. Child processes are simply placed back into the default scheduling class.

To address these issues a more comprehensive API is required. Fortunately, previous research provides some example implementations. For example, the "top half" of lottery scheduling , i.e, tickets and currencies (section 2.1.4.3 and [WW94]) could be directly applied to manage credit allocations amongst users and processes. A file based namespace, similar to Eclipse's `/reserv` filesystem (section 2.3.4 and [BBG+99c]), would offer an alternative API for managing credits and would allow resources consumers to make resource requests. Furthermore, it could easily be extended to provide support for multiple resources.

It has to be stressed that these issues are orthogonal to the actual congestion pricing mechanism described in this section. For the implementation of the pricing mechanism it is merely necessary to identify distinct consumers of resources, account resource consumption to them, and to be able to identify resource congestion. However, for a deployment in a general-purpose operating system, abstractions similar to the two described above would need to be provided.

5.2.5 Example application: `mplayer`

The previous sections described the implementation of the decentralised resource management architecture over FreeBSD and the interface provided to user-level applications. To evaluate the suitability and the effort required to modify applications to make use of this new facility a small number of applications where adapted. The most substantial one was a multi-media decoder and player called `mplayer`[21].

`mplayer` is a modular application which can handle several different media formats and decoders and is primarily written for Linux but also compiles and runs under FreeBSD. Existing decoders are imported into the main source code distribution and small wrappers are provided

[21] `http://www.mplayerhq.hu`

to access them through a common API. These decoders include various MPEG2, MPEG4 and DivX decoders. `mplayer` also includes a number of output modules both for audio and video and a general framework for applying filters to them before playback. Examples for video filters are horizontal and vertical de-blocking filters and noise filters. They are typically used to reduced the effect of artifacts introduced by the video encoding scheme being used.

Video playback in `mplayer` is driven by the audio playback — displaying of video frames is synchronised with audio playback to prevent unpleasant effects such as, dialogues being out of sync with lip movements. This is achieved by using timestamps embedded in combined audio/video streams in combination with the audio card as the main timing source. After a video frame has been decoded its timestamp is compared with the timestamp of the current audio sample. If the video timestamp is later then the audio timestamp `mplayer` sleeps for the appropriate amount of time. If the video frame is late it is displayed immediately.

`mplayer` performs some adaption depending on the time difference, or *skew*, between video frames and audio samples. For example, it may drop the decoding of entire frames if video decoding lags behind by a larger margin. Furthermore, if post-processing filters are enabled `mplayer` can be configured to increase or decrease the number of filters applied to a video frame depending on whether the previous frame was too late or too early. In the default configuration `mplayer` uses six distinct levels of post-processing filters and uses increasingly more CPU resources to improve image quality. It is worth pointing out that the image quality improvements depend on the video encoding used and, while the improvements are noticeable in certain scenes, the difference between different filters is small enough to allow for a frame by frame change of filters without irritating the viewer.

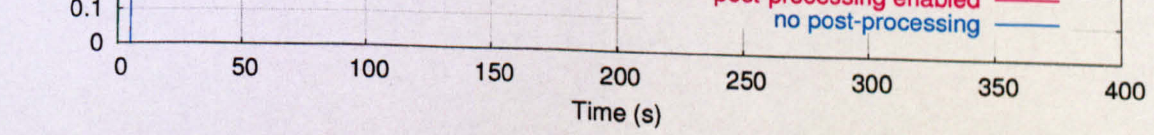


Figure 5.12: CPU consumption for decoding a video sequence with different filters

Figure 5.12 shows the resource demand, averaged over one second intervals, for decoding and displaying a DivX encoded video sequence on an otherwise idle AMD Athlon 1.3GHz system with different, fixed levels of post-processing enabled. The bottom line represents the baseline resource usage of the video decoder without any post-processing. For the next line post-processing was enabled but no filters were activated. The slightly higher CPU usage can be attributed to per-frame overheads such as an additional memory copy introduced by the post-processing module. The top three lines show the CPU demand for three different levels (1, 3, and 6) of post-processing filters enabled. On this system the resource requirement for the different post-processing filters covers a range of about 20% of the available CPU time, leaving room for dynamic adaption, trading off resource consumption against image quality[22]. It is worth pointing out that the absolute numbers for CPU resources consumed, as shown in the graph, are somewhat misleading and it has been observed that the same quality levels can be achieved with less CPU resources used. The reason for this phenomenon can be found in the stream synchronisation code: `mplayer` occasionally "busy-waits" instead of sleeping if the time difference between the current video frame timestamp and the current audio sample timestamp is relatively small. Furthermore, a certain amount of buffering of decoded video frames may also help for smoothing out resource demand over multiple frames making the overall resource demand somewhat more elastic (I observed a similar effect using a modified MJPEG player described in [Neu99]).

Figure 5.12 also indicates that `mplayer` is a non-elastic resource consumer with a non-convex utility curve. The application has a relatively high initial resource demand of 45% to 50% of the CPU for the chosen video clip on the particular system used. If the `mplayer` process receives less CPU resources the video playback will become unwatchable to a user. However, for resource allocations above this minimum allocation its utility curve can be assumed convex. This application therefore falls into the second category of applications with non-convex utility curves as identified in section 4.4.1. For the evaluation, presented in section 6.3, it is ensured that the `mplayer` application has sufficient resource available to operate in the convex region of its utility curve.

[22]Further reduction in CPU resource demands could be achieved by modifying the individual decoders, e.g., as described for a simple MPEG decoder in section 5.1.5.1. However, the MPEG4 decoders available in `mplayer` are significantly more complex and more optimised so this would have required a significant additional effort.

5.2.5.1 Introducing congestion pricing

To evaluate the benefits of congestion pricing based feedback I modified the standard `mplayer` source code to use the interface provided by the FreeBSD prototype. These changes where made to the main loop in `mplayer`. Within this loop the following actions are performed continuously until the end of a file is reached:

1. Decode audio sample and place it in play buffer if not empty.
2. Decode a video frame.
3. Deal with GUI events (if enabled).
4. Determine when to display the frame, then sleep, if necessary, and display the frame.
5. Calculate the timestamps for audio and video and the delay between them.
6. Adjust the post-processing levels based on the time spent sleeping.
7. Deal with keyboard events.

To take feedback signals from the kernel into account only step 6, the adjustment of post-processing filters, has been replaced. Instead of adjusting the output quality only based on the time spent sleeping, the modifications also determine the current charging rate and decide whether to increment or decrement the resource demand along with the output quality. This adaption is not performed for every frame as the sample interval would be too short to reliably determine charging rates. Instead, the current prototype performs the adaption step for resource requests every ten frames and sets the maximum allowed post-processing level for the next ten frames. For intermediate frames the post-processing level is set between this maximum level and zero, based on the sleep time as before.

Figure 5.13 shows the pseudo code for the changes made. Lines 2–10 essentially perform the WTP algorithm as the example given in section 5.2.4. However, the requested share is not directly adjusted. Instead, in line 14–17, it is decided if the current share is sufficient by checking if the application slept during the past ten frames. If that is the case the share is not increased. Next, the maximum level of post-processing is decided based on the ratio of current charging rate to credit allocation rate. Three distinct cases are distinguished. If the charges are higher than the current credit allocation, the maximum level of post-processing is reduced (lines 19–25). More specifically, if the post-processing level is at the maximum, it is reduced by two levels and if it is already at the minimum a frame is dropped. Otherwise, the level is only reduced

```
every 10 frames do {
    /* Get the data */
    rtprio(RTP_LOOKUP, 0, &rtp);
    eco_sched(ECO_GET_SHARE, &share))
    eco_sched(ECO_GET_DCHARGES, &charges)
    t_delta=time difference since last call;
    charging_rate   = charges/t_delta;

    /* used WTP algorithm */
    wtp_share_delta = wtp(kappa, rtp.prio, charging_rate);

    c_ratio     = charges/(float)rtp.prio;

    /* decide what to do */
    if (wtp_share_d > 0) /* we can afford a bigger share */
        if (eq_sleep_time > 2 * 1/frames_per_second)
            wtp_share_d = 0; /* don't increase share */

    if (c_ratio > 1) {/* charges > than credid allocation */
        if (max_quality == eco_q_max)
            max_quality -= 2;
         else if (max_quality >= 1)
            max_quality--;
         else
            drop_frame = 2;

    } else if (c_ratio <= .95)  {/* charges << than credit alloc*/
         if (max_quality < eco_q_max)
             max_quality ++;

    } else { /* charges within 5% of credit allocation */
         if (max_quality == eco_q_max)
             max_quality--;
    }

    share += wtp_share_d;         /* adjust share  */
    res = eco_sched(ECO_SET_SHARE, &share);

    output_quality = max_quality; /* set new output quality */

} else {
    /* for intermediate frames we still do adaption  */
    if (output_quality<max_quality && aq_sleep_time>0)
        ++output_quality;
    else
        if (output_quality>1 && aq_sleep_time<0)
            --output_quality;
    else if (output_quality>0 && aq_sleep_time<-0.050f) // 50ms
            output_quality=0;
}
set_video_quality(sh_video,output_quality);
```

Figure 5.13: Pseudo code for `mplayer` adaption

by one. If the charges are significantly lower than the credit allocation the post-processing level is raised (lines 27–29). If the charges are within 5% of the credit allocation and the level is at the maximum the post-processing level is reduced as a precaution (lines 31–34). In all other cases the output quality remains the same. Lines 36–39 simply request a new share of the CPU and set the output quality to the new maximum level. Lines 41–51 perform the adaption for intermediate frames as the unmodified `mplayer` does.

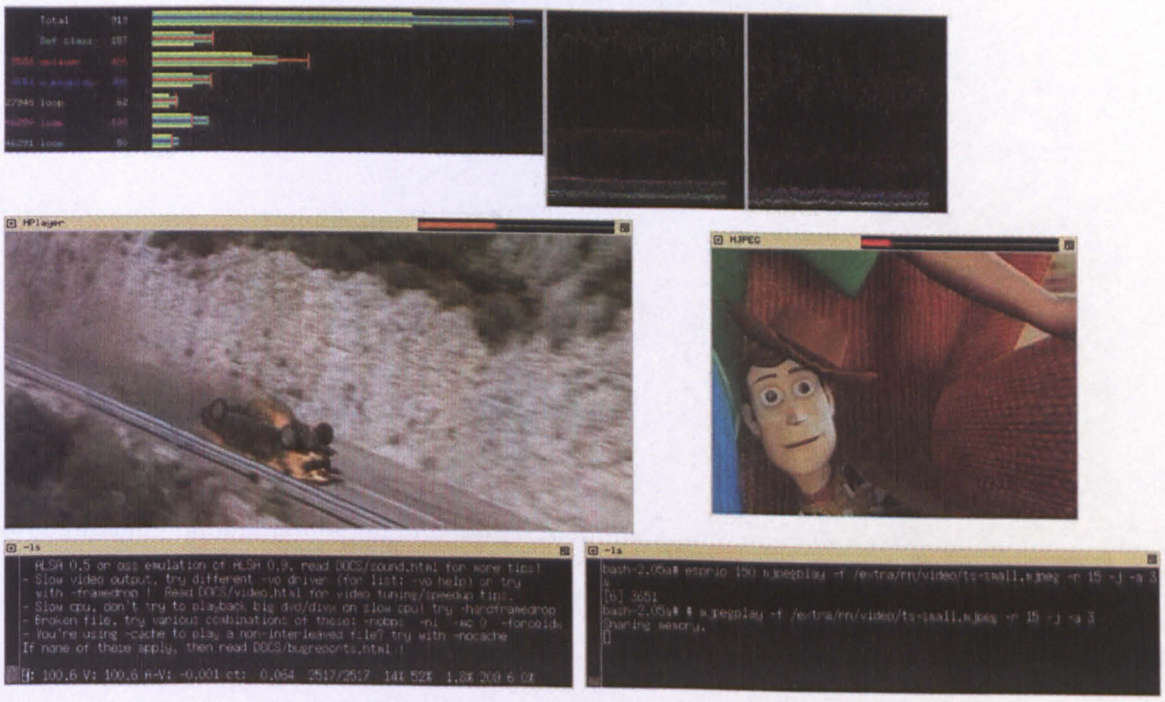
This adaption strategy is fairly simple. It performs the WTP algorithm with some adjustments to the output quality of the application. It does not implement any hysteresis when adjusting the level of post-processing as the resulting image quality changes are relatively small. One could imagine more complex adaption strategies, for example, strategies which utilise statistics about per-frame decoding times or which take the size of change in requested share into account when adjusting the output quality. However, it is worth pointing out that even this simple strategy performs adaption specific to an application — it proactively adjusts the maximum output quality based on the feedback provided — a task difficult to achieve with any centralised resource management approach.

`mplayer` has also been modified to write per frame information, such as quality level and audio/video timestamps to a log file. This information is used for the evaluation of different adaption strategies and has also been useful for debugging purposes.

5.2.6 User Interface

The description has so far focused on the implementation of congestion pricing mechanisms within the FreeBSD kernel and a description of the API presented to the application developer. The prototype also includes a set of utilities for end-users, both command-line utilities and graphical user interfaces.

As described above, a user can add or remove applications to and from the new scheduling class using the `ecoprio(1)` command. This command is modelled after the standard FreeBSD commands `rtprio(1)` and `idprio(1)`, used to change the scheduling class of processes to the real-time class and the idle class respectively. `ecoprio(1)` can also be used to adjust the credit allocation of a running process. A new command was introduced, `eco_ps(1)`, which, analogous to the standard `ps(1)` utility, displays information about processes currently in the new scheduling class. This information includes a process' credit allocation, current charging rate and resource current consumption.



```
 ALSA 0.5 or oss emulation of ALSA 0.9, read DOCS/sound.html for more tips!
- Slow video output, try different -vo driver (for list: -vo help) or try
  with -framedrop !  Read DOCS/video.html for video tuning/speedup tips.
- Slow cpu, don't try to playback big dvd/divx on slow cpu! try -hardframedrop
- Broken file, try various combinations of these: -nobps  -ni  -mc 0  -forceidx
- You're using -cache to play a non-interleaved file? try with -nocache
If none of these apply, then read DOCS/bugreports.html !

A: 100.6 V: 100.6 A-V: -0.001 ct:  0.064  2517/2517  14% 52%  1.8% 208 6 0%
```

```
bash-2.05a# esprio 150 mjpegplay -f /extra/rn/video/ts-small.mjpeg -r 15 -j -a 3
&
[6] 3651
bash-2.05a# # mjpegplay -f /extra/rn/video/ts-small.mjpeg -r 15 -j -a 3
Sharing memory.
```

Figure 5.14: Graphical User Interfaces and sample applications

In addition to these text-based utilities two graphical utilities are provided: a standalone application, called eco_bars and a modified window manager[23]. Both tools are shown in figure 5.14.

The eco_bars application consists of the three windows, located at the top of the screenshot. The leftmost window shows a horizontal bar for the default scheduling class and for each process executing in the new scheduling class. Each process' bar is composed of three different bars. The outermost bar (in yellow) shows the current charging rate for that process. The middle bar (in green) shows the rate at which the process consumes the CPU resource and the innermost bar indicates the requested share for that process. The colour of the innermost bar is either red, indicating that the process performs adaption, or blue, indicating that the process is non-adaptive. These horizontal bars provide visual feedback to a user on the current state of the system. The small red vertical line indicates the current credit allocation for each process. A user can move this line using the mouse to change the credit allocation for each application. The

[23]Both tools are modelled after similar user interfaces available under Nemesis. The equivalent to eco_bars, known as qosbars, was widely used with Nemesis, however, the window manager integration was only demonstrated at the final Pegasus II workshop (Cambridge, UK, November 1999), and has not, unfortunately, been described in detail anywhere.

horizontal bars then provide the user with visual feedback on how resource allocations change. The other two windows of the `eco_bars` application provide an animated history of resource consumption and charging rates for each of the processes.

The second user interface component is embedded in the window manager. The effect is visible within the window-manager decoration for two sample applications in the middle of the screenshot: `mplayer` and an motion JPEG player. The title bar for these two applications contains a black area with a red bar indicating the credit allocation to these applications. Again the user can adjust the credit allocation using the mouse. Furthermore, if the user moves the mouse pointer into one of these windows, the credit allocation of the corresponding application is automatically boosted by a fixed percentage. This is an example of a simple user policy implemented entirely in user-space. Naturally, more complex policies can be implemented, e.g., through configuration files specifying application-specific policies for credit allocations.

The window manager based user interface was implemented by modifying an existing window manager, `vtwm`. It required only minor modifications to the main window manager code, essentially adding hooks to selected parts of the window drawing and event handling code. The core functionality is implemented in a separate file in less than 300 hundred lines of code. Since in the X-Window system the window manager has no detailed knowledge about the processes to which the windows belong — in particular the window manager has no knowledge of the process ids — an application wishing to be managed in the fashion outlined above has to communicate its identity to the window manager. In the current prototype this is accomplished using the standard X-Window feature of window properties: an application stores its process id under a well known name, `ECOPID`, for one of its windows. This requires the addition of 5 lines of code to the window initialisation code and could be provided as a standard library call. If a window does not have this property then it is handled by the window manager without modifications to the decoration.

5.2.7 Summary

In this section a prototype implementation of the congestion pricing mechanism for CPU resource in FreeBSD has been described. The three main components of this implementation are a new scheduling class providing processes with the ability to make absolute requests for CPU resources, a request-usage based pricing mechanism allowing flexible run-time configuration, and a token-bucket based credit account mechanism. Furthermore, the kernel has been instrumented to provide detailed tracing information about scheduling decisions for off-line analysis.

The implementation required around 70 lines of kernel code to be changed. The single biggest change was required in the `chooseproc()` function in order to add the reservation scheduling class to the other scheduling classes. Most other changes were required to instrument the kernel with the tracing code. The reservation scheduling class itself, inclusive of the code for the pricing and accounting mechanism, is implemented in around 900 lines of heavily commented code, while the tracing facility requires around 350 lines of kernel code.

The implementation serves mainly as a proof-of-concept prototype and its limitations and their possible solutions have been addressed. In summary, CPU reservations are only provided to processes — there is no separate abstraction of a resource principal and resources may be consumed without being accounted for, e.g., due to interrupt processing — and the round-robin scheduling of processes is not sufficient to provide timeliness guarantees. Subsequently, the congestion pricing mechanism is only applied within the constraints of the implementation of the reservation scheduling class. It has been argued that these issues can be addressed by proposals to introduce more flexible resource management abstractions into FreeBSD (reviewed in section 2.3) and that the application of congestion pricing to operating system resources is orthogonal to these approaches.

Despite its limitations, the prototype has two important features. Firstly, it provides a mechanism to prevent processes in the default scheduling class from being starved by processes with reservations in the higher priority scheduling class. This is achieved by providing a configurable reservation to the default scheduling class. Secondly, and more importantly, the use of a separate scheduling class allows *only* those processes that can benefit from being more actively involved in the management of their resource demands, to be exposed to the feedback — other applications are simply allotted resources by the default scheduler as in an unmodified system. This is arguably a better approach than that of requiring all applications to participate and also having to provide default strategies for those applications that have not been instrumented to adjust their resource demands.

This section has been rounded off with a description of how applications can make use of these new features by describing the modifications made to an existing multimedia video player and with a presentation of a number of tools and user interfaces enabling end-users to use of the system more easily.

Chapter 6
Evaluation

In the previous two chapters a general discourse on how the concept of congestion pricing can be applied in the context of operating systems has been given (chapter 4) and two prototype implementation have been described in detail (chapter 5). In this chapter a detailed evaluation of these prototypes is provided. In section 6.1 the simulator is used to evaluate the effect of the different design options discussed in chapter 4 while section 6.2 presents results from the FreeBSD prototype. In section 6.3 the benefits the decentralised resource management architecture offers to applications is investigated.

In section 6.4 my general experience with the decentralised system is presented and sections 6.5 and 6.6 discuss what impact this system has on application developers and users respectively. Section 6.7 provides a summary of this chapter.

6.1 Simulation results

The simulator is a convenient tool to evaluate the different design options, such as different pricing mechanisms and adaption strategies, discussed in sections 4.2.1 and 4.4. Initially, only elastic consumers are used, i.e., consumers which can consume an arbitrary amount of a resource albeit with a diminishing marginal utility. Using these consumers, first, different pricing mechanisms are compared in section 6.1.1, then, in section 6.1.2, different consumer adaption strategies are compared under changing task sets.

The two sections that follow section 6.1.2 then investigate non-elastic consumers. First, in section 6.1.3 non-elastic, bursty applications are used, while in section 6.1.4 consumers are used which attempt to "play" the system.

6.1.1 Different pricing mechanisms

In this section the impact of different pricing mechanisms, discussed in section 4.2.1, are evaluated. For this purpose, a task set of ten elastic consumers is used, each is assigned a different credit allocation by the user (in multiples of $150\ credits/s$, with a total sum of $8250\ credits/s$). All ten consumers adapt their resource demand every $100ms$ according to the WTP strategy, described in section 4.4, with a convergence constant of $\kappa = 0.1$. All ten consumers use a period of $10ms$ and make the same initial request of a slice of $100\mu s$.

In section 4.2.1 two different approaches for determining shadow prices for CPU resources were discussed: consumers are charged after congestion has occurred; and prices are based on the probability of congestion. These two approaches are compared in the first experiment. The results are shown as a time-line in figure 6.1. On the left hand side a pricing mechanism is used which charges all tasks, in proportion to their usage, after a deadline has been missed until the CPU is next idle (as shown in figure 4.2). On the right hand side the pricing mechanism uses a probability function of the form $p(y) = e^{a(y-b)}$ with $a = 15$ and $b = 0.95$ with the probability kept between $0.0 <= p(y) <= 1.0$ (as in the second graph in figure 4.3). The factor a determines the slope of the function and the factor b ensures that the maximum charging probability (1.0) is reached at the target utilisation of 95%.

For each of the pricing mechanisms, figure 6.1 shows four different aspects: (1) the top graph shows the service rate achieved by each of the ten consumers (the service rate is averaged over $100ms$); (2) the second graph shows the amount of resource requested by each of the consumers; (3) the third graph summarises the top two graphs by showing the total utilisation and the sum of requests for the experiment; (4) the bottom graph shows the charging rate for each of the ten consumers.

A number of key observations can be made from these graphs. First, and most importantly, the service rate graphs demonstrate that, after an initial settling period, with both pricing mechanisms, the consumers receive a service rate proportional to their credit allocations. This is important because, although each consumer is attempting to maximise its utility *independently*, given its credit allocation, the overall system still provides a socially optimal allocation of the CPU. Furthermore, the credit allocation provides a meaningful measure for service differentiation. If a user assigns twice as many credits to one consumer compared to another consumer, the user can expect the first consumer to receive twice as many resources.

The second important observation is that all consumers manage to control their charging rate close to the rate at which credits are allocated to them. Thus, the WTP strategy seems to work well in this environment.

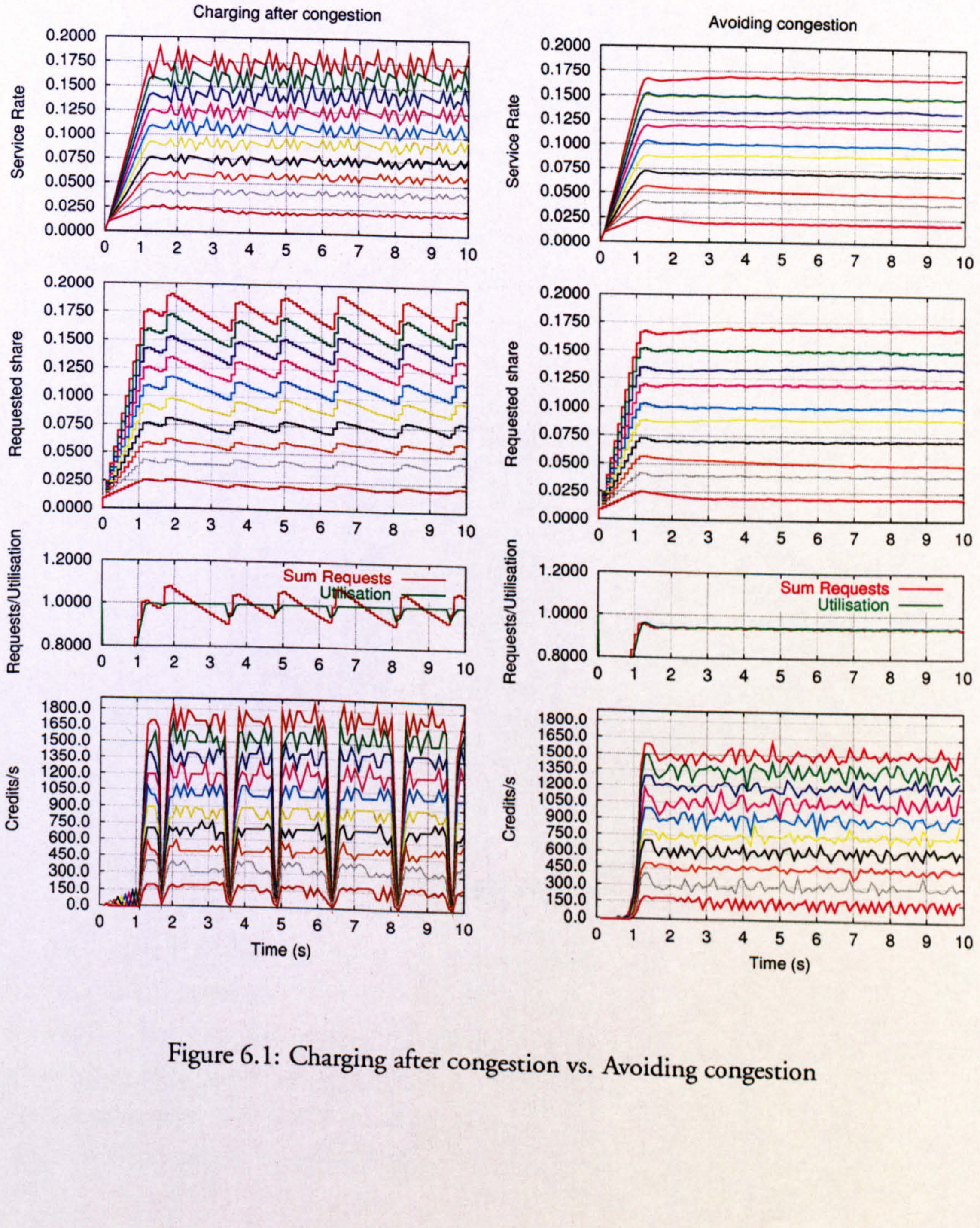


Figure 6.1: Charging after congestion vs. Avoiding congestion

Comparing the two pricing mechanisms, it is apparent that the mechanism using prices based on the probability of congestion provides much "smoother" results. This is not unexpected, as it provides some feedback signals to the consumers before the resource becomes congested. The pricing scheme, which charges after congestion occurs, does not provide any feedback to the consumers until a deadline is missed, and then consumers observe charges for every minimum time unit they consume. The result is that the consumers periodically drive the system into overload, then observe charges and reduce their resource demand. This can be observed in the oscillations of the requested share and the graph showing the resource utilisation and sum of resource requests made by all consumers. In contrast, the pricing mechanism based on the probability of congestion does not exhibit this behaviour and the system settles with a steady state after an initial period. Furthermore, due to the offset of the pricing function, it achieves a target utilisation of around 95%, and, since the system is not in overload, all resource requests are satisfied.

Consumer	Charging after Congestion			Avoiding Congestion		
w_i	Service time	Charges	Missed Ddlns	Service time	Charges	Missed Ddlns
150	0.213s	1414	556(55%)	0.202s	1397	0(0%)
300	0.389s	2714	564(56%)	0.381s	2680	0(0%)
450	0.551s	4046	576(57%)	0.497s	4122	0(0%)
600	0.706s	5366	590(59%)	0.667s	5414	0(0%)
750	0.865s	6695	606(60%)	0.844s	6726	0(0%)
900	1.016s	8015	622(62%)	0.953s	8139	0(0%)
1050	1.178s	9325	647(64%)	1.133s	9418	0(0%)
1200	1.336s	10661	673(67%)	1.275s	10769	0(0%)
1350	1.497s	11982	719(72%)	1.420s	12123	0(0%)
1500	1.655s	13291	776(77%)	1.596s	13422	0(0%)

Figure 6.2: Summary of experiments

The results from this experiment are further summarised in table 6.2 where rows contain statistics about consumers, i.e., the overall service time received, the total charges incurred, and the number and percentage of missed deadlines. As one would expect, the pricing mechanism which charges consumers after congestion occurred, results in a significant number of missed deadlines (between 55% and 77% of deadlines) while with the second pricing mechanism no deadlines were missed. One can also see that the service time and total charges are roughly proportional to the credit allocation. The inaccuracy of proportionality can be attributed to both the fluctuation of service rates and charges, and the initial settlement period.

In the this experiment it takes over a second, or more than ten iterations, until the stable state is reached. In section 4.2.1.1 a pricing function providing "negative charges" is proposed to give consumers an indication of when a resource is only lightly loaded. Figure 6.3 shows the result of an experiment with such a pricing function and the same task set as in the previous experiment. "Negative charges" are based on a linear charging probability function ranging from -1.0 for a utilisation of 0 and no charges for a utilisation of 0.75. Above this utilisation, the same exponential probability function is used as in the previous experiment.

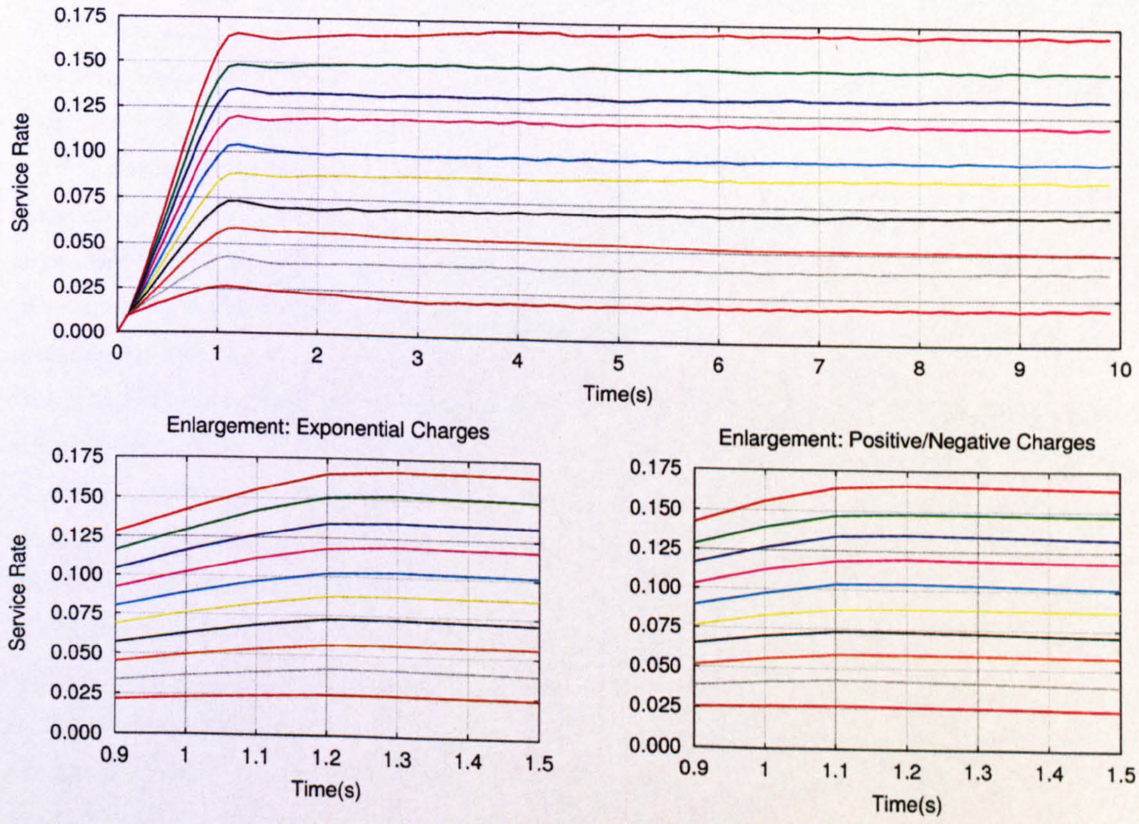


Figure 6.3: Exponential prices vs. Positive/Negative Charges

Not surprisingly, the results for the steady state are comparable to the pricing scheme using the exponential function in the previous experiment, as, for a higher utilisation, the same function is used. However, as can be seen from the enlargement at the bottom of figure 6.3, with this pricing mechanism, the system reaches the steady state slightly earlier.

In summary, for homogeneous task sets, as used for the experiments in this section, the decentralised approach yields results comparable to a proportional weighted fair scheduler *although* the individual consumers are acting independently of each other. For general system stability it appears beneficial to provide some early feedback to applications before congestion occurs while only a slight improvement is noticeable if the system also provides feedback if the resource is idle.

6.1.2 Different strategies and changing workloads

In the previous experiments a static task set with all consumers using the same adaption strategy (WTP) was used. In this section, two different aspects are evaluated: firstly, different strategies are used, and, secondly, the task set is changed over time, to evaluate how the decentralised system responds under changing conditions.

For this evaluation, again, a task set of ten elastic consumers is used, however, only five of them are using the WTP strategy as in the previous experiment. The remaining five consumers are using a PID controller, as described in section 4.4, to adjust their resource demand in accordance to the charges they observe. The tasks have different credit allocations assigned (in multiples of 250 $credits/s$). The tasks also use different periods, ranging from $5ms$ to $20ms$, and have different initial resource allocations. During the experiment, the task set is changed every 50 seconds. A new task joins the set at time $50s$, the user then decides that certain consumers are more or less important and changes their credit allocation rate at time $100s$ and $150s$ respectively. Finally, a consumer leaves the task set at time $200s$. For this experiment, the same simple exponential price function is used, as in the initial experiment.

Figure 6.4 shows the results of this experiment. The top graph shows the service rates achieved by each consumer over time. The service rates are batch-averaged over $100ms$ as in the previous experiments. The second graph shows the rate at which the consumers incur charges in $credits/s$. The final graph shows the current price for resource usage (in red) and the utilisation of the CPU (in green), as measured by the pricing mechanisms (using the EWMA filter described in section 5.1.9). The price graph is so highly variable as the exponential price function amplifies the slight changes in system utilisation. The graph also contains the sum of all resource requests (in blue).

There are two key observations to be made from these results. Firstly, different (sensible) strategies can be used for elastic consumers and the decentralised system still provides good weighted proportionally fair resource allocation. Both WTP and PID seem rational strategies

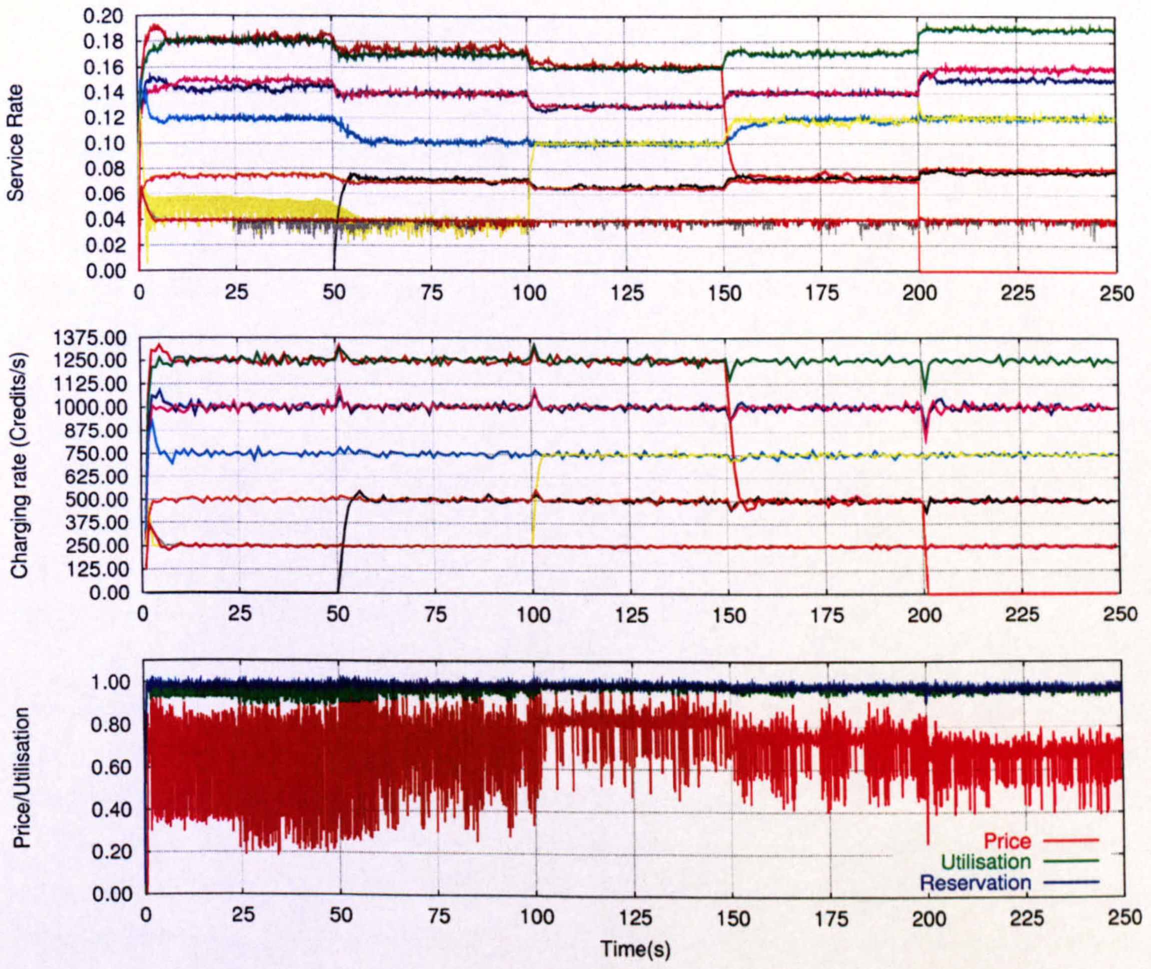


Figure 6.4: Change of preferences and task set. Different strategies

to use for elastic users as they both manage to control the charging rate well (as can be seen from the second graph). Secondly, the overall system reacts quite quickly to changes in the task set, despite its consumers acting independently of each other. Thus, the weighted proportional fairness is maintained. This is important, as it allows a user to dynamically change the credit allocations of his consumers should the user's preference for consumers change. However, as with the initial settling period in the previous experiments, it may take a number of iterations of the adaption for the system to settle to a new stable state.

6.1.2.1 Effects of different adaption parameters

The previous experiments demonstrated that consumers can choose their adaption strategy. However, consumers may also choose the various parameters, e.g., the intervals in which they adapt, that are used by their adaption strategy. In this section, the impact of these parameters is analysed. This analysis focuses on the WTP strategy as it is much simpler, and as demonstrated in the previous experiment, is similarly effective as the more complex PID controller. In fact, the PID controller requires five parameters to be specified and, in practise, it is quite difficult to choose the right parameters to achieve rapid convergence while preventing extreme reactions or heavy oscillations.

A WTP consumer essentially requires two parameters to be specified: an interval, over which to perform adaption and the convergence constant κ. The constant κ influences by how much the new request is adjusted given an observed charging rate and credit allocation. The adaption interval determines how quickly a consumer reacts to the feedback signal.

For an initial analysis, the first experiment from section 6.1.1, using, again, the exponential price function is repeated multiple times with different values of κ. Figure 6.5 summarises the results. The graph shows the average service rate (with min/max values as error-bars) achieved by each of the consumers in the steady state[1] for different values for κ.

The graph shows that the *average* service rate is largely independent of the choice of κ — on average the consumers receive a service rate that is proportional to their credit allocations. However, for larger values of κ they observe much larger variations of services rates as indicated by the min/max values.

Next, the effects of the convergence constant κ and the adaption interval are investigated in more detail. Combinations of these two parameters may have an impact on how quickly a consumer adapts to changing conditions or on how variable a consumer's resource requests are when the task set is stable. To illustrate this impact, a series of simulations with systematically adjusted values of κ and the adaption interval have been executed. As in the previous experiments, a task set of 10 elastic consumers with different, evenly distributed, credit allocations is used. These experiments are summarised using three different measures: the percentage of missed deadlines, the time it takes for the consumer with the highest credit allocations to reach the stable state[2],

[1]The first 10 seconds of each 20 second-long experiment are discarded. With the smallest value of κ (0.01) the steady state is reached after about 8 seconds.

[2]The length of the initial transient interval is calculated with a variant of the initial data deletion method using relative changes, as described in [Jai91, pp 424–426]. First, the overall mean of the service rate is calculated. Then, the mean is calculated without the first, second, third, . . . , observation. For each of these means, the relative change

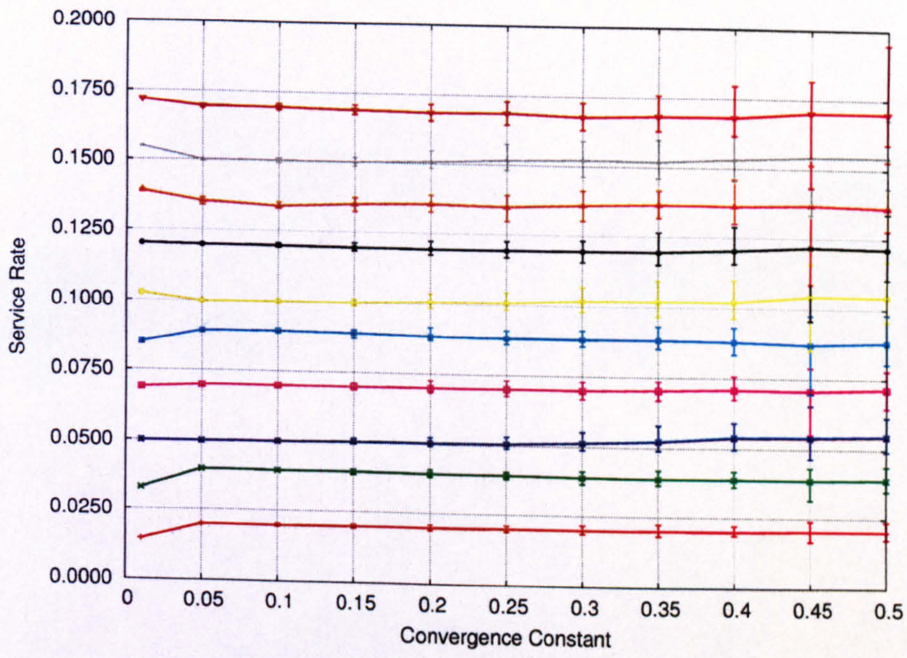


Figure 6.5: Impact of κ on service rate: Mean and Max/Min values in stable state

and the variance for this task for the first 20 seconds after reaching the stable state. The results are shown in figure 6.6. Note, however, that the absolute values given in these graphs also depend on the behaviour of the other consumers. The graphs, therefore, only present a qualitative illustration rather than a quantitative evaluation.

The top-left panel shows the percentage of missed deadlines for all consumers. Only for larger values of κ are deadlines missed, especially for short adaption intervals. This is not surprising, as for larger values of κ the individual consumers may overreact to the feedback signals and drive the system into overload. This effect will be more pronounced if the consumers "overreact" more often, i.e., for shorter adaption intervals.

The top-right panel shows the variance (σ^2) of the service rate once the stable state is reached. The results are what is to be expected. The variance is highest when the consumers adjust the resource requests more often and by a larger amount (larger κ). The variance is lowest when the resource requests are changed only slightly in large intervals.

However, as the bottom panel illustrates, there is a tradeoff between the variability a consumer may experience in the stable state and the rate at which it can adjust to changes. The bottom panel shows the time it takes for one consumer to reach the stable state allocation with a

to the overall mean is calculated. If the difference of relative changes for subsequent means approaches zero, the "knee" of the graph can be identified.

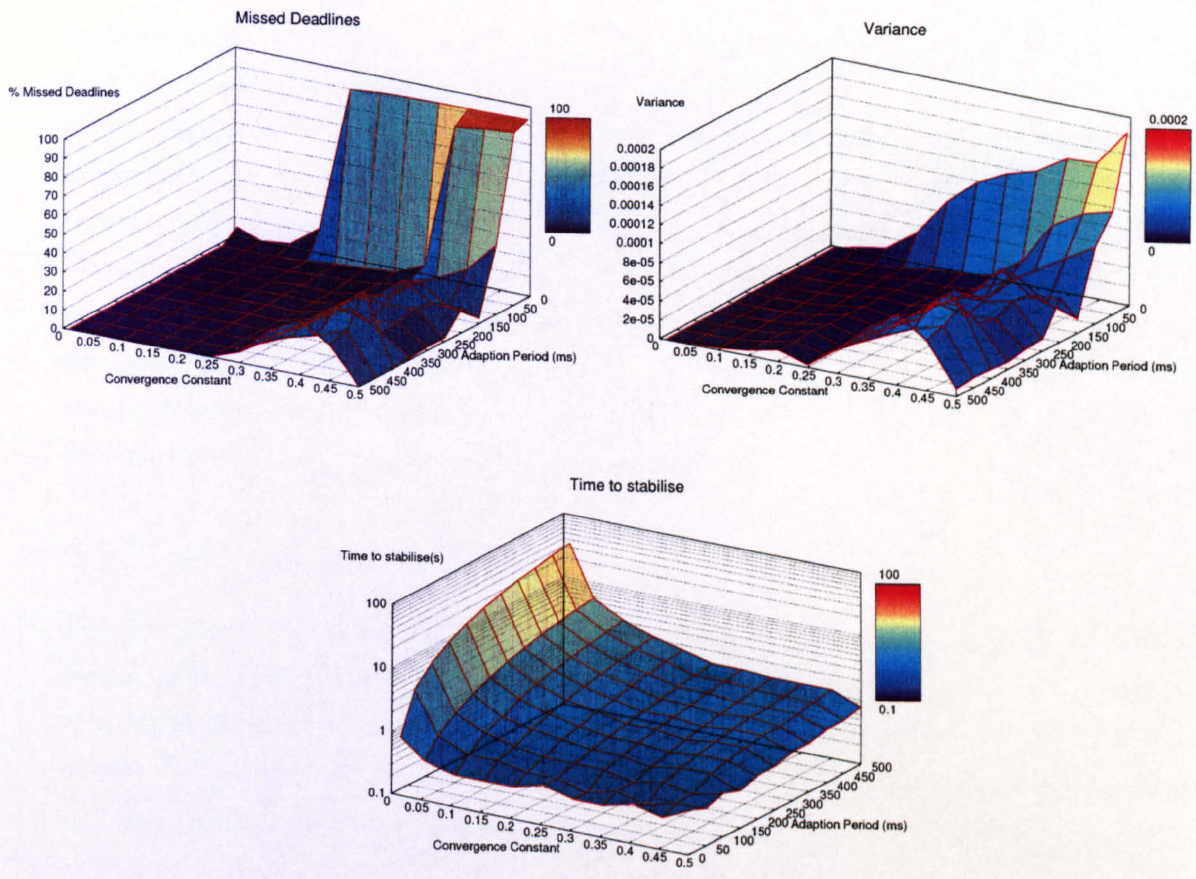


Figure 6.6: Impact of κ and adaption period

static task set. Note that, for better visibility, the axis, showing the adaption interval, is reversed and the z-axis is in a log-scale. Again, the results are what is to be expected. A consumer which adjusts its requests by only a small amount (small κ) in large time intervals, will take a longer time to reach the stable state. Conversely, a consumer adjusting the resource requests more often, and by a larger amount, will reach the stable state earlier.

The drop in missed deadlines and in variance for $\kappa = 0.5$ and long adaption intervals ($< 200ms$) is counterintuitive. The variance should increase with larger values of κ and not decrease as the top right-hand panel of figure 6.6 shows. Despite an inspection of the data it is not entirely clear why this is the case. It could possibly be attributed to interactions between different consumers which is triggered by this particular combination of parameters. Another explanation is a possible interaction of the trace analysis tools, for example, a correlation of the parameters used for the experiments and the way service rates and, subsequently the knee and variance, are calculated. This issue requires further investigation.

To summarise, for large values of κ and, especially, for short adaption intervals, the system might be driven into overload, and the steady state allocation is highly variable. But, by choosing such a combination, the consumer will be able to react more quickly to changes in the task set. Conversely, for small values of κ, the service rate will be less variable in the steady state. For small values of κ, the impact of the adaption interval is not as significant as for larger values. Individual consumers therefore have a wide range of "sensible" values to choose from. It is also conceivable that a consumer may adjust their convergence "constant" over time to reflect different modes of operation or changes in the overall dynamic of the task set.

6.1.3 Non-elastic applications

The consumers used in the experiments so far have been elastic consumers, i.e., consumers capable of consuming as many resources as are available. In this section, non-elastic consumers with bursty resource demands are investigated. In section 4.2.1.2 it has been argued that for non-elastic consumers a combined pricing scheme, with charges based on the resource requests and resource usage, should be used. Charges of the form $\text{charge}_i = \beta\Big(x_i p(\sum x_i)\Big) + (1 - \beta)\Big(r_i p(\sum r_i)\Big)$ have been proposed, where x_i is the resource consumption of consumer i and r_i its resource request. The factor β determines the ratio of request and usage price.

To evaluate this pricing scheme a combination of bursty workloads, modelled using Markov processes (as described in section 5.1.5), elastic consumers, and a non-adaptive constant rate consumer is used. The results are shown in figure 6.7, with graphs of the achieved service rates[3], requested shares, and charging rates for each consumer. The bottom graph shows the CPU utilisation and the sum of requests with the resulting prices.

Three Markov processes (labelled `Markov1` to `Markov3`) and the non-adaptive constant rate consumer (labelled `CBR`) are started first. The first two Markov processes use the same simple ON/OFF transition probability matrix, but have different amounts of credit assigned (300 and 600 credits/s). The third Markov process generates a more bursty workload by using a four state

[3]For this graph the service rates are averaged over the longer interval of one second. With an interval of 100ms, as used for the other simulation results, the highly variable service rates of the Markov workloads would obscure the service rates of the elastic consumers and the graph would become illegible. However, this longer averaging interval does not have a significant smoothing effect on the service rates for elastic consumers, as their service rates a quite stable.

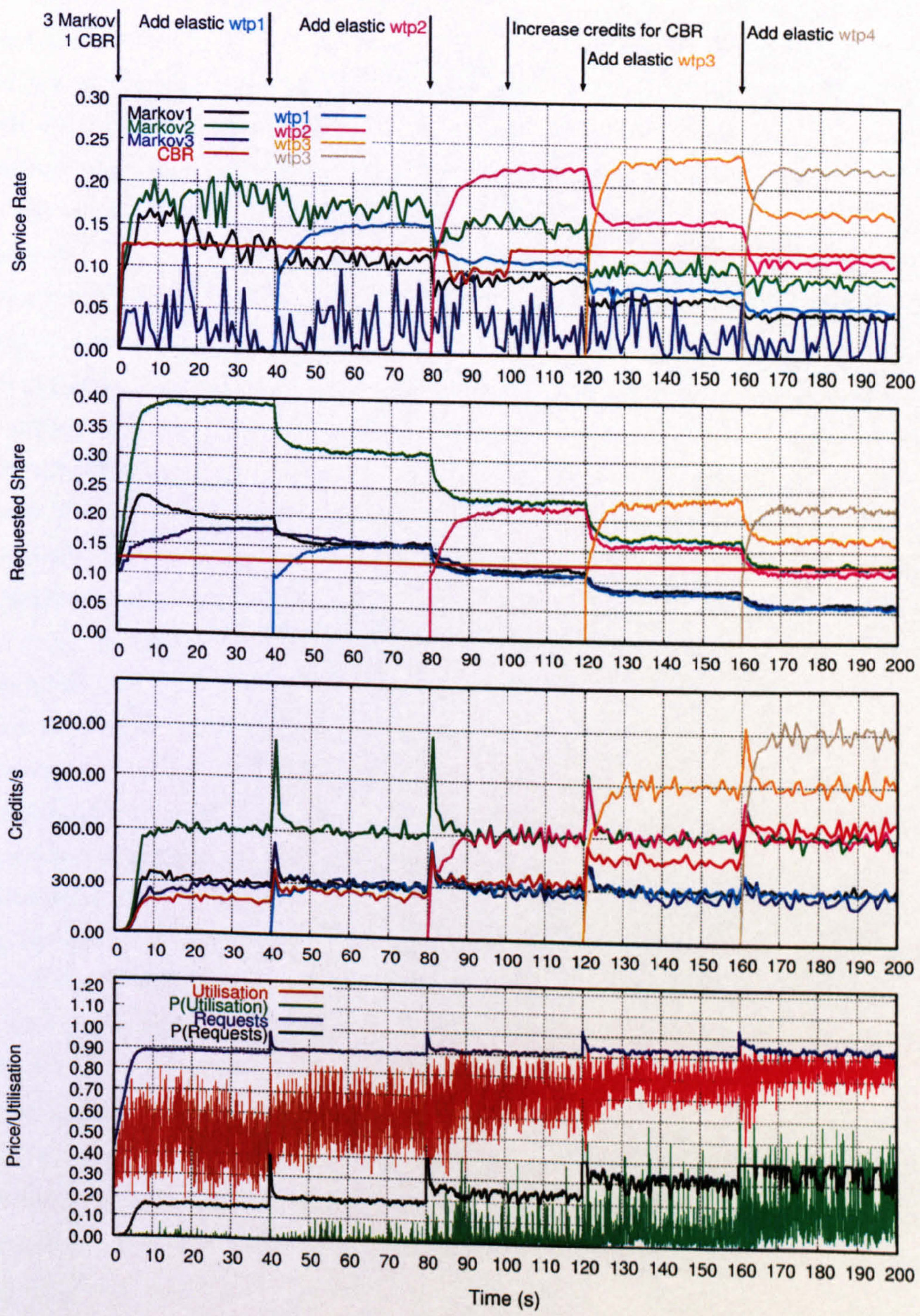


Figure 6.7: Request-Usage prices and non-elastic consumers

transition probability matrix and has 300 credits/s assigned. The non-adaptive constant rate consumer tries to consume $1.25ms$ worth of CPU time every $10ms$[4]. Initially, it has enough credits (270 credits/s) assigned to achieve this rate. At times $40s$, $80s$, $120s$ and $160s$ elastic users with different credit allocations (300, 600, 900, 1200 credits respectively) are started. All consumers, with the exception of the non-adaptive consumer, use WTP as their adaption strategy. The charges are split 40% for requests and 60% for usage.

Initially (i.e., in the first 40 seconds), the Markov processes request a much larger share than they actually consume, especially the consumer labelled Markov2 (requesting 40% while consuming a maximum of 20%). However, the processes can afford to request more resource than they consume, since, as the bottom graph shows, charges are almost exclusively based on resource requests and not usage during the initial period. Due to the $40 : 60$ split of charges consumers can afford this high reservation. However, with the first elastic user joining the task set, the sum of requests, and thus the charges for requests, increases and the Markov consumers reduce their resource requests. This has a bigger impact on the service rate for the consumer labelled Markov1 as its credit allocation is half of that of consumer Markov2. Usage prices are still very low as, due to the bursty behaviour of the Markov processes, the overall resource utilisation is still low.

With a more elastic consumer joining, the other consumers are forced to reduce their resource requests further, since both the prices for request and usage rise. With the fourth elastic consumer joining at time 160s the price for request reaches its maximum and the higher charges for usage are becoming more dominant as the resource utilisation rises.

An interesting observation is that all the adaptive applications manage to keep their credit spending rates constant around their credit allocations, although charges are based on both their requests and their resource usage and the resource utilisation is quite bursty. Furthermore, as in the previous experiments, elastic consumers achieve service rates proportional to their credit allocation.

The non-adaptive, constant rate consumer (labelled CBR) initially sustains its desired service rate of 12.5%, as its credit allocation covers the charges it incurs for its resource request and usage. However, around the 84th second the charging rate exceeds its credit allocation and its service rate drops below the desired rate. If the consumer were an MP3 player, this would result in a breakup of the audio signal. A user could then increase the credit allocation to such an application. Thus, in the experiment, the credit allocation for the CBR consumer is increased to

[4]An example for such an application is an MP3 player (see section 6.2).

660 credits/s at time 100s. The new credit allocation is sufficient to sustain the required service rate again.

In summary, this experiment demonstrates that the combined Request-Usage pricing scheme is suitable for non-elastic and non-adaptive consumers as well as elastic consumers. If the resource is only slightly loaded, it allows bursty consumers to request resources at a peak rate, or above it, without impacting other consumers. However, if the load increases, they are forced to reduce their resource requests, as they will be increasingly charged for their resource usage as well. At very high resource load, they essentially behave like elastic consumers. Furthermore, the decentralised system also allows non-adaptive consumers to participate. By simply ignoring the charges, like the CBR consumer, they are guaranteed a service rate proportional to the total number of credits in the system. If this is not sufficient, the user could increase the credit allocation for those consumers.

An interesting direction for future work is to dynamically change the parameter β, which determines the ratio of request and usage charges. By changing β, the emphasis of charges could be placed more on charges for requests, if, for example, there is a significant difference between the sum of requests and actual resource utilisation. This would encourage consumers to estimate their resource requests more accurately. Likewise, if the utilisation is *closer* to the sum of resource requests, the emphasis could be placed on usage based prices, as usage based charges are more meaningful (congestion occurs through usage not through requests).

6.1.4 Ill-behaving consumers

The experiments so far only used well-behaving consumers, i.e., consumers which only made small adjustments to their resource requests on each adaption and which reacted to feedback signals. In this section, a number of more aggressive strategies and their impact on the overall system are evaluated. Two different strategies are presented. Both aggressively increase their resource requests if they are charged less than they can afford. They simply double their resource demand. However, if their charges exceed their credit allocation, the first strategy immediately decreases its resource demand to a small share (to 5% in the experiment) while the second strategy only reduces its request by a small fraction (1% in the experiment). Furthermore, these strategies also adapt in much shorter intervals (every $10ms$ in the experiments) in order to exploit small fluctuations of the price. By aggressively increasing their resource demand, these consumers may drive the price up and other consumers may suffer from this.

To evaluate the impact such aggressive consumers have on well-behaved consumers, a task set of well-behaved consumers is initially used. Once that task set has reached a stable state, an aggressive consumer is introduced to the task set and the impact can be observed. Then, the task set is reduced to observe the behaviour of the aggressive strategy under changing load and price conditions. The set of well-behaved consumers is consists of 5 elastic consumers with different credit allocations (multiples of 250 $credits/s$) and the simple two state Markov workload, also used in section 6.1.3. These consumers all use the WTP strategy and adjust their resource requests every $100ms$. The pricing mechanism is the same combined Request-Usage scheme, that is also used in section 6.1.3. A new consumer is introduced after 15 seconds and an elastic consumer leaves the task set after 30 seconds. Figure 6.8 shows the results of this experiment.

The three panels show the service rates achieved by the consumers when a consumer with a different strategy joins the task set after 15 seconds. For the top panel, a consumer, labelled `Aggressive1`, joins, which doubles its resource requests if its charges are below its credit allocation and reduce its share to 5% when charges exceed the credit allocation. The consumer, labelled `Aggressive2` in the middle panel, also doubles it resource request of the charging rate does not exceed its credit allocation but only reduces its resource request by 1% if it does. For a comparison, the bottom panel shows the service times if the new consumer joining is also using the WTP strategy. Thus, by comparing the first two graphs with the bottom graph, the impact of a more aggressive consumer can be assessed.

The first variation of an aggressive consumer (top graph) does not seem to have a significant impact on the other consumers. While it aggressively increases its resource requests it also backs down too radically. This has the effect of harming its own performance and it actually receives less service time that if it used a better-behaved strategy, such as WTP. In fact, since it receives less service time than an equivalent consumer using WTP, the other consumers benefit from its behaviour and receive slightly more service time.

However, the second aggressive consumer, as shown in the middle graph disrupts the other consumers noticeably. For example, the consumer labelled `WTP5` receives less CPU time after the aggressive consumer joins, compared to the bottom graph where another WTP consumer joins. When the consumer labelled `WTP3` leaves the task set after 30 seconds, the consumers labelled `WTP4` and `WTP5` are unable to increase their share as in the other two experiments. This is because of the aggressive consumer increasing its share first, and thus driving the price up to a level which prevents the other consumers from increasing their share. Furthermore, the aggressive consumer manages to receive a higher service rate when compared to the WTP

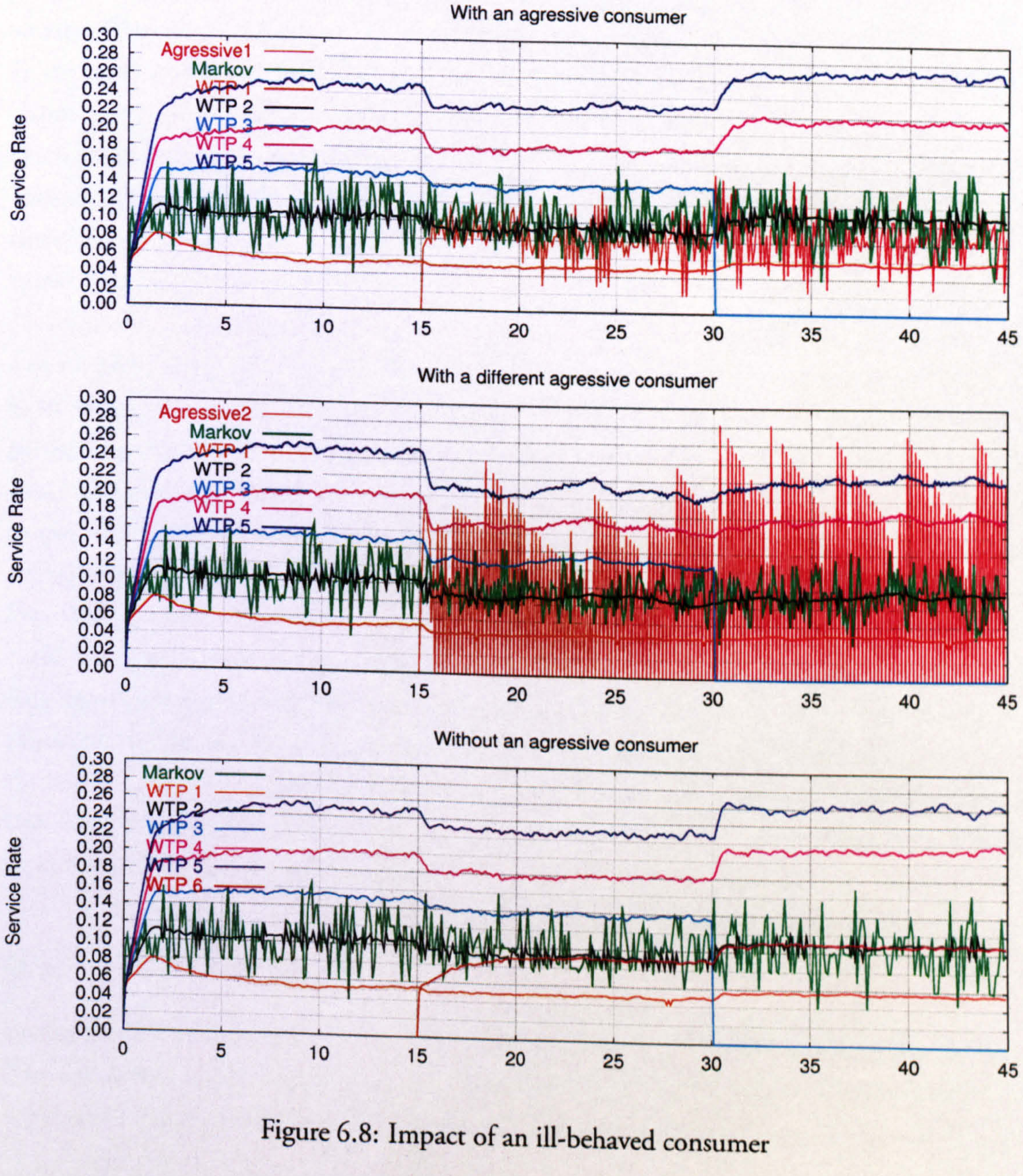


Figure 6.8: Impact of an ill-behaved consumer

consumer with the same credit allocation in the bottom graph[5], and causes all consumers to miss a considerable proportion of their deadlines (a total of 8% of the deadlines are missed).

However, this experiment also demonstrates that the credit account mechanism, described in section 4.3, prevents the aggressive consumer from causing greater damage to the other consumers. The other consumers still make reasonable progress, albeit not quite as good as would be the case without the aggressive consumer. This can be attributed to the credit account mechanism: the service rate of the aggressive consumer frequently drops to zero because it is spending credits at a significantly higher rate than credits are allocated to it. Thus, with an empty credit account, the consumer is unable to pay for both its resource request and consumption and, therefore, the consumer is not allocated CPU. This mechanism prevents the aggressive consumer from driving the other consumers out of the system.

However, as the increased service rate it receives indicates, the aggressive consumer is still able to "play" the system to some extent, even though it suffers from frequent suspensions due to an empty account. Apart from the measures already discussed in section 4.4.2, such as social pressure on users with such aggressive consumers, a number of other, more technical deterrents could be deployed to prevent consumers from behaving too aggressively. An obvious measure is to prevent consumers from making rapid changes in their resource requests. The two example strategies used in this section double their resource requests every $10ms$ if their charges are low. Policing the increase in resource requests to an upper limit, of, for example 20%, prevents rapid increases in demand while not interfering with well-behaved consumers, which typically only increase their resource requests moderately. Another measure could identify ill-behaving consumers by monitoring credit accounts. Consumers which are frequently suspended due to the lack of credits, such as the second aggressive consumer in the experiment, could be penalised and suspended for increasingly longer periods of time, or the user or system administrator could be notified of such consumers and then they could take appropriate action.

6.2 FreeBSD results

In this section an evaluation of the congestion pricing implementation in FreeBSD is given. The simulation results suggest that a combined Request-Usage pricing mechanism, using an exponential pricing function, is a suitable scheme to support a variety of different application

[5]Due to the short averaging interval of $100ms$ this is not apparent from the graph. However, the aggressive consumer receives around 3.37 seconds of CPU time during the experiment, while the WTP consumer in the bottom graph receives around 2.93 seconds of CPU.

scenarios. Therefore, the FreeBSD prototype implements such a scheme. Charges for request and usage are divided into a 40:60 ratio. For simplicity, elastic consumers are implemented as an endless loop but, essentially, any compute bound process could be used. These elastic consumers use the WTP strategy for adjusting their resource request and execute the WTP algorithm once every second.

In general, the baseline behaviour of the FreeBSD prototype is comparable to the simulation results. Therefore, a slightly more complex experiment is presented in this section. This experiment is similar to the one described in section 6.1.3 using both adaptive and non-adaptive applications. The results of this experiment[6] are presented in figure 6.9. The graphs show the service rates, requested shares, charging rates, and prices as in the simulation experiments.

The task set for the experiment initially contains four elastic adaptive consumers. These consumers have different credit allocations (initially set to 25, 50, 125, and 150 $credits/s$) and are labelled `loop1` to `loop4` respectively. In addition, a non-adaptive MP3 application, decoding a variable bit-rate MP3 sound file, is used with an initial credit allocation of 100 $credits/s$ and a resource reservation of 10%. This consumer is labelled `mpg123`. The default scheduling class has a credit allocation of 200 $credits/s$ and a CPU reservation of 20%. There is no adaption performed for the default scheduling class (however, this could optionally be enabled). Within the default scheduling class an elastic consumer (labelled `loop0`) and a parallel build of the FreeBSD kernel are executed. After 290 seconds this elastic consumer is placed into the CPU reservation class and then it adjusts its resources demands using the WTP strategy. The processes of the kernel compilation and other processes of the default scheduling class are summarised and labelled `Default` in the graphs.

The kernel build is started first and after 20 seconds the elastic consumers are added to the task set. These adaptive tasks quickly manage to receive a service rate proportional to their credit allocations. In the default scheduling class the CPU is shared between the elastic consumer (`loop0`) and the other processes. The burstiness of these allocations is caused by the bursty resource demand of the compilation processes.

The MP3 decoder is started 40 seconds into the experiment and it initially receives 10% of the CPU, which is proportional to its credit allocation of 100 $credits/s$. However, this allocation is not sufficient and the sound frequently breaks up. Therefore, its credit allocation is increased to 175 $credits/s$ after 140 seconds. Since this application is non-adaptive its resource request is automatically increased to 17.5%, the share it can pay for at the maximum charging rate (see section 5.2.4). After the credit allocation increase, the MP3 decoder receives enough

[6]Performed in multi-user mode on the same Intel P200 hardware used in sections 5.1.8 and 5.2.3.

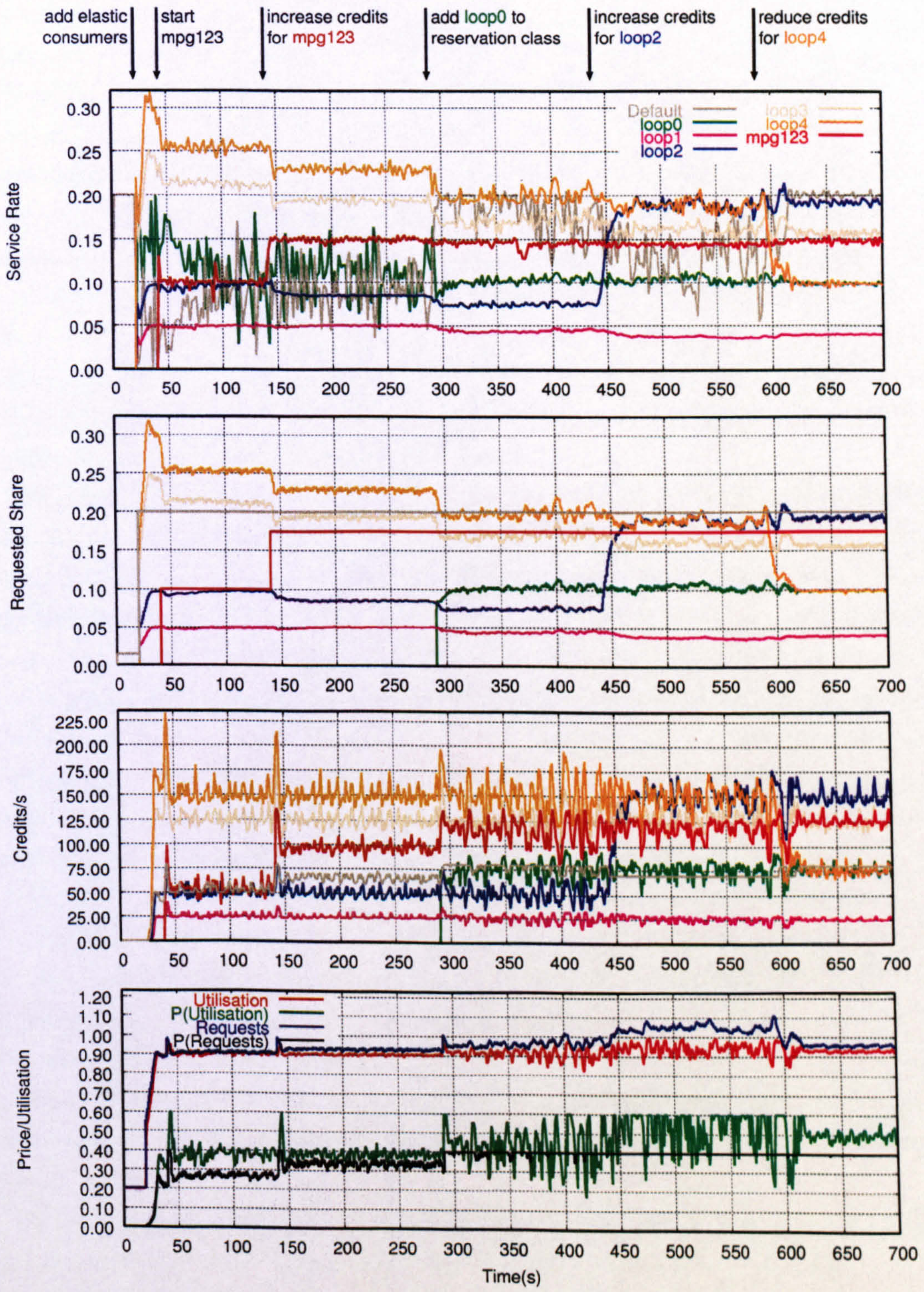


Figure 6.9: FreeBSD: Congestion prices for CPU

CPU resource to decode and play back the sound file without breaking up, consuming CPU resource at a constant rate of around 15%, leaving 2.5% percent of its resource request unused.

With the increase of the credit allocation and share for the MP3 decoder the adaptive consumers are forced to reduce their resource requests as the prices rise. Further changes to the task set result in similar changes. After 390 seconds the elastic consumer `loop0` is taken out of the default scheduling class and placed into the CPU reservation scheduling class with a credit allocation of 75 $credits/s$. After 440 seconds the credit allocation for `loop2` is increased from 50 to 150 $credits/s$ and after 590 seconds the credit allocation for `loop4` is reduced from 150 to 75 $credits/s$. On each of these changes the individual consumers quickly adjust their resource requests and their resulting service rates remain proportional to their credit allocations. Furthermore, elastic consumers with the same credit allocation achieve similar service rates (e.g., consumer `loop2` and `loop4` between 440 and 590 seconds).

The slightly higher variations of the service rates after around 300 seconds can be attributed to the elastic consumer `loop0` leaving the default scheduling class. Due to the bursty resource demand of the remaining processes in that class, the overall resource utilisation and thus the usage price fluctuates stronger. Thus, the charges vary more and the consumers adjust their resource requests slightly more than before the change. However, the resulting service rates are still reasonably stable and not lower than the minimum guaranteed service rates (as defined by the proportion of credit allocation to the maximum charging rate). In fact, especially during the period from 450 seconds to 600 seconds, the elastic consumers achieve a higher service rate than their proportional fair share, since the resource demand of the default scheduling class is significantly lower than its resource request of 20%.

This effect can also be observed in the charging rates, shown in the third graph. Up to 290 seconds the elastic consumers manage to stabilise their charging rates close to their credit allocations. The charging rates vary more strongly after consumer `loop0` leaves the default scheduling class. However, on average, the charging rates are maintained at the same rates at which credits are allocated. Furthermore, some of the variations visible in the graph can be attributed to quantisation errors; the charging rate is calculated by the analysis tools over one second intervals, but, these intervals do not exactly match the one second time quantum of the scheduler nor the one second adaption intervals

The bottom graph summarises utilisation and the sum of resource requests and the resulting prices. Resource requests are largely matched by the resource utilisation except during the period where the resource usage of the processes in the default scheduling class is particular bursty (450-600 seconds). The resulting prices clearly show the 40:60 split between request and usage prices.

A number of key observations can be made from this experiment. First, with the decentralised resource management, independently acting elastic consumers can achieve weighted proportional fair resource allocations. Second, constant rate and non-adaptive applications, such as the MP3 decoder, can be accommodated by the same operating system mechanisms. Third, placing the default scheduling class as a dummy process in the CPU reservation scheduling class effectively prevents starvation of its processes. And, most importantly, the results obtained from the FreeBSD prototype are comparable to the results obtained with the simulator (especially when compared to the experiment with the non-elastic consumers presented in section 6.1.3).

6.3 Application use

In the previous experiment a set of simple applications make use of the decentralised resource management architecture to achieve proportional fair sharing of the CPU. In this section a set of experiments is presented which demonstrates how a more complex and realistic application can make use of the resource management architecture. For this purpose an instance of the modified version of `mplayer`, introduced in section 5.2.5, is executed with varying background loads and its performance is compared to instances, both adaptive and non-adaptive, using CPU reservations and the default FreeBSD scheduler. The aim is to evaluate what benefits the decentralised architecture offers to real applications.

The `mplayer` application is used to decode and display a 400s long video clip at 25 frames per second (10000 frames in total). The modified version of `mplayer` can be configured through command line options to decode the video frames at a fixed level of post-processing or to adapt the amount of post-processing performed based on the audio/video skew of the previous frame and, additionally, adapt based on the feedback provided by congestion prices. The first two modes are available in the standard `mplayer` distribution. The last mode has been described in detail in section 5.2.5.1. As the primary measure to compare performance between different configurations the audio/video skew is used as this is the main measure `mplayer` attempts to minimise. Specifically the following configurations are compared:

Adapt Q, BE: `mplayer` is executed in the default scheduling class and performs its default quality adaption policy.

Fixed Q, 60%: `mplayer` is executed with a fixed reservation of 60% of the CPU and performs the maximum amount of post-processing for all frames.

Adapt Q, 60%: `mplayer` is executed with a fixed reservation of 60% of the CPU and performs its default quality adaption policy.

Feedback, 500: `mplayer` receives a credit allocation of 500 credits per second and performs adaption based on the feedback provided by congestion prices. This should result in a resource allocation of at least 50% of the CPU.

These configurations are representative of different approaches to resource management. The first configuration represents a traditional Unix operating system, using a "traditional" Unix scheduling algorithm. The next two configurations represent reservation based scheduling algorithms and are chosen to evaluate the effectiveness of `mplayer`'s default adaption policy. The final configuration is used to evaluate the impact of the decentralised resource management architecture on the performance of `mplayer`.

To compare the performance of these different configurations they were exposed to the same background load while decoding the aforementioned video clip. During each experiment, the background load is varied every $100s$. For all experiments using reservations (either fixed or dynamically adjusted) the following load is used: Initially, only a light load is used consisting of a constant rate application consuming 5% of the CPU and a process executing in the default scheduling class with a bursty resource demand[7]. The default scheduling class has a CPU reservation of 5%. After $100s$ the load is increased by starting three processes with reservations of 5%, 5%, and 10% and a bursty resource demand. After a further $100s$ all four processes with bursty resource demand are replaced with entirely compute bound processes: one executing in the default scheduling class and three with fixed resource reservations of 5%, 5%, and 10% respectively. The compute bound process in the default scheduling class has the effect that the system has no slack CPU resources available (recall the relative priorities of the different scheduling classes described in section 5.2.1). For the final $100s$ of each experiment, the compute bound process with fixed reservations are replaced with elastic compute bound consumers performing the WTP to adjust their resource demand. This background load essentially competes with `mplayer` process for available CPU resources and generates a high background load. For the only configurations not using reservations for the `mplayer` process, **Auto Q, BE**, the background load is generated by starting the same background processes as above, albeit in the default scheduling class without resource reservations.

[7]The bursty demand is generated using a two state Markov process. In the "On" state the process consumes as much CPU as possible and sleeps in the "Off" state. Each state has a period of $1s$ and the probability for state changes was set to 50% for both states. For simplicity, this configuration was used for all bursty background loads.

All experiments were executed on a 1.3 GHz Athlon computer with 256MB of memory. For the experiments using the CPU reservation scheduling class a scheduling quantum of $100ms$ and a preemption interval of $1ms$ were configured. For these experiments, the X server process was given its own CPU reservation equivalent of 9%. This reservation is sufficient for the work performed by the X server and was necessary to avoid the scheduling of the X server from influencing the performance of the measured applications.

Figure 6.10 gives a summary of the results. For each graph four subgraphs are shown corresponding to the four phases of each experiment. The top graph shows the service rates received by the `mplayer` process for the different configurations, the second graph shows the level of post-processing performed by the adaptive configurations. The third graph plots the skew between audio and video playback in milliseconds and the fourth graph summarises this data as a cumulative distribution function (CDF) for the four different phases (at 25 frames per second a frame should be displayed every $40ms$). Ideally, the CDF would be a vertical line at $0ms$ indicating perfect synchronisation between the audio and video playback. A positive skew indicates that the video frame has been displayed after the corresponding audio sample has been played while a negative skew indicates that the video frame has been displayed before the corresponding audio sample. The latter can occur if the previous video frames have been displayed too late (positive skew) and `mplayer` attempts to "catch up". Note, that for clarity of presentation not every graph shows results from all configurations.

Looking at the results in more detail, it becomes clear that the default FreeBSD scheduler is not suitable for this type of workload. Even under the light background load in the first phase the service rate received by the `mplayer` process is highly variable. This variability is too high to be compensated by the default adaption strategy. As a result, the audio/video skew is highly variable (for clarity not shown in the third graph) and, as can been seen from the corresponding CDF in the bottom graph, only a small fraction of video frames are displayed in sync with the audio playback. A large fraction of video frames are displayed when the previous or next frames respectively should have been displayed, resulting in unacceptable playback performance for an end-user. This effect is exaggerated when the background load is increased in the second phase making the video clip unwatchable with 80% of the frames being dropped; in the final two phases, with further increased background load, less than 0.1% of the frames are displayed.

The performance of the other, reservation based configurations is in stark contrast to the performance of `mplayer` under the default scheduling class. For all tested configurations the audio/video skew is predominately within an acceptable range. However, there are differences between these configurations. With a light background load in the first phase the modified

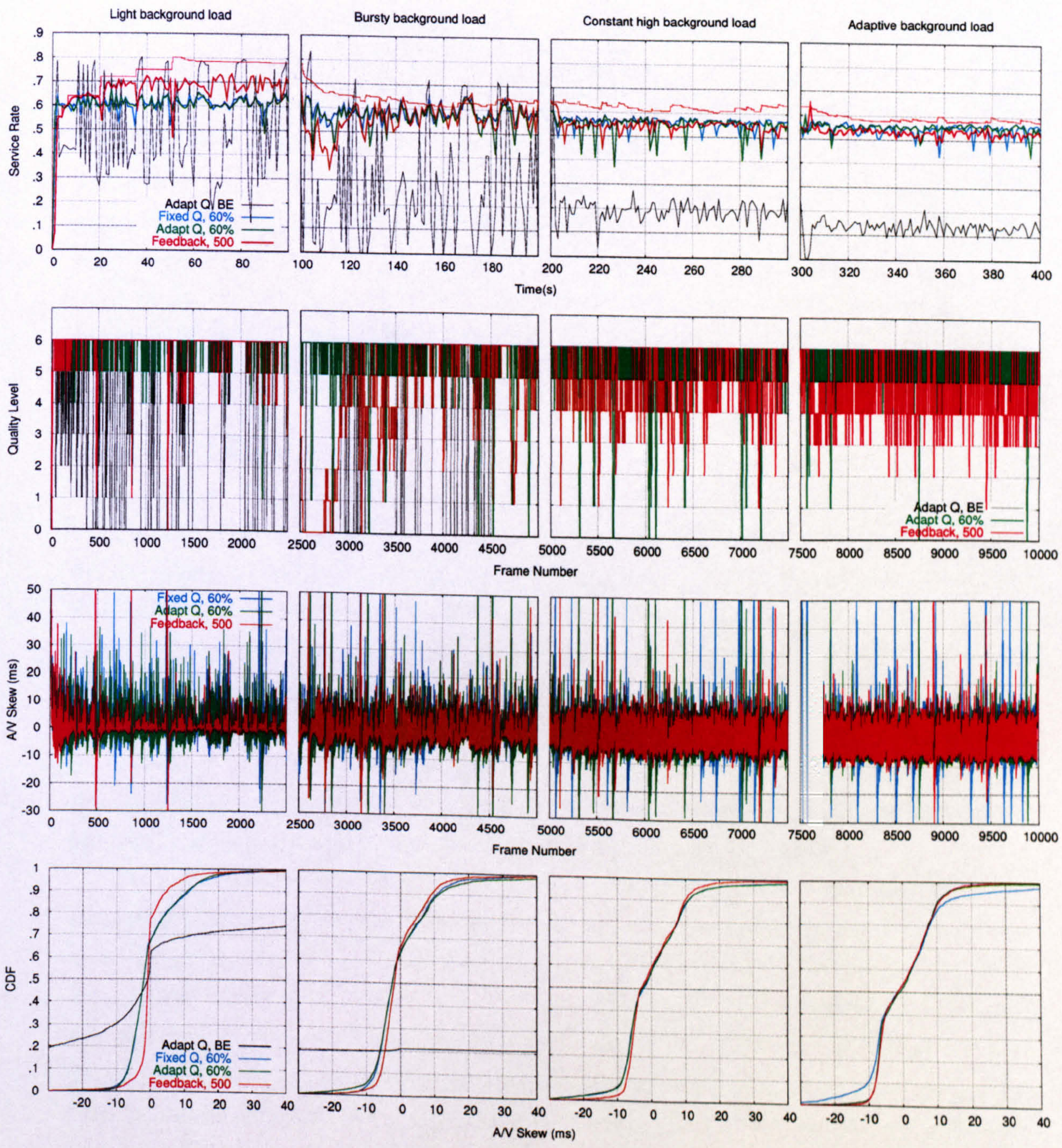


Figure 6.10: `mplayer` performance comparison in different configurations

`mplayer`, adjusting its resource reservation based on the feedback provided by the operating system performs better than the instances of `mplayer` with a fixed reservation (both the adaptive and non-adaptive versions). This is not surprising since the modified version of `mplayer` receives a bigger share of the CPU due to its, dynamically adjusted, higher resource requests (the requested resource share for the modified version of `mplayer` is shown as a thin red line in the top graph, with the received share of the CPU drawn as a thick red line). In contrast, the instances with fixed reservations have to compete with the other processes for slack CPU resources on a best effort basis. The resulting better performance is particular visible in the third graph showing the audio/video skew. After an initial settling period, the skew for the feedback based `mplayer` is predominantly within $5ms$ of the ideal time while for the adaptive `mplayer` instance with fixed reservation of 60% of the CPU the skew is more variable, but typically within $10ms$. This is also reflected in the CDFs of the audio/video skew, shown in the bottom graph. Interestingly, the configuration with the fixed maximum level of post-processing levels almost matches the performance of the adaptive configuration, indicating that 60% of the CPU on *this* particular system is just about sufficient to decode *this* video clip at the highest quality[8].

With a higher background load, as in the second and third phase of each experiment the difference between the feedback based instance of `mplayer` and the instances of `mplayer` with fixed reservations becomes less pronounced. Due to the higher load the feedback based instance of `mplayer` is forced to reduce its requested share close to the 60% of the CPU[9]. However, since it reduces its level of post-processing more aggressively based on the feedback provided by the operating system its audio/video skew is less variable when compared with the other configurations. The reduction of post-processing is particularly visible in the third phase where the feedback based `mplayer` frequently chooses a level of post-processing of 4 while the adaptive `mplayer` with a fixed reservation rarely drops its post-processing level below 5.

The forth phase further exaggerates this effect by adding adaptive background loads, essentially driving the system into full utilisation. The added competition forces the feedback based `mplayer` to further reduce its requested share to slightly below the 60% guaranteed to the `mplayer` instances in the other configurations. The feedback based `mplayer` compensates this constraint resource availability by reducing the level of post-processing it performs. In this phase, the adaptive `mplayer` with fixed reservation matches the performance of the feedback based `mplayer`, although with a slightly higher overall quality. The performance of `mplayer` with a fixed post-processing level, however, drops slightly behind, clearly visible in the high spikes in its audio/video skew and the wider audio/video CDF graph. The higher skew is clearly

[8]In fact, this is the reason why this reservation was chosen.

[9]The `mplayer` process does not always consume its entire resource reservation because it may block, waiting for the X server to render a decoded frame. The X server requires up to $2ms$ of CPU time to draw a frame.

visible to a user, the video playback appears slightly unsteady and jittery. This is not surprising, considering that almost 10% of the frames are displayed more than half a intra-frame time before or after their ideal time. This frequently leads to rapidly varying intervals between frames displayed, which can be perceived as irritating by a human.

6.3.1 Credit allocation versus CPU reservations

A general problem with reservation based resource allocation schemes is that someone, possibly a end-user, has to decide what fixed fraction of a resource to assign to particular consumers. The problem is that a too high reservation results in inefficient sharing of resources by leaving them under-utilised, while too small a reservation may result in poor performance. This problem is made more complicated since the resource demand of a given application may depend on its input parameters. For example, the resource demand of the `mplayer` application depends not only on the decoding scheme of a given video clip but also varies between different video clips using the same encoding scheme. With the decentralised resource management architecture, the question arises if the assignment of credits is similarly problematic.

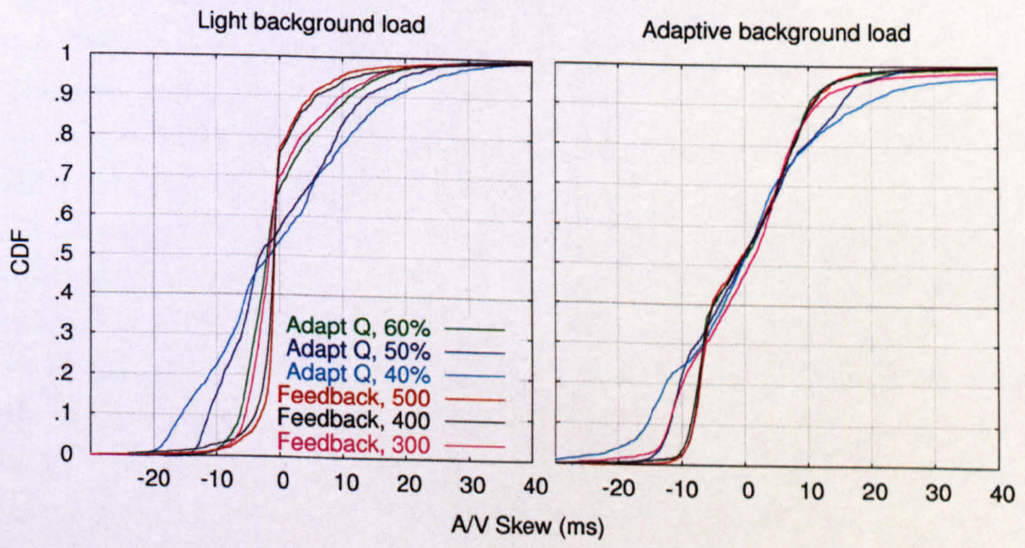


Figure 6.11: Sensitivity to credit allocation

To investigate this issue, the previous experiments were repeated with different, lower credit allocations for the feedback based `mplayer` (400 $credits/s$ and 300 $credits/s$ instead of 500 $credits/s$) and lower resource reservations for the default adaptive `mplayer` (50% and 40% instead of 60%) respectively. Figure 6.11 shows the resulting cumulative distributions of the per-frame audio/video skew of the first and last phase of each experiment for these six different configurations.

From the graphs it is clearly visible, that the performance of the feedback based `mplayer` configurations are almost identical. There is hardly a difference for the configurations with 400 and 500 $credits/s$, both under light and high background load. The configuration with only 300 $credits/s$ has a slightly higher skew under the light background load as it requires more time to settle to a steady state initially. Under the high background load this configuration performs slightly worse than the other two feedback based configurations.

In comparison, the configurations with fixed reservations are significantly more sensitive to the amount of CPU reserved for them. In particular, the configuration with a reservation of 40% exhibits a significant skew between its audio and video playback times in both phases of the experiment, but especially under the higher load. During this phase of the experiment it becomes unpleasant for a viewer to watch the decoded video with more than 10% of frames displayed more than $20ms$ out of sync and with rapidly changing intra-frame times. The other configurations, especially the ones with a 50% reservation or credit allocations of 400 or 500 $credits/s$ are still quite watchable. Subjectively, the skew and the related changes of intra-frame times are significantly more irritating than the changes of post-processing levels. However, this may depend on the encoding scheme used for the video stream.

The main reasons for the better performance are that the feedback based configuration can more dynamically adjust both its resource requests and the post-processing level. The former is probably more significant, especially under lighter load conditions where in a reservation based resource management architecture consumers have to compete with others for slack resources. However, under high load conditions the adaption of modes of operations, e.g., change of post-processing levels in the case of `mplayer`, also contributes to better performance.

In general, it is important to note that for the feedback based configurations used in this section the minimum guaranteed share of the CPU based on their credit allocation is 30%, 40% and 50% respectively and that even the configuration with the lowest credit allocation compares quite favourably with the configuration with the highest fixed reservation of 60%. This obviously depends somewhat on the background load used. However, the comparison demonstrates that the feedback based configurations are less susceptible to the credit allocation than reservation based schemes are to changes in reservations.

6.3.2 Summary

In this section the benefits of the decentralised resource management architecture to applications have been evaluated. This evaluation was performed using a modified, adaptive multimedia decoder application, which utilises the feedback provided by the operating system to adjust its resource demand and change its mode of operation. The performance of this application has been compared to an unmodified version of that application executing both on a traditional Unix CPU scheduler and a reservation based scheduler.

Two main observations can be made from these experiments: First, the application performs better when adapting to the feedback provided by the system. This can primarily be attributed to its ability to dynamically change its resource demand, but also due to its ability to change its mode of operation based on the feedback. More importantly, however, the second observation is that the feedback based system appear to be easier to configure for the user. The performance of the individual applications, and the overall system performance, is less sensitive to the allocation of credits to applications than a comparable reservation based system. This is because, the applications are actively involved in the way resources are shared amongst them.

6.4 Experience with the system

In the decentralised resource management system individual consumers act independently of each other and compete for limited resources. This is essentially a system where multiple feedback loops, one for each consumer, may indirectly interact with each other — which is, from a control theoretical point of view, a fairly complex system. Even in controlled systems, i.e., where parameters such as the WTP's convergence constant κ can be set globally, issues such as oscillations, stability and sensitivity to change and different parameter settings may arise. In this section some of my experiences with this type of system are presented.

In general, as, for example the experiment presented in section 6.1.2 shows, task sets with just well-behaved, elastic consumers do not create any unwanted interactions between different consumers, even if they use different strategies for their adaption. Resource allocations are stable and there are very little variations of service and charging rates. However, when designing this experiment, I had significant problems finding the correct parameter settings for the PID controller used by half of the consumers in the experiment. For different choices of parameters, consumers with the PID controller would either react very slowly to changing conditions, or would wildly overreact to the feedback provided (unlike WTP's κ, the effect of some of the parameters of the PID controller are non-intuitive). The wild overreaction of half the consumers in the task set also had a knock-on effect on the well-behaved consumers using the WTP strategy, as available prices and resources would fluctuate rapidly. Due to the high proportion of misconfigured consumers, this effect was far more severe than in the system with just one ill-behaving consumer as presented in section 6.1.4.

From figures 6.7 and 6.8 it can also be observed that bursty or ill-behaved consumers have an effect on the "smoothness" of the achieved service rates and observed charging rates of elastic consumers. The higher variability in resource utilisation, and thus the price, results in higher variations of the feedback signals. One possible solution would be to make the price less variable, for example, by averaging the utilisation over longer periods. However, this would potentially introduce delays into the appropriate feedback being delivered to consumers in overload situations. I have experimented with different time scales for establishing prices based on utilisation. This experience showed that it is better to provide fine grained but highly variable feedback, as used in the experiments presented in this chapter, instead of a more coarse grained feedback (utilisation is averaged with an agile EWMA filter over twice the length of the longest period used by any consumer). Consumers can then average the observed charges at timescales suitable to them. Furthermore, consumers could also vary these intervals or change parameters of their adaption strategy, for example WTP's κ, over time.

It is worth pointing out, however, that the variations in service rate for well-behaved consumers does not effect their "guaranteed" resource allocations. Due to the design of the pricing function, an elastic consumer is ensured to receive a service rate which is proportional to their credit allocation and the maximum charging rate ($w_i/(max\ charging\ rate)$). Thus, adaptive applications only are exposed to these variations if they either receive more service time than their minimum rate or if they overreact to charges and reduce their resource requests below the minimum rate.

The Request-Usage pricing scheme splits charges between requested resources and resources used. The ratio of the split can be controlled by a single variable. The need for this pricing scheme arose when experimenting with non-elastic consumers. In the first experiments I conducted with non-elastic consumers using the WTP strategy, these would constantly increase their resource requests as they were only being charged for their resource consumption. While the usage charges prevented the system from being driven into constant overload, nevertheless a significant proportion of deadlines would be missed. Furthermore, the EDF scheduler is known to behave unpredictably under overload, thus allowing consumers to make arbitrarily large requests could potentially be exploited by malicious consumers. The Request-Usage pricing scheme prevents this by charging consumers for their resource request as well as their usage.

With this pricing scheme the question arises on how to split the charges. In practice, placing the emphasis on usage charges can cause the same effect as with only usage based charges. Placing the emphasis on request based charges may result in a lower overall system utilisation. Experiments with different mixes of bursty and elastic consumers and ratios ranging from 40 : 60 to 70 : 30 have worked well. It would, however, be interesting to explore these issues in more detail as part of future work.

The mathematical foundation of congestion prices assumes elastic resource consumers with continuous, convex utility functions. Many real world applications do not satisfy this assumption. In my experience this is not a problem. For example, in section 4.4.1 two typical categories of applications with non-convex utility curves were presented. In this evaluation chapter, it has been demonstrated with two representative applications, `mpg123` and `mplayer` respectively, that the decentralised resource management architecture can support these types of application.

And finally, in my experience with the FreeBSD prototype, I have been unable, even with ill-behaved or misconfigured consumers, to place the system into a state where it was completely unresponsive and required rebooting (as it is possible when using the real-time scheduling class [NHNW93]). This can be in part attributed to the credit account mechanisms (as demonstrated in section 6.1.4) and in part to the resource "reservation" for the default scheduling class. Thus, even if misbehaving consumers may negatively affect other consumers of the system, the user or system administrator still has the opportunity to stop these consumers. However, in experiments with more realistic application scenarios, some limitations of the prototype have (not unexpectedly) become apparent. For example, the server architecture of the X window system causes significant distortions if more than one application makes heavy use of the display. Other limitations include the limited timing and scheduling granularity and the audio device driver only allowing one client playing back audio samples at a time. While limitations prohibit the use of the prototype in the real-world it has to be stressed that they are only artifacts of the current prototype and are not inherent to the decentralised resource management architecture itself.

6.5 Application developer issues

The decentralised resource management architecture removes resource management policies out of centralised entities and places them into the applications. Thus, application developers also have to be concerned with managing resources for their applications. In this section the impact on application developers is evaluated in the form of a discussion of the issues involved. This discussion takes the FreeBSD prototype as an example since it provides a more realistic environment than the simulator.

Application developers are potentially faced with the additional task of managing resources for their applications. Applications receive feedback from the resources and have to adapt their resource demand accordingly. With the decentralised architecture, developers have three options.

First, developers could opt to not participate in the management of resources. These applications then receive either best-effort service, if placed into the default scheduling class by the users, or a fixed share, if placed into the resource reservation class (like the MP3 decoder in section 6.2). Furthermore, default adaption strategies could be supplied automatically to these and legacy applications. For example, Eclipse/BSD (described in section 2.3.4) uses a modified `libc` which intercepts certain system calls and loads either generic or application specific *requirement brokers* to enable unmodified legacy applications to take advantage of Eclipse/BSD's resource management features [BGSS99]. A similar approach can be adopted for the decentralised resource management architecture.

Second, developers could choose a standard adaption strategy, like WTP, from a system supplied library to adjust the resource demand of the application, but without changing the mode of operation of the application in response to charges. This option is suitable for simple elastic consumers. However, it is unclear what advantage this would provide over the automatic approach described above.

Third, application developers can write applications to actively participate in the management of resources. The decentralised architecture is particularly targeted at these types of application. Typically, these applications will be complex and resource intensive and it is assumed that at least a small group of expert developers have good knowledge of how these applications behave in resource constraint environments, and that benchmarking and profiling should be good practice during the development cycles of these applications. Therefore, some developers should have an understanding on how an application may adapt to varying resource availability. This understanding may be used to adjust the mode of operation of such applications in response to observed charging rates. It is not expected that this requires substantial changes to applications which already are adaptive, however, these applications can be enabled to be more proactive in their resource adaption.

As anecdotal evidence, it took the author of this dissertation less than a day to identify a "knob" to adjust the decoding quality, and thus influence the resource requirement of the MPEG decoder, mpeg2play, used in section 5.1.5.1. Similarly, it required about three days to make an already adaptive Motion JPEG player[10], which adjusted the decoding quality when the previous frame missed its deadline, more elastic to resource changes by making it multi-threaded [Neu99]. Modifying the mplayer application was slightly more complicated, partially because it represents a more complex and larger piece of software and partially because it is written quite poorly (e.g., the main() function is close to 3000 lines of code using a substantial number of global variables). However, after the appropriate "knob" and variables had been identified, the changes required to adjust mplayer to respond to feedback from the operating system were local and straightforward. This personal experience seems to suggest that writing adaptive multimedia applications is reasonably easy and adjusting existing adaptive applications to take advantage of the feedback provided by the operating system is straightforward. Existing adaptive multimedia applications adapt after they have observed a resource shortage, i.e., after they were unable to display a decoded video frame on time. It seems straightforward to change this in order to react to congestion charges instead.

In general, adaptive applications are written in a way that they adapt after a performance degradation has been detected. With congestion pricing the same techniques can be applied. But, instead of reacting to the observed performance, however, applications would react to congestion charges. Thus, I believe that for expert developers of adaptive applications, it should be reasonably straightforward to develop applications which take advantage of the decentralised resource management system.

6.6 User issues

The decentralised resource management may also have an impact on how end-users interact with the system. They have to manage limited credits available to them and allocate them to their applications. The current FreeBSD implementation provides them with an interface similar to one which allows users to assign priorities to their applications. However, users actually have better control over the resource allocations as credit allocations result in a more comprehensible resource allocation, than, for example, nice values.

For server systems, this type of control might be sufficient to simply provide a programmable interface to the system. However, for the system to be used by end-users, a more comprehen-

[10]Originally written by Neil Stratford at the Computer Laboratory, University of Cambridge, U.K.

sible user-interface needs to be provided. The simple graphical user interface presented in section 5.2.6 may help users to control credit allocations to their applications and provides them with graphical feedback about the allocation of resources. However, more sophisticated tools are conceivable. For example, users could deploy user agents which manage credit allocations for them in accordance with their preferences. However, these higher level issues are beyond the scope of this dissertation.

6.7 Summary

In this chapter a detailed evaluation of the congestion pricing mechanisms for CPU resource management has been presented. The main evaluation was performed using the simulation environment described in section 5.1. Simulations were used to evaluate different pricing mechanisms and different consumer adaption strategies with a variety of different types of consumer. In section 6.2 the simulation results have been confirmed with the FreeBSD prototype.

The key observation from the experiments is that the decentralised resource management framework can achieve weighted proportional fair sharing of CPU resources with individually acting elastic consumers, using a variety of pricing mechanisms and adaption strategies. This confirms the theoretical results (presented in sections 3.2.2.1 and 4.1). For elastic consumers, resource allocations are weighted proportionally fair to their credit allocations.

A number of different pricing mechanisms have been compared and a simple exponential price function based on the resource utilisation provides good results for well-behaved elastic resource consumers. Such a price function helps to prevent resource congestion (which is desirable) as it provides some feedback signals to consumers before congestion occurs. This is in particular contrast to an alternative pricing mechanism which only applies charges after resource congestion has been detected.

For non-elastic consumers, the combined Request-Usage pricing mechanism, discussed in section 4.2.1.2 provides good results, requiring non-elastic consumers to make "sensible" resource requests based on the overall demand for the resource. For a lightly loaded resource, a bursty consumer may request a share of the resource which satisfies its peak resource demand, while for a heavier loaded resource, such a consumer will be forced to reduce its resource requests. Furthermore, it has also been demonstrated that this pricing scheme can also accommodate non-adaptive applications. They simply receive resources either proportional to their credit allocation or to their resource reservation (whichever is smaller). The combined Request-Usage pricing scheme does not affect elastic consumers as the combined charges are the same as in the scheme using just usage based prices.

In section 6.1.4 it has been demonstrated that the credit account management scheme provides some protection against ill-behaved or malicious consumers. Such consumers may still gain a small, unfair advantage over well-behaved consumers, however, they are not able to completely monopolise the resource. A number of additional measures, such as detecting ill-behaved consumers or preventing resource request changes that are too radical, have been proposed to further limit the impact of ill-behaved consumers.

The decentralised resource management architecture enables applications to actively participate in the management of the resource they consume. In section 6.3 an evaluation of how real applications can benefit from this participation was presented. The evaluation has shown that applications can benefit from the feedback in two ways: they can benefit from controlling the amount of resources they receive and they can proactively adjust their mode of operation. Both enable applications to provide better performance to end-users.

Sections 6.4 to 6.6 provided a qualitative rather than quantitative evaluation of the system, describing personal experience (section 6.4) and discussing the impact on application developers (section 6.5) and users (section 6.6). Described in personal experience, some of the design decisions implicit in the implementations have been motivated.

For application developers and end-users, it has been argued that the decentralised system is not radically different from more traditional operating systems. Developers of adaptive applications may benefit from the feedback provided by the system in order to trigger application adaption (instead of observing deteriorating application performance) and to end-users the system can be presented as a simple priority based resource allocation.

Chapter 7
Conclusions

In this chapter a summary of the dissertation is given, future work is discussed and the conclusions are presented.

7.1 Summary

This dissertation is concerned with the management of resources in operating systems. It has been argued that certain types of application, such as multimedia and server applications, can actively adjust their resource demand and that they may have an approximate notion of their future resource demand. In chapter 2, it was argued that current approaches to resource management in operating systems do not leverage this potentially useful ability. Instead, resources are typically managed "magically" without notifying the resource consumers of changes in resource allocation or availability.

In contrast, I have proposed a radically different approach: allowing and encouraging resource-aware applications to manage resources themselves. Applications consume resources at a rate determined by their demand and are provided with feedback by the operating system. Applications interpret this feedback in order to adjust their resource demands. This effectively decentralises resource management as applications, rather than the operating system, make policy decisions on the allocation of resources between competing consumers. The operating system then only has to provide mechanisms for multiplexing resources and a feedback mechanism, but no resource allocation policies.

This decentralised resource management system has been described in microeconomic terms. Feedback is provided to applications by charging them for their resource consumption and applications are given the incentive to adjust their resource demand by limiting the credits available for paying these charges. Resource prices are based on congestion prices, which expose a consumer of a resource to the negative, external impact it causes on the other consumers of the same resource. In chapter 3 it has been argued that this is the correct microeconomic model for resources managed by an operating system. This argument is based on standard economic practice, which suggests that prices should be matched by cost, and a sound theoretical framework, which previously had been applied for managing resources in communication networks.

This dissertation has been focused on the operating system mechanisms required to decentralise the management of resources. I have identified four essential mechanisms: pricing & charging, credit accounts, resource accounting, and multiplexing.

In chapter 4 the two core mechanisms, pricing & charging and credit accounts, have been described in detail. The pricing mechanism has been identified as the most important one. Four different approaches for identifying the cost of congestion, and therefore the price, for resources managed by an operating system have been identified (request based, goal directed, queue based, and resource specific) and the applicability of these approaches to the various resources were discussed. A particular emphasis was placed on the management of CPU resources and a request based approach has been proposed for this particular resource: consumers request a share of the resource and prices are based on both resource utilisation and resource requests.

The management of credits is the second important mechanism. The availability of credits needs to be limited in order to assign a value to them. This provides consumers of resources with an incentive to adjust their resource demand when they incur congestion charges. To prevent arbitrary accumulation of unspent credits, a simple token bucket scheme for managing credits was introduced. It was argued that such a scheme provides some protection from ill-behaving consumers and that more complex, potentially user- or application-specific, credit allocation and management policies can be built on top of it at user-level.

The remaining two mechanisms, resource accounting and multiplexing, are not specific to the decentralised resource management framework. Resource accounting is an essential precondition to effectively manage resources and multiplexing falls in the well researched domain of scheduling algorithms. Prior work in these areas, applicable in the context of this dissertation, was reviewed extensively in chapter 2.

Chapter 4 also contains a discussion of possible application adaption strategies. This discussion presented two example strategies which are suitable for consumers wishing to maintain a charging rate close to their credit allocation — given the account mechanism, which is a rational objective.

In chapter 5 two prototype implementations of the congestion pricing framework for CPU resources were presented. A generic simulation environment for scheduling algorithms was described in detail. This simulator models the key abstractions of the Nemesis operating system and is suitable for implementing a large variety of scheduling algorithms, resource management mechanisms and policies, and both simple and realistic workloads. The implementation of congestion pricing is provided as a simple extension of the Nemesis task model and uses the Nemesis scheduler Atropos as the multiplexing mechanism. The second prototype is a modification of the FreeBSD kernel. This proof-of-concept implementation introduces a new scheduling class which allows processes to make resource requests and which provides feedback in the form of congestion charges. A key feature of this implementation is that only resource-aware applications need to actively participate in the decentralised resource management, while all other processes are managed by the default scheduling class as in an unmodified kernel.

Chapter 6 has provided a detailed evaluation of the decentralised resource management framework using both the simulator and the FreeBSD prototype. The results confirm the theoretical model, namely that, for elastic consumers, the decentralised approach, with independently acting consumers, can yield a socially optimal resource allocation where resources are shared in a weighted proportional manner. Furthermore, it has been demonstrated that the congestion pricing framework is suitable for a variety of non-elastic and even non-adaptive applications. Using a modified existing application, the multimedia system `mplayer`, it has been demonstrated that applications can benefit from the feedback provided by the operating system and provide better performance to end-users. And finally, the experimental results have demonstrated that both implementations, despite following radically different operating system design philosophies, are comparable. This supports the claim that the operating system mechanisms presented in this dissertation are generally applicable to the resources managed by operating systems.

7.2 Future Work

This dissertation is mainly concerned with operating system mechanisms to decentralise resource management and has mainly used a single resource, namely the CPU, for evaluation. There are two main directions in which this work can be extended — other resources and application-specific adaption strategies.

In section 4.2.2 the application of congestion pricing to resources other than the CPU was discussed. It was argued that, for all resources managed by an operating system, congestion and the cost of congestion can be identified. However, this discussion was oversimplified and left many implementation issues unanswered, such as timescales and shared resources. Future work could address these issues in greater detail. A further step in this direction would address the issues arising from handling multiple resources within the same decentralised framework. Some of these issues were discussed in section 4.5; however, detailed experimentation is required to acquire a deeper understanding of the issues involved.

Decentralising resource management enables resource-aware applications to perform application specific adaption to changing resource availability. The evaluation using the `mplayer` application provided some promising initial results of how applications can benefit from the feedback provided, however a more detailed study of these resource-aware applications is certainly very interesting, especially if all resources, managed by an operating system provide feedback signals to them. Such work could leverage some of the techniques developed in the area of QoS management (see section 2.2, in particular Q-RAM), applying them in the context of the applications themselves. Thus, it would address the problems with centralised QoS managers discussed in section 2.2.5.

A related area of future work is the development of appropriate user interfaces and user agents for the decentralised architecture. The command line tools and graphical user interfaces presented in section 5.2.6 only provide very rudimentary means for a user to adjust the credit allocations to his applications. More sophisticated interfaces could be developed, especially systems which would require less interaction from the end user. Examples range from a mechanism to specify sensible default values for certain applications to sophisticated user agents managing credit allocations on behalf of a user. These policies could be built on top of the presented operating system mechanisms and would not require changes to the general model described in this dissertation.

In terms of implementations, it would be advantageous to combine advanced resource management abstractions (described in section 2.3), such as resource containers or reservation domains, with congestion pricing. Such an approach would be especially beneficial, if not strictly necessary, when dealing with multiple resources, as traditional operating systems do not provide accurate enough accounting of resource consumptions to resource consumers. An implementation using the Nemesis operating system would be of particular interest, as Nemesis provides advanced resource management facilities for *all* operating system resources.

7.3 Conclusions

One of the most important tasks an operating system has to perform is the allocation of resources to competing consumers. It has been argued that existing approaches attempt to perform this task "magically" without involvement from the resource consumers. While this might be appropriate for a range of applications and utilities, it is the premise of this dissertation that a number of application domains exist where applications are resource-aware, i.e., applications from these domains can actively control their resource demands and might have an (implicit) notion of their future resource requirements. Example application domains are multimedia applications and a range of networked server applications. It is the thesis of this dissertation that getting these applications actively involved in the management of operating system resources is valuable and beneficial to the performance of these applications and their users. Therefore, this dissertation has presented operating system mechanisms which enable applications to actively manage resources themselves.

These mechanisms are based on the principled approach of congestion or shadow prices — a well understood microeconomic theory which also has been proposed to manage bandwidth and congestion in communication networks. The theoretical model of shadow prices and the results of its application to congestion pricing in networks suggest that, for congestable resources, independently and selfishly acting consumers can achieve a socially optimal resource allocation if the charges are based on the cost of resource congestion.

In this dissertation it has been demonstrated that operating system resources can become congested and that congestion pricing can be applied in this context. Congestion pricing can be implemented in an operating system and its implementation is inexpensive (in terms of performance overhead) and simple. Furthermore, the implementation provides the desired results, which are in line with the theoretical model: resources are shared in a weighted proportional fashion by independently acting consumers.

Applications are encouraged to adjust their resource demand by being charged for their resource consumption. Simple elastic applications can change their resource demand according to one of the charging rate controlling strategies presented. More complex, resource-aware applications can adjust their mode of operation in response to the feedback signals. Typically, expert developers of resource aware applications are best suited to decide on a specific application adaption strategy. However, for simple applications, a developer could choose a system-provided adaption strategy. Users are presented with an easy to understand interface; they simply assign credits to their consumers, in a similar way as they assign priorities to processes in traditional systems. However, more elaborate, user-friendly schemes can be built on top of the mechanisms provided by the decentralised architecture.

In conclusions, this dissertation presents and evaluates a novel, decentralised approach to resource management in operating systems which allows applications to actively participate in the management of resources. To the best of my knowledge it presents the first application and implementation of the general shadow pricing theory to resources managed by operating systems. It has been demonstrated that this approach is both viable and useful to applications, developers and users.

Bibliography

[ABLL92] Thomas E. Anderson, Brian N. Bershad, Edward D. Lazowska, and Henry M. Levy. Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism. *ACM Transactions on Computer Systems*, 10(1):53–79, February 1992. Also published in the Proc. of the 13th ACM SIGOPS Symposium on Operating Systems Principles (SOSP'91), pp. 95–109. (cited on pages: 37, 130)

[ACH98] Cristina Aurrecoechea, Andrew T. Campbell, and Linda Hauw. A survey of QoS architectures. *Multimedia Systems Journal*, 6(3):138–151, May 1998. (cited on pages: 30)

[AD00] Mohit Aron and Peter Druschel. Soft timers: efficient microsecond software timer support for network processing. *ACM Transactions on Computer Systems*, 18(3):197–228, August 2000. A shorter version has been published in the Proc. of the 17th ACM SIGOPS Symposium on Operating Systems Principles (SOSP'99), pp. 232–246. (cited on pages: 121)

[ADZ00] Mohit Aron, Peter Druschel, and Willy Zwaenepoel. Cluster Reserves: A Mechanism for Resource Management in Cluster-based Network Servers. In *Proceedings of the International Conference on Measurements and Modeling of Computer Systems*, Santa Clara, CA, USA, June 2000. (cited on pages: 40)

[Ago96] Agorics, Inc. Auctions – Going, Going, Gone! A Survey of Auction Types, 1996. http://www.agorics.com/auctions/auction1.html. (cited on pages: 47, 49)

[ATM99] The ATM Forum, Mountain View, CA, USA. *Traffic Management Specification Version 4.1*, March 1999. (cited on pages: 59)

[Axe90] Robert Axelrod. *The Evolution of Co-operation*. Penguin Books, April 1990. (cited on pages: 4)

[Bar96] Paul R. Barham. *Devices in a Multi-Service Operating System.* PhD thesis, University of Cambridge Computer Laboratory, October 1996. Available as Technical Report No. 403. (cited on pages: 37, 38, 40)

[Bar97] Paul R. Barham. A fresh approach to File System Quality of Service. In *Proceedings of the 7th International Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV'97)*, St. Louis, MO, USA, May 1997. (cited on pages: 38, 79)

[Bar98] Paul Barham. Atropos improvements. Posted to Nemesis.Live Newsgroup, June 1998. see also ntsc/generic/atropos.c. (cited on pages: 19, 104)

[BBDS97] Richard Black, Paul Barham, Austin Donnelly, and Neil Stratford. Protocol Implementation in a Vertically Structured Operating System. In *Proceedings of the 22nd Annual Conference on Local Computer Networks (LCN'97)*, pages 179–188, November 1997. (cited on pages: 38, 79)

[BBG$^+$99a] J. Blanquer, J. Bruno, E. Gabber, M. Mcshea, B. Özden, and A. Silberschatz. Resource Management for QoS in Eclipse/BSD. In *Proceedings of the FreeBSD 1999 Conference,*, Berkeley, California, October 1999. (cited on pages: 41)

[BBG$^+$99b] John Bruno, José Brustoloni, Eran Gabber, Banu Özden, and Abraham Silberschatz. Disk Scheduling with Quality of Service Guarantees. In *Proceedings of the International Conference on Multimedia Computing and Systems*, Centro Affari, Florence, ITALY, June 1999. (cited on pages: 42)

[BBG$^+$99c] John Bruno, José Brustoloni, Eran Gabber, Banu Özden, and Abraham Silberschatz. Retrofitting Quality of Service into a Time-Sharing Operating System. In *Proceedings of the USENIX 1999 Annual Technical Conference*, pages 15–26, Monterey, CA, USA, June 1999. (cited on pages: 41, 123, 132)

[BCC$^+$94] Gordon Blair, Andrew Campbell, Geoff Coulson, David Hutchison, Michael Papathomas, and Philippe Robin. On the Implementation of a Quality of Service Controlled ATM Based Communications System in Chorus. Technical Report MPG-94-11, Distributed Multimedia Research Group, Department of Computing, Lancaster University, 1994. (cited on pages: 31)

[BDM99] Guarav Banga, Peter Druschel, and Jeffrey C. Mogul. Resource Containers: A new facility for resource management in server systems. In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation (OSDI'99)*, pages 45–58, New Orleans, LA, USA, February 1999. (cited on pages: 39, 40, 80, 123)

[BGÖS97] J. Bruno, E. Gabber, B. Özden, and A. Silberschatz. Move-To-Rear List Scheduling: a new scheduling algorithm for providing QoS guarantees. In *Proceedings of the 5th ACM International Multimedia Conference*, pages 63–73, Seattle, WA, USA, November 1997. (cited on pages: 42)

[BGÖS98] John Bruno, Eran Gabber, Banu Özden, and Abraham Silberschatz. The eclipse operating system: Providing quality of service via reservation domains. In *Proceedings of the USENIX 1998 Annual Technical Conference*, New Orleans, LA, USA, June 1998. (cited on pages: 41)

[BGSS99] José Brustoloni, Eran Gabber, Abraham Silberschatz, and Amit Singh. Quality of service support for legacy applications. In *Proceedings of the 9th International Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV'99)*, pages 3–11, Basking Ridge, NJ, USA, June 1999. (cited on pages: 41, 171)

[BJ95] G. Bollella and K. Jeffay. Support For Real-Time Computing Within General Purpose Operating Systems: Supporting co-resident operating systems. In *Proceedings of the 1st IEEE Real-Time Technology and Applications Symposium*, pages 4–14, Chicago, IL USA, May 1995. (cited on pages: 26)

[Bla95] Richard J. Black. *Explicit Network Scheduling.* PhD thesis, University of Cambridge Computer Laboratory, April 1995. Available as Technical Report no. 361. (cited on pages: 101)

[BMP98] A. Bavier, B. Montz, and L. Peterson. Predicting MPEG Execution Time. In *Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems*, pages 131–140, Madison, WI, USA, June 1998. (cited on pages: 35, 106)

[BN98] Li Boachun and Klara Nahrstedt. A Control Theoretical Model for Quality of Service Adaptations. In *Proceedings of the 6th International Workshop on Quality of Service*, pages 145–153, Napa Valey, CA, USA, May 1998. (cited on pages: 27)

[Bog94] Nathaniel R. Bogan. Economic Allocation of Computation Time with Computational Markets. Master's thesis, Department of Electrical Engineering and Computer Science, MIT, May 1994. (cited on pages: 47, 54, 55)

[BP00] A. Bavier and L. Peterson. The Power of Virtual Time for Multimedia Scheduling. In *Proceedings of the 10th International Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV'00)*, Chapel Hill, NC, USA, June 2000. (cited on pages: 24)

[BPM99] A. Bavier, L. Peterson, and D. Mosberger. BERT: A Scheduler for Best Effort and Real-time Tasks. Technical Report TR-602-99, Department of Computer Science, Princeton University, March 1999. (cited on pages: 20, 23, 24)

[BZ96] Jon Bennett and Hui Zhang. WF^{2}Q: Worst-case Fair Queueing. In *Proceedings of INFO-COM'96*, San Francisco, CA, USA, March 1996. (cited on pages: 20, 22, 42)

[BZ97] Jon Bennett and Hui Zhang. Hierarchical Packet Fair Queueing Algorithms. *IEEE/ACM Transactions on Networking*, 5(5):675–689, October 1997. (cited on pages: 22, 23)

[CAT$^+$01] J. Chase, D. Anderson, P. Thakar, A. Vahdat, and R. Doyle. Managing Energy and Server Resources in Hosting Centers. In *Proceedings of the 18th ACM SIGOPS Symposium on Operating Systems Principles (SOSP'01)*, pages 103–116, Chateau Lake Louise, Banff, Canada, October 2001. (cited on pages: 7, 40, 96)

[CBRS93] Geoff Coulson, Gordon Blair, Philippe Robin, and Doug Shepherd. Extending the Chorus Micro-kernel to Support Continuous Media Applications. In *Proceedings of the 4th International Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV'93)*, pages 49–60, Lancaster, UK, November 1993. (cited on pages: 31)

[CC00] Brent Chun and David Culler. Market-based Proportional Resource Sharing for Clusters. Technical Report UCB-CSD-1092, University of California at Berkeley, Computer Science Division, January 2000. (cited on pages: 96)

[CCG$^+$93] Andrew Campbell, Geoff Coulson, Francisco Garcia, David Hutchinson, and Helmut Leopold. Integrated Quality of Service for Multimedia Communications. In *Proceedings of INFOCOM'93*, San Francisco, CA, USA, March 1993. (cited on pages: 31)

[CCH94] Andrew Campbell, Geoff Coulson, and David Hutchison. A Quality of Service Architecture. *ACM Computer Communication Review*, 24(2):6–27, 1994. (cited on pages: 31)

[CJ98] George M. Candea and Michael B. Jones. Vassal: Loadable Scheduler Support for Multi-Policy Scheduling. In *Proceedings of the 2nd USENIX Windows NT Symposium*, Seattle, WA, USA, August 1998. (cited on pages: 13, 26)

[CKSW98] C. Courcoubetis, F. P. Kelly, V. A. Siris, and R. Weber. A Study of Simple Usage-based Charging Schemes for Broadband Networks. In *Proceedings of IFIP TC6 International Conference on Broadband Communications (BC'98)*, Stuttgart, Germany, April 1998. (cited on pages: 63, 64)

[CKW97] C. Courcoubetis, F. P. Kelly, and R. R. Weber. Measurement-based charging in communication networks. Technical Report 1997-19, University of Cambridge Statistical Laboratory, 16 Mill Lane, Cambridge CB2 1SB, UK, 1997. (cited on pages: 63)

[CS98] C. Courcoubetis and V. A. Siris. An Evaluation of Pricing Schemes that are based on Effective Usage. In *Proceedings of IEEE International Conference on Communications (ICC'98)*, Atlanta, GA, USA, June 1998. (cited on pages: 63, 64)

[CSS96] C. Courcoubetis, V. A. Siris, and G. D. Stamoulis. Integration of Pricing and Flow Control for Available Bit Rate Services in ATM Networks. In *Proceedings of IEEE Globecom' 96*, London, U.K., November 1996. (cited on pages: 62)

[CT93] David Clarke and Brendan Tangney. Microeconomic theory applied to distributed systems. Technical Report TCD-CS-93-30, Department of Computer Science, Trinity College, Dublin, Trinity College, Dublin, Ireland, December 1993. (cited on pages: 46)

[CW98] John Q. Cheng and Michael P. Wellman. The WALRAS Algorithm: A Convergent Distributed Implementation of General Equilibrium Outcomes. *Computational Economics*, 12(1):1–12, August 1998. (cited on pages: 53)

[DB96] Peter Druschel and Guarav Banga. Lazy Receiver Processing (LRP): A Network Subsystem Architecture for Server Systems. In *Proceedings of the 2nd Symposium on Operating Systems Design and Implementation (OSDI'96)*, Seattle, WA, USA, October 1996. (cited on pages: 40, 79)

[DC99] Kenneth J. Duda and David R. Cheriton. Borrowed-Virtual-Time (BVT) scheduling: supporting latency-sensitive threads in a general-purpose scheduler. In *Proceedings of the 17th ACM SIGOPS Symposium on Operating Systems Principles (SOSP'99)*, pages 261–276, Kiawah Island Resort, SC, USA, December 1999. (cited on pages: 20, 22, 23, 24, 26, 104)

[DKS90] Alan Demers, Srinivasan Keshav, and Scott Shenker. Analysis and Simulation of a Fair Queueing Algorithm. *Journal of Internetworking: Research and Experience*, 1:3–26, 1990. Also in Proc. of SIGCOMM'89, pp 3–12. (cited on pages: 20, 21)

[Ell99] Carla Ellis. The Case for Higher-Level Power Management. In *Proceedings of the 7th Workshop on Hot Topics in Operating Systems*, Rio Rico, AZ, USA, March 1999. (cited on pages: 80)

[Eva00] Jason Evans. Kernel-Scheduled Entities for FreeBSD. http://people.freebsd.org/ jasone/kse/, November 2000. (cited on pages: 130)

[FJ93] S. Floyd and V. Jacobson. Random Early Detection Gateways for Congestion Avoidance. *IEEE/ACM Transactions on Networking*, 1(4):397–413, August 1993. (cited on pages: 75)

[Flo94] Sally Floyd. TCP and Explicit Congestion Notification. *ACM Computer Communication Review*, 24(5):10–23, October 1994. (cited on pages: 62, 75)

[FP97] Norival Figueira and Joseph Pasquale. A Schedulability Condition for Deadline-Ordered Service Disciplines. *IEEE/ACM Transactions on Networking*, 4(2):232–244, April 1997. (cited on pages: 24)

[FS96] Bryan Ford and Sai Susarla. CPU Inheritance Scheduling. In *Proceedings of the 2nd Symposium on Operating Systems Design and Implementation (OSDI'96)*, pages 91–105, Seattle, WA, USA, October 1996. (cited on pages: 26, 27)

[FS99] Jason Flinn and M. Satyanarayanan. Energy-aware adaptation for mobile applications. In *Proceedings of the 17th ACM SIGOPS Symposium on Operating Systems Principles (SOSP'99)*, pages 48–79, Kiawah Island Resort, SC, USA, December 1999. (cited on pages: 80, 81)

[GGV96] Pawan Goyal, Xingang Guo, and Harrick M. Vin. A Hierachichal CPU Scheduler for Multimedia Operating Systems. In *Proceedings of the 2nd Symposium on Operating Systems Design and Implementation (OSDI'96)*, pages 107–121, Seattle, WA, USA, October 1996. (cited on pages: 20, 23, 24, 26)

[GK99a] R. Gibbens and P. Key. The use of games to assess user strategies for differential Quality of Service in the Internet. In *Proceedings of the Workshop on Internet Service Quality Economics (ISQE)*, Cambridge, MA, USA, December 1999. (cited on pages: 87)

[GK99b] R. J. Gibbens and F. P. Kelly. Resource pricing and the evolution of congestion control. *Automatica*, 35(12), 1999. (cited on pages: 59, 61, 62, 73, 81, 86)

[GK01] Richard Gibbens and Peter Key. Distributed Control and Resource Marking Using Best-Effort Routers. *IEEE Network*, pages 54–59, May/June 2001. (cited on pages: 63, 75)

[GKT00] R. Gibbens, P. Key, and S. Turner. Properties of the Virtual Queue Marking Algorithm. In *Proceedings of the 7th UK Teletraffic Symposium*, Dublin, Ireland, May 2000. (cited on pages: 63, 75)

[GLS00] Ayalvadi Ganesh, Koenraad Laevens, and Richard Steinberg. Dynamics of congestion pricing. Technical Report MSR-TR-2000-70, Microsoft Research, Cambridge, U.K., June 2000. (cited on pages: 63)

[GVC96] Pawan Goyal, Harrick M. Vin, and Haichen Chen. Start-time fair queueing: a scheduling algorithm for integrated services packet switching networks. In *Proceedings of the ACM SIGCOMM Annual Technical Conference*, pages 157–168, Palo Alto, CA USA, August 1996. (cited on pages: 22, 23)

[Han99a] Steven Hand. *Providing Quality of Service in Memory Management.* PhD thesis, University of Cambridge Computer Laboratory, November 1999. (cited on pages: 78, 114)

[Han99b] Steven Hand. Self-Paging in the Nemesis Operating System. In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation (OSDI'99)*, pages 73–86, New Orleans, LA, USA, February 1999. (cited on pages: 38, 78)

[Har68] Garret Hardin. The tragedy of the commons. *Science*, 162:1243–1248, December 1968. (cited on pages: 47)

[HHKP03] Steven Hand, Tim Harris, Evangelos Kotsovinos, and Ian Pratt. Controlling the XenoServer Open Platform. In *Proceedings of IEEE OPENARCH'03*, San Francisco, CA, USA, April 2003. (cited on pages: 94, 96)

[Hyd94] Eoin Hyden. *Operating System Support for Quality of Service.* PhD thesis, University of Cambridge Computer Laboratory, February 1994. Available as Technical Report No. 340. (cited on pages: 15, 18, 197)

[IBM01] IBM. Partitioning for the IBM Eserver pSeries 690 System. White paper, IBM Corp., October 2001. (cited on pages: 7)

[ID01] Sitaram Iyer and Peter Druschel. Anticipatory scheduling: A disk scheduling framework to overcome deceptive idleness in synchronous I/O. In *Proceedings of the 18th ACM SIGOPS Symposium on Operating Systems Principles (SOSP'01)*, pages 117–130, Chateau Lake Louise, Banff, Canada, October 2001. (cited on pages: 78)

[Int99] Intel Corporation and Microsoft Corporation and Toshiba Corporation. *Advanced Configuration and Power Interface Specification (ACPI)*, February 1999. Revision 1.0b. (cited on pages: 81)

[ISO96] ISO/IEC. ISO/IEC 9945-1: Information Technology – Portable Operating System Interface (POSIX), Part 1: System Application Program Interface (API) [C Language]. International Standard, July 1996. (cited on pages: 14)

[Jai91] Raj Jain. *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, and Modelling.* John Wiley & Son, 1991. (cited on pages: 148)

[JLDB95] M. B. Jones, P. J. Leach, R. Draves, and J. S. Barrera. Modular Real-Time Resource Management in the Rialto Operating System. In *Proceedings of the 5th Workshop on Hot Topics in Operating Systems*, Orcas Island, WA, USA, May 1995. (cited on pages: 36)

[JR00] Michael B. Jones and John Regehr. Predictable Scheduling for Digital Audio. Technical Report MSR-TR-2000-87, Microsoft Research, December 2000. (cited on pages: 29)

[JRR97] Michael B. Jones, Daniela Rosu, and Marcel-Catalin Rosu. CPU Reservations and Time Constraints: Efficient, Predictable Scheduling of Independent Activities. In *Proceedings of the 16th ACM SIGOPS Symposium on Operating Systems Principles (SOSP'97)*, pages 198–211, Saint-Malo, France, October 1997. (cited on pages: 19, 43)

[JS01] Michael B. Jones and Stefan Saroiu. Predictability Requirements of a Soft Modem. In *Proceedings of the International Conference on Measurements and Modeling of Computer Systems*, Cambridge, MA, USA, June 2001. (cited on pages: 29)

[JSMA98] K. Jeffay, F. D. Smith, A. Moorthy, and J. H. Anderson. Proportional Share Scheduling of Operating System Services for Real-Time Applications. In *Proceedings of the 19th IEEE Real-Time Systems Symposium*, pages 480–491, Madrid, Spain, December 1998. (cited on pages: 20)

[KEG$^+$97] Frans Kaashoek, Dawson Engler, Gregory Ganger, Héctor Briceño, Russel Hunt, David Mazières, Thomas Pinckney, Robert Grimm, John Jannotti, and Kenneth Mackenzie. Application Performance and Flexibilty on Exokernel Systems. In *Proceedings of the 16th ACM SIGOPS Symposium on Operating Systems Principles (SOSP'97)*, pages 52–65, Saint-Malo, France, October 1997. (cited on pages: 37)

[Kel96] Frank P. Kelly. Notes on effective bandwidths. In F. P. Kelly, S. Zachary, and I. B. Ziedins, editors, *Stochastic Networks: Theory and Applications*, number 4 in Royal Statistical Society Lecture Note Series, pages 141–168. Oxford University Press, September 1996. (cited on pages: 63, 64)

[Kel97a] F. P. Kelly. Charging and accounting for bursty connections. In Lee W. McKnight and Joseph P. Bailey, editors, *Internet Economics*, pages 253–278. MIT Press, Cambridge, MA, 1997. (cited on pages: 63, 64, 77)

[Kel97b] Frank P. Kelly. Charging and rate control for elastic traffic. *European Transactions on Telecommunications*, 8:33–37, 1997. (cited on pages: 59, 60)

[Key01] Peter Key. Resource Pricing for Differentiated Services. Invited talk at GI Fachtagung Kommunikation in Verteilten Systemen, February 2001. Paper available at http://research.microsoft.com/users/pbk/. (cited on pages: 59, 86)

[KM99] Peter Key and Derek McAuley. Differential QoS and Pricing in Networks: where flow-control meets game theory. *IEE Proceedings Software*, 146(2), March 1999. (cited on pages: 62, 69, 87)

[KMBL99] Peter Key, Derek McAuley, Paul Barham, and Koenraad Laevens. Congestion pricing for congestion avoidance. Technical Report MSR-TR-99-15, Microsoft Research, Cambridge, U.K., February 1999. (cited on pages: 61, 62, 66, 70, 73, 84)

[KMT98] Frank Kelly, Aman Maulloo, and David Tan. Rate control in Communication Networks: Shadow Prices, Proportional Fairness and Stability. *Journal of the Operational Research Society*, 49(3):237–252, March 1998. (cited on pages: 59, 60, 61, 69, 70, 84)

[KN97] Kiwook Kim and Klara Nahrstedt. QoS Translation and Admission Control for MPEG Video. In *Proceedings of the 5th International Workshop on Quality of Service*, New York, NY, USA, May 1997. (cited on pages: 33)

[KN01] Minkyong Kim and Brian Noble. Mobile Network Estimation. In *Proceedings of the ACM Conference on Mobile Computing and Networking*, Rome, Italy, June 2001. (cited on pages: 111, 116)

[KS01a] S. Kunniyur and R. Srikant. A Time Scale Decomposition Approach to Adaptive ECN Marking. In *Proceedings of INFOCOM'01*, Anchorage, Alaska, April 2001. (cited on pages: 63)

[KS01b] S. Kunniyur and R. Srikant. Analysis and Design of an Adaptive Virtual Queue (AVQ) Algorithm for Active Queue Management. In *Proceedings of the ACM SIGCOMM Annual Technical Conference*, San Diego, CA, USA, August 2001. (cited on pages: 63)

[Lar75] J. Larmouth. Scheduling for a Share of the Machine. *Software Practice and Experience*, 5:29–49, 1975. Also available as University of Cambridge Computer Laboratory Technical Report no. 2. (cited on pages: 2)

[LCC$^+$75] R. Levin, E. Cohen, W. Corwin, F. Pollack, and W. Wulf. Policy/mechanism separation in Hydra . In *Proceedings of the 5th ACM SIGOPS Symposium on Operating Systems Principles (SOSP'75)*, pages 132–140, Austin, TX, USA, November 1975. (cited on pages: 92)

[Lee99] C. Lee. *On Quality of Service Management.* PhD thesis, Department of Computer Science, Carnegie Mellon University, August 1999. Also availble as technical report CMU-CS-99-165. (cited on pages: 33, 34)

[LG91] Didier Le Gall. MPEG: a video compression standard for multimedia applications. *Communications of the Association for Computing Machinery*, 34(4):46–58, April 1991. (cited on pages: 105)

[LL73] C. L. Liu and James W. Layland. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *Journal of the Association for Computing Machinery*, 20(1):46–61, January 1973. (cited on pages: 15, 16, 72, 104)

[LLRS99] Chen Lee, John Lehoczky, Raj Rajkumar, and Dan Siewiorek. On Quality of Service Optimization with Discrete QoS Options. In *Proceedings of the 5th IEEE Real-Time Technology and Application Symposium*, Vancouver, Canada, June 1999. (cited on pages: 33)

[LLS$^+$91] Jane W. S. Liu, Kwei-Jay Lin, Wei-Kuan Shih, Albert Chuang shi Yu, Jen-Yao Chung, and Wei Zhao. Algorithms for scheduling imprecise computations. *IEEE Computer*, 24(5):58–68, May 1991. (cited on pages: 28)

[LLS$^+$99] Chen Lee, John Lehoczky, Dan Siewiorek, Raj Rajkumar, and Jeff Hansen. A Scalable Solution to the Multi-Resource QoS Problem. In *Proceedings of the 20th IEEE Real-Time Systems Symposium*, Phoenix, Arizona, USA, December 1999. (cited on pages: 33)

[LMB$^+$96] I. M. Leslie, D. McAuley, R. Black, T. Roscoe, P. Barham, D. Evers, R. Fairbairns, and E. Hyden. The Design and Implementation of an Operating System to Support Distributed Multimedia Applications. *IEEE Journal on Selected Areas In Communications*, 14(7):1280–1297, September 1996. (cited on pages: 19, 26, 27, 37, 38, 100)

[LSA$^+$00] C. Lu, J. Stankovic, T. Abdelzaher, G. Tao, S. Son, and M. Marley. Performance Specification and Metrics for Adaptive Real-Time Systems. In *Proceedings of the 21th IEEE Real-Time Systems Symposium*, Disney World, Orlando, Florida, USA, December 2000. (cited on pages: 28)

[LSD89] J. P. Lehoczky, L. Sha, and Y. Ding. The Rate Monotonic Scheduling Algorithm: Exact Characterization and Average Case Behaviour. In *Proceedings of the 10th IEEE Real-Time Systems Symposium*, pages 166–171, December 1989. (cited on pages: 16)

[LSTS99] C. Lu, J. Stankovic, G. Tao, and S. Son. Design and Evaluation of a Feedback Control EDF Scheduling Algorithm. In *Proceedings of the 20th IEEE Real-Time Systems Symposium*, Phoenix, Arizona, USA, December 1999. (cited on pages: 27, 28, 87)

[LY96] K. Lakshman and R. Yavatkar. AQUA: An Adaptive End-System Quality of Service Architecture. In W. Effelsberg, O. Spaniol, A. Danthine, and D. Ferrari, editors, *High-Speed Networking for Multimedia Applications*. Kluwer Academic Publishers, 1996. (cited on pages: 34)

[LYF97] K. Lakshman, R. Yavatkar, and R. Finkel. Integrated CPU and Network-I/O QoS Management in an Endsystem. In *Proceedings of the 5th International Workshop on Quality of Service*, New York, NY, USA, May 1997. (cited on pages: 34)

[MBKQ96] M. K. McKusick, K. Bostic, M. J. Karels, and J. S. Quaterman. *The Design and Implementation of the 4.4 BSD Operation System*. Addison Wesley, Reading, MA, USA, 1996. (cited on pages: 12, 78, 119)

[McD01] Ian McDonald. *Memory Management in a Distributed System of Single Address Space Operating Systems supporting Quality of Service*. PhD thesis, Department of Computing Science, University of Glasgow, 2001. (cited on pages: 78)

[MD88a] M. S. Miller and K. E. Drexler. Comparative ecology: A computational perspective. In B. A. Huberman, editor, *The ecology of Computation*, pages 51–76. North-Holland, Amsterdam, Netherlands, 1988. (cited on pages: 51)

[MD88b] M. S. Miller and K. E. Drexler. Incentive engineering for computation resource management. In B. A. Huberman, editor, *The ecology of Computation*, pages 231–266. North-Holland, Amsterdam, Netherlands, 1988. (cited on pages: 47, 51)

[MD88c] M. S. Miller and K. E. Drexler. Markets and computation: Agoric open systems. In B. A. Huberman, editor, *The ecology of Computation*, pages 133–176. North-Holland, Amsterdam, Netherlands, 1988. (cited on pages: 51)

[Mic01] Microsoft. *Platform SDK: DLL's, Processes and Threads*, November 01. (cited on pages: 13)

[Mil90] F. Miller. Predictive Deadline Multi-Processing. *ACM Operating Systems Review*, 24(4):52–63, October 1990. (cited on pages: 17)

[MKH+96] Mark S. Miller, David Krieger, Norman Hardy, Chris Hibbert, and E. Dean Tribble. An Automated Auction in ATM Network Bandwidth. In S. Clearwater, editor, *Market-Based Control: A Paradigm for Distributed Resource Allocation*, pages 96–125. World Scientific, 1996. (cited on pages: 47, 51, 52)

[Mos97] David Mosberger. Linux/Alpha or How to Make Your Applications Fly. In *Proceedings of the Linux Expo '97,*, Chapel Hill, NC, USA, March 1997. (cited on pages: 106)

[MP96] David Mosberger and Larry L. Peterson. Making paths explicit in the scout operating system. In *Proceedings of the 2nd Symposium on Operating Systems Design and Implementation (OSDI'96)*, pages 153–167, Seattle, WA, USA, October 1996. (cited on pages: 23)

[MR97] Jeffrey C. Mogul and K. K. Ramakrishnan. Eliminating receive livelock in an interrupt-driven kernel. *ACM Transactions on Computer Systems*, 15(3):217–252, August 1997. A shorter version has been published in the Proc. of the USENIX 1996 Annual Technical Conference, pp. 99-111. (cited on pages: 40)

[MS96] L. McVoy and C. Staelin. lmbench: Portable Tools for Performance Analysis. In *Proceedings of the USENIX 1996 Annual Technical Conference*, pages 279–295, San Diego, CA, USA, January 1996. (cited on pages: 126)

[MST94] C. W. Mercer, S. Savage, and H. Tokuda. Processor Capacity Reserves for Multimedia Operating Systems. In *Proceedings of the IEEE International Conference on Multimedia Computing and Systems*, May 1994. (cited on pages: 18, 40, 41)

[MT97] David Mosberger-Tang. SCOUT: A path-based operating system. Technical Report TR97-06, The Department of Computer Science, University of Arizona, May 1997. (cited on pages: 23)

[MTHH97] Mark S. Miller, E. Dean Tribble, Norman Hardy, and Chris Hibbert. Diverse goods arbitration system and method for allocating resources in a distributed computer system. US Patent No: 5640569, June 1997. Filed 28/04/1995. (cited on pages: 51, 52)

[MV95a] Jeffrey K. MacKie-Mason and Hal R. Varian. Pricing Congestable Network Resources. *IEEE Journal on Selected Areas In Communications*, 13(7):1141–1149, September 1995. (cited on pages: 7, 57, 58, 83)

[MV95b] Jeffrey K. MacKie-Mason and Hal R. Varian. Pricing the Internet. In B. Kahin and J. Keller, editors, *Public Access to the Internet*, pages 269–314. MIT Press, Cambridge, MA, USA, 1995. (cited on pages: 46, 49, 57, 58)

[MV95c] Jeffrey K. MacKie-Mason and Hal R. Varian. Some FAQs about Usage-Based Pricing. *Computer Networks and ISDN Systems*, 28:257–265, 1995. (cited on pages: 56)

[MV96] Jeffrey K. MacKie-Mason and Hal R. Varian. Some Economics of the Internet. In W. Sichel, editor, *Networks, Infrastructure and the New Task for Regulation*. University of Michigan Press, Ann Arbor, 1996. (cited on pages: 46)

[MV97] Jeffrey K. MacKie-Mason and Hal R. Varian. Economic FAQs About the Internet. In Lee W. McKnight and Joseph P. Bailey, editors, *Internet Economics*, pages 27–62. MIT Press, Cambridge, MA, 1997. (cited on pages: 66)

[Nag87] J. Nagle. On Packet Switches with Infinite Storage. *IEEE Transactions on Communications*, 35:435–438, April 1987. (cited on pages: 20)

[NB98] Rolf Neugebauer and Richard Black. Stateful API Architecture. PEGASUS II Technical Report 4.5.1, Department of Computing Science, University of Glasgow, September 1998. (cited on pages: 39)

[NB00] Rolf Neugebauer and Richard Black. Supporting Foreign Personalities in a Single Address Space Operating System. In *Proceedings of the 3rd ECOOP Workshop on Object-Oriented Operating Systems*, pages 42–48, Sophia-Antipolis, France, June 2000. (cited on pages: 39)

[ND99] Rolf Neugebauer and Michael Dales. Unix functionality. PEGASUS II Technical Report 4.5.2, Department of Computing Science, University of Glasgow, March 1999. (cited on pages: 39)

[NDH$^+$99] Rolf Neugebauer, Michael Dales, Matthew Holgate, Neetu Nangia, and Richard Black. A Unix-like personality supporting Quality of Service. Poster presented at the 17th ACM SIGOPS Symposium on Operating Systems Principles (SOSP'99), Kiawah Island Resort, SC, USA, December 1999. (cited on pages: 39)

[Neu99] Rolf Neugebauer. How Elastic are Real Applications? In *Proceedings of the 9th International Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV'99)*, pages 197–200, Basking Ridge, NJ, USA, June 1999. (cited on pages: i, 107, 134, 172)

[Neu01] Rolf Neugebauer. Decentralising Resource Management: No Policy — Just Mechanisms. Poster presented at the 18th ACM SIGOPS Symposium on Operating Systems Principles (SOSP'01), Chateau Lake Louise, Banff, Canada, December 2001. (cited on pages: i)

[NH99] Neetu Nangia and Matt Holgate. Xlib functionality. PEGASUS II Technical Report 4.5.3, Department of Computing Science, University of Glasgow, March 1999. (cited on pages: 39)

[NhCN98] Klara Nahrstedt, Hao hua Chu, and Srinivas Naraya. QoS-aware Resource Management for Distributed Multimedia Applications. *Journal on High-Speed Networking, Special Issue on Multimedia Networking*, 8(3-4):227–255, 1998. (cited on pages: 31)

[NHK96] Klara Nahrstedt, Ashfaq Hossain, and Sung-Mo Kang. A Probe-based Algorithm for QoS Specification and Adaptation. In *Proceedings of the 4th International Workshop on Quality of Service*, pages 89–100, Paris, France, March 1996. (cited on pages: 32)

[NHNW93] J. Nieh, J. G. Hanko, J. D. Northcutt, and G. A. Wall. SVR4 UNIX Scheduler Unacceptable for Multimedia Applications. In *Proceedings of the 4th International Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV'93)*, Lancaster, UK, November 1993. (cited on pages: 14, 26, 29, 119, 121, 170)

[Nie70] Norman Nielsen. The Allocation of Computer Resources — Is pricing the Answer? *Communications of the Association for Computing Machinery*, 13(8):467–474, August 1970. (cited on pages: 2)

[NL97] Jason Nieh and Monica S. Lam. The Design, Implementation and Evaluation of SMART: A Scheduler for Multimedia Applications. In *Proceedings of the 16th ACM SIGOPS Symposium on Operating Systems Principles (SOSP'97)*, pages 184–197, Saint-Malo, France, October 1997. (cited on pages: 12, 20, 24, 72, 197)

[NM00] Rolf Neugebauer and Derek McAuley. Congestion Prices as Feedback Signals: An Approach to QoS Management. In *Proceedings of the 9th ACM SIGOPS European Workshop*, pages 91–96, Kolding, Denmark, September 2000. (cited on pages: i)

[NM01] Rolf Neugebauer and Derek McAuley. Energy is just another resource: Energy accounting and energy pricing in the Nemesis OS. In *Proceedings of the 8th Workshop on Hot Topics in Operating Systems*, pages 59–64, Schloss Elmau, Germany, May 2001. (cited on pages: i, 38, 80)

[NS95] Klara Nahrstedt and Jonathan Smith. The QoS Broker,. *IEEE Multimedia Magazine*, 2(1):53–67, 1995. (cited on pages: 31)

[NS96] Klara Nahrstedt and Jonathan Smith. Design, Implementation and Experiences with the OMEGA End-point Architercture. *IEEE Journal on Selected Areas In Communications*, 14(7), September 1996. (cited on pages: 31)

[Odl01] Andrew Odlyzko. Internet pricing and the history of communications. *Computer Networks (Amsterdam, Netherlands: 1999)*, 36(5–6):493–517, 2001. (cited on pages: 65)

[OR98] Shui Oikawa and Raj Rajkumar. Linux/RK: A Portable Resource Kernel in Linux. In *Proceedings of the 19th IEEE Real-Time Systems Symposium*, Madrid, Spain, December 1998. (cited on pages: 33, 40)

[Par94] Craig Partridge. *Gigabit Networking.* Addison-Wesley, Reading, MA, USA, 1994. (cited on pages: 84, 85)

[PD00] Larry L. Peterson and Bruce S. Davie. *Computer networks: A systems approach.* Morgan Kaufman Publishers, San Francisco, CA, USA, 2nd edition, 2000. (cited on pages: 59)

[PDP93] F. Panzieri, L. Donatiello, and L. Poretti. Real Time Systems: A Tutorial. Technical Report UBLCS-93-22, Laboratory of Computer Science, University of Bologna, Piazza di Porta S. Donata, 5, 40127 Bologna, Italy, October 1993. (cited on pages: 14, 15, 17)

[PG93] Abhay K. Parekh and Robert G. Gallager. A generalized processor sharing approach to flow control in integrated services networks: the single-node case. *IEEE/ACM Transactions on Networking*, 1(3):344–357, June 1993. (cited on pages: 20, 21)

[PMG99] David Petrou, John W. Milford, and Garth A. Gibson. Implementing Lottery Scheduling: Matching the Specializations in Traditional Schedulers. In *Proceedings of the USENIX 1999 Annual Technical Conference*, pages 1–14, Monterey, CA, USA, June 1999. (cited on pages: 85)

[Pra97] Ian Pratt. *The User-Safe Device I/O Architecture.* PhD thesis, University of Cambridge Computer Laboratory, August 1997. (cited on pages: 29)

[Ree98] Dickon Reed. A new audio device driver abstraction. In *Proceedings of the 8th International Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV'98)*, pages 87–90, Cambridge, UK, July 1998. (cited on pages: 38)

[RJMO98] R. Rajkumar, K. Juvva, A. Molano, and S. Oikawa. Resource Kernels: A Resource-Centric Approach to Real-Time Systems. In *Proceedings of the SPIE/ACM Conference on Multimedia Computing and Networking*, January 1998. (cited on pages: 33, 40)

[RJS00] John Regehr, Michael Jones, and John Stankovic. Operating System Support for Multimedia: The Programming Model Matters. Technical Report MSR-TR-2000-89, Microsoft Research, Redmond, WA, USA, September 2000. (cited on pages: 11)

[RK79] David P. Reed and Rajendra K. Kanodia. Synchronization with eventcounts and sequencers. *Communications of the Association for Computing Machinery*, 22(2):115–123, February 1979. (cited on pages: 101)

[RLLS97] R. Rajkumar, C. Lee, J. Lehoczky, and D. Siewiorek. A Resource Allocation Model for QoS Management. In *Proceedings of the 18th IEEE Real-Time Systems Symposium*, San Francisco, CA, USA, December 1997. (cited on pages: 33)

[RLLS98] Raj Rajkumar, Chen Lee, John Lehoczky, and Dan Siewiorek. Practical Solutions for QoS-based Resource Allocation Problems. In *Proceedings of the 19th IEEE Real-Time Systems Symposium*, Madrid, Spain, December 1998. (cited on pages: 33)

[Ros95] Timothy Roscoe. *The Structure of a Multi-Service Operating System.* PhD thesis, University of Cambridge Computer Laboratory, August 1995. Available as Technical Report no. 376. (cited on pages: 19, 101, 104)

[RPM$^+$99] Dickon Reed, Ian Pratt, Paul Menage, Stephen Early, and Neil Stratford. Xenoservers: Accounted execution of untrusted code . In *Proceedings of the 7th Workshop on Hot Topics in Operating Systems*, Rio Rico, AZ, USA, March 1999. (cited on pages: 7, 94)

[SAWJ$^+$96a] I. Stoica, H. Abdel-Wahab, K. Jeffay, S. Baruah, J. Gehrke, and G. Plaxton. A Proportional Share Resource Allocation Algorithm for Real-Time, Time-Shared Systems. In *Proceedings of the 17th IEEE Real-Time Systems Symposium*, pages 288–299, Washington, DC, USA, December 1996. (cited on pages: 20, 21, 22, 23, 24, 196)

[SAWJ$^+$96b] I. Stoica, H. Abdel-Wahab, K. Jeffay, S. Baruah, J. Gehrke, and G. Plaxton. A Proportional Share Resource Allocation Algorithm for Real-Time, Time-Shared Systems. Technical Report TR-96-38, Computer Science Department, University of North Carolina, September 1996. Extended version of [SAWJ$^+$96a]. (cited on pages: 22)

[SAWJ97] I. Stoica, H. Abdel-Wahab, and K. Jeffay. On the Duality between Resource Reservation and Proportional Share Resource Allocation. In *Proceedings of the SPIE/ACM Conference on Multimedia Computing and Networking*, pages 207–214, San Jose, CA, USA, February 1997. (cited on pages: 26)

[SCO90] Margo Seltzer, Peter Chen, and John Ousterhout. Disk Scheduling Revisited. In *Proceedings of the Winter 1990 USENIX Conference*, pages 313–324, Washington, D.C., USA, January 1990. (cited on pages: 78)

[SGG$^+$99] David C. Steere, Ashvin Goel, Joshua Gruenberg, Dylan McNamee, Calton Pu, and Jonathan Walpole. A Feedback-driven Proportion Allocator for Real-Rate Scheduling. In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation (OSDI'99)*, pages 145–158, New Orleans, LA, USA, February 1999. (cited on pages: 27, 28, 87)

[She95a] Scott Shenker. Fundamental Design Issues for the Future Internet. *IEEE Journal on Selected Areas In Communications*, 13(7):1176–1188, September 1995. (cited on pages: 69)

[She95b] Scott Shenker. Service models and pricing policies for and integrated services Internet. In B. Kahin and J. Keller, editors, *Public Access to the Internet*, pages 315–337. MIT Press, Cambridge, MA, USA, 1995. (cited on pages: 56)

[Shn92] B. Shneiderman. *Designing the User Interface: Strategies for Effective Human-Computer Interaction.* Addison-Wesley, Reading, MA, USA, 2nd edition, 1992. Quoted after [NL97]. (cited on pages: 108)

[SHS99] David Sullivan, Robert Haas, and Margo Seltzer. Tickets and Currencies Revisted: Extensions to Multi-Resource Lottery Scheduling. In *Proceedings of the 7th Workshop on Hot Topics in Operating Systems*, Rio Rico, AZ, USA, March 1999. (cited on pages: 26, 90)

[SLR86] L. Sha, J. P. Lehoczky, and R. Rajkumar. Solutions for some practical problems in prioritized preemptive scheduling. In *Proceedings of the 6th IEEE Real-Time Systems Symposium*, pages 181–191, December 1986. Quoted from [Hyd94]. (cited on pages: 17)

[SM99a] N. Stratford and R. Mortier. An Economic Approach to Adaptive Resource Management. In *Proceedings of the 7th Workshop on Hot Topics in Operating Systems*, Rio Rico, AZ, USA, March 1999. (cited on pages: 4, 7)

[SM99b] Neil Stratford and Richard Mortier. QoS User Agent Software. PEGASUS II Deliverable Report 4.3.3, University of Cambridge Computer Laboratory, New Museums Site, Pembroke Street, Cambridge CB2 3QG, UK, March 1999. (cited on pages: 4, 85)

[Sol98] David A. Solomon. *Inside Windows NT.* Microsoft Press, 2nd edition edition, 1998. (cited on pages: 13)

[SS00] David G. Sullivan and Margo I. Seltzer. Isolation with flexibility: a resource management framework for central servers. In *Proceedings of the USENIX 2000 Annual Technical Conference*, pages 337–350, San Diego, CA, USA, June 2000. (cited on pages: 26, 90, 91)

[SSS99] Elizabeth Shriver, Christopher Small, and Keith Smith. Why Does File System Prefetching Work? In *Proceedings of the USENIX 1999 Annual Technical Conference*, Monterey, CA, USA, June 1999. (cited on pages: 78)

[Str98] Neil Stratford. Separate sdoms and adoms. Personal communication, July 1998. (cited on pages: 38)

[Str00] Neil Stratford. Congestion Pricing: A Testbed Implementation. Presentation at Multi-Service Networks 2000, July 2000.

http://research.microsoft.com/research/network/talks/Neil_S_Cos2k.pdf. (cited on pages: 63, 81)

[Sut68] I. E. Sutherland. A Futures Market in Computer Time. *Communications of the Association for Computing Machinery*, 11(6):449–451, June 1968. (cited on pages: 2)

[SV98] Prashant Shenoy and Harrick M. Vin. Cello: A Disk Scheduling Framework for Next Generation Operating Systems. In *Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems*, pages 44–55, Madison, WI, USA, June 1998. Extended version available as Technical Report CS-TR-97-27 from the Department of Computer Sciences, Univ. of Texas at Austin. (cited on pages: 78)

[Tha98] M. T. Tham. Dealing with Measurement Noise — A gentle Introduction to Noise Filtering. http://lorien.ncl.ac.uk/ming/filter/filter.htm, 1998. (cited on pages: 116)

[Tho78] Ken Thompson. UNIX implementation. *The Bell Technical Journal*, 57(6):1931–1946, July-August 1978. Special issue on the UNIX Time-Sharing System. (cited on pages: 12)

[Vah96] Uresh Vahalia. *Unix internals: The new frontier.* Prentice-Hall, 1996. (cited on pages: 13, 26, 78)

[Var92] Hal Varian. *Microeconomic Analysis.* W.W. Norton & Company, New York, London, 3rd edition edition, 1992. (cited on pages: 53, 56)

[VL87] G. Varghese and A. Lauck. Hashed and Hierarchical Timing Wheels: Data Structures for the Efficient Implementation of a Timer Facility. In *Proceedings of the 11th ACM SIGOPS Symposium on Operating Systems Principles (SOSP'87)*, pages 25–38, Austin, TX, USA, November 1987. (cited on pages: 121)

[VLE00] Amin Vahdat, Alvin Lebeck, and Carla Ellis. Every Joule is Precious: The Case for Revisiting Operating System Design for Energy Efficiency. In *Proceedings of the 9th ACM SIGOPS European Workshop*, pages 31–36, Kolding, Denmark, September 2000. (cited on pages: 80)

[VM94] Hal Varian and Jeffrey K. MacKie-Mason. Generalized vickrey auctions. http://www-personal.umich.edu/ jmm/papers.html#gva, July 1994. Working Paper. (cited on pages: 49)

[vRD00] Guido van Rossum and Fred Drake, Jr. *Python Reference Manual*, October 2000. http://www.python.org/doc/current/ref/ref.html. (cited on pages: 99)

[Wal91] Gregory Wallace. The JPEG still picture compression standard. *Communications of the Association for Computing Machinery*, 34(4):30–44, April 1991. (cited on pages: 106)

[Wel96] Michael P. Wellman. Market-oriented programming: Some early lessons. In S. Clearwater, editor, *Market-Based Control: A Paradigm for Distributed Resource Allocation*, pages 74–95. World Scientific, 1996. (cited on pages: 46, 47, 52, 53)

[WHH$^+$92] Carl A. Waldspurger, Tad Hogg, Bernardo A. Huberman, Jeff O. Kephart, and Scott Stornetta. Spawn: A Distributed Computational Economy. *IEEE Transactions on Software Engineering*, 18(2):103–117, February 1992. (cited on pages: 46, 47, 50)

[WW94] Carl A. Waldspurger and William E. Weihl. Lottery Scheduling: Flexible Proportional-Share Resource Mangement. In *Proceedings of the 1st Symposium on Operating Systems Design and Implementation (OSDI'94)*, pages 1–11, Monterey, CA, USA, November 1994. (cited on pages: 20, 25, 85, 104, 132)

[WW95] Carl A. Waldspurger and William E. Weihl. Stride Scheduling: Deterministic Proportional-Share Resource Mangement. Technical Memorandum MIT/LCS/TM-528, June 1995. MIT Laboratory for Computer Science. (cited on pages: 20, 26, 104)

[WWWM98] William Walsh, Michael Wellman, Peter Wurman, and Jeffrey MacKie-Mason. Some Economics of Market-Based Distributed Scheduling. In *Proceedings of the 18th International Conference on Distributed Computing Systems*, Amsterdam, Netherlands, 1998. (cited on pages: 49)

[YD00] Karim Yaghmour and Michael R. Dagenais. Measuring and Characterizing System Behaviour Using Kernel-Level Event Logging. In *Proceedings of the USENIX 2000 Annual Technical Conference*, San Diego, CA, USA, June 2000. (cited on pages: 123)

[YWI96] Hirofumi Yamaki, Michale P. Wellman, and Toru Ishida. A Market-Based Approach to Allocating QoS for Multimedia Applications. In *Proceedings of the 2nd International Conference on Multiagent Systems*, Kyoto, Japan, December 1996. (cited on pages: 47, 53)

[ZFE$^+$01] H. Zeng, X. Fan, C. Ellis, A. Lebeck, and A. Vahdat. ECOSystem: Managing Energy as a First Class Operating System Resource. Technical Report CS-2001-01, Duke University Computer Science Department, March 2001. (cited on pages: 40, 80, 81)

[Zha91] Lixia Zhang. VirtualClock: A New Traffic Control Algorithm for Packet-Switched Networks. *ACM Transactions on Computer Systems*, 9(2):101–124, May 1991. (cited on pages: 20)